If the orientation creates a kinematic loop.

Examples

Setup variables for the examples:

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame
>>> q1, q2, q3 = symbols('q1, q2, q3')
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B1 = ReferenceFrame('B1')
>>> B2 = ReferenceFrame('B2')
>>> B3 = ReferenceFrame('B3')
```

```
>>> B.orient_space_fixed(N, (q1, q2, q3), '312')
>>> B.dcm(N)
Matrix([
[ sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3), sin(q1)*cos(q2),␣
↪sin(q1)*sin(q2)*cos(q3) - sin(q3)*cos(q1)],
[-sin(q1)*cos(q3) + sin(q2)*sin(q3)*cos(q1), cos(q1)*cos(q2),␣
↪sin(q1)*sin(q3) + sin(q2)*cos(q1)*cos(q3)],
[                           sin(q3)*cos(q2),        -sin(q2),        ␣
↪                  cos(q2)*cos(q3)]])
```

is equivalent to:

```
>>> B1.orient_axis(N, N.z, q1)
>>> B2.orient_axis(B1, N.x, q2)
>>> B3.orient_axis(B2, N.y, q3)
>>> B3.dcm(N).simplify()
Matrix([
[ sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3), sin(q1)*cos(q2),␣
↪sin(q1)*sin(q2)*cos(q3) - sin(q3)*cos(q1)],
[-sin(q1)*cos(q3) + sin(q2)*sin(q3)*cos(q1), cos(q1)*cos(q2),␣
↪sin(q1)*sin(q3) + sin(q2)*cos(q1)*cos(q3)],
[                           sin(q3)*cos(q2),        -sin(q2),        ␣
↪                  cos(q2)*cos(q3)]])
```

It is worth noting that space-fixed and body-fixed rotations are related by the order of the rotations, i.e. the reverse order of body fixed will give space fixed and vice versa.

```
>>> B.orient_space_fixed(N, (q1, q2, q3), '231')
>>> B.dcm(N)
Matrix([
[cos(q1)*cos(q2), sin(q1)*sin(q3) + sin(q2)*cos(q1)*cos(q3), -
↪sin(q1)*cos(q3) + sin(q2)*sin(q3)*cos(q1)],
[       -sin(q2),                           cos(q2)*cos(q3),         ␣
↪                  sin(q3)*cos(q2)],
[sin(q1)*cos(q2), sin(q1)*sin(q2)*cos(q3) - sin(q3)*cos(q1), ␣
↪sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3)]])
```

```
>>> B.orient_body_fixed(N, (q3, q2, q1), '132')
>>> B.dcm(N)
Matrix([
[cos(q1)*cos(q2), sin(q1)*sin(q3) + sin(q2)*cos(q1)*cos(q3), -
↪sin(q1)*cos(q3) + sin(q2)*sin(q3)*cos(q1)],
[       -sin(q2),                           cos(q2)*cos(q3),                ␣
↪                   sin(q3)*cos(q2)],
[sin(q1)*cos(q2), sin(q1)*sin(q2)*cos(q3) - sin(q3)*cos(q1), ␣
↪sin(q1)*sin(q2)*sin(q3) + cos(q1)*cos(q3)]])
```

orientnew(*newname, rot_type, amounts, rot_order='', variables=None, indices=None, latexs=None*)

Returns a new reference frame oriented with respect to this reference frame.

See `ReferenceFrame.orient()` for detailed examples of how to orient reference frames.

Parameters

newname : str

Name for the new reference frame.

rot_type : str

The method used to generate the direction cosine matrix. Supported methods are:

- `'Axis'`: simple rotations about a single common axis
- `'DCM'`: for setting the direction cosine matrix directly
- `'Body'`: three successive rotations about new intermediate axes, also called "Euler and Tait-Bryan angles"
- `'Space'`: three successive rotations about the parent frames' unit vectors
- `'Quaternion'`: rotations defined by four parameters which result in a singularity free direction cosine matrix

amounts :

Expressions defining the rotation angles or direction cosine matrix. These must match the `rot_type`. See examples below for details. The input types are:

- `'Axis'`: 2-tuple (expr/sym/func, Vector)
- `'DCM'`: Matrix, shape(3,3)
- `'Body'`: 3-tuple of expressions, symbols, or functions
- `'Space'`: 3-tuple of expressions, symbols, or functions
- `'Quaternion'`: 4-tuple of expressions, symbols, or functions

rot_order : str or int, optional

If applicable, the order of the successive of rotations. The string `'123'` and integer `123` are equivalent, for example. Required for `'Body'` and `'Space'`.

indices : tuple of str

Enables the reference frame's basis unit vectors to be accessed by Python's square bracket indexing notation using the provided three indice strings and alters the printing of the unit vectors to reflect this choice.

latexs : tuple of str

Alters the LaTeX printing of the reference frame's basis unit vectors to the provided three valid LaTeX strings.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame, vlatex
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = ReferenceFrame('N')
```

Create a new reference frame A rotated relative to N through a simple rotation.

```
>>> A = N.orientnew('A', 'Axis', (q0, N.x))
```

Create a new reference frame B rotated relative to N through body-fixed rotations.

```
>>> B = N.orientnew('B', 'Body', (q1, q2, q3), '123')
```

Create a new reference frame C rotated relative to N through a simple rotation with unique indices and LaTeX printing.

```
>>> C = N.orientnew('C', 'Axis', (q0, N.x), indices=('1', '2', '3'),
... latexs=(r'\hat{\mathbf{c}}_1',r'\hat{\mathbf{c}}_2',
... r'\hat{\mathbf{c}}_3'))
>>> C['1']
C['1']
>>> print(vlatex(C['1']))
\hat{\mathbf{c}}_1
```

partial_velocity(*frame*, **gen_speeds*)

Returns the partial angular velocities of this frame in the given frame with respect to one or more provided generalized speeds.

Parameters

frame : ReferenceFrame

The frame with which the angular velocity is defined in.

gen_speeds : functions of time

The generalized speeds.

Returns

partial_velocities : tuple of Vector

The partial angular velocity vectors corresponding to the provided generalized speeds.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> u1, u2 = dynamicsymbols('u1, u2')
>>> A.set_ang_vel(N, u1 * A.x + u2 * N.y)
>>> A.partial_velocity(N, u1)
A.x
>>> A.partial_velocity(N, u1, u2)
(A.x, N.y)
```

set_ang_acc(*otherframe, value*)

Define the angular acceleration Vector in a ReferenceFrame.

Defines the angular acceleration of this ReferenceFrame, in another. Angular acceleration can be defined with respect to multiple different ReferenceFrames. Care must be taken to not create loops which are inconsistent.

Parameters

otherframe : ReferenceFrame

A ReferenceFrame to define the angular acceleration in

value : Vector

The Vector representing angular acceleration

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> V = 10 * N.x
>>> A.set_ang_acc(N, V)
>>> A.ang_acc_in(N)
10*N.x
```

set_ang_vel(*otherframe, value*)

Define the angular velocity vector in a ReferenceFrame.

