Inertia

See the Inertia (Dyadics) section in 'Advanced Topics' part of *sympy.physics.vector* (page 1592) docs.

Rigid Body

Rigid bodies are created in a similar fashion as particles. The RigidBody class generates objects with four attributes: mass, center of mass, a reference frame, and an inertia tuple:

```
>>> 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')
>>> I = outer(A.x, A.x)
>>> # create a rigid body
>>> B = RigidBody('B', P, A, m, (I, P))
```

The mass is specified exactly as is in a particle. Similar to the Particle's .point, the RigidBody's center of mass, .masscenter must be specified. The reference frame is stored in an analogous fashion and holds information about the body's orientation and angular velocity. Finally, the inertia for a rigid body needs to be specified about a point. In *sympy.physics.mechanics* (page 1675), you are allowed to specify any point for this. The most common is the center of mass, as shown in the above code. If a point is selected which is not the center of mass, ensure that the position between the point and the center of mass has been defined. The inertia is specified as a tuple of length two with the first entry being a Dyadic and the second entry being a Point of which the inertia dyadic is defined about.

Dyadic

In *sympy.physics.mechanics* (page 1675), dyadics are used to represent inertia ([Kane1985], [WikiDyadics], [WikiDyadicProducts]). A dyadic is a linear polynomial of component unit dyadics, similar to a vector being a linear polynomial of component unit vectors. A dyadic is the outer product between two vectors which returns a new quantity representing the juxtaposition of these two vectors. For example:

$$\mathbf{\hat{a}_x} \otimes \mathbf{\hat{a}_x} = \mathbf{\hat{a}_x}\mathbf{\hat{a}_x}$$
$$\mathbf{\hat{a}_x} \otimes \mathbf{\hat{a}_y} = \mathbf{\hat{a}_x}\mathbf{\hat{a}_y}$$

Where $\mathbf{\hat{a}_x}\mathbf{\hat{a}_x}$ and $\mathbf{\hat{a}_x}\mathbf{\hat{a}_y}$ are the outer products obtained by multiplying the left side as a column vector by the right side as a row vector. Note that the order is significant.

Some additional properties of a dyadic are:

$$(x\mathbf{v}) \otimes \mathbf{w} = \mathbf{v} \otimes (x\mathbf{w}) = x(\mathbf{v} \otimes \mathbf{w})$$
$$\mathbf{v} \otimes (\mathbf{w} + \mathbf{u}) = \mathbf{v} \otimes \mathbf{w} + \mathbf{v} \otimes \mathbf{u}$$
$$(\mathbf{v} + \mathbf{w}) \otimes \mathbf{u} = \mathbf{v} \otimes \mathbf{u} + \mathbf{w} \otimes \mathbf{u}$$

A vector in a reference frame can be represented as $\begin{bmatrix} a \\ b \\ c \end{bmatrix}$ or $a\mathbf{\hat{i}} + b\mathbf{\hat{j}} + c\mathbf{\hat{k}}$. Similarly, a dyadic can be represented in tensor form:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

or in dyadic form:

$$a_{11}\mathbf{\hat{a}_x}\mathbf{\hat{a}_x} + a_{12}\mathbf{\hat{a}_x}\mathbf{\hat{a}_y} + a_{13}\mathbf{\hat{a}_x}\mathbf{\hat{a}_z} + a_{21}\mathbf{\hat{a}_y}\mathbf{\hat{a}_x} + a_{22}\mathbf{\hat{a}_y}\mathbf{\hat{a}_y} + a_{23}\mathbf{\hat{a}_y}\mathbf{\hat{a}_z} + a_{31}\mathbf{\hat{a}_z}\mathbf{\hat{a}_x} + a_{32}\mathbf{\hat{a}_z}\mathbf{\hat{a}_y} + a_{33}\mathbf{\hat{a}_z}\mathbf{\hat{a}_z}$$

Just as with vectors, the later representation makes it possible to keep track of which frames the dyadic is defined with respect to. Also, the two components of each term in the dyadic need not be in the same frame. The following is valid:

$$\mathbf{\hat{a}_x} \otimes \mathbf{\hat{b}_y} = \mathbf{\hat{a}_x}\mathbf{\hat{b}_y}$$

Dyadics can also be crossed and dotted with vectors; again, order matters:

$$\begin{aligned}
\mathbf{\hat{a}_x}\mathbf{\hat{a}_x} \cdot \mathbf{\hat{a}_x} &= \mathbf{\hat{a}_x} \\
\mathbf{\hat{a}_y}\mathbf{\hat{a}_x} \cdot \mathbf{\hat{a}_x} &= \mathbf{\hat{a}_y} \\
\mathbf{\hat{a}_x}\mathbf{\hat{a}_y} \cdot \mathbf{\hat{a}_x} &= 0 \\
\mathbf{\hat{a}_x} \cdot \mathbf{\hat{a}_x}\mathbf{\hat{a}_x} &= \mathbf{\hat{a}_x} \\
\mathbf{\hat{a}_x} \cdot \mathbf{\hat{a}_x}\mathbf{\hat{a}_y} &= \mathbf{\hat{a}_y} \\
\mathbf{\hat{a}_x} \cdot \mathbf{\hat{a}_y}\mathbf{\hat{a}_x} &= 0 \\
\mathbf{\hat{a}_x} \times \mathbf{\hat{a}_y}\mathbf{\hat{a}_x} &= \mathbf{\hat{a}_z}\mathbf{\hat{a}_x} \\
\mathbf{\hat{a}_x} \times \mathbf{\hat{a}_x}\mathbf{\hat{a}_x} &= 0 \\
\mathbf{\hat{a}_y}\mathbf{\hat{a}_x} \times \mathbf{\hat{a}_z} &= -\mathbf{\hat{a}_y}\mathbf{\hat{a}_y}
\end{aligned}$$

One can also take the time derivative of dyadics or express them in different frames, just like with vectors.

Linear Momentum

The linear momentum of a particle P is defined as:

$$L_P = m\mathbf{v}$$

where m is the mass of the particle P and $\mathbf{v}$ is the velocity of the particle in the inertial frame.[Likins1973]_.

Similarly the linear momentum of a rigid body is defined as:

$$L_B = m\mathbf{v}^*$$

where m is the mass of the rigid body, B, and $\mathbf{v}^*$ is the velocity of the mass center of B in the inertial frame.

Angular Momentum

The angular momentum of a particle P about an arbitrary point O in an inertial frame N is defined as:

$$^N\mathbf{H}^{P/O} = \mathbf{r} \times m\mathbf{v}$$

where $\mathbf{r}$ is a position vector from point O to the particle of mass m and $\mathbf{v}$ is the velocity of the particle in the inertial frame.

Similarly the angular momentum of a rigid body B about a point O in an inertial frame N is defined as:

$$^N\mathbf{H}^{B/O} =^N \mathbf{H}^{B/B^*} +^N \mathbf{H}^{B^*/O}$$

where the angular momentum of the body about it's mass center is:

$$^N\mathbf{H}^{B/B^*} = \mathbf{I}^* \cdot \omega$$

and the angular momentum of the mass center about O is:

$$^N\mathbf{H}^{B^*/O} = \mathbf{r}^* \times m\mathbf{v}^*$$

where $\mathbf{I}^*$ is the central inertia dyadic of rigid body B, ω is the inertial angular velocity of B, $\mathbf{r}^*$ is a position vector from point O to the mass center of B, m is the mass of B and $\mathbf{v}^*$ is the velocity of the mass center in the inertial frame.

Using momenta functions in Mechanics

The following example shows how to use the momenta functions in *sympy.physics.mechanics* (page 1675).

One begins by creating the requisite symbols to describe the system. Then the reference frame is created and the kinematics are done.

```
>>> from sympy import symbols
>>> from sympy.physics.mechanics import dynamicsymbols, ReferenceFrame
>>> from sympy.physics.mechanics import RigidBody, Particle, Point, outer
>>> from sympy.physics.mechanics import linear_momentum, angular_momentum
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> m, M, l1 = symbols('m M l1')
>>> q1d = dynamicsymbols('q1d')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> Ac = O.locatenew('Ac', l1 * N.x)
>>> P = Ac.locatenew('P', l1 * N.x)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, q1d * N.z)
>>> Ac.v2pt_theory(O, N, a)
l1*q1d*N.y
>>> P.v2pt_theory(O, N, a)
2*l1*q1d*N.y
```

Finally, the bodies that make up the system are created. In this case the system consists of a particle Pa and a RigidBody A.

```
>>> Pa = Particle('Pa', P, m)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, M, (I, Ac))
```

Then one can either choose to evaluate the momenta of individual components of the system or of the entire system itself.

```
>>> linear_momentum(N,A)
M*l1*q1d*N.y
>>> angular_momentum(O, N, Pa)
4*l1**2*m*q1d*N.z
>>> linear_momentum(N, A, Pa)
(M*l1*q1d + 2*l1*m*q1d)*N.y
>>> angular_momentum(O, N, A, Pa)
(M*l1**2*q1d + 4*l1**2*m*q1d + q1d)*N.z
```

It should be noted that the user can determine either momenta in any frame in *sympy.physics.mechanics* (page 1675) as the user is allowed to specify the reference frame when calling the function. In other words the user is not limited to determining just inertial linear and angular momenta. Please refer to the docstrings on each function to learn more about how each function works precisely.

Kinetic Energy

The kinetic energy of a particle P is defined as

$$T_P = \frac{1}{2}m\mathbf{v^2}$$

where m is the mass of the particle P and $\mathbf{v}$ is the velocity of the particle in the inertial frame.

Similarly the kinetic energy of a rigid body B is defined as

$$T_B = T_t + T_r$$

where the translational kinetic energy is given by:

$$T_t = \frac{1}{2}m\mathbf{v}^* \cdot \mathbf{v}^*$$

and the rotational kinetic energy is given by:

$$T_r = \frac{1}{2}\omega \cdot \mathbf{I}^* \cdot \omega$$

where m is the mass of the rigid body, $\mathbf{v}^*$ is the velocity of the mass center in the inertial frame, ω is the inertial angular velocity of the body and $\mathbf{I}^*$ is the central inertia dyadic.

Potential Energy

Potential energy is defined as the energy possessed by a body or system by virtue of its position or arrangement.

Since there are a variety of definitions for potential energy, this is not discussed further here. One can learn more about this in any elementary text book on dynamics.

Lagrangian

The Lagrangian of a body or a system of bodies is defined as:

$$\mathcal{L} = T - V$$

where T and V are the kinetic and potential energies respectively.

Using energy functions in Mechanics

The following example shows how to use the energy functions in *sympy.physics.mechanics* (page 1675).

As was discussed above in the momenta functions, one first creates the system by going through an identical procedure.

```
>>> from sympy import symbols
>>> from sympy.physics.mechanics import dynamicsymbols, ReferenceFrame, outer
>>> from sympy.physics.mechanics import RigidBody, Particle
>>> from sympy.physics.mechanics import kinetic_energy, potential_energy,␣
↪Point
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> m, M, l1, g, h, H = symbols('m M l1 g h H')
>>> omega = dynamicsymbols('omega')
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> O.set_vel(N, 0 * N.x)
>>> Ac = O.locatenew('Ac', l1 * N.x)
>>> P = Ac.locatenew('P', l1 * N.x)
>>> a = ReferenceFrame('a')
>>> a.set_ang_vel(N, omega * N.z)
>>> Ac.v2pt_theory(O, N, a)
l1*omega*N.y
>>> P.v2pt_theory(O, N, a)
2*l1*omega*N.y
>>> Pa = Particle('Pa', P, m)
>>> I = outer(N.z, N.z)
>>> A = RigidBody('A', Ac, a, M, (I, Ac))
```

The user can then determine the kinetic energy of any number of entities of the system:

```
>>> kinetic_energy(N, Pa)
2*l1**2*m*omega**2
```

(continues on next page)

(continued from previous page)

```
>>> kinetic_energy(N, Pa, A)
M*l1**2*omega**2/2 + 2*l1**2*m*omega**2 + omega**2/2
```

It should be noted that the user can determine either kinetic energy relative to any frame in *sympy.physics.mechanics* (page 1675) as the user is allowed to specify the reference frame when calling the function. In other words the user is not limited to determining just inertial kinetic energy.