Defines the angular velocity of this ReferenceFrame, in another. Angular velocity can be defined with respect to multiple different ReferenceFrames. Care must be taken to not create loops which are inconsistent.

Parameters

otherframe : ReferenceFrame

A ReferenceFrame to define the angular velocity in

value : Vector

The Vector representing angular velocity

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> V = 10 * N.x
>>> A.set_ang_vel(N, V)
>>> A.ang_vel_in(N)
10*N.x
```

variable_map(*otherframe*)

Returns a dictionary which expresses the coordinate variables of this frame in terms of the variables of otherframe.

If Vector.simp is True, returns a simplified version of the mapped values. Else, returns them without simplification.

Simplification of the expressions may take time.

Parameters

otherframe : ReferenceFrame

The other frame to map the variables to

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> A = ReferenceFrame('A')
>>> q = dynamicsymbols('q')
>>> B = A.orientnew('B', 'Axis', [q, A.z])
>>> A.variable_map(B)
{A_x: B_x*cos(q(t)) - B_y*sin(q(t)), A_y: B_x*sin(q(t)) + B_
↪y*cos(q(t)), A_z: B_z}
```

property x

The basis Vector for the ReferenceFrame, in the x direction.

property y

The basis Vector for the ReferenceFrame, in the y direction.

property z

The basis Vector for the ReferenceFrame, in the z direction.

Vector

class sympy.physics.vector.vector.**Vector**(*inlist*)

The class used to define vectors.

It along with ReferenceFrame are the building blocks of describing a classical mechanics system in PyDy and sympy.physics.vector.

Attributes

simp	(Boolean) Let certain methods use trigsimp on their outputs

angle_between(*vec*)

Returns the smallest angle between Vector 'vec' and self.

> **Warning:** Python ignores the leading negative sign so that might give wrong results. `-A.x.angle_between()` would be treated as `-(A.x.angle_between())`, instead of `(-A.x).angle_between()`.

Parameter

vec
[Vector] The Vector between which angle is needed.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> A = ReferenceFrame("A")
>>> v1 = A.x
>>> v2 = A.y
>>> v1.angle_between(v2)
pi/2
```

```
>>> v3 = A.x + A.y + A.z
>>> v1.angle_between(v3)
acos(sqrt(3)/3)
```

applyfunc(*f*)

Apply a function to each component of a vector.

cross(*other*)

The cross product operator for two Vectors.

Returns a Vector, expressed in the same ReferenceFrames as self.

Parameters

other : Vector

The Vector which we are crossing with

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame, cross
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> cross(N.x, N.y)
N.z
>>> A = ReferenceFrame('A')
>>> A.orient_axis(N, q1, N.x)
>>> cross(A.x, N.y)
N.z
>>> cross(N.y, A.x)
- sin(q1)*A.y - cos(q1)*A.z
```

diff(*var, frame, var_in_dcm=True*)

Returns the partial derivative of the vector with respect to a variable in the provided reference frame.

Parameters

var : Symbol

What the partial derivative is taken with respect to.

frame : ReferenceFrame

The reference frame that the partial derivative is taken in.

var_in_dcm : boolean

If true, the differentiation algorithm assumes that the variable may be present in any of the direction cosine matrices that relate the frame to the frames of any component of the vector. But if it is known that the variable is not present in the direction cosine matrices, false can be set to skip full reexpression in the desired frame.

Examples

```
>>> from sympy import Symbol
>>> from sympy.physics.vector import dynamicsymbols, ReferenceFrame
>>> from sympy.physics.vector import Vector
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> Vector.simp = True
>>> t = Symbol('t')
>>> q1 = dynamicsymbols('q1')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [q1, N.y])
>>> A.x.diff(t, N)
- sin(q1)*q1'*N.x - cos(q1)*q1'*N.z
>>> A.x.diff(t, N).express(A)
- q1'*A.z
>>> B = ReferenceFrame('B')
>>> u1, u2 = dynamicsymbols('u1, u2')
```

(continues on next page)

(continued from previous page)

```
>>> v = u1 * A.x + u2 * B.y
>>> v.diff(u2, N, var_in_dcm=False)
B.y
```

doit(***hints*)

Calls .doit() on each term in the Vector

dot(*other*)

Dot product of two vectors.

Returns a scalar, the dot product of the two Vectors

Parameters

other : Vector

The Vector which we are dotting with

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dot
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> dot(N.x, N.x)
1
>>> dot(N.x, N.y)
0
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> dot(N.y, A.y)
cos(q1)
```

dt(*otherframe*)

Returns a Vector which is the time derivative of the self Vector, taken in frame otherframe.

Calls the global time_derivative method

Parameters

otherframe : ReferenceFrame

The frame to calculate the time derivative in

express(*otherframe, variables=False*)

Returns a Vector equivalent to this one, expressed in otherframe. Uses the global express method.

Parameters

otherframe : ReferenceFrame

The frame for this Vector to be described in

variables : boolean

If True, the coordinate symbols(if present) in this Vector are re-expressed in terms otherframe

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q1 = dynamicsymbols('q1')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [q1, N.y])
>>> A.x.express(N)
cos(q1)*N.x - sin(q1)*N.z
```

free_dynamicsymbols(*reference_frame*)

Returns the free dynamic symbols (functions of time `t`) in the measure numbers of the vector expressed in the given reference frame.

Parameters

reference_frame : ReferenceFrame

The frame with respect to which the free dynamic symbols of the given vector is to be determined.

Returns

set

Set of functions of time `t`, e.g. `Function('f')(me.dynamicsymbols._t)`.

free_symbols(*reference_frame*)

Returns the free symbols in the measure numbers of the vector expressed in the given reference frame.

Parameters

reference_frame : ReferenceFrame

The frame with respect to which the free symbols of the given vector is to be determined.

Returns

set of Symbol

set of symbols present in the measure numbers of `reference_frame`.

property func

Returns the class Vector.

magnitude()

Returns the magnitude (Euclidean norm) of self.

Warning: Python ignores the leading negative sign so that might give wrong results. `-A.x.magnitude()` would be treated as `-(A.x.magnitude())`, instead of `(-A.x).magnitude()`.

normalize()

Returns a Vector of magnitude 1, codirectional with self.

outer(*other*)

Outer product between two Vectors.

A rank increasing operation, which returns a Dyadic from two Vectors

Parameters

other : Vector

The Vector to take the outer product with

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer
>>> N = ReferenceFrame('N')
>>> outer(N.x, N.x)
(N.x|N.x)
```

separate()

The constituents of this vector in different reference frames, as per its definition.

Returns a dict mapping each ReferenceFrame to the corresponding constituent Vector.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R1 = ReferenceFrame('R1')
>>> R2 = ReferenceFrame('R2')
>>> v = R1.x + R2.x
>>> v.separate() == {R1: R1.x, R2: R2.x}
True
```

simplify()

Returns a simplified Vector.

subs(**args, **kwargs*)

Substitution on the Vector.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy import Symbol
>>> N = ReferenceFrame('N')
>>> s = Symbol('s')
>>> a = N.x * s
>>> a.subs({s: 2})
2*N.x
```

to_matrix(*reference_frame*)

Returns the matrix form of the vector with respect to the given frame.

Parameters

reference_frame : ReferenceFrame

The reference frame that the rows of the matrix correspond to.

Returns

matrix : ImmutableMatrix, shape(3,1)

The matrix that gives the 1D vector.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame
>>> a, b, c = symbols('a, b, c')
>>> N = ReferenceFrame('N')
>>> vector = a * N.x + b * N.y + c * N.z
>>> vector.to_matrix(N)
Matrix([
[a],
[b],
[c]])
>>> beta = symbols('beta')
>>> A = N.orientnew('A', 'Axis', (beta, N.x))
>>> vector.to_matrix(A)
Matrix([
[                          a],
[ b*cos(beta) + c*sin(beta)],
[-b*sin(beta) + c*cos(beta)]])
```

xreplace(*rule*)

Replace occurrences of objects within the measure numbers of the vector.

Parameters

rule : dict-like

Expresses a replacement rule.

Returns

Vector

Result of the replacement.

Examples

```
>>> from sympy import symbols, pi
>>> from sympy.physics.vector import ReferenceFrame
>>> A = ReferenceFrame('A')
>>> x, y, z = symbols('x y z')
>>> ((1 + x*y) * A.x).xreplace({x: pi})
(pi*y + 1)*A.x
>>> ((1 + x*y) * A.x).xreplace({x: pi, y: 2})
(1 + 2*pi)*A.x
```

Replacements occur only if an entire node in the expression tree is matched:

```
>>> ((x*y + z) * A.x).xreplace({x*y: pi})
(z + pi)*A.x
>>> ((x*y*z) * A.x).xreplace({x*y: pi})
x*y*z*A.x
```

Dyadic

class sympy.physics.vector.dyadic.**Dyadic**(*inlist*)

A Dyadic object.

See: https://en.wikipedia.org/wiki/Dyadic_tensor Kane, T., Levinson, D. Dynamics Theory and Applications. 1985 McGraw-Hill

A more powerful way to represent a rigid body's inertia. While it is more complex, by choosing Dyadic components to be in body fixed basis vectors, the resulting matrix is equivalent to the inertia tensor.

applyfunc(*f*)

Apply a function to each component of a Dyadic.

cross(*other*)

For a cross product in the form: Dyadic x Vector.

Parameters

other : Vector

The Vector that we are crossing this Dyadic with

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer, cross
>>> N = ReferenceFrame('N')
>>> d = outer(N.x, N.x)
>>> cross(d, N.y)
(N.x|N.z)
```

doit(***hints*)

Calls .doit() on each term in the Dyadic

dot(*other*)

The inner product operator for a Dyadic and a Dyadic or Vector.

Parameters

other : Dyadic or Vector

The other Dyadic or Vector to take the inner product with

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer
>>> N = ReferenceFrame('N')
>>> D1 = outer(N.x, N.y)
>>> D2 = outer(N.y, N.y)
>>> D1.dot(D2)
(N.x|N.y)
>>> D1.dot(N.y)
N.x
```

dt(*frame*)

Take the time derivative of this Dyadic in a frame.

This function calls the global time_derivative method

Parameters

frame : ReferenceFrame

The frame to take the time derivative in

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer, ␣
↪dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')
>>> q = dynamicsymbols('q')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> d = outer(N.x, N.x)
>>> d.dt(B)
- q'*(N.y|N.x) - q'*(N.x|N.y)
```

express(*frame1*, *frame2=None*)

Expresses this Dyadic in alternate frame(s)

The first frame is the list side expression, the second frame is the right side; if Dyadic is in form A.x|B.y, you can express it in two different frames. If no second frame is given, the Dyadic is expressed in only one frame.

Calls the global express function

Parameters

frame1 : ReferenceFrame

The frame to express the left side of the Dyadic in

frame2 : ReferenceFrame

If provided, the frame to express the right side of the Dyadic in

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer,␣
↪dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')
>>> q = dynamicsymbols('q')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> d = outer(N.x, N.x)
>>> d.express(B, N)
cos(q)*(B.x|N.x) - sin(q)*(B.y|N.x)
```

property func

Returns the class Dyadic.

simplify()

Returns a simplified Dyadic.

subs(**args*, ***kwargs*)

Substitution on the Dyadic.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy import Symbol
>>> N = ReferenceFrame('N')
>>> s = Symbol('s')
>>> a = s*(N.x|N.x)
>>> a.subs({s: 2})
2*(N.x|N.x)
```

to_matrix(*reference_frame*, *second_reference_frame=None*)

Returns the matrix form of the dyadic with respect to one or two reference frames.

Parameters

reference_frame : ReferenceFrame

The reference frame that the rows and columns of the matrix correspond to. If a second reference frame is provided, this only corresponds to the rows of the matrix.

second_reference_frame : ReferenceFrame, optional, default=None

The reference frame that the columns of the matrix correspond to.

Returns

matrix : ImmutableMatrix, shape(3,3)

The matrix that gives the 2D tensor form.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame, Vector
>>> Vector.simp = True
>>> from sympy.physics.mechanics import inertia
>>> Ixx, Iyy, Izz, Ixy, Iyz, Ixz = symbols('Ixx, Iyy, Izz, Ixy, Iyz,␣
↪Ixz')
>>> N = ReferenceFrame('N')
>>> inertia_dyadic = inertia(N, Ixx, Iyy, Izz, Ixy, Iyz, Ixz)
>>> inertia_dyadic.to_matrix(N)
Matrix([
[Ixx, Ixy, Ixz],
[Ixy, Iyy, Iyz],
[Ixz, Iyz, Izz]])
>>> beta = symbols('beta')
>>> A = N.orientnew('A', 'Axis', (beta, N.x))
>>> inertia_dyadic.to_matrix(A)
Matrix([
[                           Ixx,                                           ␣
↪     Ixy*cos(beta) + Ixz*sin(beta),                                           ␣
↪          -Ixy*sin(beta) + Ixz*cos(beta)],
[ Ixy*cos(beta) + Ixz*sin(beta), Iyy*cos(2*beta)/2 + Iyy/2 +␣
↪Iyz*sin(2*beta) - Izz*cos(2*beta)/2 + Izz/2,                            -
↪Iyy*sin(2*beta)/2 + Iyz*cos(2*beta) + Izz*sin(2*beta)/2],
[-Ixy*sin(beta) + Ixz*cos(beta),                -Iyy*sin(2*beta)/2 +␣
↪Iyz*cos(2*beta) + Izz*sin(2*beta)/2, -Iyy*cos(2*beta)/2 + Iyy/2 -␣
↪Iyz*sin(2*beta) + Izz*cos(2*beta)/2 + Izz/2]])
```

xreplace(*rule*)

Replace occurrences of objects within the measure numbers of the Dyadic.

Parameters

rule : dict-like

Expresses a replacement rule.

Returns

Dyadic

Result of the replacement.