For potential energies, the user must first specify the potential energy of every entity of the system using the *sympy.physics.mechanics.rigidbody.RigidBody.potential_energy* (page 1749) property. The potential energy of any number of entities comprising the system can then be determined:

```
>>> Pa.potential_energy = m * g * h
>>> A.potential_energy = M * g * H
>>> potential_energy(A, Pa)
H*M*g + g*h*m
```

One can also determine the Lagrangian for this system:

```
>>> from sympy.physics.mechanics import Lagrangian
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> Lagrangian(N, Pa, A)
-H*M*g + M*l1**2*omega**2/2 - g*h*m + 2*l1**2*m*omega**2 + omega**2/2
```

Please refer to the docstrings to learn more about each function.

Kane's Method in Physics/Mechanics

sympy.physics.mechanics (page 1675) provides functionality for deriving equations of motion using Kane's method [Kane1985]. This document will describe Kane's method as used in this module, but not how the equations are actually derived.

Structure of Equations

In *sympy.physics.mechanics* (page 1675) we are assuming there are 5 basic sets of equations needed to describe a system. They are: holonomic constraints, non-holonomic constraints, kinematic differential equations, dynamic equations, and differentiated non-holonomic equations.

$$\begin{aligned}\mathbf{f_h}(q,t) &= 0\\ \mathbf{k_{nh}}(q,t)u + \mathbf{f_{nh}}(q,t) &= 0\\ \mathbf{k_{kq}}(q,t)\dot{q} + \mathbf{k_{ku}}(q,t)u + \mathbf{f_k}(q,t) &= 0\\ \mathbf{k_d}(q,t)\dot{u} + \mathbf{f_d}(q,\dot{q},u,t) &= 0\\ \mathbf{k_{dnh}}(q,t)\dot{u} + \mathbf{f_{dnh}}(q,\dot{q},u,t) &= 0\end{aligned}$$

In *sympy.physics.mechanics* (page 1675) holonomic constraints are only used for the linearization process; it is assumed that they will be too complicated to solve for the dependent coordinate(s). If you are able to easily solve a holonomic constraint, you should consider redefining your problem in terms of a smaller set of coordinates. Alternatively, the time-differentiated holonomic constraints can be supplied.

Kane's method forms two expressions, F_r and F_r^*, whose sum is zero. In this module, these expressions are rearranged into the following form:

$$\mathbf{M}(q,t)\dot{u} = \mathbf{f}(q,\dot{q},u,t)$$

For a non-holonomic system with o total speeds and m motion constraints, we will get o - m equations. The mass-matrix/forcing equations are then augmented in the following fashion:

$$\mathbf{M}(q,t) = \begin{bmatrix}\mathbf{k_d}(q,t)\\ \mathbf{k_{dnh}}(q,t)\end{bmatrix}$$

$$_{(\mathbf{forcing})}(q,\dot{q},u,t) = \begin{bmatrix}-\mathbf{f_d}(q,\dot{q},u,t)\\ -\mathbf{f_{dnh}}(q,\dot{q},u,t)\end{bmatrix}$$

Kane's Method in Physics/Mechanics

The formulation of the equations of motion in *sympy.physics.mechanics* (page 1675) starts with creation of a KanesMethod object. Upon initialization of the KanesMethod object, an inertial reference frame needs to be supplied. along with some basic system information, such as coordinates and speeds

```
>>> from sympy.physics.mechanics import *
>>> N = ReferenceFrame('N')
>>> q1, q2, u1, u2 = dynamicsymbols('q1 q2 u1 u2')
>>> q1d, q2d, u1d, u2d = dynamicsymbols('q1 q2 u1 u2', 1)
>>> KM = KanesMethod(N, [q1, q2], [u1, u2])
```

It is also important to supply the order of coordinates and speeds properly if there are dependent coordinates and speeds. They must be supplied after independent coordinates and speeds or as a keyword argument; this is shown later.

```
>>> q1, q2, q3, q4 = dynamicsymbols('q1 q2 q3 q4')
>>> u1, u2, u3, u4 = dynamicsymbols('u1 u2 u3 u4')
>>> # Here we will assume q2 is dependent, and u2 and u3 are dependent
>>> # We need the constraint equations to enter them though
>>> KM = KanesMethod(N, [q1, q3, q4], [u1, u4])
```

Additionally, if there are auxiliary speeds, they need to be identified here. See the examples for more information on this. In this example u4 is the auxiliary speed.

```
>>> KM = KanesMethod(N, [q1, q3, q4], [u1, u2, u3], u_auxiliary=[u4])
```

Kinematic differential equations must also be supplied; there are to be provided as a list of expressions which are each equal to zero. A trivial example follows:

```
>>> kd = [q1d - u1, q2d - u2]
```

Turning on mechanics_printing() makes the expressions significantly shorter and is recommended. Alternatively, the mprint and mpprint commands can be used.

If there are non-holonomic constraints, dependent speeds need to be specified (and so do dependent coordinates, but they only come into play when linearizing the system). The constraints need to be supplied in a list of expressions which are equal to zero, trivial motion and configuration constraints are shown below:

```
>>> N = ReferenceFrame('N')
>>> q1, q2, q3, q4 = dynamicsymbols('q1 q2 q3 q4')
>>> q1d, q2d, q3d, q4d = dynamicsymbols('q1 q2 q3 q4', 1)
>>> u1, u2, u3, u4 = dynamicsymbols('u1 u2 u3 u4')
>>> #Here we will assume q2 is dependent, and u2 and u3 are dependent
>>> speed_cons = [u2 - u1, u3 - u1 - u4]
>>> coord_cons = [q2 - q1]
>>> q_ind = [q1, q3, q4]
>>> q_dep = [q2]
>>> u_ind = [u1, u4]
>>> u_dep = [u2, u3]
>>> kd = [q1d - u1, q2d - u2, q3d - u3, q4d - u4]
>>> KM = KanesMethod(N, q_ind, u_ind, kd,
...           q_dependent=q_dep,
...           configuration_constraints=coord_cons,
...           u_dependent=u_dep,
...           velocity_constraints=speed_cons)
```

A dictionary returning the solved $\dot{q}$'s can also be solved for:

```
>>> mechanics_printing(pretty_print=False)
>>> KM.kindiffdict()
{q1': u1, q2': u2, q3': u3, q4': u4}
```

The final step in forming the equations of motion is supplying a list of bodies and particles, and a list of 2-tuples of the form `(Point, Vector)` or `(ReferenceFrame, Vector)` to represent applied forces and torques.

```
>>> N = ReferenceFrame('N')
>>> q, u = dynamicsymbols('q u')
>>> qd, ud = dynamicsymbols('q u', 1)
>>> P = Point('P')
>>> P.set_vel(N, u * N.x)
>>> Pa = Particle('Pa', P, 5)
>>> BL = [Pa]
>>> FL = [(P, 7 * N.x)]
>>> KM = KanesMethod(N, [q], [u], [qd - u])
>>> (fr, frstar) = KM.kanes_equations(BL, FL)
>>> KM.mass_matrix
Matrix([[5]])
>>> KM.forcing
Matrix([[7]])
```

When there are motion constraints, the mass matrix is augmented by the $k_{dnh}(q, t)$ matrix, and the forcing vector by the $f_{dnh}(q, \dot{q}, u, t)$ vector.

There are also the "full" mass matrix and "full" forcing vector terms, these include the kinematic differential equations; the mass matrix is of size (n + o) x (n + o), or square and the size of all coordinates and speeds.

```
>>> KM.mass_matrix_full
Matrix([
[1, 0],
[0, 5]])
```

(continues on next page)

(continued from previous page)

```
>>> KM.forcing_full
Matrix([
[u],
[7]])
```

Exploration of the provided examples is encouraged in order to gain more understanding of the `KanesMethod` object.

Lagrange's Method in Physics/Mechanics

sympy.physics.mechanics (page 1675) provides functionality for deriving equations of motion using Lagrange's method. This document will describe Lagrange's method as used in this module, but not how the equations are actually derived.

Structure of Equations

In *sympy.physics.mechanics* (page 1675) we are assuming there are 3 basic sets of equations needed to describe a system; the constraint equations, the time differentiated constraint equations and the dynamic equations.

$$\begin{aligned} \mathbf{m_c}(q,t)\dot{q} + \mathbf{f_c}(q,t) &= 0 \\ \mathbf{m_{dc}}(\dot{q},q,t)\ddot{q} + \mathbf{f_{dc}}(\dot{q},q,t) &= 0 \\ \mathbf{m_d}(\dot{q},q,t)\ddot{q} + \square_{\mathbf{c}}(q,t)\lambda + \mathbf{f_d}(\dot{q},q,t) &= 0 \end{aligned}$$

In this module, the expressions formed by using Lagrange's equations of the second kind are rearranged into the following form:

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

where in the case of a system without constraints:

$$x = \ddot{q}$$

For a constrained system with n generalized speeds and m constraints, we will get n - m equations. The mass-matrix/forcing equations are then augmented in the following fashion:

$$\begin{aligned} x &= \begin{bmatrix} \ddot{q} \\ \lambda \end{bmatrix} \\ \mathbf{M}(q,t) &= \begin{bmatrix} \mathbf{m_d}(q,t) & \square_{\mathbf{c}}(q,t) \end{bmatrix} \\ \mathbf{F}(\dot{q},q,t) &= \begin{bmatrix} \mathbf{f_d}(q,\dot{q},t) \end{bmatrix} \end{aligned}$$

Lagrange's Method in Physics/Mechanics

The formulation of the equations of motion in *sympy.physics.mechanics* (page 1675) using Lagrange's Method starts with the creation of generalized coordinates and a Lagrangian. The Lagrangian can either be created with the `Lagrangian` function or can be a user supplied function. In this case we will supply the Lagrangian.

```
>>> from sympy.physics.mechanics import *
>>> q1, q2 = dynamicsymbols('q1 q2')
>>> q1d, q2d = dynamicsymbols('q1 q2', 1)
>>> L = q1d**2 + q2d**2
```

To formulate the equations of motion we create a `LagrangesMethod` object. The Lagrangian and generalized coordinates need to be supplied upon initialization.

```
>>> LM = LagrangesMethod(L, [q1, q2])
```

With that the equations of motion can be formed.

```
>>> mechanics_printing(pretty_print=False)
>>> LM.form_lagranges_equations()
Matrix([
[2*q1''],
[2*q2'']])
```

It is possible to obtain the mass matrix and the forcing vector.

```
>>> LM.mass_matrix
Matrix([
[2, 0],
[0, 2]])

>>> LM.forcing
Matrix([
[0],
[0]])
```

If there are any holonomic or non-holonomic constraints, they must be supplied as keyword arguments (`hol_coneqs` and `nonhol_coneqs` respectively) in a list of expressions which are equal to zero. Modifying the example above, the equations of motion can then be generated:

```
>>> LM = LagrangesMethod(L, [q1, q2], hol_coneqs=[q1 - q2])
```

When the equations of motion are generated in this case, the Lagrange multipliers are introduced; they are represented by `lam1` in this case. In general, there will be as many multipliers as there are constraint equations.

```
>>> LM.form_lagranges_equations()
Matrix([
[ lam1 + 2*q1''],
[-lam1 + 2*q2'']])
```

Also in the case of systems with constraints, the 'full' mass matrix is augmented by the $k_{dc}(q, t)$ matrix, and the forcing vector by the $f_{dc}(q, \dot{q}, t)$ vector. The 'full' mass matrix is of size (2n + o) x (2n + o), i.e. it's a square matrix.

```
>>> LM.mass_matrix_full
Matrix([
[1, 0, 0,  0,  0],
[0, 1, 0,  0,  0],
[0, 0, 2,  0, -1],
```

(continues on next page)

(continued from previous page)

```
[0, 0, 0,  2,  1],
[0, 0, 1, -1,  0]])
>>> LM.forcing_full
Matrix([
[q1'],
[q2'],
[  0],
[  0],
[  0]])
```

If there are any non-conservative forces or moments acting on the system, they must also be supplied as keyword arguments in a list of 2-tuples of the form `(Point, Vector)` or `(ReferenceFrame, Vector)` where the `Vector` represents the non-conservative forces and torques. Along with this 2-tuple, the inertial frame must also be specified as a keyword argument. This is shown below by modifying the example above:

```
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, q1d * N.x)
>>> FL = [(P, 7 * N.x)]
>>> LM = LagrangesMethod(L, [q1, q2], forcelist=FL, frame=N)
>>> LM.form_lagranges_equations()
Matrix([
[2*q1'' - 7],
[    2*q2'']])
```