Examples

```
>>> from sympy import symbols, pi
>>> from sympy.physics.vector import ReferenceFrame, outer
>>> N = ReferenceFrame('N')
>>> D = outer(N.x, N.x)
>>> x, y, z = symbols('x y z')
>>> ((1 + x*y) * D).xreplace({x: pi})
(pi*y + 1)*(N.x|N.x)
>>> ((1 + x*y) * D).xreplace({x: pi, y: 2})
(1 + 2*pi)*(N.x|N.x)
```

Replacements occur only if an entire node in the expression tree is matched:

```
>>> ((x*y + z) * D).xreplace({x*y: pi})
(z + pi)*(N.x|N.x)
>>> ((x*y*z) * D).xreplace({x*y: pi})
x*y*z*(N.x|N.x)
```

Kinematics (Docstrings)

Point

class sympy.physics.vector.point.**Point**(*name*)

This object represents a point in a dynamic system.

It stores the: position, velocity, and acceleration of a point. The position is a vector defined as the vector distance from a parent point to this point.

Parameters

name : string

The display name of the Point

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame,␣
↪dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')
>>> O = Point('O')
>>> P = Point('P')
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> O.set_vel(N, u1 * N.x + u2 * N.y + u3 * N.z)
>>> O.acc(N)
u1'*N.x + u2'*N.y + u3'*N.z
```

symbols() can be used to create multiple Points in a single step, for example:

```
>>> from sympy.physics.vector import Point, ReferenceFrame,␣
↪dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> from sympy import symbols
>>> N = ReferenceFrame('N')
>>> u1, u2 = dynamicsymbols('u1 u2')
>>> A, B = symbols('A B', cls=Point)
>>> type(A)
<class 'sympy.physics.vector.point.Point'>
>>> A.set_vel(N, u1 * N.x + u2 * N.y)
>>> B.set_vel(N, u2 * N.x + u1 * N.y)
>>> A.acc(N) - B.acc(N)
(u1' - u2')*N.x + (-u1' + u2')*N.y
```

a1pt_theory(*otherpoint*, *outframe*, *interframe*)

Sets the acceleration of this point with the 1-point theory.

The 1-point theory for point acceleration looks like this:

^N a^P = ^B a^P + ^N a^O + ^N alpha^B x r^OP + ^N omega^B x (^N omega^B x r^OP) + 2 ^N omega^B x ^B v^P

where O is a point fixed in B, P is a point moving in B, and B is rotating in frame N.

Parameters

otherpoint : Point

The first point of the 1-point theory (O)

outframe : ReferenceFrame

The frame we want this point's acceleration defined in (N)

fixedframe : ReferenceFrame

The intermediate frame in this calculation (B)

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q = dynamicsymbols('q')
>>> q2 = dynamicsymbols('q2')
>>> qd = dynamicsymbols('q', 1)
>>> q2d = dynamicsymbols('q2', 1)
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B.set_ang_vel(N, 5 * B.y)
>>> O = Point('O')
>>> P = O.locatenew('P', q * B.x)
>>> P.set_vel(B, qd * B.x + q2d * B.y)
>>> O.set_vel(N, 0)
>>> P.a1pt_theory(O, N, B)
(-25*q + q'')*B.x + q2''*B.y - 10*q'*B.z
```

a2pt_theory(*otherpoint*, *outframe*, *fixedframe*)

Sets the acceleration of this point with the 2-point theory.

The 2-point theory for point acceleration looks like this:

^N a^P = ^N a^O + ^N alpha^B x r^OP + ^N omega^B x (^N omega^B x r^OP)

where O and P are both points fixed in frame B, which is rotating in frame N.

Parameters

otherpoint : Point

The first point of the 2-point theory (O)

outframe : ReferenceFrame

The frame we want this point's acceleration defined in (N)

fixedframe : ReferenceFrame

The frame in which both points are fixed (B)

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame,␣
↪dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q = dynamicsymbols('q')
>>> qd = dynamicsymbols('q', 1)
>>> N = ReferenceFrame('N')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> O = Point('O')
>>> P = O.locatenew('P', 10 * B.x)
>>> O.set_vel(N, 5 * N.x)
>>> P.a2pt_theory(O, N, B)
- 10*q'**2*B.x + 10*q''*B.y
```

`acc`(*frame*)

The acceleration Vector of this Point in a ReferenceFrame.

Parameters

frame : ReferenceFrame

The frame in which the returned acceleration vector will be defined in.

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_acc(N, 10 * N.x)
>>> p1.acc(N)
10*N.x
```

`locatenew`(*name*, *value*)

Creates a new point with a position defined from this point.

Parameters

name : str

The name for the new point

value : Vector

The position of the new point relative to this point

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> N = ReferenceFrame('N')
>>> P1 = Point('P1')
>>> P2 = P1.locatenew('P2', 10 * N.x)
```

partial_velocity(*frame, *gen_speeds*)

Returns the partial velocities of the linear velocity vector of this point in the given frame with respect to one or more provided generalized speeds.

Parameters

frame : ReferenceFrame

The frame with which the velocity is defined in.

gen_speeds : functions of time

The generalized speeds.

Returns

partial_velocities : tuple of Vector

The partial velocity vectors corresponding to the provided generalized speeds.

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> from sympy.physics.vector import dynamicsymbols
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> p = Point('p')
>>> u1, u2 = dynamicsymbols('u1, u2')
>>> p.set_vel(N, u1 * N.x + u2 * A.y)
>>> p.partial_velocity(N, u1)
N.x
>>> p.partial_velocity(N, u1, u2)
(N.x, A.y)
```

pos_from(*otherpoint*)

Returns a Vector distance between this Point and the other Point.

Parameters

otherpoint : Point

The otherpoint we are locating this one relative to

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p2 = Point('p2')
>>> p1.set_pos(p2, 10 * N.x)
>>> p1.pos_from(p2)
10*N.x
```

set_acc(*frame, value*)

Used to set the acceleration of this Point in a ReferenceFrame.

Parameters

frame : ReferenceFrame

The frame in which this point's acceleration is defined

value : Vector

The vector value of this point's acceleration in the frame

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_acc(N, 10 * N.x)
>>> p1.acc(N)
10*N.x
```

set_pos(*otherpoint, value*)

Used to set the position of this point w.r.t. another point.

Parameters

otherpoint : Point

The other point which this point's location is defined relative to

value : Vector

The vector which defines the location of this point

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p2 = Point('p2')
>>> p1.set_pos(p2, 10 * N.x)
>>> p1.pos_from(p2)
10*N.x
```

set_vel(*frame*, *value*)

Sets the velocity Vector of this Point in a ReferenceFrame.

Parameters

frame : ReferenceFrame

The frame in which this point's velocity is defined

value : Vector

The vector value of this point's velocity in the frame

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_vel(N, 10 * N.x)
>>> p1.vel(N)
10*N.x
```

v1pt_theory(*otherpoint*, *outframe*, *interframe*)

Sets the velocity of this point with the 1-point theory.

The 1-point theory for point velocity looks like this:

^N v^P = ^B v^P + ^N v^O + ^N omega^B x r^OP

where O is a point fixed in B, P is a point moving in B, and B is rotating in frame N.

Parameters

otherpoint : Point

The first point of the 1-point theory (O)

outframe : ReferenceFrame

The frame we want this point's velocity defined in (N)

interframe : ReferenceFrame

The intermediate frame in this calculation (B)

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q = dynamicsymbols('q')
>>> q2 = dynamicsymbols('q2')
>>> qd = dynamicsymbols('q', 1)
>>> q2d = dynamicsymbols('q2', 1)
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B.set_ang_vel(N, 5 * B.y)
```

(continues on next page)

(continued from previous page)

```
>>> O = Point('O')
>>> P = O.locatenew('P', q * B.x)
>>> P.set_vel(B, qd * B.x + q2d * B.y)
>>> O.set_vel(N, 0)
>>> P.v1pt_theory(O, N, B)
q'*B.x + q2'*B.y - 5*q*B.z
```

v2pt_theory(*otherpoint, outframe, fixedframe*)

Sets the velocity of this point with the 2-point theory.

The 2-point theory for point velocity looks like this:

^N v^P = ^N v^O + ^N omega^B x r^OP

where O and P are both points fixed in frame B, which is rotating in frame N.

Parameters

otherpoint : Point

The first point of the 2-point theory (O)

outframe : ReferenceFrame

The frame we want this point's velocity defined in (N)

fixedframe : ReferenceFrame

The frame in which both points are fixed (B)

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame,␣
↪dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q = dynamicsymbols('q')
>>> qd = dynamicsymbols('q', 1)
>>> N = ReferenceFrame('N')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> O = Point('O')
>>> P = O.locatenew('P', 10 * B.x)
>>> O.set_vel(N, 5 * N.x)
>>> P.v2pt_theory(O, N, B)
5*N.x + 10*q'*B.y
```

vel(*frame*)

The velocity Vector of this Point in the ReferenceFrame.

Parameters

frame : ReferenceFrame

The frame in which the returned velocity vector will be defined in

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame,␣
↪dynamicsymbols
>>> N = ReferenceFrame('N')
>>> p1 = Point('p1')
>>> p1.set_vel(N, 10 * N.x)
>>> p1.vel(N)
10*N.x
```

Velocities will be automatically calculated if possible, otherwise a `ValueError` will be returned. If it is possible to calculate multiple different velocities from the relative points, the points defined most directly relative to this point will be used. In the case of inconsistent relative positions of points, incorrect velocities may be returned. It is up to the user to define prior relative positions and velocities of points in a self-consistent way.

```
>>> p = Point('p')
>>> q = dynamicsymbols('q')
>>> p.set_vel(N, 10 * N.x)
>>> p2 = Point('p2')
>>> p2.set_pos(p, q*N.x)
>>> p2.vel(N)
(Derivative(q(t), t) + 10)*N.x
```

kinematic_equations

`sympy.physics.vector.functions.`**`get_motion_params`**(*frame*, ***kwargs*)

Returns the three motion parameters - (acceleration, velocity, and position) as vectorial functions of time in the given frame.

If a higher order differential function is provided, the lower order functions are used as boundary conditions. For example, given the acceleration, the velocity and position parameters are taken as boundary conditions.

The values of time at which the boundary conditions are specified are taken from timevalue1(for position boundary condition) and timevalue2(for velocity boundary condition).

If any of the boundary conditions are not provided, they are taken to be zero by default (zero vectors, in case of vectorial inputs). If the boundary conditions are also functions of time, they are converted to constants by substituting the time values in the dynamicsymbols._t time Symbol.

This function can also be used for calculating rotational motion parameters. Have a look at the Parameters and Examples for more clarity.

Parameters

frame : ReferenceFrame

The frame to express the motion parameters in

acceleration : Vector

Acceleration of the object/frame as a function of time

velocity : Vector

Velocity as function of time or as boundary condition of velocity at time = timevalue1

position : Vector

Velocity as function of time or as boundary condition of velocity at time = timevalue1

timevalue1 : sympyfiable

Value of time for position boundary condition

timevalue2 : sympyfiable

Value of time for velocity boundary condition

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, get_motion_params,␣
↪dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> from sympy import symbols
>>> R = ReferenceFrame('R')
>>> v1, v2, v3 = dynamicsymbols('v1 v2 v3')
>>> v = v1*R.x + v2*R.y + v3*R.z
>>> get_motion_params(R, position = v)
(v1''*R.x + v2''*R.y + v3''*R.z, v1'*R.x + v2'*R.y + v3'*R.z, v1*R.x +␣
↪v2*R.y + v3*R.z)
>>> a, b, c = symbols('a b c')
>>> v = a*R.x + b*R.y + c*R.z
>>> get_motion_params(R, velocity = v)
(0, a*R.x + b*R.y + c*R.z, a*t*R.x + b*t*R.y + c*t*R.z)
>>> parameters = get_motion_params(R, acceleration = v)
>>> parameters[1]
a*t*R.x + b*t*R.y + c*t*R.z
>>> parameters[2]
a*t**2/2*R.x + b*t**2/2*R.y + c*t**2/2*R.z
```

sympy.physics.vector.functions.**kinematic_equations**(*speeds, coords, rot_type, rot_order=''*)

Gives equations relating the qdot's to u's for a rotation type.

Supply rotation type and order as in orient. Speeds are assumed to be body-fixed; if we are defining the orientation of B in A using by rot_type, the angular velocity of B in A is assumed to be in the form: speed[0]*B.x + speed[1]*B.y + speed[2]*B.z

Parameters

speeds : list of length 3

The body fixed angular velocity measure numbers.

coords : list of length 3 or 4

The coordinates used to define the orientation of the two frames.

rot_type : str

The type of rotation used to create the equations. Body, Space, or Quaternion only

rot_order : str or int

If applicable, the order of a series of rotations.

Examples

```
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy.physics.vector import kinematic_equations, vprint
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> q1, q2, q3 = dynamicsymbols('q1 q2 q3')
>>> vprint(kinematic_equations([u1,u2,u3], [q1,q2,q3], 'body', '313'),
...     order=None)
[-(u1*sin(q3) + u2*cos(q3))/sin(q2) + q1', -u1*cos(q3) + u2*sin(q3) + q2
↪', (u1*sin(q3) + u2*cos(q3))*cos(q2)/sin(q2) - u3 + q3']
```

sympy.physics.vector.functions.**partial_velocity**(*vel_vecs, gen_speeds, frame*)

Returns a list of partial velocities with respect to the provided generalized speeds in the given reference frame for each of the supplied velocity vectors.

The output is a list of lists. The outer list has a number of elements equal to the number of supplied velocity vectors. The inner lists are, for each velocity vector, the partial derivatives of that velocity vector with respect to the generalized speeds supplied.

Parameters

vel_vecs : iterable

An iterable of velocity vectors (angular or linear).

gen_speeds : iterable

An iterable of generalized speeds.

frame : ReferenceFrame

The reference frame that the partial derivatives are going to be taken in.