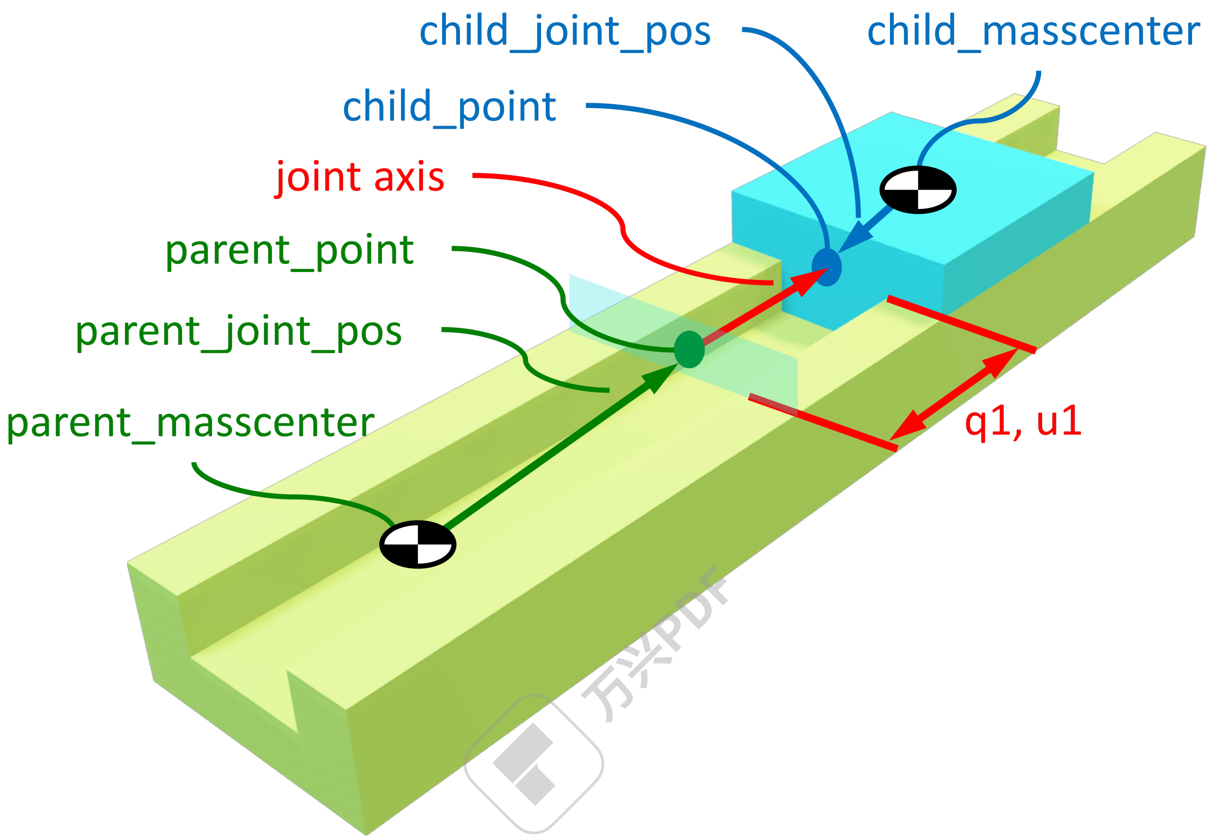
Exploration of the provided examples is encouraged in order to gain more understanding of the `LagrangesMethod` object.

Joints Framework in Physics/Mechanics

sympy.physics.mechanics (page 1675) provides a joints framework. This system consists of two parts. The first are the *joints* (page 1770) themselves, which are used to create connections between *bodies* (page 1756). The second part is the *JointsMethod* (page 1780), which is used to form the equations of motion. Both of these parts are doing what we can call "book-keeping": keeping track of the relationships between *bodies* (page 1756).

Joints in Physics/Mechanics

The general task of the *joints* (page 1770) is creating kinematic relationships between *bodies* (page 1756). Each joint has a setup as shown in the image below (this is the example of the *PrismaticJoint* (page 1777)).



As can be seen in this image, each joint needs several objects in order to define the relationships. First off it needs two bodies: the parent body (shown in green) and the child body (shown in blue). Both of these bodies have a mass center from which the position of the joint is defined. In the parent body the vector from the mass center to the `parent_point` is called the `parent_joint_pos`. For the child body these are called the `child_point` and `child_joint_pos`. The orientation of the joint in each body is defined by the `parent_axis` and `child_axis`. These two vectors are aligned as explained in the *Joint* (page 1770) notes and are in the image parallel to the red vector. As last the joint also needs *dynamicsymbols()* (page 1666) as generalized coordinates and speeds. In the case of the *PrismaticJoint* (page 1777) shown above, the generalized coordinate `q_1` distance along the joint axis. And the generalized speed `u_1` is its velocity.

With the information listed above, the joint defines the following relationships. It first defines the kinematic differential equations, which relate the generalized coordinates to the generalized speeds. Next, it orients the parent and child body with respect to each other. After which it also defines their velocity relationships.

The following code shows the creation of a *PrismaticJoint* (page 1777) as shown above with arbitrary linked position vectors:

```
>>> from sympy.physics.mechanics import *
>>> mechanics_printing(pretty_print=False)
```

(continues on next page)

(continued from previous page)

```
>>> q1, u1 = dynamicsymbols('q1, u1')
>>> parent = Body('parent')
>>> child = Body('child')
>>> joint = PrismaticJoint(
...     'slider', parent, child, q1, u1,
...     parent_joint_pos=parent.frame.x / 2 + parent.frame.y / 10,
...     child_joint_pos=-(child.frame.x + child.frame.y) / 10,
...     parent_axis=parent.frame.x, child_axis=child.frame.x)
>>> joint.kdes
[u1 - q1']
>>> child.masscenter.pos_from(parent.masscenter)
(q1 + 1/2)*parent_frame.x + 1/10*parent_frame.y + 1/10*child_frame.x + 1/
↪10*child_frame.y
>>> child.masscenter.vel(parent.frame)
u1*parent_frame.x
```

JointsMethod in Physics/Mechanics

After defining the entire system you can use the *JointsMethod* (page 1780) to parse the system and form the equations of motion. In this process the *JointsMethod* (page 1780) only does the “book-keeping” of the joints. It uses another method, like the *KanesMethod* (page 1764), as its backend for forming the equations of motion.

In the code below we form the equations of motion of the single *PrismaticJoint* (page 1777) above.

```
>>> method = JointsMethod(parent, joint)
>>> method.form_eoms()
Matrix([[-child_mass*u1']])
>>> type(method.method)  # The method working in the backend
<class 'sympy.physics.mechanics.kane.KanesMethod'>
```

Symbolic Systems in Physics/Mechanics

The $SymbolicSystem$ class in physics/mechanics is a location for the pertinent information of a multibody dynamic system. In its most basic form it contains the equations of motion for the dynamic system, however, it can also contain information regarding the loads that the system is subject to, the bodies that the system is comprised of and any additional equations the user feels is important for the system. The goal of this class is to provide a unified output format for the equations of motion that numerical analysis code can be designed around.

SymbolicSystem Example Usage

This code will go over the manual input of the equations of motion for the simple pendulum that uses the Cartesian location of the mass as the generalized coordinates into *SymbolicSystem*.

The equations of motion are formed in the physics/mechanics/examples. In that spot the variables q1 and q2 are used in place of x and y and the reference frame is rotated 90 degrees.

```
>>> from sympy import atan, symbols, Matrix
>>> from sympy.physics.mechanics import (dynamicsymbols, ReferenceFrame,
...                                      Particle, Point)
>>> import sympy.physics.mechanics.system as system
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
```

The first step will be to initialize all of the dynamic and constant symbols.

```
>>> x, y, u, v, lam = dynamicsymbols('x y u v lambda')
>>> m, l, g = symbols('m l g')
```

Next step is to define the equations of motion in multiple forms:

[1] Explicit form where the kinematics and dynamics are combined
x’ = F_1(x, t, r, p)

[2] Implicit form where the kinematics and dynamics are combined
M_2(x, p) x’ = F_2(x, t, r, p)

[3] Implicit form where the kinematics and dynamics are separate
M_3(q, p) u’ = F_3(q, u, t, r, p) q’ = G(q, u, t, r, p)

where

x : states, e.g. [q, u] t : time r : specified (exogenous) inputs p : constants q : generalized coordinates u : generalized speeds F_1 : right hand side of the combined equations in explicit form F_2 : right hand side of the combined equations in implicit form F_3 : right hand side of the dynamical equations in implicit form M_2 : mass matrix of the combined equations in implicit form M_3 : mass matrix of the dynamical equations in implicit form G : right hand side of the kinematical differential equations

```
>>> dyn_implicit_mat = Matrix([[1, 0, -x/m],
...                            [0, 1, -y/m],
...                            [0, 0, l**2/m]])
>>> dyn_implicit_rhs = Matrix([0, 0, u**2 + v**2 - g*y])
>>> comb_implicit_mat = Matrix([[1, 0, 0, 0, 0],
...                             [0, 1, 0, 0, 0],
...                             [0, 0, 1, 0, -x/m],
...                             [0, 0, 0, 1, -y/m],
...                             [0, 0, 0, 0, l**2/m]])
>>> comb_implicit_rhs = Matrix([u, v, 0, 0, u**2 + v**2 - g*y])
>>> kin_explicit_rhs = Matrix([u, v])
>>> comb_explicit_rhs = comb_implicit_mat.LUsolve(comb_implicit_rhs)
```

Now the reference frames, points and particles will be set up so this information can be passed into *system.SymbolicSystem* in the form of a bodies and loads iterable.

```
>>> theta = atan(x/y)
>>> omega = dynamicsymbols('omega')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [theta, N.z])
>>> A.set_ang_vel(N, omega * N.z)
>>> O = Point('O')
>>> O.set_vel(N, 0)
>>> P = O.locatenew('P', l * A.x)
>>> P.v2pt_theory(O, N, A)
l*omega*A.y
>>> Pa = Particle('Pa', P, m)
```

Now the bodies and loads iterables need to be initialized.

```
>>> bodies = [Pa]
>>> loads = [(P, g * m * N.x)]
```

The equations of motion are in the form of a differential algebraic equation (DAE) and DAE solvers need to know which of the equations are the algebraic expressions. This information is passed into $SymbolicSystem$ as a list specifying which rows are the algebraic equations. In this example it is a different row based on the chosen equations of motion format. The row index should always correspond to the mass matrix that is being input to the $SymbolicSystem$ class but will always correspond to the row index of the combined dynamics and kinematics when being accessed from the $SymbolicSystem$ class.

```
>>> alg_con = [2]
>>> alg_con_full = [4]
```

An iterable containing the states now needs to be created for the system. The $SymbolicSystem$ class can determine which of the states are considered coordinates or speeds by passing in the indexes of the coordinates and speeds. If these indexes are not passed in the object will not be able to differentiate between coordinates and speeds.

```
>>> states = (x, y, u, v, lam)
>>> coord_idxs = (0, 1)
>>> speed_idxs = (2, 3)
```

Now the equations of motion instances can be created using the above mentioned equations of motion formats.

```
>>> symsystem1 = system.SymbolicSystem(states, comb_explicit_rhs,
...                                    alg_con=alg_con_full, bodies=bodies,
...                                    loads=loads)
>>> symsystem2 = system.SymbolicSystem(states, comb_implicit_rhs,
...                                    mass_matrix=comb_implicit_mat,
...                                    alg_con=alg_con_full,
...                                    coord_idxs=coord_idxs)
>>> symsystem3 = system.SymbolicSystem(states, dyn_implicit_rhs,
...                                    mass_matrix=dyn_implicit_mat,
...                                    coordinate_derivatives=kin_explicit_
↪rhs,
...                                    alg_con=alg_con,
...                                    coord_idxs=coord_idxs,
...                                    speed_idxs=speed_idxs)
```

Like coordinates and speeds, the bodies and loads attributes can only be accessed if they are specified during initialization of the *SymbolicSystem* class. Lastly here are some attributes accessible from the *SymbolicSystem* class.

```
>>> symsystem1.states
Matrix([
[     x],
[     y],
[     u],
[     v],
[lambda]])
>>> symsystem2.coordinates
Matrix([
[x],
[y]])
>>> symsystem3.speeds
Matrix([
[u],
[v]])
>>> symsystem1.comb_explicit_rhs
Matrix([
[                          u],
[                          v],
[(-g*y + u**2 + v**2)*x/l**2],
[(-g*y + u**2 + v**2)*y/l**2],
[m*(-g*y + u**2 + v**2)/l**2]])
>>> symsystem2.comb_implicit_rhs
Matrix([
[                 u],
[                 v],
[                 0],
[                 0],
[-g*y + u**2 + v**2]])
>>> symsystem2.comb_implicit_mat
Matrix([
[1, 0, 0, 0,      0],
[0, 1, 0, 0,      0],
[0, 0, 1, 0,   -x/m],
[0, 0, 0, 1,   -y/m],
[0, 0, 0, 0, l**2/m]])
>>> symsystem3.dyn_implicit_rhs
Matrix([
[                 0],
[                 0],
[-g*y + u**2 + v**2]])
>>> symsystem3.dyn_implicit_mat
Matrix([
[1, 0,   -x/m],
[0, 1,   -y/m],
[0, 0, l**2/m]])
>>> symsystem3.kin_explicit_rhs
Matrix([
[u],
```

(continues on next page)

(continued from previous page)

```
[v]])
>>> symsystem1.alg_con
[4]
>>> symsystem1.bodies
(Pa,)
>>> symsystem1.loads
((P, g*m*N.x),)
```

Linearization in Physics/Mechanics

sympy.physics.mechanics (page 1675) includes methods for linearizing the generated equations of motion (EOM) about an operating point (also known as the trim condition). Note that this operating point doesn't have to be an equilibrium position, it just needs to satisfy the equations of motion.

Linearization is accomplished by taking the first order Taylor expansion of the EOM about the operating point. When there are no dependent coordinates or speeds this is simply the jacobian of the right hand side about q and u. However, in the presence of constraints more care needs to be taken. The linearization methods provided here handle these constraints correctly.

Background

In *sympy.physics.mechanics* (page 1675) we assume all systems can be represented in the following general form:

$$\begin{aligned}
f_c(q,t) &= 0_{l\times 1}\\
f_v(q,u,t) &= 0_{m\times 1}\\
f_a(q,\dot{q},u,\dot{u},t) &= 0_{m\times 1}\\
f_0(q,\dot{q},t) + f_1(q,u,t) &= 0_{n\times 1}\\
f_2(q,u,\dot{u},t) + f_3(q,\dot{q},u,r,t) + f_4(q,\lambda,t) &= 0_{(o-m+k)\times 1}
\end{aligned}$$

where

$$\begin{aligned}
q,\dot{q} &\in \mathbb{R}^n\\
u,\dot{u} &\in \mathbb{R}^o\\
r &\in \mathbb{R}^s\\
\lambda &\in \mathbb{R}^k
\end{aligned}$$

In this form,

- f_c represents the configuration constraint equations
- f_v represents the velocity constraint equations
- f_a represents the acceleration constraint equations
- f_0 and f_1 form the kinematic differential equations
- f_2, f_3, and f_4 form the dynamic differential equations
- q and $\dot{q}$ are the generalized coordinates and their derivatives

- u and $\dot{u}$ are the generalized speeds and their derivatives
- r is the system inputs
- λ is the Lagrange multipliers

This generalized form is held inside the `Linearizer` class, which performs the actual linearization. Both `KanesMethod` and `LagrangesMethod` objects have methods for forming the linearizer using the `to_linearizer` class method.

A Note on Dependent Coordinates and Speeds

If the system being linearized contains constraint equations, this results in not all generalized coordinates being independent (i.e. q_1 may depend on q_2). With l configuration constraints, and m velocity constraints, there are l dependent coordinates and m dependent speeds.

In general, you may pick any of the coordinates and speeds to be dependent, but in practice some choices may result in undesirable singularites. Methods for deciding which coordinates/speeds to make dependent is behind the scope of this guide. For more information, please see [Blajer1994].

Once the system is coerced into the generalized form, the linearized EOM can be solved for. The methods provided in *`sympy.physics.mechanics`* (page 1675) allow for two different forms of the linearized EOM:

M**,** A**, and** B

In this form, the forcing matrix is linearized into two separate matrices A and B. This is the default form of the linearized EOM. The resulting equations are:

$$M \begin{bmatrix} \delta\dot{q} \\ \delta\dot{u} \\ \delta\lambda \end{bmatrix} = A \begin{bmatrix} \delta q_i \\ \delta u_i \end{bmatrix} + B \begin{bmatrix} \delta r \end{bmatrix}$$

where

$$M \in \mathbb{R}^{(n+o+k)\times(n+o+k)}$$
$$A \in \mathbb{R}^{(n+o+k)\times(n-l+o-m)}$$
$$B \in \mathbb{R}^{(n+o+k)\times s}$$

Note that q_i and u_i are just the independent coordinates and speeds, while q and u contains both the independent and dependent coordinates and speeds.

A **and** B

In this form, the linearized EOM are brought into explicit first order form, in terms of just the independent coordinates and speeds. This form is often used in stability analysis or control theory. The resulting equations are:

$$\begin{bmatrix} \delta\dot{q}_i \\ \delta\dot{u}_i \end{bmatrix} = A \begin{bmatrix} \delta q_i \\ \delta u_i \end{bmatrix} + B \begin{bmatrix} \delta r \end{bmatrix}$$

where

$$A \in \mathbb{R}^{(n-l+o-m)\times(n-l+o-m)}$$
$$B \in \mathbb{R}^{(n-l+o-m)\times s}$$

To use this form set `A_and_B=True` in the `linearize` class method.

Linearizing Kane's Equations

After initializing the KanesMethod object and forming F_r and F_r^* using the kanes_equations class method, linearization can be accomplished in a couple ways. The different methods will be demonstrated with a simple pendulum system:

```
>>> from sympy import symbols, Matrix
>>> from sympy.physics.mechanics import *
>>> q1 = dynamicsymbols('q1')                     # Angle of pendulum
>>> u1 = dynamicsymbols('u1')                     # Angular velocity
>>> q1d = dynamicsymbols('q1', 1)
>>> L, m, t, g = symbols('L, m, t, g')

>>> # Compose world frame
>>> N = ReferenceFrame('N')
>>> pN = Point('N*')
>>> pN.set_vel(N, 0)

>>> # A.x is along the pendulum
>>> A = N.orientnew('A', 'axis', [q1, N.z])
>>> A.set_ang_vel(N, u1*N.z)

>>> # Locate point P relative to the origin N*
>>> P = pN.locatenew('P', L*A.x)
>>> vel_P = P.v2pt_theory(pN, N, A)
>>> pP = Particle('pP', P, m)

>>> # Create Kinematic Differential Equations
>>> kde = Matrix([q1d - u1])

>>> # Input the force resultant at P
>>> R = m*g*N.x

>>> # Solve for eom with kanes method
>>> KM = KanesMethod(N, q_ind=[q1], u_ind=[u1], kd_eqs=kde)
>>> fr, frstar = KM.kanes_equations([pP], [(P, R)])
```

1. Using the Linearizer class directly:

A linearizer object can be created using the to_linearizer class method. This coerces the representation found in the KanesMethod object into the generalized form described above. As the independent and dependent coordinates and speeds are specified upon creation of the KanesMethod object, there is no need to specify them here.

```
>>> linearizer = KM.to_linearizer()
```

The linearized EOM can then be formed with the linearize method of the Linearizer object:

```
>>> M, A, B = linearizer.linearize()
>>> M
Matrix([
[1,       0],
```

(continues on next page)

(continued from previous page)

```
[0, -L**2*m]])
>>> A
Matrix([
[                 0, 1],
[L*g*m*cos(q1(t)), 0]])
>>> B
Matrix(0, 0, [])
```

Alternatively, the A and B form can be generated instead by specifying A_and_B=True:

```
>>> A, B = linearizer.linearize(A_and_B=True)
>>> A
Matrix([
[                0, 1],
[-g*cos(q1(t))/L, 0]])
>>> B
Matrix(0, 0, [])
```

An operating point can also be specified as a dictionary or an iterable of dictionaries. This will evaluate the linearized form at the specified point before returning the matrices:

```
>>> op_point = {q1: 0, u1: 0}
>>> A_op, B_op = linearizer.linearize(A_and_B=True, op_point=op_point)
>>> A_op
Matrix([
[     0, 1],
[-g/L, 0]])
```

Note that the same effect can be had by applying msubs to the matrices generated without the op_point kwarg:

```
>>> assert msubs(A, op_point) == A_op
```

Sometimes the returned matrices may not be in the most simplified form. Simplification can be performed after the fact, or the Linearizer object can be made to perform simplification internally by setting the simplify kwarg to True.

2. Using the linearize class method:

The linearize method of the KanesMethod class is provided as a nice wrapper that calls to_linearizer internally, performs the linearization, and returns the result. Note that all the kwargs available in the linearize method described above are also available here:

```
>>> A, B, inp_vec = KM.linearize(A_and_B=True, op_point=op_point, new_
↪method=True)
>>> A
Matrix([
[     0, 1],
[-g/L, 0]])
```

The additional output inp_vec is a vector containing all found dynamicsymbols not included in the generalized coordinate or speed vectors. These are assumed to be inputs to the system, forming the r vector described in the background above. In this example there are no inputs, so the vector is empty:

```
>>> inp_vec
Matrix(0, 0, [])
```

What's with the new_method kwarg?

Previous releases of SymPy contained a linearization method for KanesMethod objects. This method is deprecated, and will be removed from future releases. Until then, you must set new_method=True in all calls to KanesMethod.linearize. After the old method is removed, this kwarg will no longer be needed.

Linearizing Lagrange's Equations

Linearization of Lagrange's equations proceeds much the same as that of Kane's equations. As before, the process will be demonstrated with a simple pendulum system:

```
>>> # Redefine A and P in terms of q1d, not u1
>>> A = N.orientnew('A', 'axis', [q1, N.z])
>>> A.set_ang_vel(N, q1d*N.z)
>>> P = pN.locatenew('P', L*A.x)
>>> vel_P = P.v2pt_theory(pN, N, A)
>>> pP = Particle('pP', P, m)

>>> # Solve for eom with Lagrange's method
>>> Lag = Lagrangian(N, pP)
>>> LM = LagrangesMethod(Lag, [q1], forcelist=[(P, R)], frame=N)
>>> lag_eqs = LM.form_lagranges_equations()
```

1. Using the Linearizer class directly:

A Linearizer object can be formed from a LagrangesMethod object using the to_linearizer class method. The only difference between this process and that of the KanesMethod class is that the LagrangesMethod object doesn't already have its independent and dependent coordinates and speeds specified internally. These must be specified in the call to to_linearizer. In this example there are no dependent coordinates and speeds, but if there were they would be included in the q_dep and qd_dep kwargs:

```
>>> linearizer = LM.to_linearizer(q_ind=[q1], qd_ind=[q1d])
```

Once in this form, everything is the same as it was before with the KanesMethod example:

```
>>> A, B = linearizer.linearize(A_and_B=True, op_point=op_point)
>>> A
Matrix([
[     0, 1],
[-g/L, 0]])
```

2. Using the linearize class method:

Similar to KanesMethod, the LagrangesMethod class also provides a linearize method as a nice wrapper that calls to_linearizer internally, performs the linearization, and returns the result. As before, the only difference is that the independent and dependent coordinates and speeds must be specified in the call as well:

```
>>> A, B, inp_vec = LM.linearize(q_ind=[q1], qd_ind=[q1d], A_and_B=True, op_
↪point=op_point)
>>> A
Matrix([
[     0, 1],
[-g/L, 0]])
```

Potential Issues

While the Linearizer class *should* be able to linearize all systems, there are some potential issues that could occur. These are discussed below, along with some troubleshooting tips for solving them.

1. Symbolic linearization with A_and_B=True is slow

This could be due to a number of things, but the most likely one is that solving a large linear system symbolically is an expensive operation. Specifying an operating point will reduce the expression size and speed this up. If a purely symbolic solution is desired though (for application of many operating points at a later period, for example) a way to get around this is to evaluate with A_and_B=False, and then solve manually after applying the operating point:

```
>>> M, A, B = linearizer.linearize()
>>> M_op = msubs(M, op_point)
>>> A_op = msubs(A, op_point)
>>> perm_mat = linearizer.perm_mat
>>> A_lin = perm_mat.T * M_op.LUsolve(A_op)
>>> A_lin
Matrix([
[     0, 1],
[-g/L, 0]])
```

The fewer symbols in A and M before solving, the faster this solution will be. Thus, for large expressions, it may be to your benefit to delay conversion to the A and B form until most symbols are subbed in for their numeric values.