Examples

```
>>> from sympy.physics.vector import Point, ReferenceFrame
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy.physics.vector import partial_velocity
>>> u = dynamicsymbols('u')
>>> N = ReferenceFrame('N')
>>> P = Point('P')
>>> P.set_vel(N, u * N.x)
>>> vel_vecs = [P.vel(N)]
>>> gen_speeds = [u]
>>> partial_velocity(vel_vecs, gen_speeds, N)
[[N.x]]
```

Printing (Docstrings)

init_vprinting

sympy.physics.vector.printing.**init_vprinting**(***kwargs*)

Initializes time derivative printing for all SymPy objects, i.e. any functions of time will be displayed in a more compact notation. The main benefit of this is for printing of time derivatives; instead of displaying as `Derivative(f(t),t)`, it will display `f'`. This is only actually needed for when derivatives are present and are not in a physics.vector.Vector or physics.vector.Dyadic object. This function is a light wrapper to *init_printing()* (page 2119). Any keyword arguments for it are valid here.

Initializes pretty-printer depending on the environment.

Parameters

pretty_print : bool, default=True

If `True`, use *pretty_print()* (page 2139) to stringify or the provided pretty printer; if `False`, use *sstrrepr()* (page 2178) to stringify or the provided string printer.

order : string or None, default='lex'

There are a few different settings for this parameter: `'lex'` (default), which is lexographic order; `'grlex'`, which is graded lexographic order; `'grevlex'`, which is reversed graded lexographic order; `'old'`, which is used for compatibility reasons and for long expressions; `None`, which sets it to lex.

use_unicode : bool or None, default=None

If `True`, use unicode characters; if `False`, do not use unicode characters; if `None`, make a guess based on the environment.

use_latex : string, bool, or None, default=None

If `True`, use default LaTeX rendering in GUI interfaces (png and mathjax); if `False`, do not use LaTeX rendering; if `None`, make a guess based on the environment; if `'png'`, enable LaTeX rendering with an external LaTeX compiler, falling back to matplotlib if external compilation fails; if `'matplotlib'`, enable LaTeX rendering with matplotlib; if `'mathjax'`, enable LaTeX text generation, for example MathJax rendering in IPython notebook or text rendering in LaTeX documents; if `'svg'`, enable LaTeX rendering with an external latex compiler, no fallback

wrap_line : bool

If True, lines will wrap at the end; if False, they will not wrap but continue as one line. This is only relevant if `pretty_print` is True.

num_columns : int or None, default=None

If `int`, number of columns before wrapping is set to num_columns; if `None`, number of columns before wrapping is set to terminal width. This is only relevant if `pretty_print` is `True`.

no_global : bool, default=False

If `True`, the settings become system wide; if `False`, use just for this console/session.

ip : An interactive console

This can either be an instance of IPython, or a class that derives from code.InteractiveConsole.

euler : bool, optional, default=False

Loads the euler package in the LaTeX preamble for handwritten style fonts (http://www.ctan.org/pkg/euler).

forecolor : string or None, optional, default=None

DVI setting for foreground color. `None` means that either `'Black'`, `'White'`, or `'Gray'` will be selected based on a guess of the IPython terminal color setting. See notes.

backcolor : string, optional, default='Transparent'

DVI setting for background color. See notes.

fontsize : string or int, optional, default='10pt'

A font size to pass to the LaTeX documentclass function in the preamble. Note that the options are limited by the documentclass. Consider using scale instead.

latex_mode : string, optional, default='plain'

The mode used in the LaTeX printer. Can be one of: `{'inline'|'plain'|'equation'|'equation*'}`.

print_builtin : boolean, optional, default=True

If `True` then floats and integers will be printed. If `False` the printer will only print SymPy types.

str_printer : function, optional, default=None

A custom string printer function. This should mimic *`sstrrepr()`* (page 2178).

pretty_printer : function, optional, default=None

A custom pretty printer. This should mimic *`pretty()`* (page 2139).

latex_printer : function, optional, default=None

A custom LaTeX printer. This should mimic *`latex()`* (page 2170).

scale : float, optional, default=1.0

Scale the LaTeX output when using the `'png'` or `'svg'` backends. Useful for high dpi screens.

settings :

Any additional settings for the `latex` and `pretty` commands can be used to fine-tune the output.

Examples

```
>>> from sympy import Function, symbols
>>> t, x = symbols('t, x')
>>> omega = Function('omega')
>>> omega(x).diff()
Derivative(omega(x), x)
>>> omega(t).diff()
Derivative(omega(t), t)
```

Now use the string printer:

```
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> omega(x).diff()
Derivative(omega(x), x)
>>> omega(t).diff()
omega'
```

vprint

`sympy.physics.vector.printing.`**`vprint`**`(`*expr, **settings*`)`

Function for printing of expressions generated in the sympy.physics vector package.

Extends SymPy's StrPrinter, takes the same setting accepted by SymPy's *`sstr()`* (page 2178), and is equivalent to `print(sstr(foo))`.

Parameters

expr : valid SymPy object

SymPy expression to print.

settings : args

Same as the settings accepted by SymPy's sstr().

Examples

```
>>> from sympy.physics.vector import vprint, dynamicsymbols
>>> u1 = dynamicsymbols('u1')
>>> print(u1)
u1(t)
>>> vprint(u1)
u1
```

vpprint

sympy.physics.vector.printing.**vpprint**(*expr*, ***settings*)

Function for pretty printing of expressions generated in the sympy.physics vector package.

Mainly used for expressions not inside a vector; the output of running scripts and generating equations of motion. Takes the same options as SymPy's *pretty_print()* (page 2139); see that function for more information.

Parameters

expr : valid SymPy object

SymPy expression to pretty print

settings : args

Same as those accepted by SymPy's pretty_print.

vlatex

sympy.physics.vector.printing.**vlatex**(*expr*, ***settings*)

Function for printing latex representation of sympy.physics.vector objects.

For latex representation of Vectors, Dyadics, and dynamicsymbols. Takes the same options as SymPy's *latex()* (page 2170); see that function for more information;

Parameters

expr : valid SymPy object

SymPy expression to represent in LaTeX form

settings : args

Same as latex()

Examples

```
>>> from sympy.physics.vector import vlatex, ReferenceFrame,␣
↪dynamicsymbols
>>> N = ReferenceFrame('N')
>>> q1, q2 = dynamicsymbols('q1 q2')
>>> q1d, q2d = dynamicsymbols('q1 q2', 1)
>>> q1dd, q2dd = dynamicsymbols('q1 q2', 2)
>>> vlatex(N.x + N.y)
'\\mathbf{\\hat{n}_x} + \\mathbf{\\hat{n}_y}'
>>> vlatex(q1 + q2)
'q_{1} + q_{2}'
>>> vlatex(q1d)
'\\dot{q}_{1}'
>>> vlatex(q1 * q2d)
'q_{1} \\dot{q}_{2}'
>>> vlatex(q1dd * q1 / q1d)
'\\frac{q_{1} \\ddot{q}_{1}}{\\dot{q}_{1}}'
```

Essential Functions (Docstrings)

dynamicsymbols

sympy.physics.vector.**dynamicsymbols**(*names*, *level=0*, ***assumptions*)

Uses symbols and Function for functions of time.