2. The linearized form has nan, zoo, or oo as matrix elements

There are two potential causes for this. The first (and the one you should check first) is that some choices of dependent coordinates will result in singularities at certain operating points. Coordinate partitioning in a systemic manner to avoid this is beyond the scope of this guide; see [Blajer1994] for more information.

The other potential cause for this is that the matrices may not have been in the most reduced form before the operating point was substituted in. A simple example of this behavior is:

```
>>> from sympy import sin, tan
>>> expr = sin(q1)/tan(q1)
>>> op_point = {q1: 0}
>>> expr.subs(op_point)
nan
```

Note that if this expression was simplified before substitution, the correct value results:

```
>>> expr.simplify().subs(op_point)
1
```

A good way of avoiding this hasn't been found yet. For expressions of reasonable size, using `msubs` with `smart=True` will apply an algorithm that tries to avoid these conditions. For large expressions though this is extremely time consuming.

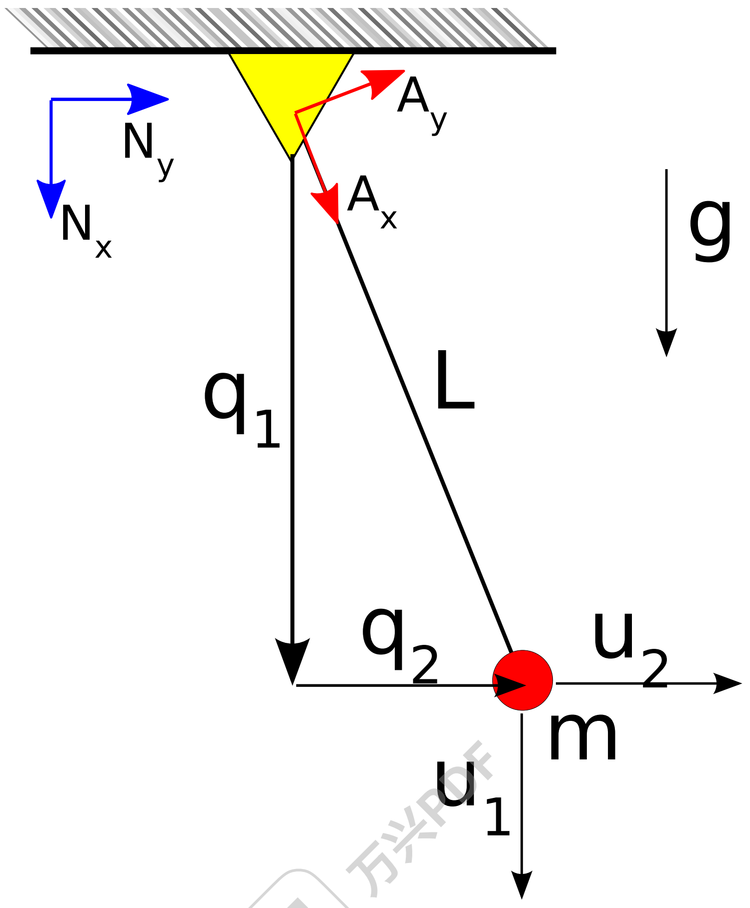
```
>>> msubs(expr, op_point, smart=True)
1
```

Further Examples

The pendulum example used above was simple, but didn't include any dependent coordinates or speeds. For a more thorough example, the same pendulum was linearized with dependent coordinates using both Kane's and Lagrange's methods:

Nonminimal Coordinates Pendulum

In this example we demonstrate the use of the functionality provided in *sympy.physics.mechanics* (page 1675) for deriving the equations of motion (EOM) for a pendulum with a nonminimal set of coordinates. As the pendulum is a one degree of freedom system, it can be described using one coordinate and one speed (the pendulum angle, and the angular velocity respectively). Choosing instead to describe the system using the x and y coordinates of the mass results in a need for constraints. The system is shown below:



The system will be modeled using both Kane's and Lagrange's methods, and the resulting EOM linearized. While this is a simple problem, it should illustrate the use of the linearization methods in the presence of constraints.

Kane's Method

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 and q_2 represent the mass x and y coordinates in the inertial N frame. Likewise, the generalized speeds u_1 and u_2 represent the velocities in these directions. We also create some `symbols` to represent the length and mass of the pendulum, as well as gravity and time.

```
>>> from sympy.physics.mechanics import *
>>> from sympy import symbols, atan, Matrix, solve
>>> # Create generalized coordinates and speeds for this non-minimal␣
↪realization
>>> # q1, q2 = N.x and N.y coordinates of pendulum
>>> # u1, u2 = N.x and N.y velocities of pendulum
>>> q1, q2 = dynamicsymbols('q1:3')
>>> q1d, q2d = dynamicsymbols('q1:3', level=1)
>>> u1, u2 = dynamicsymbols('u1:3')
>>> u1d, u2d = dynamicsymbols('u1:3', level=1)
>>> L, m, g, t = symbols('L, m, g, t')
```

Next, we create a world coordinate frame N, and its origin point N^*. The velocity of the origin is set to 0. A second coordinate frame A is oriented such that its x-axis is along the pendulum (as shown in the diagram above).

```
>>> # Compose world frame
>>> N = ReferenceFrame('N')
>>> pN = Point('N*')
>>> pN.set_vel(N, 0)

>>> # A.x is along the pendulum
>>> theta1 = atan(q2/q1)
>>> A = N.orientnew('A', 'axis', [theta1, N.z])
```

Locating the pendulum mass is then as easy as specifying its location with in terms of its x and y coordinates in the world frame. A `Particle` object is then created to represent the mass at this location.

```
>>> # Locate the pendulum mass
>>> P = pN.locatenew('P1', q1*N.x + q2*N.y)
>>> pP = Particle('pP', P, m)
```

The kinematic differential equations (KDEs) relate the derivatives of the generalized coordinates to the generalized speeds. In this case the speeds are the derivatives, so these are simple. A dictionary is also created to map $\dot{q}$ to u:

```
>>> # Calculate the kinematic differential equations
>>> kde = Matrix([q1d - u1,
...               q2d - u2])
>>> dq_dict = solve(kde, [q1d, q2d])
```

The velocity of the mass is then the time derivative of the position from the origin N^*:

```
>>> # Set velocity of point P
>>> P.set_vel(N, P.pos_from(pN).dt(N).subs(dq_dict))
```

As this system has more coordinates than degrees of freedom, constraints are needed. The configuration constraints relate the coordinates to each other. In this case the constraint is that the distance from the origin to the mass is always the length L (the pendulum doesn't get longer). Likewise, the velocity constraint is that the mass velocity in the `A.x` direction is always 0 (no radial velocity).

```
>>> f_c = Matrix([P.pos_from(pN).magnitude() - L])
>>> f_v = Matrix([P.vel(N).express(A).dot(A.x)])
>>> f_v.simplify()
```

The force on the system is just gravity, at point `P`.

```
>>> # Input the force resultant at P
>>> R = m*g*N.x
```

With the problem setup, the equations of motion can be generated using the `KanesMethod` class. As there are constraints, dependent and independent coordinates need to be provided to the class. In this case we'll use q_2 and u_2 as the independent coordinates and speeds:

```
>>> # Derive the equations of motion using the KanesMethod class.
>>> KM = KanesMethod(N, q_ind=[q2], u_ind=[u2], q_dependent=[q1],
...                  u_dependent=[u1], configuration_constraints=f_c,
...                  velocity_constraints=f_v, kd_eqs=kde)
>>> (fr, frstar) = KM.kanes_equations([pP],[(P, R)])
```

For linearization, operating points can be specified on the call, or be substituted in afterwards. In this case we'll provide them in the call, supplied in a list. The A_and_B=True kwarg indicates to solve invert the M matrix and solve for just the explicit linearized A and B matrices. The simplify=True kwarg indicates to simplify inside the linearize call, and return the presimplified matrices. The cost of doing this is small for simple systems, but for larger systems this can be a costly operation, and should be avoided.

```
>>> # Set the operating point to be straight down, and non-moving
>>> q_op = {q1: L, q2: 0}
>>> u_op = {u1: 0, u2: 0}
>>> ud_op = {u1d: 0, u2d: 0}
>>> # Perform the linearization
>>> A, B, inp_vec = KM.linearize(op_point=[q_op, u_op, ud_op], A_and_B=True,
...                              new_method=True, simplify=True)
>>> A
Matrix([
[   0, 1],
[-g/L, 0]])
>>> B
Matrix(0, 0, [])
```

The resulting A matrix has dimensions 2 x 2, while the number of total states is len(q) + len(u) = 2 + 2 = 4. This is because for constrained systems the resulting A_and_B form has a partitioned state vector only containing the independent coordinates and speeds. Written out mathematically, the system linearized about this point would be written as:

$$\begin{bmatrix} \dot{q}_2 \\ \dot{u}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ \frac{-g}{L} & 0 \end{bmatrix} \begin{bmatrix} q_2 \\ u_2 \end{bmatrix}$$

Lagrange's Method

The derivation using Lagrange's method is very similar to the approach using Kane's method described above. As before, we first create the dynamicsymbols needed to describe the system. In this case, the generalized coordinates q_1 and q_2 represent the mass x and y coordinates in the inertial N frame. This results in the time derivatives $\dot{q}_1$ and $\dot{q}_2$ representing the velocities in these directions. We also create some symbols to represent the length and mass of the pendulum, as well as gravity and time.

```
>>> from sympy.physics.mechanics import *
>>> from sympy import symbols, atan, Matrix
>>> q1, q2 = dynamicsymbols('q1:3')
>>> q1d, q2d = dynamicsymbols('q1:3', level=1)
>>> L, m, g, t = symbols('L, m, g, t')
```

Next, we create a world coordinate frame N, and its origin point N^*. The velocity of the origin is set to 0. A second coordinate frame A is oriented such that its x-axis is along the pendulum (as shown in the diagram above).

```
>>> # Compose World Frame
>>> N = ReferenceFrame('N')
>>> pN = Point('N*')
>>> pN.set_vel(N, 0)
>>> # A.x is along the pendulum
>>> theta1 = atan(q2/q1)
>>> A = N.orientnew('A', 'axis', [theta1, N.z])
```

Locating the pendulum mass is then as easy as specifying its location with in terms of its x and y coordinates in the world frame. A `Particle` object is then created to represent the mass at this location.

```
>>> # Create point P, the pendulum mass
>>> P = pN.locatenew('P1', q1*N.x + q2*N.y)
>>> P.set_vel(N, P.pos_from(pN).dt(N))
>>> pP = Particle('pP', P, m)
```

As this system has more coordinates than degrees of freedom, constraints are needed. In this case only a single holonomic constraints is needed: the distance from the origin to the mass is always the length L (the pendulum doesn't get longer).

```
>>> # Holonomic Constraint Equations
>>> f_c = Matrix([q1**2 + q2**2 - L**2])
```

The force on the system is just gravity, at point P.

```
>>> # Input the force resultant at P
>>> R = m*g*N.x
```

With the problem setup, the Lagrangian can be calculated, and the equations of motion formed. Note that the call to `LagrangesMethod` includes the Lagrangian, the generalized coordinates, the constraints (specified by `hol_coneqs` or `nonhol_coneqs`), the list of (body, force) pairs, and the inertial frame. In contrast to the `KanesMethod` initializer, independent and dependent coordinates are not partitioned inside the `LagrangesMethod` object. Such a partition is supplied later.

```
>>> # Calculate the lagrangian, and form the equations of motion
>>> Lag = Lagrangian(N, pP)
>>> LM = LagrangesMethod(Lag, [q1, q2], hol_coneqs=f_c, forcelist=[(P, R)],␣
↪frame=N)
>>> lag_eqs = LM.form_lagranges_equations()
```

Next, we compose the operating point dictionary, set in the hanging at rest position:

```
>>> # Compose operating point
>>> op_point = {q1: L, q2: 0, q1d: 0, q2d: 0, q1d.diff(t): 0, q2d.diff(t): 0}
```

As there are constraints in the formulation, there will be corresponding Lagrange Multipliers. These may appear inside the linearized form as well, and thus should also be included inside the operating point dictionary. Fortunately, the `LagrangesMethod` class provides an easy way of solving for the multipliers at a given operating point using the `solve_multipliers` method.

```
>>> # Solve for multiplier operating point
>>> lam_op = LM.solve_multipliers(op_point=op_point)
```

With this solution, linearization can be completed. Note that in contrast to the KanesMethod approach, the LagrangesMethod.linearize method also requires the partitioning of the generalized coordinates and their time derivatives into independent and dependent vectors. This is the same as what was passed into the KanesMethod constructor above:

```
>>> op_point.update(lam_op)
>>> # Perform the Linearization
>>> A, B, inp_vec = LM.linearize([q2], [q2d], [q1], [q1d],
...                             op_point=op_point, A_and_B=True)
>>> A
Matrix([
[     0, 1],
[-g/L, 0]])
>>> B
Matrix(0, 0, [])
```

The resulting A matrix has dimensions 2 x 2, while the number of total states is 2*len(q) = 4. This is because for constrained systems the resulting A_and_B form has a partitioned state vector only containing the independent coordinates and their derivatives. Written out mathematically, the system linearized about this point would be written as:

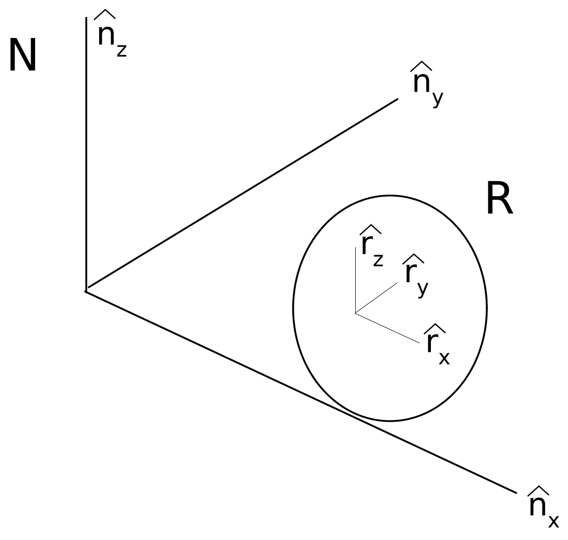
$$\begin{bmatrix}\dot{q}_2\\ \ddot{q}_2\end{bmatrix} = \begin{bmatrix}0 & 1\\ \frac{-g}{L} & 0\end{bmatrix}\begin{bmatrix}q_2\\ \dot{q}_2\end{bmatrix}$$

Examples for Physics/Mechanics

Here are some examples that illustrate how one typically uses this module. We have ordered the examples roughly according to increasing difficulty. If you have used this module to do something others might find useful or interesting, consider adding it here!

A rolling disc

The disc is assumed to be infinitely thin, in contact with the ground at only 1 point, and it is rolling without slip on the ground. See the image below.



We model the rolling disc in three different ways, to show more of the functionality of this module.

A rolling disc, with Kane's method

Here the definition of the rolling disc's kinematics is formed from the contact point up, removing the need to introduce generalized speeds. Only 3 configuration and three speed variables are need to describe this system, along with the disc's mass and radius, and the local gravity (note that mass will drop out).

```
>>> from sympy import symbols, sin, cos, tan
>>> from sympy.physics.mechanics import *
>>> q1, q2, q3, u1, u2, u3  = dynamicsymbols('q1 q2 q3 u1 u2 u3')
>>> q1d, q2d, q3d, u1d, u2d, u3d = dynamicsymbols('q1 q2 q3 u1 u2 u3', 1)
>>> r, m, g = symbols('r m g')
>>> mechanics_printing(pretty_print=False)
```

The kinematics are formed by a series of simple rotations. Each simple rotation creates a new frame, and the next rotation is defined by the new frame's basis vectors. This example uses a 3-1-2 series of rotations, or Z, X, Y series of rotations. Angular velocity for this is defined using the second frame's basis (the lean frame); it is for this reason that we defined intermediate frames, rather than using a body-three orientation.

```
>>> N = ReferenceFrame('N')
>>> Y = N.orientnew('Y', 'Axis', [q1, N.z])
>>> L = Y.orientnew('L', 'Axis', [q2, Y.x])
>>> R = L.orientnew('R', 'Axis', [q3, L.y])
>>> w_R_N_qd = R.ang_vel_in(N)
>>> R.set_ang_vel(N, u1 * L.x + u2 * L.y + u3 * L.z)
```

This is the translational kinematics. We create a point with no velocity in N; this is the contact point between the disc and ground. Next we form the position vector from the contact point to the disc's center of mass. Finally we form the velocity and acceleration of the disc.

```
>>> C = Point('C')
>>> C.set_vel(N, 0)
>>> Dmc = C.locatenew('Dmc', r * L.z)
>>> Dmc.v2pt_theory(C, N, R)
r*u2*L.x - r*u1*L.y
```

This is a simple way to form the inertia dyadic. The inertia of the disc does not change within the lean frame as the disc rolls; this will make for simpler equations in the end.

```
>>> I = inertia(L, m / 4 * r**2, m / 2 * r**2, m / 4 * r**2)
>>> mprint(I)
m*r**2/4*(L.x|L.x) + m*r**2/2*(L.y|L.y) + m*r**2/4*(L.z|L.z)
```

Kinematic differential equations; how the generalized coordinate time derivatives relate to generalized speeds.

```
>>> kd = [dot(R.ang_vel_in(N) - w_R_N_qd, uv) for uv in L]
```

Creation of the force list; it is the gravitational force at the center of mass of the disc. Then we create the disc by assigning a Point to the center of mass attribute, a ReferenceFrame to the frame attribute, and mass and inertia. Then we form the body list.

```
>>> ForceList = [(Dmc, - m * g * Y.z)]
>>> BodyD = RigidBody('BodyD', Dmc, R, m, (I, Dmc))
>>> BodyList = [BodyD]
```

Finally we form the equations of motion, using the same steps we did before. Specify inertial frame, supply generalized coordinates and speeds, supply kinematic differential equation dictionary, compute Fr from the force list and Fr* from the body list, compute the mass matrix and forcing terms, then solve for the u dots (time derivatives of the generalized speeds).

```
>>> KM = KanesMethod(N, q_ind=[q1, q2, q3], u_ind=[u1, u2, u3], kd_eqs=kd)
>>> (fr, frstar) = KM.kanes_equations(BodyList, ForceList)
>>> MM = KM.mass_matrix
>>> forcing = KM.forcing
>>> rhs = MM.inv() * forcing
>>> kdd = KM.kindiffdict()
>>> rhs = rhs.subs(kdd)
>>> rhs.simplify()
>>> mprint(rhs)
Matrix([
[(4*g*sin(q2) + 6*r*u2*u3 - r*u3**2*tan(q2))/(5*r)],
[                                       -2*u1*u3/3],
[                          (-2*u2 + u3*tan(q2))*u1]])
```

A rolling disc, with Kane's method and constraint forces

We will now revisit the rolling disc example, except this time we are bringing the non-contributing (constraint) forces into evidence. See [Kane1985] for a more thorough explanation of this. Here, we will turn on the automatic simplifcation done when doing vector operations. It makes the outputs nicer for small problems, but can cause larger vector operations to hang.

```
>>> from sympy import symbols, sin, cos, tan
>>> from sympy.physics.mechanics import *
>>> mechanics_printing(pretty_print=False)
>>> q1, q2, q3, u1, u2, u3  = dynamicsymbols('q1 q2 q3 u1 u2 u3')
>>> q1d, q2d, q3d, u1d, u2d, u3d = dynamicsymbols('q1 q2 q3 u1 u2 u3', 1)
>>> r, m, g = symbols('r m g')
```

These two lines introduce the extra quantities needed to find the constraint forces.

```
>>> u4, u5, u6, f1, f2, f3 = dynamicsymbols('u4 u5 u6 f1 f2 f3')
```

Most of the main code is the same as before.

```
>>> N = ReferenceFrame('N')
>>> Y = N.orientnew('Y', 'Axis', [q1, N.z])
>>> L = Y.orientnew('L', 'Axis', [q2, Y.x])
>>> R = L.orientnew('R', 'Axis', [q3, L.y])
>>> w_R_N_qd = R.ang_vel_in(N)
>>> R.set_ang_vel(N, u1 * L.x + u2 * L.y + u3 * L.z)
```

The definition of rolling without slip necessitates that the velocity of the contact point is zero; as part of bringing the constraint forces into evidence, we have to introduce speeds at this point, which will by definition always be zero. They are normal to the ground, along the path which the disc is rolling, and along the ground in a perpendicular direction.

```
>>> C = Point('C')
>>> C.set_vel(N, u4 * L.x + u5 * cross(Y.z, L.x) + u6 * Y.z)
>>> Dmc = C.locatenew('Dmc', r * L.z)
>>> vel = Dmc.v2pt_theory(C, N, R)
>>> I = inertia(L, m / 4 * r**2, m / 2 * r**2, m / 4 * r**2)
>>> kd = [dot(R.ang_vel_in(N) - w_R_N_qd, uv) for uv in L]
```

Just as we previously introduced three speeds as part of this process, we also introduce three forces; they are in the same direction as the speeds, and represent the constraint forces in those directions.

```
>>> ForceList = [(Dmc, - m * g * Y.z), (C, f1 * L.x + f2 * cross(Y.z, L.x) +␣
↪f3 * Y.z)]
>>> BodyD = RigidBody('BodyD', Dmc, R, m, (I, Dmc))
>>> BodyList = [BodyD]

>>> KM = KanesMethod(N, q_ind=[q1, q2, q3], u_ind=[u1, u2, u3], kd_eqs=kd,
...             u_auxiliary=[u4, u5, u6])
>>> (fr, frstar) = KM.kanes_equations(BodyList, ForceList)
>>> MM = KM.mass_matrix
>>> forcing = KM.forcing
```

(continues on next page)

(continued from previous page)

```
>>> rhs = MM.inv() * forcing
>>> kdd = KM.kindiffdict()
>>> rhs = rhs.subs(kdd)
>>> rhs.simplify()
>>> mprint(rhs)
Matrix([
[(4*g*sin(q2) + 6*r*u2*u3 - r*u3**2*tan(q2))/(5*r)],
[                                       -2*u1*u3/3],
[                          (-2*u2 + u3*tan(q2))*u1]])
>>> from sympy import trigsimp, signsimp, collect, factor_terms
>>> def simplify_auxiliary_eqs(w):
...     return signsimp(trigsimp(collect(collect(factor_terms(w), f2), m*r)))
>>> mprint(KM.auxiliary_eqs.applyfunc(simplify_auxiliary_eqs))
Matrix([
[                                      -m*r*(u1*u3 + u2') + f1],
[-m*r*u1**2*sin(q2) - m*r*u2*u3/cos(q2) + m*r*cos(q2)*u1' + f2],
[                -g*m + m*r*(u1**2*cos(q2) + sin(q2)*u1') + f3]])
```

A rolling disc using Lagrange's Method

Here the rolling disc is formed from the contact point up, removing the need to introduce generalized speeds. Only 3 configuration and 3 speed variables are needed to describe this system, along with the disc's mass and radius, and the local gravity.

```
>>> from sympy import symbols, cos, sin
>>> from sympy.physics.mechanics import *
>>> mechanics_printing(pretty_print=False)
>>> q1, q2, q3 = dynamicsymbols('q1 q2 q3')
>>> q1d, q2d, q3d = dynamicsymbols('q1 q2 q3', 1)
>>> r, m, g = symbols('r m g')
```

The kinematics are formed by a series of simple rotations. Each simple rotation creates a new frame, and the next rotation is defined by the new frame's basis vectors. This example uses a 3-1-2 series of rotations, or Z, X, Y series of rotations. Angular velocity for this is defined using the second frame's basis (the lean frame).

```
>>> N = ReferenceFrame('N')
>>> Y = N.orientnew('Y', 'Axis', [q1, N.z])
>>> L = Y.orientnew('L', 'Axis', [q2, Y.x])
>>> R = L.orientnew('R', 'Axis', [q3, L.y])
```

This is the translational kinematics. We create a point with no velocity in N; this is the contact point between the disc and ground. Next we form the position vector from the contact point to the disc's center of mass. Finally we form the velocity and acceleration of the disc.

```
>>> C = Point('C')
>>> C.set_vel(N, 0)
>>> Dmc = C.locatenew('Dmc', r * L.z)
>>> Dmc.v2pt_theory(C, N, R)
r*(sin(q2)*q1' + q3')*L.x - r*q2'*L.y
```