Creates a SymPy UndefinedFunction, which is then initialized as a function of a variable, the default being Symbol('t').

Parameters

names : str

Names of the dynamic symbols you want to create; works the same way as inputs to symbols

level : int

Level of differentiation of the returned function; d/dt once of t, twice of t, etc.

assumptions :

- **real(bool)**
  [This is used to set the dynamicsymbol as real,] by default is False.
- **positive(bool)**
  [This is used to set the dynamicsymbol as positive,] by default is False.
- **commutative(bool)**
  [This is used to set the commutative property of] a dynamicsymbol, by default is True.
- **integer(bool)**
  [This is used to set the dynamicsymbol as integer,] by default is False.

Examples

```
>>> from sympy.physics.vector import dynamicsymbols
>>> from sympy import diff, Symbol
>>> q1 = dynamicsymbols('q1')
>>> q1
q1(t)
>>> q2 = dynamicsymbols('q2', real=True)
>>> q2.is_real
True
>>> q3 = dynamicsymbols('q3', positive=True)
>>> q3.is_positive
True
>>> q4, q5 = dynamicsymbols('q4,q5', commutative=False)
>>> bool(q4*q5 != q5*q4)
True
>>> q6 = dynamicsymbols('q6', integer=True)
>>> q6.is_integer
True
```

(continues on next page)

(continued from previous page)

```
>>> diff(q1, Symbol('t'))
Derivative(q1(t), t)
```

dot

`sympy.physics.vector.functions.`**`dot`**`(`*vec1*, *vec2*`)`

Dot product convenience wrapper for Vector.dot(): Dot product of two vectors.

Returns a scalar, the dot product of the two Vectors

Parameters

other : Vector

The Vector which we are dotting with

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dot
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> dot(N.x, N.x)
1
>>> dot(N.x, N.y)
0
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> dot(N.y, A.y)
cos(q1)
```

cross

`sympy.physics.vector.functions.`**`cross`**`(`*vec1*, *vec2*`)`

Cross product convenience wrapper for Vector.cross(): The cross product operator for two Vectors.

Returns a Vector, expressed in the same ReferenceFrames as self.

Parameters

other : Vector

The Vector which we are crossing with

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame, cross
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> cross(N.x, N.y)
N.z
>>> A = ReferenceFrame('A')
>>> A.orient_axis(N, q1, N.x)
>>> cross(A.x, N.y)
N.z
>>> cross(N.y, A.x)
- sin(q1)*A.y - cos(q1)*A.z
```

outer

`sympy.physics.vector.functions.`**`outer`**`(`*vec1*, *vec2*`)`

Outer product convenience wrapper for Vector.outer(): Outer product between two Vectors.

A rank increasing operation, which returns a Dyadic from two Vectors

Parameters

other : Vector

The Vector to take the outer product with

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer
>>> N = ReferenceFrame('N')
>>> outer(N.x, N.x)
(N.x|N.x)
```

express

`sympy.physics.vector.functions.`**`express`**`(`*expr*, *frame*, *frame2=None*, *variables=False*`)`

Global function for 'express' functionality.

Re-expresses a Vector, scalar(sympyfiable) or Dyadic in given frame.

Refer to the local methods of Vector and Dyadic for details. If 'variables' is True, then the coordinate variables (CoordinateSym instances) of other frames present in the vector/scalar field or dyadic expression are also substituted in terms of the base scalars of this frame.

Parameters

expr : Vector/Dyadic/scalar(sympyfiable)

The expression to re-express in ReferenceFrame 'frame'

frame: ReferenceFrame

The reference frame to express expr in

frame2 : ReferenceFrame

The other frame required for re-expression(only for Dyadic expr)

variables : boolean

Specifies whether to substitute the coordinate variables present in expr, in terms of those of frame

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, outer,␣
↪dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')
>>> q = dynamicsymbols('q')
>>> B = N.orientnew('B', 'Axis', [q, N.z])
>>> d = outer(N.x, N.x)
>>> from sympy.physics.vector import express
>>> express(d, B, N)
cos(q)*(B.x|N.x) - sin(q)*(B.y|N.x)
>>> express(B.x, N)
cos(q)*N.x + sin(q)*N.y
>>> express(N[0], B, variables=True)
B_x*cos(q) - B_y*sin(q)
```

time_derivative

sympy.physics.vector.functions.**time_derivative**(*expr, frame, order=1*)

Calculate the time derivative of a vector/scalar field function or dyadic expression in given frame.

Parameters

expr : Vector/Dyadic/sympifyable

The expression whose time derivative is to be calculated

frame : ReferenceFrame

The reference frame to calculate the time derivative in

order : integer

The order of the derivative to be calculated

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, dynamicsymbols
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> from sympy import Symbol
>>> q1 = Symbol('q1')
>>> u1 = dynamicsymbols('u1')
>>> N = ReferenceFrame('N')
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> v = u1 * N.x
>>> A.set_ang_vel(N, 10*A.x)
>>> from sympy.physics.vector import time_derivative
>>> time_derivative(v, N)
u1'*N.x
>>> time_derivative(u1*A[0], N)
N_x*u1'
>>> B = N.orientnew('B', 'Axis', [u1, N.z])
>>> from sympy.physics.vector import outer
>>> d = outer(N.x, N.x)
>>> time_derivative(d, B)
- u1'*(N.y|N.x) - u1'*(N.x|N.y)
```

References

https://en.wikipedia.org/wiki/Rotating_reference_frame#Time_derivatives_in_the_two_frames

Docstrings for basic field functions

Field operation functions

These functions implement some basic operations pertaining to fields in general.

curl

`sympy.physics.vector.fieldfunctions.`**`curl`**(*vect, frame*)

Returns the curl of a vector field computed wrt the coordinate symbols of the given frame.

Parameters

vect : Vector

The vector operand

frame : ReferenceFrame

The reference frame to calculate the curl in

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import curl
>>> R = ReferenceFrame('R')
>>> v1 = R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z
>>> curl(v1, R)
0
>>> v2 = R[0]*R[1]*R[2]*R.x
>>> curl(v2, R)
R_x*R_y*R.y - R_x*R_z*R.z
```

divergence

`sympy.physics.vector.fieldfunctions.`**`divergence`**`(`*vect, frame*`)`

Returns the divergence of a vector field computed wrt the coordinate symbols of the given frame.

Parameters

vect : Vector

The vector operand

frame : ReferenceFrame

The reference frame to calculate the divergence in

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import divergence
>>> R = ReferenceFrame('R')
>>> v1 = R[0]*R[1]*R[2] * (R.x+R.y+R.z)
>>> divergence(v1, R)
R_x*R_y + R_x*R_z + R_y*R_z
>>> v2 = 2*R[1]*R[2]*R.y
>>> divergence(v2, R)
2*R_z
```

gradient

`sympy.physics.vector.fieldfunctions.`**`gradient`**`(`*scalar, frame*`)`

Returns the vector gradient of a scalar field computed wrt the coordinate symbols of the given frame.

Parameters

scalar : sympifiable

The scalar field to take the gradient of

frame : ReferenceFrame

The frame to calculate the gradient in

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import gradient
>>> R = ReferenceFrame('R')
>>> s1 = R[0]*R[1]*R[2]
>>> gradient(s1, R)
R_y*R_z*R.x + R_x*R_z*R.y + R_x*R_y*R.z
>>> s2 = 5*R[0]**2*R[2]
>>> gradient(s2, R)
10*R_x*R_z*R.x + 5*R_x**2*R.z
```

scalar_potential

`sympy.physics.vector.fieldfunctions.`**`scalar_potential`**(*field*, *frame*)

Returns the scalar potential function of a field in a given frame (without the added integration constant).

Parameters

field : Vector

The vector field whose scalar potential function is to be calculated

frame : ReferenceFrame

The frame to do the calculation in

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import scalar_potential, gradient
>>> R = ReferenceFrame('R')
>>> scalar_potential(R.z, R) == R[2]
True
>>> scalar_field = 2*R[0]**2*R[1]*R[2]
>>> grad_field = gradient(scalar_field, R)
>>> scalar_potential(grad_field, R)
2*R_x**2*R_y*R_z
```

scalar_potential_difference

sympy.physics.vector.fieldfunctions.**scalar_potential_difference**(*field*, *frame*, *point1*, *point2*, *origin*)

Returns the scalar potential difference between two points in a certain frame, wrt a given field.

If a scalar field is provided, its values at the two points are considered. If a conservative vector field is provided, the values of its scalar potential function at the two points are used.

Returns (potential at position 2) - (potential at position 1)

Parameters

field : Vector/sympyfiable

The field to calculate wrt

frame : ReferenceFrame

The frame to do the calculations in

point1 : Point

The initial Point in given frame

position2 : Point

The second Point in the given frame

origin : Point

The Point to use as reference point for position vector calculation

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> from sympy.physics.vector import scalar_potential_difference
>>> R = ReferenceFrame('R')
>>> O = Point('O')
>>> P = O.locatenew('P', R[0]*R.x + R[1]*R.y + R[2]*R.z)
>>> vectfield = 4*R[0]*R[1]*R.x + 2*R[0]**2*R.y
>>> scalar_potential_difference(vectfield, R, O, P, O)
2*R_x**2*R_y
>>> Q = O.locatenew('O', 3*R.x + R.y + 2*R.z)
>>> scalar_potential_difference(vectfield, R, P, Q, O)
-2*R_x**2*R_y + 18
```

Checking the type of vector field

is_conservative

sympy.physics.vector.fieldfunctions.**is_conservative**(*field*)

Checks if a field is conservative.

Parameters

field : Vector

The field to check for conservative property

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import is_conservative
>>> R = ReferenceFrame('R')
>>> is_conservative(R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z)
True
>>> is_conservative(R[2] * R.y)
False
```

is_solenoidal

sympy.physics.vector.fieldfunctions.**is_solenoidal**(*field*)

Checks if a field is solenoidal.

Parameters

field : Vector

The field to check for solenoidal property

Examples

```
>>> from sympy.physics.vector import ReferenceFrame
>>> from sympy.physics.vector import is_solenoidal
>>> R = ReferenceFrame('R')
>>> is_solenoidal(R[1]*R[2]*R.x + R[0]*R[2]*R.y + R[0]*R[1]*R.z)
True
>>> is_solenoidal(R[1] * R.y)
False
```

Classical Mechanics

Abstract

In this documentation many components of the physics/mechanics module will be discussed. *sympy.physics.mechanics* (page 1675) has been written to allow for creation of symbolic equations of motion for complicated multibody systems.

Vector

This module derives the vector-related abilities and related functionalities from *sympy.physics.vector* (page 1592). Please have a look at the documentation of *sympy.physics.vector* (page 1592) and its necessary API to understand the vector capabilities of *sympy.physics.mechanics* (page 1675).

Mechanics

In physics, mechanics describes conditions of rest (statics) or motion (dynamics). There are a few common steps to all mechanics problems. First, an idealized representation of a system is described. Next, we use physical laws to generate equations that define the system's behavior. Then, we solve these equations, sometimes analytically but usually numerically. Finally, we extract information from these equations and solutions. The current scope of the module is multi-body dynamics: the motion of systems of multiple particles and/or rigid bodies. For example, this module could be used to understand the motion of a double pendulum, planets, robotic manipulators, bicycles, and any other system of rigid bodies that may fascinate us.

Often, the objective in multi-body dynamics is to obtain the trajectory of a system of rigid bodies through time. The challenge for this task is to first formulate the equations of motion of the system. Once they are formulated, they must be solved, that is, integrated forward in time. When digital computers came around, solving became the easy part of the problem. Now, we can tackle more complicated problems, which leaves the challenge of formulating the equations.

The term "equations of motion" is used to describe the application of Newton's second law to multi-body systems. The form of the equations of motion depends on the method used to generate them. This package implements two of these methods: Kane's method and Lagrange's method. This module facilitates the formulation of equations of motion, which can then be solved (integrated) using generic ordinary differential equation (ODE) solvers.

The approach to a particular class of dynamics problems, that of forward dynamics, has the following steps:

1. describing the system's geometry and configuration,
2. specifying the way the system can move, including constraints on its motion
3. describing the external forces and moments on the system,
4. combining the above information according to Newton's second law ($\mathbf{F} = m\mathbf{a}$), and
5. organizing the resulting equations so that they can be integrated to obtain the system's trajectory through time.

Together with the rest of SymPy, this module performs steps 4 and 5, provided that the user can perform 1 through 3 for the module. That is to say, the user must provide a complete representation of the free body diagrams that themselves represent the system, with which this code can provide equations of motion in a form amenable to numerical integration. Step 5 above amounts to arduous algebra for even fairly simple multi-body systems. Thus, it is desirable to use a symbolic math package, such as SymPy, to perform this step. It is for this reason that this module is a part of SymPy. Step 4 amounts to this specific module, sympy.physics.mechanics.

Guide to Mechanics

Masses, Inertias, Particles and Rigid Bodies in Physics/Mechanics

This document will describe how to represent masses and inertias in *sympy.physics.mechanics* (page 1675) and use of the `RigidBody` and `Particle` classes.

It is assumed that the reader is familiar with the basics of these topics, such as finding the center of mass for a system of particles, how to manipulate an inertia tensor, and the definition of a particle and rigid body. Any advanced dynamics text can provide a reference for these details.

Mass

The only requirement for a mass is that it needs to be a `sympify`-able expression. Keep in mind that masses can be time varying.

Particle

Particles are created with the class `Particle` in *sympy.physics.mechanics* (page 1675). A `Particle` object has an associated point and an associated mass which are the only two attributes of the object.:

```
>>> from sympy.physics.mechanics import Particle, Point
>>> from sympy import Symbol
>>> m = Symbol('m')
>>> po = Point('po')
>>> # create a particle container
>>> pa = Particle('pa', po, m)
```

The associated point contains the position, velocity and acceleration of the particle. *sympy.physics.mechanics* (page 1675) allows one to perform kinematic analysis of points separate from their association with masses.