Forming the inertia dyadic.

```
>>> I = inertia(L, m / 4 * r**2, m / 2 * r**2, m / 4 * r**2)
>>> mprint(I)
m*r**2/4*(L.x|L.x) + m*r**2/2*(L.y|L.y) + m*r**2/4*(L.z|L.z)
>>> BodyD = RigidBody('BodyD', Dmc, R, m, (I, Dmc))
```

We then set the potential energy and determine the Lagrangian of the rolling disc.

```
>>> BodyD.potential_energy = - m * g * r * cos(q2)
>>> Lag = Lagrangian(N, BodyD)
```

Then the equations of motion are generated by initializing the `LagrangesMethod` object. Finally we solve for the generalized accelerations(q double dots) with the `rhs` method.

```
>>> q = [q1, q2, q3]
>>> l = LagrangesMethod(Lag, q)
>>> le = l.form_lagranges_equations()
>>> le.simplify(); le
Matrix([
[m*r**2*(6*sin(q2)*q3'' + 5*sin(2*q2)*q1'*q2' + 6*cos(q2)*q2'*q3' -␣
↪5*cos(2*q2)*q1''/2 + 7*q1''/2)/4],
[                      m*r*(4*g*sin(q2) - 5*r*sin(2*q2)*q1'**2/2 -␣
↪6*r*cos(q2)*q1'*q3' + 5*r*q2'')/4],
[                                 3*m*r**2*(sin(q2)*q1'' +␣
↪cos(q2)*q1'*q2' + q3'')/2]])
>>> lrhs = l.rhs(); lrhs.simplify(); lrhs
Matrix([
[                                                          q1'],
[                                                          q2'],
[                                                          q3'],
[                       -2*(2*tan(q2)*q1' + 3*q3'/cos(q2))*q2'],
[-4*g*sin(q2)/(5*r) + sin(2*q2)*q1'**2/2 + 6*cos(q2)*q1'*q3'/5],
[         (-5*cos(q2)*q1' + 6*tan(q2)*q3' + 4*q1'/cos(q2))*q2']])
```

A bicycle

The bicycle is an interesting system in that it has multiple rigid bodies, non-holonomic constraints, and a holonomic constraint. The linearized equations of motion are presented in [Meijaard2007]. This example will go through construction of the equations of motion in *sympy.physics.mechanics* (page 1675).

```
>>> from sympy import *
>>> from sympy.physics.mechanics import *
>>> print('Calculation of Linearized Bicycle \"A\" Matrix, '
...       'with States: Roll, Steer, Roll Rate, Steer Rate')
Calculation of Linearized Bicycle "A" Matrix, with States: Roll, Steer, Roll␣
↪Rate, Steer Rate
```

Note that this code has been crudely ported from Autolev, which is the reason for some of the unusual naming conventions. It was purposefully as similar as possible in order to aid initial porting & debugging. We set Vector.simp to False (in case it has been set True elsewhere), since it slows down the computations:

```
>>> Vector.simp = False
>>> mechanics_printing(pretty_print=False)
```

Declaration of Coordinates & Speeds: A simple definition for qdots, qd = u,is used in this code. Speeds are: yaw frame ang. rate, roll frame ang. rate, rear wheel frame ang. rate (spinning motion), frame ang. rate (pitching motion), steering frame ang. rate, and front wheel ang. rate (spinning motion). Wheel positions are ignorable coordinates, so they are not introduced.

```
>>> q1, q2, q3, q4, q5 = dynamicsymbols('q1 q2 q3 q4 q5')
>>> q1d, q2d, q4d, q5d = dynamicsymbols('q1 q2 q4 q5', 1)
>>> u1, u2, u3, u4, u5, u6 = dynamicsymbols('u1 u2 u3 u4 u5 u6')
>>> u1d, u2d, u3d, u4d, u5d, u6d = dynamicsymbols('u1 u2 u3 u4 u5 u6', 1)
```

Declaration of System's Parameters: The below symbols should be fairly self-explanatory.

```
>>> WFrad, WRrad, htangle, forkoffset = symbols('WFrad WRrad htangle␣
↪forkoffset')
>>> forklength, framelength, forkcg1 = symbols('forklength framelength forkcg1
↪')
>>> forkcg3, framecg1, framecg3, Iwr11 = symbols('forkcg3 framecg1 framecg3␣
↪Iwr11')
>>> Iwr22, Iwf11, Iwf22, Iframe11 = symbols('Iwr22 Iwf11 Iwf22 Iframe11')
>>> Iframe22, Iframe33, Iframe31, Ifork11 = \
...     symbols('Iframe22 Iframe33 Iframe31 Ifork11')
>>> Ifork22, Ifork33, Ifork31, g = symbols('Ifork22 Ifork33 Ifork31 g')
>>> mframe, mfork, mwf, mwr = symbols('mframe mfork mwf mwr')
```

Set up reference frames for the system: N - inertial Y - yaw R - roll WR - rear wheel, rotation angle is ignorable coordinate so not oriented Frame - bicycle frame TempFrame - statically rotated frame for easier reference inertia definition Fork - bicycle fork TempFork - statically rotated frame for easier reference inertia definition WF - front wheel, again posses an ignorable coordinate

```
>>> N = ReferenceFrame('N')
>>> Y = N.orientnew('Y', 'Axis', [q1, N.z])
>>> R = Y.orientnew('R', 'Axis', [q2, Y.x])
>>> Frame = R.orientnew('Frame', 'Axis', [q4 + htangle, R.y])
>>> WR = ReferenceFrame('WR')
>>> TempFrame = Frame.orientnew('TempFrame', 'Axis', [-htangle, Frame.y])
>>> Fork = Frame.orientnew('Fork', 'Axis', [q5, Frame.x])
>>> TempFork = Fork.orientnew('TempFork', 'Axis', [-htangle, Fork.y])
>>> WF = ReferenceFrame('WF')
```

Kinematics of the Bicycle: First block of code is forming the positions of the relevant points rear wheel contact -> rear wheel's center of mass -> frame's center of mass + frame/fork connection -> fork's center of mass + front wheel's center of mass -> front wheel contact point.

```
>>> WR_cont = Point('WR_cont')
>>> WR_mc = WR_cont.locatenew('WR_mc', WRrad * R.z)
>>> Steer = WR_mc.locatenew('Steer', framelength * Frame.z)
>>> Frame_mc = WR_mc.locatenew('Frame_mc', -framecg1 * Frame.x + framecg3 *␣
```

(continues on next page)

(continued from previous page)

```
↪Frame.z)
>>> Fork_mc = Steer.locatenew('Fork_mc', -forkcg1 * Fork.x + forkcg3 * Fork.z)
>>> WF_mc = Steer.locatenew('WF_mc', forklength * Fork.x + forkoffset * Fork.
↪z)
>>> WF_cont = WF_mc.locatenew('WF_cont', WFrad*(dot(Fork.y, Y.z)*Fork.y - \
...                                            Y.z).normalize())
```

Set the angular velocity of each frame: Angular accelerations end up being calculated automatically by differentiating the angular velocities when first needed. :: u1 is yaw rate u2 is roll rate u3 is rear wheel rate u4 is frame pitch rate u5 is fork steer rate u6 is front wheel rate

```
>>> Y.set_ang_vel(N, u1 * Y.z)
>>> R.set_ang_vel(Y, u2 * R.x)
>>> WR.set_ang_vel(Frame, u3 * Frame.y)
>>> Frame.set_ang_vel(R, u4 * Frame.y)
>>> Fork.set_ang_vel(Frame, u5 * Fork.x)
>>> WF.set_ang_vel(Fork, u6 * Fork.y)
```

Form the velocities of the points, using the 2-point theorem. Accelerations again are calculated automatically when first needed.

```
>>> WR_cont.set_vel(N, 0)
>>> WR_mc.v2pt_theory(WR_cont, N, WR)
WRrad*(u1*sin(q2) + u3 + u4)*R.x - WRrad*u2*R.y
>>> Steer.v2pt_theory(WR_mc, N, Frame)
WRrad*(u1*sin(q2) + u3 + u4)*R.x - WRrad*u2*R.y + framelength*(u1*sin(q2) +␣
↪u4)*Frame.x - framelength*(-u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle +␣
↪q4))*Frame.y
>>> Frame_mc.v2pt_theory(WR_mc, N, Frame)
WRrad*(u1*sin(q2) + u3 + u4)*R.x - WRrad*u2*R.y + framecg3*(u1*sin(q2) +␣
↪u4)*Frame.x + (-framecg1*(u1*cos(htangle + q4)*cos(q2) + u2*sin(htangle +␣
↪q4)) - framecg3*(-u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle +␣
↪q4)))*Frame.y + framecg1*(u1*sin(q2) + u4)*Frame.z
>>> Fork_mc.v2pt_theory(Steer, N, Fork)
WRrad*(u1*sin(q2) + u3 + u4)*R.x - WRrad*u2*R.y + framelength*(u1*sin(q2) +␣
↪u4)*Frame.x - framelength*(-u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle +␣
↪q4))*Frame.y + forkcg3*((sin(q2)*cos(q5) + sin(q5)*cos(htangle +␣
↪q4)*cos(q2))*u1 + u2*sin(htangle + q4)*sin(q5) + u4*cos(q5))*Fork.x + (-
↪forkcg1*((-sin(q2)*sin(q5) + cos(htangle + q4)*cos(q2)*cos(q5))*u1 +␣
↪u2*sin(htangle + q4)*cos(q5) - u4*sin(q5)) - forkcg3*(-u1*sin(htangle +␣
↪q4)*cos(q2) + u2*cos(htangle + q4) + u5))*Fork.y +␣
↪forkcg1*((sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2))*u1 +␣
↪u2*sin(htangle + q4)*sin(q5) + u4*cos(q5))*Fork.z
>>> WF_mc.v2pt_theory(Steer, N, Fork)
WRrad*(u1*sin(q2) + u3 + u4)*R.x - WRrad*u2*R.y + framelength*(u1*sin(q2) +␣
↪u4)*Frame.x - framelength*(-u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle +␣
↪q4))*Frame.y + forkoffset*((sin(q2)*cos(q5) + sin(q5)*cos(htangle +␣
↪q4)*cos(q2))*u1 + u2*sin(htangle + q4)*sin(q5) + u4*cos(q5))*Fork.x +␣
↪(forklength*((-sin(q2)*sin(q5) + cos(htangle + q4)*cos(q2)*cos(q5))*u1 +␣
↪u2*sin(htangle + q4)*cos(q5) - u4*sin(q5)) - forkoffset*(-u1*sin(htangle +␣
↪q4)*cos(q2) + u2*cos(htangle + q4) + u5))*Fork.y -␣
```

(continues on next page)

(continued from previous page)

```
↪forklength*((sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2))*u1 +␣
↪u2*sin(htangle + q4)*sin(q5) + u4*cos(q5))*Fork.z
>>> WF_cont.v2pt_theory(WF_mc, N, WF)
- WFrad*((-sin(q2)*sin(q5)*cos(htangle + q4) + cos(q2)*cos(q5))*u6 +␣
↪u4*cos(q2) + u5*sin(htangle + q4)*sin(q2))/sqrt((-sin(q2)*cos(q5) -␣
↪sin(q5)*cos(htangle + q4)*cos(q2))*(sin(q2)*cos(q5) + sin(q5)*cos(htangle +␣
↪q4)*cos(q2)) + 1)*Y.x + WFrad*(u2 + u5*cos(htangle + q4) + u6*sin(htangle +␣
↪q4)*sin(q5))/sqrt((-sin(q2)*cos(q5) - sin(q5)*cos(htangle +␣
↪q4)*cos(q2))*(sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2)) + 1)*Y.y␣
↪+ WRrad*(u1*sin(q2) + u3 + u4)*R.x - WRrad*u2*R.y + framelength*(u1*sin(q2)␣
↪+ u4)*Frame.x - framelength*(-u1*sin(htangle + q4)*cos(q2) + u2*cos(htangle␣
↪+ q4))*Frame.y + (-WFrad*(sin(q2)*cos(q5) + sin(q5)*cos(htangle +␣
↪q4)*cos(q2))*((-sin(q2)*sin(q5) + cos(htangle + q4)*cos(q2)*cos(q5))*u1 +␣
↪u2*sin(htangle + q4)*cos(q5) - u4*sin(q5))/sqrt((-sin(q2)*cos(q5) -␣
↪sin(q5)*cos(htangle + q4)*cos(q2))*(sin(q2)*cos(q5) + sin(q5)*cos(htangle +␣
↪q4)*cos(q2)) + 1) + forkoffset*((sin(q2)*cos(q5) + sin(q5)*cos(htangle +␣
↪q4)*cos(q2))*u1 + u2*sin(htangle + q4)*sin(q5) + u4*cos(q5)))*Fork.x +␣
↪(forklength*((-sin(q2)*sin(q5) + cos(htangle + q4)*cos(q2)*cos(q5))*u1 +␣
↪u2*sin(htangle + q4)*cos(q5) - u4*sin(q5)) - forkoffset*(-u1*sin(htangle +␣
↪q4)*cos(q2) + u2*cos(htangle + q4) + u5))*Fork.y + (WFrad*(sin(q2)*cos(q5)␣
↪+ sin(q5)*cos(htangle + q4)*cos(q2))*(-u1*sin(htangle + q4)*cos(q2) +␣
↪u2*cos(htangle + q4) + u5)/sqrt((-sin(q2)*cos(q5) - sin(q5)*cos(htangle +␣
↪q4)*cos(q2))*(sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2)) + 1) -␣
↪forklength*((sin(q2)*cos(q5) + sin(q5)*cos(htangle + q4)*cos(q2))*u1 +␣
↪u2*sin(htangle + q4)*sin(q5) + u4*cos(q5)))*Fork.z
```

Sets the inertias of each body. Uses the inertia frame to construct the inertia dyadics. Wheel inertias are only defined by principal moments of inertia, and are in fact constant in the frame and fork reference frames; it is for this reason that the orientations of the wheels does not need to be defined. The frame and fork inertias are defined in the 'Temp' frames which are fixed to the appropriate body frames; this is to allow easier input of the reference values of the benchmark paper. Note that due to slightly different orientations, the products of inertia need to have their signs flipped; this is done later when entering the numerical value.

```
>>> Frame_I = (inertia(TempFrame, Iframe11, Iframe22, Iframe33, 0, 0,
...                                                    Iframe31), Frame_mc)
>>> Fork_I = (inertia(TempFork, Ifork11, Ifork22, Ifork33, 0, 0, Ifork31),␣
↪Fork_mc)
>>> WR_I = (inertia(Frame, Iwr11, Iwr22, Iwr11), WR_mc)
>>> WF_I = (inertia(Fork, Iwf11, Iwf22, Iwf11), WF_mc)
```

Declaration of the RigidBody containers.

```
>>> BodyFrame = RigidBody('BodyFrame', Frame_mc, Frame, mframe, Frame_I)
>>> BodyFork = RigidBody('BodyFork', Fork_mc, Fork, mfork, Fork_I)
>>> BodyWR = RigidBody('BodyWR', WR_mc, WR, mwr, WR_I)
>>> BodyWF = RigidBody('BodyWF', WF_mc, WF, mwf, WF_I)

>>> print('Before Forming the List of Nonholonomic Constraints.')
Before Forming the List of Nonholonomic Constraints.
```

The kinematic differential equations; they are defined quite simply. Each entry in this list is equal to zero.

```
>>> kd = [q1d - u1, q2d - u2, q4d - u4, q5d - u5]
```

The nonholonomic constraints are the velocity of the front wheel contact point dotted into the X, Y, and Z directions; the yaw frame is used as it is "closer" to the front wheel (1 less DCM connecting them). These constraints force the velocity of the front wheel contact point to be 0 in the inertial frame; the X and Y direction constraints enforce a "no-slip" condition, and the Z direction constraint forces the front wheel contact point to not move away from the ground frame, essentially replicating the holonomic constraint which does not allow the frame pitch to change in an invalid fashion.

```
>>> conlist_speed = [dot(WF_cont.vel(N), Y.x),
...                  dot(WF_cont.vel(N), Y.y),
...                  dot(WF_cont.vel(N), Y.z)]
```

The holonomic constraint is that the position from the rear wheel contact point to the front wheel contact point when dotted into the normal-to-ground plane direction must be zero; effectively that the front and rear wheel contact points are always touching the ground plane. This is actually not part of the dynamic equations, but instead is necessary for the linearization process.

```
>>> conlist_coord = [dot(WF_cont.pos_from(WR_cont), Y.z)]
```

The force list; each body has the appropriate gravitational force applied at its center of mass.

```
>>> FL = [(Frame_mc, -mframe * g * Y.z), (Fork_mc, -mfork * g * Y.z),
...       (WF_mc, -mwf * g * Y.z), (WR_mc, -mwr * g * Y.z)]
>>> BL = [BodyFrame, BodyFork, BodyWR, BodyWF]
```

The N frame is the inertial frame, coordinates are supplied in the order of independent, dependent coordinates. The kinematic differential equations are also entered here. Here the independent speeds are specified, followed by the dependent speeds, along with the nonholonomic constraints. The dependent coordinate is also provided, with the holonomic constraint. Again, this is only comes into play in the linearization process, but is necessary for the linearization to correctly work.

```
>>> KM = KanesMethod(N, q_ind=[q1, q2, q5],
...             q_dependent=[q4], configuration_constraints=conlist_coord,
...             u_ind=[u2, u3, u5],
...             u_dependent=[u1, u4, u6], velocity_constraints=conlist_speed,
...             kd_eqs=kd)
>>> print('Before Forming Generalized Active and Inertia Forces, Fr and Fr*')
Before Forming Generalized Active and Inertia Forces, Fr and Fr*
>>> (fr, frstar) = KM.kanes_equations(BL, FL)
>>> print('Base Equations of Motion Computed')
Base Equations of Motion Computed
```

This is the start of entering in the numerical values from the benchmark paper to validate the eigenvalues of the linearized equations from this model to the reference eigenvalues. Look at the aforementioned paper for more information. Some of these are intermediate values, used to transform values from the paper into the coordinate systems used in this model.

```
>>> PaperRadRear  =  0.3
>>> PaperRadFront =  0.35
>>> HTA           =  evalf.N(pi/2-pi/10)
```

(continues on next page)

(continued from previous page)

```
>>> TrailPaper    =  0.08
>>> rake          =  evalf.N(-(TrailPaper*sin(HTA)-(PaperRadFront*cos(HTA))))
>>> PaperWb       =  1.02
>>> PaperFrameCgX =  0.3
>>> PaperFrameCgZ =  0.9
>>> PaperForkCgX  =  0.9
>>> PaperForkCgZ  =  0.7
>>> FrameLength   =  evalf.N(PaperWb*sin(HTA) - (rake - \
...                        (PaperRadFront - PaperRadRear)*cos(HTA)))
>>> FrameCGNorm   =  evalf.N((PaperFrameCgZ - PaperRadRear - \
...                         (PaperFrameCgX/sin(HTA))*cos(HTA))*sin(HTA))
>>> FrameCGPar    =  evalf.N((PaperFrameCgX / sin(HTA) + \
...                         (PaperFrameCgZ - PaperRadRear - \
...                          PaperFrameCgX / sin(HTA) * cos(HTA)) *␣
↪cos(HTA)))
>>> tempa         =  evalf.N((PaperForkCgZ - PaperRadFront))
>>> tempb         =  evalf.N((PaperWb-PaperForkCgX))
>>> tempc         =  evalf.N(sqrt(tempa**2 + tempb**2))
>>> PaperForkL    =  evalf.N((PaperWb*cos(HTA) - \
...                         (PaperRadFront - PaperRadRear)*sin(HTA)))
>>> ForkCGNorm    =  evalf.N(rake + (tempc * sin(pi/2 - \
...                         HTA - acos(tempa/tempc))))
>>> ForkCGPar     =  evalf.N(tempc * cos((pi/2 - HTA) - \
...                         acos(tempa/tempc)) - PaperForkL)
```

Here is the final assembly of the numerical values. The symbol 'v' is the forward speed of the bicycle (a concept which only makes sense in the upright, static equilibrium case?). These are in a dictionary which will later be substituted in. Again the sign on the *product* of inertia values is flipped here, due to different orientations of coordinate systems.

```
>>> v = Symbol('v')
>>> val_dict = {
...         WFrad: PaperRadFront,
...         WRrad: PaperRadRear,
...         htangle: HTA,
...         forkoffset: rake,
...         forklength: PaperForkL,
...         framelength: FrameLength,
...         forkcg1: ForkCGPar,
...         forkcg3: ForkCGNorm,
...         framecg1: FrameCGNorm,
...         framecg3: FrameCGPar,
...         Iwr11: 0.0603,
...         Iwr22: 0.12,
...         Iwf11: 0.1405,
...         Iwf22: 0.28,
...         Ifork11: 0.05892,
...         Ifork22: 0.06,
...         Ifork33: 0.00708,
...         Ifork31: 0.00756,
...         Iframe11: 9.2,
...         Iframe22: 11,
```

(continues on next page)

(continued from previous page)

```
...         Iframe33: 2.8,
...         Iframe31: -2.4,
...         mfork: 4,
...         mframe: 85,
...         mwf: 3,
...         mwr: 2,
...         g: 9.81,
...         q1: 0,
...         q2: 0,
...         q4: 0,
...         q5: 0,
...         u1: 0,
...         u2: 0,
...         u3: v/PaperRadRear,
...         u4: 0,
...         u5: 0,
...         u6: v/PaperRadFront}
>>> kdd = KM.kindiffdict()
>>> print('Before Linearization of the \"Forcing\" Term')
Before Linearization of the "Forcing" Term
```

Linearizes the forcing vector; the equations are set up as MM udot = forcing, where MM is the mass matrix, udot is the vector representing the time derivatives of the generalized speeds, and forcing is a vector which contains both external forcing terms and internal forcing terms, such as centripetal or Coriolis forces. This actually returns a matrix with as many rows as *total* coordinates and speeds, but only as many columns as independent coordinates and speeds. (Note that below this is commented out, as it takes a few minutes to run, which is not good when performing the doctests)

```
>>> # forcing_lin = KM.linearize()[0].subs(sub_dict)
```

As mentioned above, the size of the linearized forcing terms is expanded to include both q's and u's, so the mass matrix must have this done as well. This will likely be changed to be part of the linearized process, for future reference.

```
>>> MM_full = (KM._k_kqdot).row_join(zeros(4, 6)).col_join(
...           (zeros(6, 4)).row_join(KM.mass_matrix))
>>> print('Before Substitution of Numerical Values')
Before Substitution of Numerical Values
```

I think this is pretty self explanatory. It takes a really long time though. I've experimented with using evalf with substitution, this failed due to maximum recursion depth being exceeded; I also tried lambdifying this, and it is also not successful. (again commented out due to speed)

```
>>> # MM_full = MM_full.subs(val_dict)
>>> # forcing_lin = forcing_lin.subs(val_dict)
>>> # print('Before .evalf() call')

>>> # MM_full = MM_full.evalf()
>>> # forcing_lin = forcing_lin.evalf()
```

Finally, we construct an "A" matrix for the form xdot = A x (x being the state vector, although in this case, the sizes are a little off). The following line extracts only the minimum entries required for eigenvalue analysis, which correspond to rows and columns for lean, steer, lean rate, and steer rate. (this is all commented out due to being dependent on the above code, which is also commented out):

```
>>> # Amat = MM_full.inv() * forcing_lin
>>> # A = Amat.extract([1,2,4,6],[1,2,3,5])
>>> # print(A)
>>> # print('v = 1')
>>> # print(A.subs(v, 1).eigenvals())
>>> # print('v = 2')
>>> # print(A.subs(v, 2).eigenvals())
>>> # print('v = 3')
>>> # print(A.subs(v, 3).eigenvals())
>>> # print('v = 4')
>>> # print(A.subs(v, 4).eigenvals())
>>> # print('v = 5')
>>> # print(A.subs(v, 5).eigenvals())
```

Upon running the above code yourself, enabling the commented out lines, compare the computed eigenvalues to those is the referenced paper. This concludes the bicycle example.

Multi Degree of Freedom Holonomic System

In this example we demonstrate the use of the functionality provided in *sympy.physics.mechanics* (page 1675) for deriving the equations of motion (EOM) of a holonomic system that includes both particles and rigid bodies with contributing forces and torques, some of which are specified forces and torques. The system is shown below: