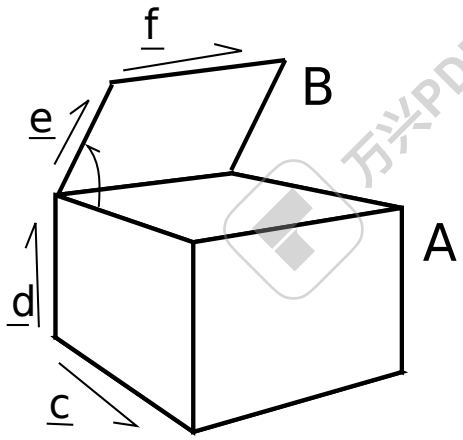
Vector Calculus

To deal with the calculus of vectors with moving object, we have to introduce the concept of a reference frame. A classic example is a train moving along its tracks, with you and a friend inside. If both you and your friend are sitting, the relative velocity between the two of you is zero. From an observer outside the train, you will both have velocity though.

We will now apply more rigor to this definition. A reference frame is a virtual "platform" which we choose to observe vector quantities from. If we have a reference frame **N**, vector **a** is said to be fixed in the frame **N** if none of its properties ever change when observed from **N**. We will typically assign a fixed orthonormal basis vector set with each reference frame; **N** will have $\hat{\mathbf{n}}_\mathbf{x}, \hat{\mathbf{n}}_\mathbf{y}, \hat{\mathbf{n}}_\mathbf{z}$ as its basis vectors.

Derivatives of Vectors

A vector which is not fixed in a reference frame therefore has changing properties when observed from that frame. Calculus is the study of change, and in order to deal with the peculiarities of vectors fixed and not fixed in different reference frames, we need to be more explicit in our definitions.



Fixed in:

	A	B
c	x	
d	x	
e		x
f		x

In the above figure, we have vectors $\mathbf{c}, \mathbf{d}, \mathbf{e}, \mathbf{f}$. If one were to take the derivative of **e** with respect to θ:

$$\frac{d\mathbf{e}}{d\theta}$$

it is not clear what the derivative is. If you are observing from frame **A**, it is clearly non-zero. If you are observing from frame **B**, the derivative is zero. We will therefore introduce the

frame as part of the derivative notation:

$\frac{^{\mathbf{A}}d\mathbf{e}}{d\theta} \neq 0$, the derivative of $\mathbf{e}$ with respect to θ in the reference frame $\mathbf{A}$
$\frac{^{\mathbf{B}}d\mathbf{e}}{d\theta} = 0$, the derivative of $\mathbf{e}$ with respect to θ in the reference frame $\mathbf{B}$
$\frac{^{\mathbf{A}}d\mathbf{c}}{d\theta} = 0$, the derivative of $\mathbf{c}$ with respect to θ in the reference frame $\mathbf{A}$
$\frac{^{\mathbf{B}}d\mathbf{c}}{d\theta} \neq 0$, the derivative of $\mathbf{c}$ with respect to θ in the reference frame $\mathbf{B}$

Here are some additional properties of derivatives of vectors in specific frames:

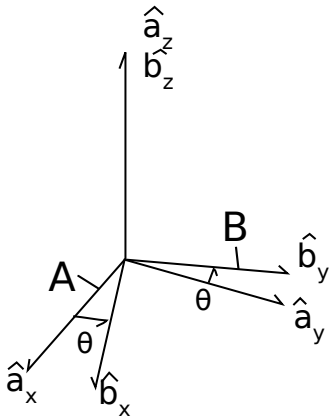
$$\frac{^{\mathbf{A}}d}{dt}(\mathbf{a}+\mathbf{b}) = \frac{^{\mathbf{A}}d\mathbf{a}}{dt} + \frac{^{\mathbf{A}}d\mathbf{b}}{dt}$$
$$\frac{^{\mathbf{A}}d}{dt}\gamma\mathbf{a} = \frac{d\gamma}{dt}\mathbf{a} + \gamma\frac{^{\mathbf{A}}d\mathbf{a}}{dt}$$
$$\frac{^{\mathbf{A}}d}{dt}(\mathbf{a}\times\mathbf{b}) = \frac{^{\mathbf{A}}d\mathbf{a}}{dt}\times\mathbf{b} + \mathbf{a}\times\frac{^{\mathbf{A}}d\mathbf{b}}{dt}$$

Relating Sets of Basis Vectors

We need to now define the relationship between two different reference frames; or how to relate the basis vectors of one frame to another. We can do this using a direction cosine matrix (DCM). The direction cosine matrix relates the basis vectors of one frame to another, in the following fashion:

$$\begin{bmatrix}\hat{\mathbf{a}}_\mathbf{x}\\ \hat{\mathbf{a}}_\mathbf{y}\\ \hat{\mathbf{a}}_\mathbf{z}\end{bmatrix} = [^{\mathbf{A}}\mathbf{C}^{\mathbf{B}}]\begin{bmatrix}\hat{\mathbf{b}}_\mathbf{x}\\ \hat{\mathbf{b}}_\mathbf{y}\\ \hat{\mathbf{b}}_\mathbf{z}\end{bmatrix}$$

When two frames (say, **A** & **B**) are initially aligned, then one frame has all of its basis vectors rotated around an axis which is aligned with a basis vector, we say the frames are related by a simple rotation. The figure below shows this:



The above rotation is a simple rotation about the Z axis by an angle θ. Note that after the rotation, the basis vectors $\hat{\mathbf{a}}_\mathbf{z}$ and $\hat{\mathbf{b}}_\mathbf{z}$ are still aligned.

This rotation can be characterized by the following direction cosine matrix:

$$^{\mathbf{A}}\mathbf{C}^{\mathbf{B}} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Simple rotations about the X and Y axes are defined by:

$$\text{DCM for x-axis rotation: } \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

$$\text{DCM for y-axis rotation: } \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

Rotation in the positive direction here will be defined by using the right-hand rule.

The direction cosine matrix is also involved with the definition of the dot product between sets of basis vectors. If we have two reference frames with associated basis vectors, their direction cosine matrix can be defined as:

$$\begin{bmatrix} C_{xx} & C_{xy} & C_{xz} \\ C_{yx} & C_{yy} & C_{yz} \\ C_{zx} & C_{zy} & C_{zz} \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{a}}_{\mathbf{x}} \cdot \hat{\mathbf{b}}_{\mathbf{x}} & \hat{\mathbf{a}}_{\mathbf{x}} \cdot \hat{\mathbf{b}}_{\mathbf{y}} & \hat{\mathbf{a}}_{\mathbf{x}} \cdot \hat{\mathbf{b}}_{\mathbf{z}} \\ \hat{\mathbf{a}}_{\mathbf{y}} \cdot \hat{\mathbf{b}}_{\mathbf{x}} & \hat{\mathbf{a}}_{\mathbf{y}} \cdot \hat{\mathbf{b}}_{\mathbf{y}} & \hat{\mathbf{a}}_{\mathbf{y}} \cdot \hat{\mathbf{b}}_{\mathbf{z}} \\ \hat{\mathbf{a}}_{\mathbf{z}} \cdot \hat{\mathbf{b}}_{\mathbf{x}} & \hat{\mathbf{a}}_{\mathbf{z}} \cdot \hat{\mathbf{b}}_{\mathbf{y}} & \hat{\mathbf{a}}_{\mathbf{z}} \cdot \hat{\mathbf{b}}_{\mathbf{z}} \end{bmatrix}$$

Additionally, the direction cosine matrix is orthogonal, in that:

$$\begin{aligned} ^{\mathbf{A}}\mathbf{C}^{\mathbf{B}} &= (^{\mathbf{B}}\mathbf{C}^{\mathbf{A}})^{-1} \\ &= (^{\mathbf{B}}\mathbf{C}^{\mathbf{A}})^{T} \end{aligned}$$

If we have reference frames **A** and **B**, which in this example have undergone a simple z-axis rotation by an amount θ, we will have two sets of basis vectors. We can then define two vectors: $\mathbf{a} = \hat{\mathbf{a}}_{\mathbf{x}} + \hat{\mathbf{a}}_{\mathbf{y}} + \hat{\mathbf{a}}_{\mathbf{z}}$ and $\mathbf{b} = \hat{\mathbf{b}}_{\mathbf{x}} + \hat{\mathbf{b}}_{\mathbf{y}} + \hat{\mathbf{b}}_{\mathbf{z}}$. If we wish to express **b** in the **A** frame, we do the following:

$$\begin{aligned} \mathbf{b} &= \hat{\mathbf{b}}_{\mathbf{x}} + \hat{\mathbf{b}}_{\mathbf{y}} + \hat{\mathbf{b}}_{\mathbf{z}} \\ \mathbf{b} &= \left[\hat{\mathbf{a}}_{\mathbf{x}} \cdot (\hat{\mathbf{b}}_{\mathbf{x}} + \hat{\mathbf{b}}_{\mathbf{y}} + \hat{\mathbf{b}}_{\mathbf{z}})\right] \hat{\mathbf{a}}_{\mathbf{x}} + \left[\hat{\mathbf{a}}_{\mathbf{y}} \cdot (\hat{\mathbf{b}}_{\mathbf{x}} + \hat{\mathbf{b}}_{\mathbf{y}} + \hat{\mathbf{b}}_{\mathbf{z}})\right] \hat{\mathbf{a}}_{\mathbf{y}} + \left[\hat{\mathbf{a}}_{\mathbf{z}} \cdot (\hat{\mathbf{b}}_{\mathbf{x}} + \hat{\mathbf{b}}_{\mathbf{y}} + \hat{\mathbf{b}}_{\mathbf{z}})\right] \hat{\mathbf{a}}_{\mathbf{z}} \\ \mathbf{b} &= (\cos(\theta) - \sin(\theta))\hat{\mathbf{a}}_{\mathbf{x}} + (\sin(\theta) + \cos(\theta))\hat{\mathbf{a}}_{\mathbf{y}} + \hat{\mathbf{a}}_{\mathbf{z}} \end{aligned}$$

And if we wish to express **a** in the **B**, we do:

$$\begin{aligned} \mathbf{a} &= \hat{\mathbf{a}}_{\mathbf{x}} + \hat{\mathbf{a}}_{\mathbf{y}} + \hat{\mathbf{a}}_{\mathbf{z}} \\ \mathbf{a} &= \left[\hat{\mathbf{b}}_{\mathbf{x}} \cdot (\hat{\mathbf{a}}_{\mathbf{x}} + \hat{\mathbf{a}}_{\mathbf{y}} + \hat{\mathbf{a}}_{\mathbf{z}})\right] \hat{\mathbf{b}}_{\mathbf{x}} + \left[\hat{\mathbf{b}}_{\mathbf{y}} \cdot (\hat{\mathbf{a}}_{\mathbf{x}} + \hat{\mathbf{a}}_{\mathbf{y}} + \hat{\mathbf{a}}_{\mathbf{z}})\right] \hat{\mathbf{b}}_{\mathbf{y}} + \left[\hat{\mathbf{b}}_{\mathbf{z}} \cdot (\hat{\mathbf{a}}_{\mathbf{x}} + \hat{\mathbf{a}}_{\mathbf{y}} + \hat{\mathbf{a}}_{\mathbf{z}})\right] \hat{\mathbf{b}}_{\mathbf{z}} \\ \mathbf{a} &= (\cos(\theta) + \sin(\theta))\hat{\mathbf{b}}_{\mathbf{x}} + (-\sin(\theta) + \cos(\theta))\hat{\mathbf{b}}_{\mathbf{y}} + \hat{\mathbf{b}}_{\mathbf{z}} \end{aligned}$$

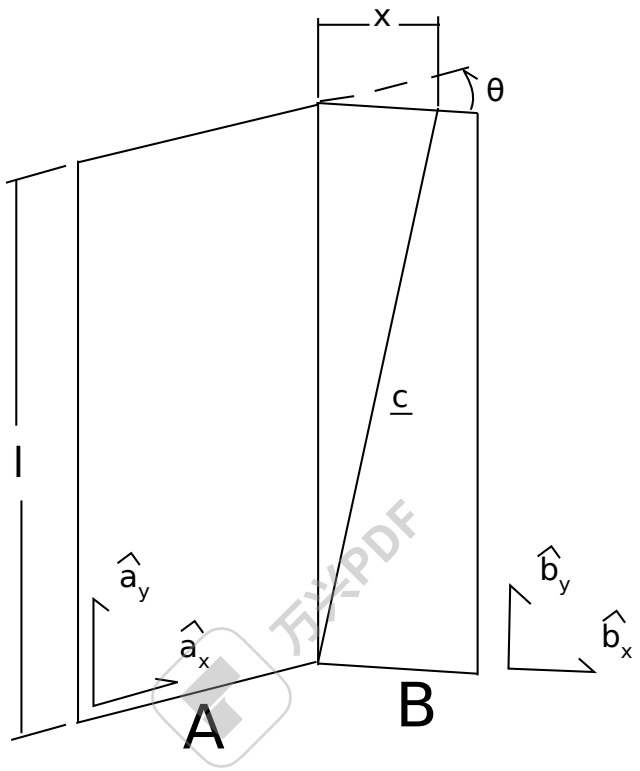
Derivatives with Multiple Frames

If we have reference frames **A** and **B** we will have two sets of basis vectors. We can then define two vectors: $\mathbf{a} = a_x\hat{\mathbf{a}}_{\mathbf{x}} + a_y\hat{\mathbf{a}}_{\mathbf{y}} + a_z\hat{\mathbf{a}}_{\mathbf{z}}$ and $\mathbf{b} = b_x\hat{\mathbf{b}}_{\mathbf{x}} + b_y\hat{\mathbf{b}}_{\mathbf{y}} + b_z\hat{\mathbf{b}}_{\mathbf{z}}$. If we want to take the derivative of **b** in the reference frame **A**, we must first express it in **A**, and the take the derivatives of the measure numbers:

$$\frac{^{\mathbf{A}}d\mathbf{b}}{dx} = \frac{d(\mathbf{b} \cdot \hat{\mathbf{a}}_{\mathbf{x}})}{dx}\hat{\mathbf{a}}_{\mathbf{x}} + \frac{d(\mathbf{b} \cdot \hat{\mathbf{a}}_{\mathbf{y}})}{dx}\hat{\mathbf{a}}_{\mathbf{y}} + \frac{d(\mathbf{b} \cdot \hat{\mathbf{a}}_{\mathbf{z}})}{dx}\hat{\mathbf{a}}_{\mathbf{z}} +$$

Examples

An example of vector calculus:



In this example we have two bodies, each with an attached reference frame. We will say that θ and x are functions of time. We wish to know the time derivative of vector **c** in both the **A** and **B** frames.

First, we need to define **c**; $\mathbf{c} = x\hat{\mathbf{b}}_\mathbf{x} + l\hat{\mathbf{b}}_\mathbf{y}$. This provides a definition in the **B** frame. We can now do the following:

$$\begin{aligned}\frac{^\mathbf{B}d\mathbf{c}}{dt} &= \frac{dx}{dt}\hat{\mathbf{b}}_\mathbf{x} + \frac{dl}{dt}\hat{\mathbf{b}}_\mathbf{y} \\ &= \dot{x}\hat{\mathbf{b}}_\mathbf{x}\end{aligned}$$

To take the derivative in the **A** frame, we have to first relate the two frames:

$$^\mathbf{A}\mathbf{C}^\mathbf{B} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

Now we can do the following:

$$\begin{aligned}\frac{^{\mathbf{A}}d\mathbf{c}}{dt} &= \frac{d(\mathbf{c}\cdot\hat{\mathbf{a}}_\mathbf{x})}{dt}\hat{\mathbf{a}}_\mathbf{x} + \frac{d(\mathbf{c}\cdot\hat{\mathbf{a}}_\mathbf{y})}{dt}\hat{\mathbf{a}}_\mathbf{y} + \frac{d(\mathbf{c}\cdot\hat{\mathbf{a}}_\mathbf{z})}{dt}\hat{\mathbf{a}}_\mathbf{z} \\ &= \frac{d(\cos(\theta)x)}{dt}\hat{\mathbf{a}}_\mathbf{x} + \frac{d(l)}{dt}\hat{\mathbf{a}}_\mathbf{y} + \frac{d(-\sin(\theta)x)}{dt}\hat{\mathbf{a}}_\mathbf{z} \\ &= (-\dot{\theta}\sin(\theta)x + \cos(\theta)\dot{x})\hat{\mathbf{a}}_\mathbf{x} + (\dot{\theta}\cos(\theta)x + \sin(\theta)\dot{x})\hat{\mathbf{a}}_\mathbf{z}\end{aligned}$$

Note that this is the time derivative of **c** in **A**, and is expressed in the **A** frame. We can express it in the **B** frame however, and the expression will still be valid:

$$\begin{aligned}\frac{^{\mathbf{A}}d\mathbf{c}}{dt} &= (-\dot{\theta}\sin(\theta)x + \cos(\theta)\dot{x})\hat{\mathbf{a}}_\mathbf{x} + (\dot{\theta}\cos(\theta)x + \sin(\theta)\dot{x})\hat{\mathbf{a}}_\mathbf{z} \\ &= \dot{x}\hat{\mathbf{b}}_\mathbf{x} - \theta x\hat{\mathbf{b}}_\mathbf{z}\end{aligned}$$

Note the difference in expression complexity between the two forms. They are equivalent, but one is much simpler. This is an extremely important concept, as defining vectors in the more complex forms can vastly slow down formulation of the equations of motion and increase their length, sometimes to a point where they cannot be shown on screen.

Using Vectors and Reference Frames

We have waited until after all of the relevant mathematical relationships have been defined for vectors and reference frames to introduce code. This is due to how vectors are formed. When starting any problem in *sympy.physics.vector* (page 1592), one of the first steps is defining a reference frame (remember to import sympy.physics.vector first):

```
>>> from sympy.physics.vector import *
>>> N = ReferenceFrame('N')
```

Now we have created a reference frame, **N**. To have access to any basis vectors, first a reference frame needs to be created. Now that we have made and object representing **N**, we can access its basis vectors:

```
>>> N.x
N.x
>>> N.y
N.y
>>> N.z
N.z
```

Vector Algebra, in physics.vector

We can now do basic algebraic operations on these vectors.:

```
>>> N.x == N.x
True
>>> N.x == N.y
False
>>> N.x + N.y
N.x + N.y
>>> 2 * N.x + N.y
2*N.x + N.y
```

Remember, don't add a scalar quantity to a vector (`N.x + 5`); this will raise an error. At this point, we'll use SymPy's Symbol in our vectors. Remember to refer to SymPy's Gotchas and Pitfalls when dealing with symbols.:

```
>>> from sympy import Symbol, symbols
>>> x = Symbol('x')
>>> x * N.x
x*N.x
>>> x*(N.x + N.y)
x*N.x + x*N.y
```

In *sympy.physics.vector* (page 1592) multiple interfaces to vector multiplication have been implemented, at the operator level, method level, and function level. The vector dot product can work as follows:

```
>>> N.x.dot(N.x)
1
>>> N.x.dot(N.y)
0
>>> dot(N.x, N.x)
1
>>> dot(N.x, N.y)
0
```

The "official" interface is the function interface; this is what will be used in all examples. This is to avoid confusion with the attribute and methods being next to each other, and in the case of the operator operation priority. The operators used in *sympy.physics.vector* (page 1592) for vector multiplication do not posses the correct order of operations; this can lead to errors. Care with parentheses is needed when using operators to represent vector multiplication.

The cross product is the other vector multiplication which will be discussed here. It offers similar interfaces to the dot product, and comes with the same warnings.

```
>>> N.x.cross(N.x)
0
>>> N.x.cross(N.z)
- N.y
>>> cross(N.x, N.y)
N.z
>>> cross(N.x, (N.y + N.z))
- N.y + N.z
```

Two additional operations can be done with vectors: normalizing the vector to length 1, and getting its magnitude. These are done as follows:

```
>>> (N.x + N.y).normalize()
sqrt(2)/2*N.x + sqrt(2)/2*N.y
>>> (N.x + N.y).magnitude()
sqrt(2)
```

Vectors are often expressed in a matrix form, especially for numerical purposes. Since the matrix form does not contain any information about the reference frame the vector is defined in, you must provide a reference frame to extract the measure numbers from the vector. There is a convenience function to do this:

```
>>> (x * N.x + 2 * x * N.y + 3 * x * N.z).to_matrix(N)
Matrix([
  [  x],
  [2*x],
  [3*x]])
```

Vector Calculus, in physics.vector

We have already introduced our first reference frame. We can take the derivative in that frame right now, if we desire:

```
>>> (x * N.x + N.y).diff(x, N)
N.x
```

SymPy has a `diff` function, but it does not currently work with *sympy.physics.vector* (page 1592) Vectors, so please use `Vector`'s `diff` method. The reason for this is that when differentiating a `Vector`, the frame of reference must be specified in addition to what you are taking the derivative with respect to; SymPy's `diff` function doesn't fit this mold.

The more interesting case arise with multiple reference frames. If we introduce a second reference frame, **A**, we now have two frames. Note that at this point we can add components of **N** and **A** together, but cannot perform vector multiplication, as no relationship between the two frames has been defined.

```
>>> A = ReferenceFrame('A')
>>> A.x + N.x
N.x + A.x
```

If we want to do vector multiplication, first we have to define and orientation. The `orient` method of `ReferenceFrame` provides that functionality.

```
>>> A.orient(N, 'Axis', [x, N.y])
```

If we desire, we can view the DCM between these two frames at any time. This can be calculated with the `dcm` method. This code: `N.dcm(A)` gives the dcm ${}^{\mathbf{A}}\mathbf{C}^{\mathbf{N}}$.

This orients the **A** frame relative to the **N** frame by a simple rotation around the Y axis, by an amount x. Other, more complicated rotation types include Body rotations, Space rotations, quaternions, and arbitrary axis rotations. Body and space rotations are equivalent to doing 3 simple rotations in a row, each about a basis vector in the new frame. An example follows:

```
>>> N = ReferenceFrame('N')
>>> Bp = ReferenceFrame('Bp')
>>> Bpp = ReferenceFrame('Bpp')
>>> B = ReferenceFrame('B')
>>> q1,q2,q3 = symbols('q1 q2 q3')
>>> Bpp.orient(N,'Axis', [q1, N.x])
>>> Bp.orient(Bpp,'Axis', [q2, Bpp.y])
>>> B.orient(Bp,'Axis', [q3, Bp.z])
>>> N.dcm(B)
Matrix([
[                        cos(q2)*cos(q3),                                  -
↪sin(q3)*cos(q2),          sin(q2)],
```

(continues on next page)

(continued from previous page)

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

Space orientations are similar to body orientation, but applied from the frame to body. Body and space rotations can involve either two or three axes: 'XYZ' works, as does 'YZX', 'ZXZ', 'YXY', etc. What is key is that each simple rotation is about a different axis than the previous one; 'ZZX' does not completely orient a set of basis vectors in 3 space.

Sometimes it will be more convenient to create a new reference frame and orient relative to an existing one in one step. The `orientnew` method allows for this functionality, and essentially wraps the `orient` method. All of the things you can do in `orient`, you can do in `orientnew`.

```
>>> C = N.orientnew('C', 'Axis', [q1, N.x])
```

Quaternions (or Euler Parameters) use 4 value to characterize the orientation of the frame. This and arbitrary axis rotations are described in the `orient` and `orientnew` method help, or in the references [Kane1983].

Finally, before starting multiframe calculus operations, we will introduce another *sympy.physics.vector* (page 1592) tool: `dynamicsymbols`. `dynamicsymbols` is a shortcut function to create undefined functions of time within SymPy. The derivative of such a 'dynamicsymbol' is shown below.

```
>>> from sympy import diff
>>> q1, q2, q3 = dynamicsymbols('q1 q2 q3')
>>> diff(q1, Symbol('t'))
Derivative(q1(t), t)
```

The 'dynamicsymbol' printing is not very clear above; we will also introduce a few other tools here. We can use `vprint` instead of print for non-interactive sessions.

```
>>> q1
q1(t)
>>> q1d = diff(q1, Symbol('t'))
>>> vprint(q1)
q1
>>> vprint(q1d)
q1'
```

For interactive sessions use `init_vprinting`. There also exist analogs for SymPy's `vprint`, `vpprint`, and `latex`, `vlatex`.

```
>>> from sympy.physics.vector import init_vprinting
>>> init_vprinting(pretty_print=False)
>>> q1
q1
>>> q1d
q1'
```

A 'dynamicsymbol' should be used to represent any time varying quantity in *sympy.physics.vector* (page 1592), whether it is a coordinate, varying position, or force. The primary use of a 'dynamicsymbol' is for speeds and coordinates (of which there will be more discussion in the Kinematics Section of the documentation).

Now we will define the orientation of our new frames with a 'dynamicsymbol', and can take derivatives and time derivatives with ease. Some examples follow.

```
>>> N = ReferenceFrame('N')
>>> B = N.orientnew('B', 'Axis', [q1, N.x])
>>> (B.y*q2 + B.z).diff(q2, N)
B.y
>>> (B.y*q2 + B.z).dt(N)
(-q1' + q2')*B.y + q2*q1'*B.z
```

Note that the output vectors are kept in the same frames that they were provided in. This remains true for vectors with components made of basis vectors from multiple frames:

```
>>> (B.y*q2 + B.z + q2*N.x).diff(q2, N)
N.x + B.y
```


How Vectors are Coded

What follows is a short description of how vectors are defined by the code in *sympy.physics.vector* (page 1592). It is provided for those who want to learn more about how this part of *sympy.physics.vector* (page 1592) works, and does not need to be read to use this module; don't read it unless you want to learn how this module was implemented.

Every `Vector`'s main information is stored in the `args` attribute, which stores the three measure numbers for each basis vector in a frame, for every relevant frame. A vector does not exist in code until a `ReferenceFrame` is created. At this point, the `x`, `y`, and `z` attributes of the reference frame are immutable `Vector`'s which have measure numbers of [1,0,0], [0,1,0], and [0,0,1] associated with that `ReferenceFrame`. Once these vectors are accessible, new vectors can be created by doing algebraic operations with the basis vectors. A vector can have components from multiple frames though. That is why `args` is a list; it has as many elements in the list as there are unique `ReferenceFrames` in its components, i.e. if there are `A` and `B` frame basis vectors in our new vector, `args` is of length 2; if it has `A`, `B`, and `C` frame basis vector, `args` is of length three.

Each element in the `args` list is a 2-tuple; the first element is a SymPy `Matrix` (this is where the measure numbers for each set of basis vectors are stored) and the second element is a `ReferenceFrame` to associate those measure numbers with.

`ReferenceFrame` stores a few things. First, it stores the name you supply it on creation (`name` attribute). It also stores the direction cosine matrices, defined upon creation with the `orientnew` method, or calling the `orient` method after creation. The direction cosine matrices are represented by SymPy's `Matrix`, and are part of a dictionary where the keys are the

ReferenceFrame and the value the Matrix; these are set bi-directionally; in that when you orient A to N you are setting A's orientation dictionary to include N and its Matrix, but you are also setting N's orientation dictionary to include A and its Matrix (that DCM being the transpose of the other).

Vector: Kinematics

This document will give some mathematical background to describing a system's kinematics as well as how to represent the kinematics in *sympy.physics.vector* (page 1592).

Introduction to Kinematics

The first topic is rigid motion kinematics. A rigid body is an idealized representation of a physical object which has mass and rotational inertia. Rigid bodies are obviously not flexible. We can break down rigid body motion into translational motion, and rotational motion (when dealing with particles, we only have translational motion). Rotational motion can further be broken down into simple rotations and general rotations.

Translation of a rigid body is defined as a motion where the orientation of the body does not change during the motion; or during the motion any line segment would be parallel to itself at the start of the motion.

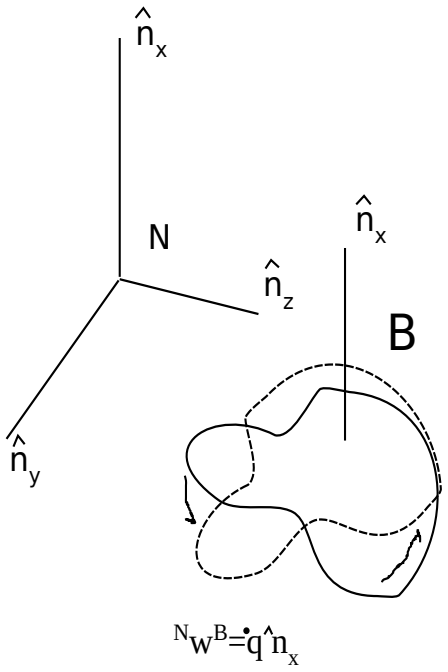
Simple rotations are rotations in which the orientation of the body may change, but there is always one line which remains parallel to itself at the start of the motion.

General rotations are rotations which there is not always one line parallel to itself at the start of the motion.

Angular Velocity

The angular velocity of a rigid body refers to the rate of change of its orientation. The angular velocity of a body is written down as: ${}^{\mathbf{N}}\omega^{\mathbf{B}}$, or the angular velocity of **B** in **N**, which is a vector. Note that here, the term rigid body was used, but reference frames can also have angular velocities. Further discussion of the distinction between a rigid body and a reference frame will occur later when describing the code representation.

Angular velocity is defined as being positive in the direction which causes the orientation angles to increase (for simple rotations, or series of simple rotations).



The angular velocity vector represents the time derivative of the orientation. As a time derivative vector quantity, like those covered in the Vector & ReferenceFrame documentation, this quantity (angular velocity) needs to be defined in a reference frame. That is what the **N** is in the above definition of angular velocity; the frame in which the angular velocity is defined in.

The angular velocity of **B** in **N** can also be defined by:

$$^{\mathbf{N}}\omega^{\mathbf{B}} = (\frac{^{\mathbf{N}}d\hat{\mathbf{b}}_{\mathbf{y}}}{dt} \cdot \hat{\mathbf{b}}_{\mathbf{z}})\hat{\mathbf{b}}_{\mathbf{x}} + (\frac{^{\mathbf{N}}d\hat{\mathbf{b}}_{\mathbf{z}}}{dt} \cdot \hat{\mathbf{b}}_{\mathbf{x}})\hat{\mathbf{b}}_{\mathbf{y}} + (\frac{^{\mathbf{N}}d\hat{\mathbf{b}}_{\mathbf{x}}}{dt} \cdot \hat{\mathbf{b}}_{\mathbf{y}})\hat{\mathbf{b}}_{\mathbf{z}}$$

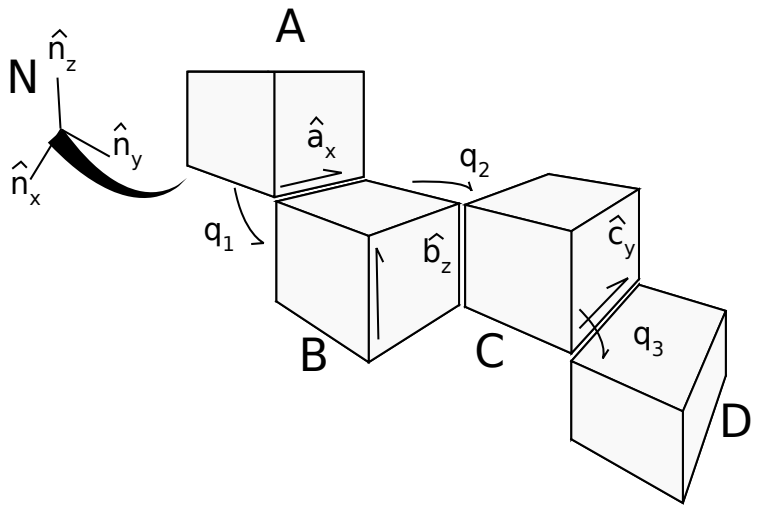
It is also common for a body's angular velocity to be written as:

$$^{\mathbf{N}}\omega^{\mathbf{B}} = w_x\hat{\mathbf{b}}_{\mathbf{x}} + w_y\hat{\mathbf{b}}_{\mathbf{y}} + w_z\hat{\mathbf{b}}_{\mathbf{z}}$$

There are a few additional important points relating to angular velocity. The first is the addition theorem for angular velocities, a way of relating the angular velocities of multiple bodies and frames. The theorem follows:

$$^{\mathbf{N}}\omega^{\mathbf{D}} = ^{\mathbf{N}}\omega^{\mathbf{A}} + ^{\mathbf{A}}\omega^{\mathbf{B}} + ^{\mathbf{B}}\omega^{\mathbf{C}} + ^{\mathbf{C}}\omega^{\mathbf{D}}$$

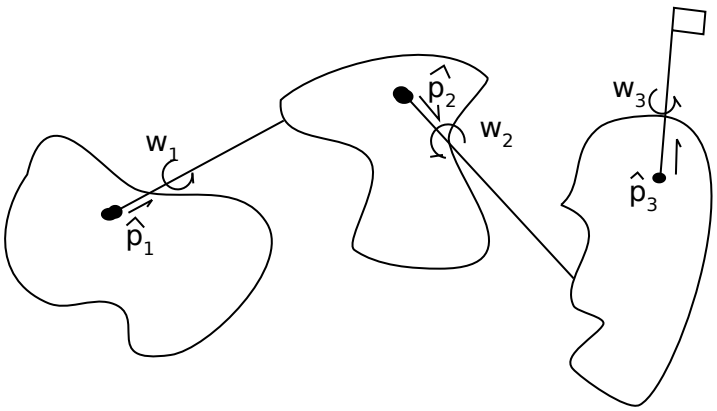
This is also shown in the following example:



$$
\begin{aligned}
{}^{\mathbf{N}}\omega^{\mathbf{A}} &= 0 \\
{}^{\mathbf{A}}\omega^{\mathbf{B}} &= \dot{q}_1 \hat{\mathbf{a}}_{\mathbf{x}} \\
{}^{\mathbf{B}}\omega^{\mathbf{C}} &= -\dot{q}_2 \hat{\mathbf{b}}_{\mathbf{z}} \\
{}^{\mathbf{C}}\omega^{\mathbf{D}} &= \dot{q}_3 \hat{\mathbf{c}}_{\mathbf{y}} \\
{}^{\mathbf{N}}\omega^{\mathbf{D}} &= \dot{q}_1 \hat{\mathbf{a}}_{\mathbf{x}} - \dot{q}_2 \hat{\mathbf{b}}_{\mathbf{z}} + \dot{q}_3 \hat{\mathbf{c}}_{\mathbf{y}}
\end{aligned}
$$

Note the signs used in the angular velocity definitions, which are related to how the displacement angle is defined in this case.

This theorem makes defining angular velocities of multibody systems much easier, as the angular velocity of a body in a chain needs to only be defined to the previous body in order to be fully defined (and the first body needs to be defined in the desired reference frame). The following figure shows an example of when using this theorem can make things easier.



Here we can easily write the angular velocity of the body **D** in the reference frame of the first body **A**:

$$
{}^{\mathbf{A}}\omega^{\mathbf{D}} = w_1 \hat{\mathbf{p}}_{\mathbf{1}} + w_2 \hat{\mathbf{p}}_{\mathbf{2}} + w_3 \hat{\mathbf{p}}_{\mathbf{3}}
$$

It is very important to remember to only use this with angular velocities; you cannot use this theorem with the velocities of points.

There is another theorem commonly used: the derivative theorem. It provides an alternative method (which can be easier) to calculate the time derivative of a vector in a reference frame:

$$\frac{^{\mathbf{N}}d\mathbf{v}}{dt} = \frac{^{\mathbf{B}}d\mathbf{v}}{dt} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{v}$$

The vector **v** can be any vector quantity: a position vector, a velocity vector, angular velocity vector, etc. Instead of taking the time derivative of the vector in **N**, we take it in **B**, where **B** can be any reference frame or body, usually one in which it is easy to take the derivative on **v** in (**v** is usually composed only of the basis vector set belonging to **B**). Then we add the cross product of the angular velocity of our newer frame, ${}^{\mathbf{N}}\omega^{\mathbf{B}}$ and our vector quantity **v**. Again, you can choose any alternative frame for this. Examples follow:

Angular Acceleration

Angular acceleration refers to the time rate of change of the angular velocity vector. Just as the angular velocity vector is for a body and is specified in a frame, the angular acceleration vector is for a body and is specified in a frame: ${}^{\mathbf{N}}\alpha^{\mathbf{B}}$, or the angular acceleration of **B** in **N**, which is a vector.

Calculating the angular acceleration is relatively straight forward:

$$^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{^{\mathbf{N}}d^{\mathbf{N}}\omega^{\mathbf{B}}}{dt}$$

Note that this can be calculated with the derivative theorem, and when the angular velocity is defined in a body fixed frame, becomes quite simple:

$$^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{^{\mathbf{N}}d^{\mathbf{N}}\omega^{\mathbf{B}}}{dt}$$

$$^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{^{\mathbf{B}}d^{\mathbf{N}}\omega^{\mathbf{B}}}{dt} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times {}^{\mathbf{N}}\omega^{\mathbf{B}}$$

$$\text{if } {}^{\mathbf{N}}\omega^{\mathbf{B}} = w_x\hat{\mathbf{b}}_{\mathbf{x}} + w_y\hat{\mathbf{b}}_{\mathbf{y}} + w_z\hat{\mathbf{b}}_{\mathbf{z}}$$

$$\text{then } {}^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{^{\mathbf{B}}d^{\mathbf{N}}\omega^{\mathbf{B}}}{dt} + \underbrace{{}^{\mathbf{N}}\omega^{\mathbf{B}} \times {}^{\mathbf{N}}\omega^{\mathbf{B}}}_{\text{this is 0 by definition}}$$

$$^{\mathbf{N}}\alpha^{\mathbf{B}} = \frac{dw_x}{dt}\hat{\mathbf{b}}_{\mathbf{x}} + \frac{dw_y}{dt}\hat{\mathbf{b}}_{\mathbf{y}} + \frac{dw_z}{dt}\hat{\mathbf{b}}_{\mathbf{z}}$$

$$^{\mathbf{N}}\alpha^{\mathbf{B}} = \dot{w}_x\hat{\mathbf{b}}_{\mathbf{x}} + \dot{w}_y\hat{\mathbf{b}}_{\mathbf{y}} + \dot{w}_z\hat{\mathbf{b}}_{\mathbf{z}}$$

Again, this is only for the case in which the angular velocity of the body is defined in body fixed components.

Point Velocity & Acceleration

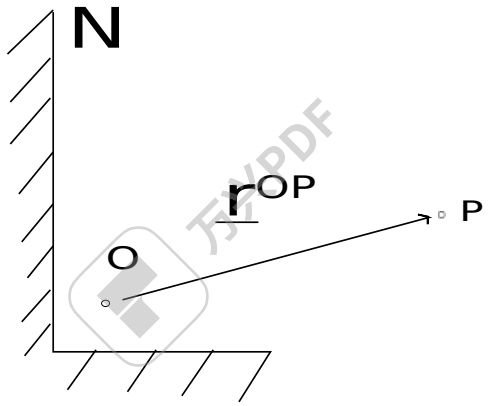
Consider a point, P: we can define some characteristics of the point. First, we can define a position vector from some other point to P. Second, we can define the velocity vector of P in a reference frame of our choice. Third, we can define the acceleration vector of P in a reference frame of our choice.

These three quantities are read as:

$$\mathbf{r}^{OP}\text{, the position vector from } O \text{ to } P$$
$$^{\mathbf{N}}\mathbf{v}^{P}\text{, the velocity of } P \text{ in the reference frame } \mathbf{N}$$
$$^{\mathbf{N}}\mathbf{a}^{P}\text{, the acceleration of } P \text{ in the reference frame } \mathbf{N}$$

Note that the position vector does not have a frame associated with it; this is because there is no time derivative involved, unlike the velocity and acceleration vectors.

We can find these quantities for a simple example easily:

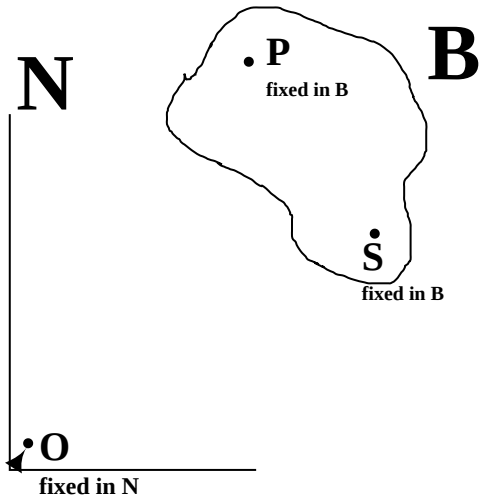


$$\text{Let's define: } \mathbf{r}^{OP} = q_x\hat{\mathbf{n}}_\mathbf{x} + q_y\hat{\mathbf{n}}_\mathbf{y}$$
$$^{\mathbf{N}}\mathbf{v}^{P} = \frac{^{\mathbf{N}}d\mathbf{r}^{OP}}{dt}$$
$$\text{then we can calculate: } ^{\mathbf{N}}\mathbf{v}^{P} = \dot{q}_x\hat{\mathbf{n}}_\mathbf{x} + \dot{q}_y\hat{\mathbf{n}}_\mathbf{y}$$
$$\text{and : } ^{\mathbf{N}}\mathbf{a}^{P} = \frac{^{\mathbf{N}}d\,^{\mathbf{N}}\mathbf{v}^{P}}{dt}$$
$$^{\mathbf{N}}\mathbf{a}^{P} = \ddot{q}_x\hat{\mathbf{n}}_\mathbf{x} + \ddot{q}_y\hat{\mathbf{n}}_\mathbf{y}$$

It is critical to understand in the above example that the point O is fixed in the reference frame $\mathbf{N}$. There is no addition theorem for translational velocities; alternatives will be discussed later though. Also note that the position of every point might not always need to be defined to form the dynamic equations of motion. When you don't want to define the position vector of a point, you can start by just defining the velocity vector. For the above example:

$$\text{Let us instead define the velocity vector as: } ^{\mathbf{N}}\mathbf{v}^{P} = u_x\hat{\mathbf{n}}_\mathbf{x} + u_y\hat{\mathbf{n}}_\mathbf{y}$$
$$\text{then acceleration can be written as: } ^{\mathbf{N}}\mathbf{a}^{P} = \dot{u}_x\hat{\mathbf{n}}_\mathbf{x} + \dot{u}_y\hat{\mathbf{n}}_\mathbf{y}$$

There will often be cases when the velocity of a point is desired and a related point's velocity is known. For the cases in which we have two points fixed on a rigid body, we use the 2-Point Theorem:

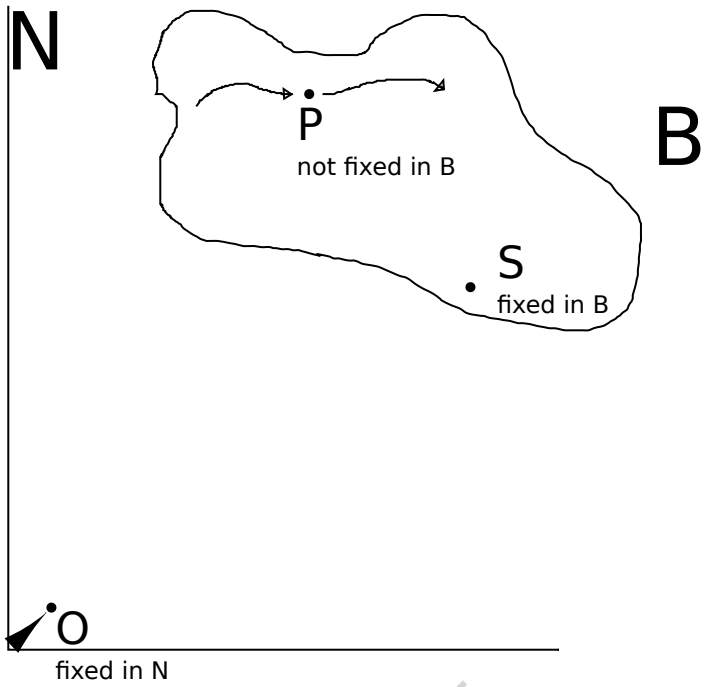


Let's say we know the velocity of the point S and the angular velocity of the body **B**, both defined in the reference frame **N**. We can calculate the velocity and acceleration of the point P in **N** as follows:

$$^{\mathbf{N}}\mathbf{v}^{P} = {}^{\mathbf{N}}\mathbf{v}^{S} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{r}^{SP}$$

$$^{\mathbf{N}}\mathbf{a}^{P} = {}^{\mathbf{N}}\mathbf{a}^{S} + {}^{\mathbf{N}}\alpha^{\mathbf{B}} \times \mathbf{r}^{SP} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times ({}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{r}^{SP})$$

When only one of the two points is fixed on a body, the 1 point theorem is used instead.

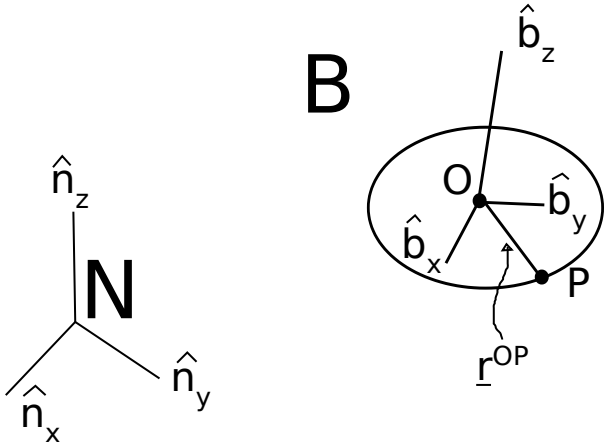


Here, the velocity of point S is known in the frame **N**, the angular velocity of **B** is known in **N**, and the velocity of the point P is known in the frame associated with body **B**. We can then write the velocity and acceleration of P in **N** as:

$$^{\mathbf{N}}\mathbf{v}^{P} = {}^{\mathbf{B}}\mathbf{v}^{P} + {}^{\mathbf{N}}\mathbf{v}^{S} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{r}^{SP}$$

$$^{\mathbf{N}}\mathbf{a}^{P} = {}^{\mathbf{B}}\mathbf{a}^{S} + {}^{\mathbf{N}}\mathbf{a}^{O} + {}^{\mathbf{N}}\alpha^{\mathbf{B}} \times \mathbf{r}^{SP} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times ({}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{r}^{SP}) + 2{}^{\mathbf{N}}\omega^{\mathbf{B}} \times {}^{\mathbf{B}}\mathbf{v}^{P}$$

Examples of applications of the 1 point and 2 point theorem follow.



This example has a disc translating and rotating in a plane. We can easily define the angular velocity of the body **B** and velocity of the point O:

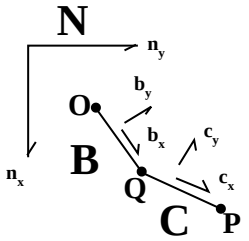
$$
\begin{aligned}
{}^{\mathbf{N}}\omega^{\mathbf{B}} &= u_3 \hat{\mathbf{n}}_{\mathbf{z}} = u_3 \hat{\mathbf{b}}_{\mathbf{z}} \\
{}^{\mathbf{N}}\mathbf{v}^{O} &= u_1 \hat{\mathbf{n}}_{\mathbf{x}} + u_2 \hat{\mathbf{n}}_{\mathbf{y}}
\end{aligned}
$$

and accelerations can be written as:

$$
\begin{aligned}
{}^{\mathbf{N}}\alpha^{\mathbf{B}} &= \dot{u}_3 \hat{\mathbf{n}}_{\mathbf{z}} = \dot{u}_3 \hat{\mathbf{b}}_{\mathbf{z}} \\
{}^{\mathbf{N}}\mathbf{a}^{O} &= \dot{u}_1 \hat{\mathbf{n}}_{\mathbf{x}} + \dot{u}_2 \hat{\mathbf{n}}_{\mathbf{y}}
\end{aligned}
$$

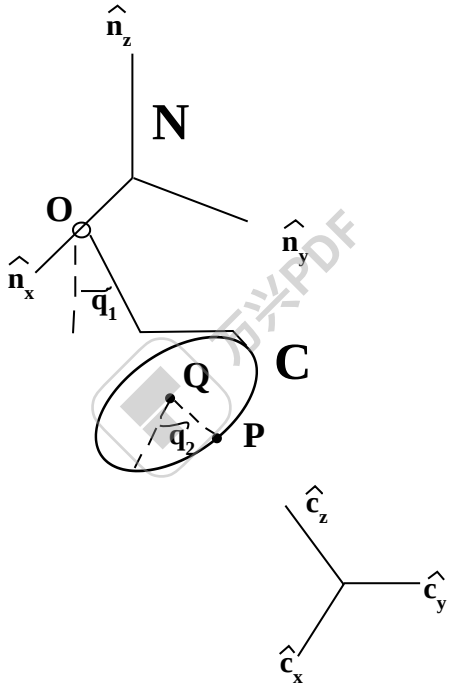
We can use the 2 point theorem to calculate the velocity and acceleration of point P now.

$$
\begin{aligned}
\mathbf{r}^{OP} &= R\hat{\mathbf{b}}_{\mathbf{x}} \\
{}^{\mathbf{N}}\mathbf{v}^{P} &= {}^{\mathbf{N}}\mathbf{v}^{O} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{r}^{OP} \\
{}^{\mathbf{N}}\mathbf{v}^{P} &= u_1 \hat{\mathbf{n}}_{\mathbf{x}} + u_2 \hat{\mathbf{n}}_{\mathbf{y}} + u_3 \hat{\mathbf{b}}_{\mathbf{z}} \times R\hat{\mathbf{b}}_{\mathbf{x}} = u_1 \hat{\mathbf{n}}_{\mathbf{x}} + u_2 \hat{\mathbf{n}}_{\mathbf{y}} + u_3 R\hat{\mathbf{b}}_{\mathbf{y}} \\
{}^{\mathbf{N}}\mathbf{a}^{P} &= {}^{\mathbf{N}}\mathbf{a}^{O} + {}^{\mathbf{N}}\alpha^{\mathbf{B}} \times \mathbf{r}^{OP} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times ({}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{r}^{OP}) \\
{}^{\mathbf{N}}\mathbf{a}^{P} &= \dot{u}_1 \hat{\mathbf{n}}_{\mathbf{x}} + \dot{u}_2 \hat{\mathbf{n}}_{\mathbf{y}} + \dot{u}_3 \hat{\mathbf{b}}_{\mathbf{z}} \times R\hat{\mathbf{b}}_{\mathbf{x}} + u_3 \hat{\mathbf{b}}_{\mathbf{z}} \times (u_3 \hat{\mathbf{b}}_{\mathbf{z}} \times R\hat{\mathbf{b}}_{\mathbf{x}}) \\
{}^{\mathbf{N}}\mathbf{a}^{P} &= \dot{u}_1 \hat{\mathbf{n}}_{\mathbf{x}} + \dot{u}_2 \hat{\mathbf{n}}_{\mathbf{y}} + R\dot{u}_3 \hat{\mathbf{b}}_{\mathbf{y}} - Ru_3^2 \hat{\mathbf{b}}_{\mathbf{x}}
\end{aligned}
$$



In this example we have a double pendulum. We can use the two point theorem twice here in order to find the velocity of points Q and P; point O's velocity is zero in $\mathbf{N}$.

$$\mathbf{r}^{OQ} = l\hat{\mathbf{b}}_{\mathbf{x}}$$
$$\mathbf{r}^{QP} = l\hat{\mathbf{c}}_{\mathbf{x}}$$
$${}^{\mathbf{N}}\omega^{\mathbf{B}} = u_1\hat{\mathbf{b}}_{\mathbf{z}}$$
$${}^{\mathbf{N}}\omega^{\mathbf{C}} = u_2\hat{\mathbf{c}}_{\mathbf{z}}$$
$${}^{\mathbf{N}}\mathbf{v}^{Q} = {}^{\mathbf{N}}\mathbf{v}^{O} + {}^{\mathbf{N}}\omega^{\mathbf{B}} \times \mathbf{r}^{OQ}$$
$${}^{\mathbf{N}}\mathbf{v}^{Q} = u_1 l\hat{\mathbf{b}}_{\mathbf{y}}$$
$${}^{\mathbf{N}}\mathbf{v}^{P} = {}^{\mathbf{N}}\mathbf{v}^{Q} + {}^{\mathbf{N}}\omega^{\mathbf{C}} \times \mathbf{r}^{QP}$$
$${}^{\mathbf{N}}\mathbf{v}^{Q} = u_1 l\hat{\mathbf{b}}_{\mathbf{y}} + u_2\hat{\mathbf{c}}_{\mathbf{z}} \times l\hat{\mathbf{c}}_{\mathbf{x}}$$
$${}^{\mathbf{N}}\mathbf{v}^{Q} = u_1 l\hat{\mathbf{b}}_{\mathbf{y}} + u_2 l\hat{\mathbf{c}}_{\mathbf{y}}$$

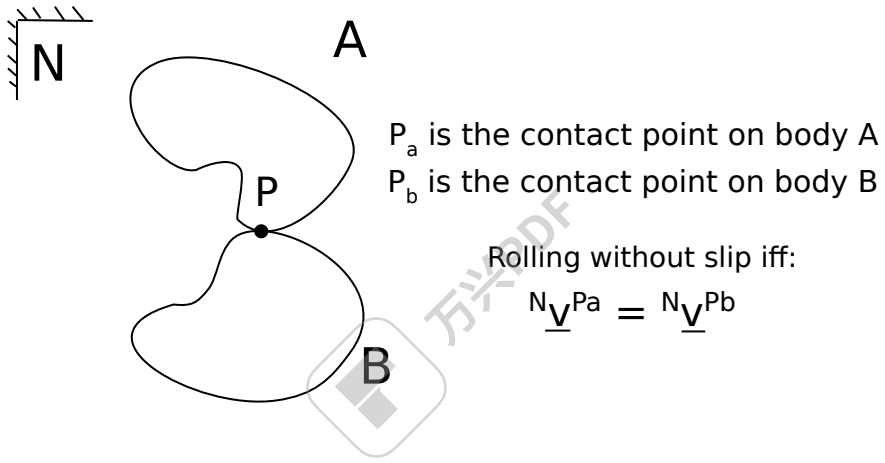


In this example we have a particle moving on a ring; the ring is supported by a rod which can rotate about the $\hat{\mathbf{n}}_{\mathbf{x}}$ axis. First we use the two point theorem to find the velocity of the center point of the ring, Q, then use the 1 point theorem to find the velocity of the particle on the

ring.

$$
\begin{aligned}
{}^{\mathbf{N}}\omega^{\mathbf{C}} &= u_1 \hat{\mathbf{n}}_{\mathbf{x}} \\
\mathbf{r}^{OQ} &= -l\hat{\mathbf{c}}_{\mathbf{z}} \\
{}^{\mathbf{N}}\mathbf{v}^{Q} &= u_1 l\hat{\mathbf{c}}_{\mathbf{y}} \\
\mathbf{r}^{QP} &= R(cos(q_2)\hat{\mathbf{c}}_{\mathbf{x}} + sin(q_2)\hat{\mathbf{c}}_{\mathbf{y}}) \\
{}^{\mathbf{C}}\mathbf{v}^{P} &= Ru_2(-sin(q_2)\hat{\mathbf{c}}_{\mathbf{x}} + cos(q_2)\hat{\mathbf{c}}_{\mathbf{y}}) \\
{}^{\mathbf{N}}\mathbf{v}^{P} &= {}^{\mathbf{C}}\mathbf{v}^{P} + {}^{\mathbf{N}}\mathbf{v}^{Q} + {}^{\mathbf{N}}\omega^{\mathbf{C}} \times \mathbf{r}^{QP} \\
{}^{\mathbf{N}}\mathbf{v}^{P} &= Ru_2(-sin(q_2)\hat{\mathbf{c}}_{\mathbf{x}} + cos(q_2)\hat{\mathbf{c}}_{\mathbf{y}}) + u_1 l\hat{\mathbf{c}}_{\mathbf{y}} + u_1\hat{\mathbf{c}}_{\mathbf{x}} \times R(cos(q_2)\hat{\mathbf{c}}_{\mathbf{x}} + sin(q_2)\hat{\mathbf{c}}_{\mathbf{y}} \\
{}^{\mathbf{N}}\mathbf{v}^{P} &= -Ru_2 sin(q_2)\hat{\mathbf{c}}_{\mathbf{x}} + (Ru_2 cos(q_2) + u_1 l)\hat{\mathbf{c}}_{\mathbf{y}} + Ru_1 sin(q_2)\hat{\mathbf{c}}_{\mathbf{z}}
\end{aligned}
$$

A final topic in the description of velocities of points is that of rolling, or rather, rolling without slip. Two bodies are said to be rolling without slip if and only if the point of contact on each body has the same velocity in another frame. See the following figure:



This is commonly used to form the velocity of a point on one object rolling on another fixed object, such as in the following example:

Kinematics in physics.vector

It should be clear by now that the topic of kinematics here has been mostly describing the correct way to manipulate vectors into representing the velocities of points. Within *sympy.physics.vector* (page 1592) there are convenient methods for storing these velocities associated with frames and points. We'll now revisit the above examples and show how to represent them in `sympy`.

The topic of reference frame creation has already been covered. When a `ReferenceFrame` is created though, it automatically calculates the angular velocity of the frame using the time derivative of the DCM and the angular velocity definition.

```
>>> from sympy import Symbol, sin, cos
>>> from sympy.physics.vector import *
>>> init_vprinting(pretty_print=False)
>>> N = ReferenceFrame('N')
```

(continues on next page)

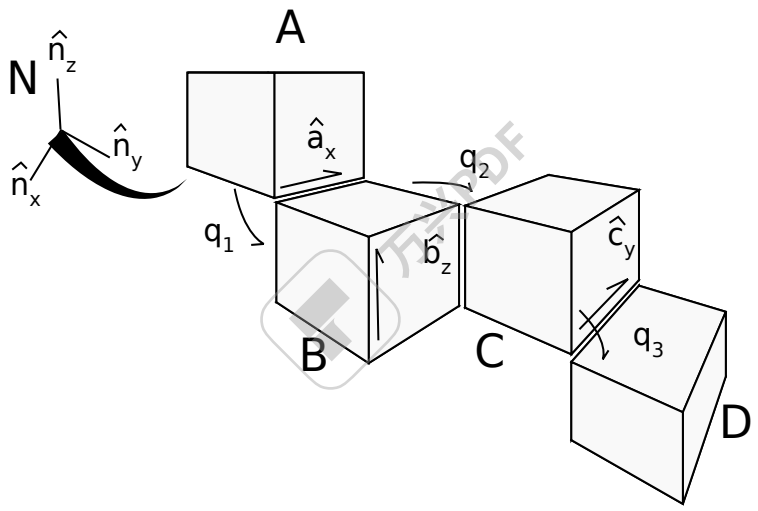
(continued from previous page)

```
>>> q1 = dynamicsymbols('q1')
>>> A = N.orientnew('A', 'Axis', [q1, N.x])
>>> A.ang_vel_in(N)
q1'*N.x
```

Note that the angular velocity can be defined in an alternate way:

```
>>> B = ReferenceFrame('B')
>>> u1 = dynamicsymbols('u1')
>>> B.set_ang_vel(N, u1 * B.y)
>>> B.ang_vel_in(N)
u1*B.y
>>> N.ang_vel_in(B)
- u1*B.y
```

Both upon frame creation during `orientnew` and when calling `set_ang_vel`, the angular velocity is set in both frames involved, as seen above.



Here we have multiple bodies with angular velocities defined relative to each other. This is coded as:

```
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> B = ReferenceFrame('B')
>>> C = ReferenceFrame('C')
>>> D = ReferenceFrame('D')
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> A.set_ang_vel(N, 0)
>>> B.set_ang_vel(A, u1 * A.x)
>>> C.set_ang_vel(B, -u2 * B.z)
>>> D.set_ang_vel(C, u3 * C.y)
>>> D.ang_vel_in(N)
u1*A.x - u2*B.z + u3*C.y
```

In *sympy.physics.vector* (page 1592) the shortest path between two frames is used when finding the angular velocity. That would mean if we went back and set:

```
>>> D.set_ang_vel(N, 0)
>>> D.ang_vel_in(N)
0
```

The path that was just defined is what is used. This can cause problems though, as now the angular velocity definitions are inconsistent. It is recommended that you avoid doing this.

Points are a translational analog to the rotational `ReferenceFrame`. Creating a `Point` can be done in two ways, like `ReferenceFrame`:

```
>>> O = Point('O')
>>> P = O.locatenew('P', 3 * N.x + N.y)
>>> P.pos_from(O)
3*N.x + N.y
>>> Q = Point('Q')
>>> Q.set_pos(P, N.z)
>>> Q.pos_from(P)
N.z
>>> Q.pos_from(O)
3*N.x + N.y + N.z
```

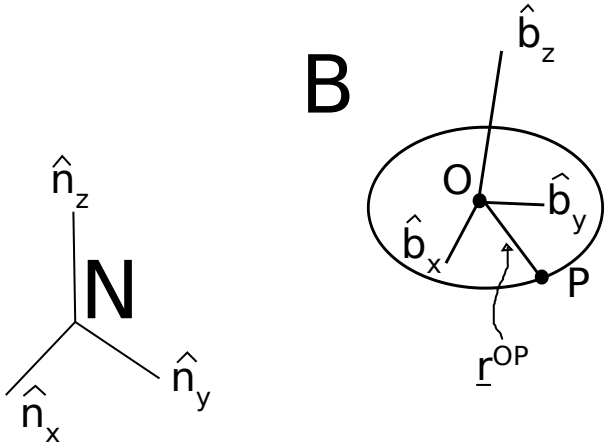
Similar to `ReferenceFrame`, the position vector between two points is found by the shortest path (number of intermediate points) between them. Unlike rotational motion, there is no addition theorem for the velocity of points. In order to have the velocity of a `Point` in a `ReferenceFrame`, you have to set the value.

```
>>> O = Point('O')
>>> O.set_vel(N, u1*N.x)
>>> O.vel(N)
u1*N.x
```

For both translational and rotational accelerations, the value is computed by taking the time derivative of the appropriate velocity, unless the user sets it otherwise.

```
>>> O.acc(N)
u1'*N.x
>>> O.set_acc(N, u2*u1*N.y)
>>> O.acc(N)
u1*u2*N.y
```

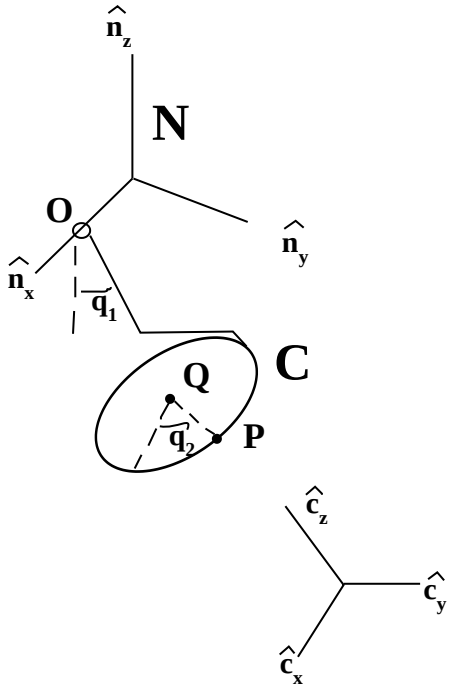
Next is a description of the 2 point and 1 point theorems, as used in `sympy`.



First is the translating, rotating disc.

```
>>> N = ReferenceFrame('N')
>>> u1, u2, u3 = dynamicsymbols('u1 u2 u3')
>>> R = Symbol('R')
>>> B = ReferenceFrame('B')
>>> O = Point('O')
>>> O.set_vel(N, u1 * N.x + u2 * N.y)
>>> P = O.locatenew('P', R * B.x)
>>> B.set_ang_vel(N, u3 * B.z)
>>> P.v2pt_theory(O, N, B)
u1*N.x + u2*N.y + R*u3*B.y
>>> P.a2pt_theory(O, N, B)
u1'*N.x + u2'*N.y - R*u3**2*B.x + R*u3'*B.y
```

We will also cover implementation of the 1 point theorem.



This is the particle moving on a ring, again.

```
>>> N = ReferenceFrame('N')
>>> u1, u2 = dynamicsymbols('u1 u2')
>>> q1, q2 = dynamicsymbols('q1 q2')
>>> l = Symbol('l')
>>> R = Symbol('R')
>>> C = N.orientnew('C', 'Axis', [q1, N.x])
>>> C.set_ang_vel(N, u1 * N.x)
>>> O = Point('O')
>>> O.set_vel(N, 0)
>>> Q = O.locatenew('Q', -l * C.z)
>>> P = Q.locatenew('P', R * (cos(q2) * C.x + sin(q2) * C.y))
>>> P.set_vel(C, R * u2 * (-sin(q2) * C.x + cos(q2) * C.y))
>>> Q.v2pt_theory(O, N, C)
l*u1*C.y
>>> P.v1pt_theory(Q, N, C)
- R*u2*sin(q2)*C.x + (R*u2*cos(q2) + l*u1)*C.y + R*u1*sin(q2)*C.z
```

Potential Issues/Advanced Topics/Future Features in Physics/Vector Module

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

Inertia (Dyadics)

A dyadic tensor is a second order tensor formed by the juxtaposition of a pair of vectors. There are various operations defined with respect to dyadics, which have been implemented in *vector* (page 1592) in the form of class *sympy.physics.vector.dyadic.Dyadic* (page 1648). To know more, refer to the *sympy.physics.vector.dyadic.Dyadic* (page 1648) and *sympy.physics.vector.vector.Vector* (page 1641) class APIs. Dyadics are used to define the inertia of bodies within *sympy.physics.mechanics* (page 1675). Inertia dyadics can be defined explicitly but the `inertia` function is typically much more convenient for the user:

```
>>> from sympy.physics.mechanics import ReferenceFrame, inertia
>>> N = ReferenceFrame('N')

Supply a reference frame and the moments of inertia if the object
is symmetrical:

>>> inertia(N, 1, 2, 3)
(N.x|N.x) + 2*(N.y|N.y) + 3*(N.z|N.z)

Supply a reference frame along with the products and moments of inertia
for a general object:

>>> inertia(N, 1, 2, 3, 4, 5, 6)
(N.x|N.x) + 4*(N.x|N.y) + 6*(N.x|N.z) + 4*(N.y|N.x) + 2*(N.y|N.y) + 5*(N.y|N.
↪z) + 6*(N.z|N.x) + 5*(N.z|N.y) + 3*(N.z|N.z)
```

Notice that the `inertia` function returns a dyadic with each component represented as two unit vectors separated by a `|`. Refer to the *sympy.physics.vector.dyadic.Dyadic* (page 1648) section for more information about dyadics.

Inertia is often expressed in a matrix, or tensor, form, especially for numerical purposes. Since the matrix form does not contain any information about the reference frame(s) the inertia dyadic is defined in, you must provide one or two reference frames to extract the measure numbers from the dyadic. There is a convenience function to do this:

```
>>> inertia(N, 1, 2, 3, 4, 5, 6).to_matrix(N)
Matrix([
[1, 4, 6],
[4, 2, 5],
[6, 5, 3]])
```

Common Issues

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

Printing

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

Substitution

Substitution into large expressions can be slow, and take a few minutes.

Acceleration of Points

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

Advanced Interfaces

Here we will cover advanced options in: `ReferenceFrame`, `dynamicsymbols`, and some associated functionality.

ReferenceFrame

`ReferenceFrame` is shown as having a `.name` attribute and `.x`, `.y`, and `.z` attributes for accessing the basis vectors, as well as a fairly rigidly defined print output. If you wish to have a different set of indices defined, there is an option for this. This will also require a different interface for accessing the basis vectors.

```
>>> from sympy.physics.vector import ReferenceFrame, vprint, vpprint, vlatex
>>> N = ReferenceFrame('N', indices=['i', 'j', 'k'])
>>> N['i']
N['i']
>>> N.x
N['i']
>>> vlatex(N.x)
'\\mathbf{\\hat{n}_{i}}'
```

Also, the latex output can have custom strings; rather than just indices though, the entirety of each basis vector can be specified. The custom latex strings can occur without custom indices, and also overwrites the latex string that would be used if there were custom indices.

```
>>> from sympy.physics.vector import ReferenceFrame, vlatex
>>> N = ReferenceFrame('N', latexs=['n1','\\mathbf{n}_2','cat'])
>>> vlatex(N.x)
'n1'
>>> vlatex(N.y)
'\\mathbf{n}_2'
>>> vlatex(N.z)
'cat'
```

dynamicsymbols

The `dynamicsymbols` function also has 'hidden' functionality; the variable which is associated with time can be changed, as well as the notation for printing derivatives.

```
>>> from sympy import symbols
>>> from sympy.physics.vector import dynamicsymbols, vprint
>>> q1 = dynamicsymbols('q1')
>>> q1
q1(t)
>>> dynamicsymbols._t = symbols('T')
>>> q2 = dynamicsymbols('q2')
>>> q2
q2(T)
>>> q1
q1(t)
>>> q1d = dynamicsymbols('q1', 1)
>>> vprint(q1d)
q1'
>>> dynamicsymbols._str = 'd'
>>> vprint(q1d)
q1d
>>> dynamicsymbols._str = '\''
>>> dynamicsymbols._t = symbols('t')
```

Note that only dynamic symbols created after the change are different. The same is not true for the $._str$ attribute; this affects the printing output only, so dynamic symbols created before or after will print the same way.

Also note that `Vector`'s `.dt` method uses the `._t` attribute of `dynamicsymbols`, along with a number of other important functions and methods. Don't mix and match symbols representing time.

Scalar and Vector Field Functionality

Introduction

Vectors and Scalars

In physics, we deal with two kinds of quantities – scalars and vectors.

A scalar is an entity which only has a magnitude – no direction. Examples of scalar quantities include mass, electric charge, temperature, distance, etc.

A vector, on the other hand, is an entity that is characterized by a magnitude and a direction. Examples of vector quantities are displacement, velocity, magnetic field, etc.

A scalar can be depicted just by a number, for e.g. a temperature of 300 K. On the other hand, vectorial quantities like acceleration are usually denoted by a vector. Given a vector $\mathbf{V}$, the magnitude of the corresponding quantity can be calculated as the magnitude of the vector itself $\|\mathbf{V}\|$, while the direction would be specified by a unit vector in the direction of the original vector, $\hat{\mathbf{V}} = \frac{\mathbf{V}}{\|\mathbf{V}\|}$.

For example, consider a displacement of $(3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}})$ m, where , as per standard convention, $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$ and $\hat{\mathbf{k}}$ represent unit vectors in the $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$ directions respectively. Therefore, it can be concluded that the distance traveled is $\|3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}}\|$ m = $5\sqrt{2}$ m. The direction of travel is given by the unit vector $\frac{3}{5\sqrt{2}}\hat{\mathbf{i}} + \frac{4}{5\sqrt{2}}\hat{\mathbf{j}} + \frac{5}{5\sqrt{2}}\hat{\mathbf{k}}$.

Fields

In general, a $field$ is a vector or scalar quantity that can be specified everywhere in space as a function of position (Note that in general a field may also be dependent on time and other custom variables). In this module, we deal with 3-dimensional spaces only. Hence, a field is defined as a function of the x, y and z coordinates corresponding to a location in 3D space.

For example, temperate in 3 dimensional space (a temperature field) can be written as $T(x, y, z)$ – a scalar function of the position. An example of a scalar field in electromagnetism is the electric potential.

In a similar manner, a vector field can be defined as a vectorial function of the location (x, y, z) of any point in space.

For instance, every point on the earth may be considered to be in the gravitational force field of the earth. We may specify the field by the magnitude and the direction of acceleration due to gravity (i.e. force per unit mass) $g(x, y, z)$ at every point in space.

To give an example from electromagnetism, consider an electric potential of form $2x^2y$, a scalar field in 3D space. The corresponding conservative electric field can be computed as the gradient of the electric potential function, and expressed as $4xy\hat{\mathbf{i}} + 2x^2\hat{\mathbf{j}}$. The magnitude of this electric field can in turn be expressed as a scalar field of the form $\sqrt{4x^4 + 16x^2y^2}$.

Implementation of fields in sympy.physics.vector

In *sympy.physics.vector* (page 1592), every *ReferenceFrame* (page 1629) instance is assigned basis vectors corresponding to the X, Y and Z directions. These can be accessed using the attributes named `x`, `y` and `z` respectively. Hence, to define a vector $\mathbf{v}$ of the form $3\hat{\mathbf{i}} + 4\hat{\mathbf{j}} + 5\hat{\mathbf{k}}$ with respect to a given frame $\mathbf{R}$, you would do

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> v = 3*R.x + 4*R.y + 5*R.z
```

Vector math and basic calculus operations with respect to vectors have already been elaborated upon in other sections of this module's documentation.

On the other hand, base scalars (or coordinate variables) are implemented as special SymPy *Symbol* (page 976)s assigned to every frame, one for each direction from X, Y and Z. For a frame `R`, the X, Y and Z base scalar *Symbol* (page 976)s can be accessed using the `R[0]`, `R[1]` and `R[2]` expressions respectively.

Therefore, to generate the expression for the aforementioned electric potential field $2x^2y$, you would have to do

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> electric_potential
2*R_x**2*R_y
```

In string representation, `R_x` denotes the X base scalar assigned to *ReferenceFrame* (page 1629) `R`. Essentially, `R_x` is the string representation of `R[0]`.

Scalar fields can be treated just as any other SymPy expression, for any math/calculus functionality. Hence, to differentiate the above electric potential with respect to x (i.e. `R[0]`), you would have to use the *diff* (page 1048) function.

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> from sympy import diff
>>> diff(electric_potential, R[0])
4*R_x*R_y
```

Like vectors (and vector fields), scalar fields can also be re-expressed in other frames of reference, apart from the one they were defined in – assuming that an orientation relationship exists between the concerned frames. This can be done using the *sympy.physics.vector.vector.Vector.express* (page 1644) method, in a way similar to vectors - but with the `variables` parameter set to `True`.

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> from sympy.physics.vector import dynamicsymbols, express
>>> q = dynamicsymbols('q')
>>> R1 = R.orientnew('R1', rot_type = 'Axis', amounts = [q, R.z])
>>> express(electric_potential, R1, variables=True)
2*(R1_x*sin(q(t)) + R1_y*cos(q(t)))*(R1_x*cos(q(t)) - R1_y*sin(q(t)))**2
```

Moreover, considering scalars can also be functions of time just as vectors, differentiation with respect to time is also possible. Depending on the *Symbol* (page 976)s present in the expression and the frame with respect to which the time differentiation is being done, the output will change/remain the same.

```
>>> from sympy.physics.vector import ReferenceFrame
>>> R = ReferenceFrame('R')
>>> electric_potential = 2*R[0]**2*R[1]
>>> q = dynamicsymbols('q')
>>> R1 = R.orientnew('R1', rot_type = 'Axis', amounts = [q, R.z])
>>> from sympy.physics.vector import time_derivative
>>> time_derivative(electric_potential, R)
0
>>> time_derivative(electric_potential, R1).simplify()
2*(R1_x*cos(q(t)) - R1_y*sin(q(t)))*(3*R1_x**2*cos(2*q(t))/2 -
R1_x**2/2 - 3*R1_x*R1_y*sin(2*q(t)) - 3*R1_y**2*cos(2*q(t))/2 -
R1_y**2/2)*Derivative(q(t), t)
```

Field operators and other related functions

Here we describe some basic field-related functionality implemented in sympy.physics.vector

Curl

A curl is a mathematical operator that describes an infinitesimal rotation of a vector in 3D space. The direction is determined by the right-hand rule (along the axis of rotation), and the magnitude is given by the magnitude of rotation.

In the 3D Cartesian system, the curl of a 3D vector $\mathbf{F}$, denoted by $\nabla \times \mathbf{F}$ is given by -

$\nabla \times \mathbf{F} = \left(\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z}\right)\hat{\mathbf{i}} + \left(\frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x}\right)\hat{\mathbf{j}} + \left(\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y}\right)\hat{\mathbf{k}}$

where F_x denotes the X component of vector $\mathbf{F}$.

To compute the curl of a vector field in *sympy.physics.vector* (page 1592), you would do

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

Divergence

Divergence is a vector operator that measures the magnitude of a vector field's source or sink at a given point, in terms of a signed scalar.

The divergence operator always returns a scalar after operating on a vector.

In the 3D Cartesian system, the divergence of a 3D vector $\mathbf{F}$, denoted by $\nabla \cdot \mathbf{F}$ is given by -

$\nabla \cdot \mathbf{F} = \frac{\partial U}{\partial x} + \frac{\partial V}{\partial y} + \frac{\partial W}{\partial z}$

where U, V and W denote the X, Y and Z components of $\mathbf{F}$ respectively.

To compute the divergence of a vector field in *sympy.physics.vector* (page 1592), you would do

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

Gradient

Consider a scalar field $f(x, y, z)$ in 3D space. The gradient of this field is defined as the vector of the 3 partial derivatives of f with respect to x, y and z in the X, Y and Z directions respectively.

In the 3D Cartesian system, the gradient of a scalar field f, denoted by ∇f is given by -

$\nabla f = \frac{\partial f}{\partial x}\mathbf{\hat{i}} + \frac{\partial f}{\partial y}\mathbf{\hat{j}} + \frac{\partial f}{\partial z}\mathbf{\hat{k}}$

To compute the gradient of a scalar field in *sympy.physics.vector* (page 1592), you would do

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

Conservative and Solenoidal fields

In vector calculus, a conservative field is a field that is the gradient of some scalar field. Conservative fields have the property that their line integral over any path depends only on the end-points, and is independent of the path between them. A conservative vector field is also said to be 'irrotational', since the curl of a conservative field is always zero.

In physics, conservative fields represent forces in physical systems where energy is conserved.

To check if a vector field is conservative in *sympy.physics.vector* (page 1592), use the *sympy.physics.vector.fieldfunctions.is_conservative* (page 1674) function.

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

A solenoidal field, on the other hand, is a vector field whose divergence is zero at all points in space.

To check if a vector field is solenoidal in *sympy.physics.vector* (page 1592), use the *sympy.physics.vector.fieldfunctions.is_solenoidal* (page 1674) function.

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

Scalar potential functions

We have previously mentioned that every conservative field can be defined as the gradient of some scalar field. This scalar field is also called the 'scalar potential field' corresponding to the aforementioned conservative field.

The *sympy.physics.vector.fieldfunctions.scalar_potential* (page 1672) function in *sympy.physics.vector* (page 1592) calculates the scalar potential field corresponding to a given conservative vector field in 3D space - minus the extra constant of integration, of course.

Example of usage -

```
>>> from sympy.physics.vector import ReferenceFrame, scalar_potential
>>> R = ReferenceFrame('R')
>>> conservative_field = 4*R[0]*R[1]*R[2]*R.x + 2*R[0]**2*R[2]*R.y +␣
↪2*R[0]**2*R[1]*R.z
>>> scalar_potential(conservative_field, R)
2*R_x**2*R_y*R_z
```

Providing a non-conservative vector field as an argument to *sympy.physics.vector.fieldfunctions.scalar_potential* (page 1672) raises a `ValueError`.

The scalar potential difference, or simply 'potential difference', corresponding to a conservative vector field can be defined as the difference between the values of its scalar potential function at two points in space. This is useful in calculating a line integral with respect to a conservative function, since it depends only on the endpoints of the path.

This computation is performed as follows in *sympy.physics.vector* (page 1592).

```
>>> from sympy.physics.vector import ReferenceFrame, Point
>>> from sympy.physics.vector import scalar_potential_difference
```

(continues on next page)

(continued from previous page)

```
>>> R = ReferenceFrame('R')
>>> O = Point('O')
>>> P = O.locatenew('P', 1*R.x + 2*R.y + 3*R.z)
>>> vectfield = 4*R[0]*R[1]*R.x + 2*R[0]**2*R.y
>>> scalar_potential_difference(vectfield, R, O, P, O)
4
```

If provided with a scalar expression instead of a vector field, *sympy.physics.vector.fieldfunctions.scalar_potential_difference* (page 1673) returns the difference between the values of that scalar field at the two given points in space.

Vector API

Essential Classes

CoordinateSym

class sympy.physics.vector.frame.**CoordinateSym**(*name*, *frame*, *index*)

A coordinate symbol/base scalar associated wrt a Reference Frame.

Ideally, users should not instantiate this class. Instances of this class must only be accessed through the corresponding frame as 'frame[index]'.

CoordinateSyms having the same frame and index parameters are equal (even though they may be instantiated separately).

Parameters

name : string

The display name of the CoordinateSym

frame : ReferenceFrame

The reference frame this base scalar belongs to

index : 0, 1 or 2

The index of the dimension denoted by this coordinate variable

Examples

```
>>> from sympy.physics.vector import ReferenceFrame, CoordinateSym
>>> A = ReferenceFrame('A')
>>> A[1]
A_y
>>> type(A[0])
<class 'sympy.physics.vector.frame.CoordinateSym'>
>>> a_y = CoordinateSym('a_y', A, 1)
>>> a_y == A[1]
True
```

ReferenceFrame

class sympy.physics.vector.frame.**ReferenceFrame**(*name*, *indices=None*, *latexs=None*, *variables=None*)

A reference frame in classical mechanics.

ReferenceFrame is a class used to represent a reference frame in classical mechanics. It has a standard basis of three unit vectors in the frame's x, y, and z directions.

It also can have a rotation relative to a parent frame; this rotation is defined by a direction cosine matrix relating this frame's basis vectors to the parent frame's basis vectors. It can also have an angular velocity vector, defined in another frame.

ang_acc_in(*otherframe*)

Returns the angular acceleration Vector of the ReferenceFrame.

Effectively returns the Vector:

`N_alpha_B`

which represent the angular acceleration of B in N, where B is self, and N is otherframe.

Parameters

otherframe : ReferenceFrame

The ReferenceFrame which the angular acceleration is returned in.

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
```

ang_vel_in(*otherframe*)

Returns the angular velocity Vector of the ReferenceFrame.

Effectively returns the Vector:

^N omega ^B

which represent the angular velocity of B in N, where B is self, and N is otherframe.

Parameters

otherframe : ReferenceFrame

The ReferenceFrame which the angular velocity is returned in.

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
```

dcm(*otherframe*)

Returns the direction cosine matrix of this reference frame relative to the provided reference frame.

The returned matrix can be used to express the orthogonal unit vectors of this frame in terms of the orthogonal unit vectors of `otherframe`.

Parameters

otherframe : ReferenceFrame

The reference frame which the direction cosine matrix of this frame is formed relative to.

Examples

The following example rotates the reference frame A relative to N by a simple rotation and then calculates the direction cosine matrix of N relative to A.

```
>>> from sympy import symbols, sin, cos
>>> from sympy.physics.vector import ReferenceFrame
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> A = ReferenceFrame('A')
>>> A.orient_axis(N, q1, N.x)
>>> N.dcm(A)
Matrix([
[1,       0,        0],
[0, cos(q1), -sin(q1)],
[0, sin(q1),  cos(q1)]])
```

The second row of the above direction cosine matrix represents the `N.y` unit vector in N expressed in A. Like so:

```
>>> Ny = 0*A.x + cos(q1)*A.y - sin(q1)*A.z
```

Thus, expressing `N.y` in A should return the same result:

```
>>> N.y.express(A)
cos(q1)*A.y - sin(q1)*A.z
```

Notes

It is important to know what form of the direction cosine matrix is returned. If `B.dcm(A)` is called, it means the "direction cosine matrix of B rotated relative to A". This is the matrix ${}^{B}\mathbf{C}^{A}$ shown in the following relationship:

$$\begin{bmatrix} \hat{\mathbf{b}}_1 \\ \hat{\mathbf{b}}_2 \\ \hat{\mathbf{b}}_3 \end{bmatrix} = {}^{B}\mathbf{C}^{A} \begin{bmatrix} \hat{\mathbf{a}}_1 \\ \hat{\mathbf{a}}_2 \\ \hat{\mathbf{a}}_3 \end{bmatrix}.$$

${}^{B}\mathbf{C}^{A}$ is the matrix that expresses the B unit vectors in terms of the A unit vectors.

orient(*parent*, *rot_type*, *amounts*, *rot_order*='')

Sets the orientation of this reference frame relative to another (parent) reference frame.

Note: It is now recommended to use the `.orient_axis, .orient_body_fixed, .orient_space_fixed, .orient_quaternion` methods for the different rotation types.

Parameters

parent : ReferenceFrame

Reference frame that this reference frame will be rotated relative to.

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'`.

Warns

UserWarning

If the orientation creates a kinematic loop.

orient_axis(*parent*, *axis*, *angle*)

Sets the orientation of this reference frame with respect to a parent reference frame by rotating through an angle about an axis fixed in the parent reference frame.

Parameters

parent : ReferenceFrame

Reference frame that this reference frame will be rotated relative to.

axis : Vector

Vector fixed in the parent frame about about which this frame is rotated. It need not be a unit vector and the rotation follows the right hand rule.

angle : sympifiable

Angle in radians by which it the frame is to be rotated.

Warns

UserWarning

If the orientation creates a kinematic loop.

Examples

Setup variables for the examples:

```
>>> from sympy import symbols
>>> from sympy.physics.vector import ReferenceFrame
>>> q1 = symbols('q1')
>>> N = ReferenceFrame('N')
>>> B = ReferenceFrame('B')
>>> B.orient_axis(N, N.x, q1)
```

The `orient_axis()` method generates a direction cosine matrix and its transpose which defines the orientation of B relative to N and vice versa. Once orient is called, `dcm()` outputs the appropriate direction cosine matrix:

```
>>> B.dcm(N)
Matrix([
[1,       0,      0],
[0,  cos(q1), sin(q1)],
[0, -sin(q1), cos(q1)]])
>>> N.dcm(B)
Matrix([
[1,       0,        0],
[0, cos(q1), -sin(q1)],
[0, sin(q1),  cos(q1)]])
```

The following two lines show that the sense of the rotation can be defined by negating the vector direction or the angle. Both lines produce the same result.

```
>>> B.orient_axis(N, -N.x, q1)
>>> B.orient_axis(N, N.x, -q1)
```

orient_body_fixed(*parent*, *angles*, *rotation_order*)

Rotates this reference frame relative to the parent reference frame by right hand rotating through three successive body fixed simple axis rotations. Each subsequent axis of rotation is about the "body fixed" unit vectors of a new intermediate reference frame. This type of rotation is also referred to rotating through the Euler and Tait-Bryan Angles.

Parameters

parent : ReferenceFrame

Reference frame that this reference frame will be rotated relative to.

angles : 3-tuple of sympifiable

Three angles in radians used for the successive rotations.

rotation_order : 3 character string or 3 digit integer

Order of the rotations about each intermediate reference frames' unit vectors. The Euler rotation about the X, Z', X'' axes can be specified by the strings `'XZX'`, `'131'`, or the integer 131. There are 12 unique valid rotation orders (6 Euler and 6 Tait-Bryan): zxz, xyx, yzy, zyz, xzx, yxy, xyz, yzx, zxy, xzy, zyx, and yxz.

Warns

UserWarning

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')
```

For example, a classic Euler Angle rotation can be done by:

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

(continues on next page)

(continued from previous page)

```
↪sin(q1)*cos(q3) + sin(q3)*cos(q1)*cos(q2)],
[sin(q2)*cos(q3), -sin(q1)*cos(q2)*cos(q3) - sin(q3)*cos(q1), -
↪sin(q1)*sin(q3) + cos(q1)*cos(q2)*cos(q3)]])
```

This rotates reference frame B relative to reference frame N through `q1` about `N.x`, then rotates B again through `q2` about `B.y`, and finally through `q3` about `B.x`. It is equivalent to three successive `orient_axis()` calls:

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

Acceptable rotation orders are of length 3, expressed in as a string `'XYZ'` or `'123'` or integer 123. Rotations about an axis twice in a row are prohibited.

```
>>> B.orient_body_fixed(N, (q1, q2, 0), 'ZXZ')
>>> B.orient_body_fixed(N, (q1, q2, 0), '121')
>>> B.orient_body_fixed(N, (q1, q2, q3), 123)
```

orient_explicit(*parent*, *dcm*)

Sets the orientation of this reference frame relative to a parent reference frame by explicitly setting the direction cosine matrix.

Parameters

parent : ReferenceFrame

Reference frame that this reference frame will be rotated relative to.

dcm : Matrix, shape(3, 3)

Direction cosine matrix that specifies the relative rotation between the two reference frames.

Warns

UserWarning

If the orientation creates a kinematic loop.

Examples

Setup variables for the examples:

```
>>> from sympy import symbols, Matrix, sin, cos
>>> from sympy.physics.vector import ReferenceFrame
>>> q1 = symbols('q1')
>>> A = ReferenceFrame('A')
>>> B = ReferenceFrame('B')
>>> N = ReferenceFrame('N')
```

A simple rotation of `A` relative to `N` about `N.x` is defined by the following direction cosine matrix:

```
>>> dcm = Matrix([[1, 0, 0],
...               [0, cos(q1), -sin(q1)],
...               [0, sin(q1), cos(q1)]])
>>> A.orient_explicit(N, dcm)
>>> A.dcm(N)
Matrix([
[1,       0,      0],
[0,  cos(q1), sin(q1)],
[0, -sin(q1), cos(q1)]])
```

This is equivalent to using `orient_axis()`:

```
>>> B.orient_axis(N, N.x, q1)
>>> B.dcm(N)
Matrix([
[1,       0,      0],
[0,  cos(q1), sin(q1)],
[0, -sin(q1), cos(q1)]])
```

Note carefully that `N.dcm(B)` (the transpose) would be passed into `orient_explicit()` for `A.dcm(N)` to match `B.dcm(N)`:

```
>>> A.orient_explicit(N, N.dcm(B))
>>> A.dcm(N)
Matrix([
[1,       0,      0],
[0,  cos(q1), sin(q1)],
[0, -sin(q1), cos(q1)]])
```

orient_quaternion(*parent*, *numbers*)

Sets the orientation of this reference frame relative to a parent reference frame via an orientation quaternion. An orientation quaternion is defined as a finite rotation a unit vector, `(lambda_x, lambda_y, lambda_z)`, by an angle `theta`. The orientation quaternion is described by four parameters:

- `q0 = cos(theta/2)`
- `q1 = lambda_x*sin(theta/2)`
- `q2 = lambda_y*sin(theta/2)`
- `q3 = lambda_z*sin(theta/2)`

See Quaternions and Spatial Rotation on Wikipedia for more information.

Parameters

parent : ReferenceFrame

Reference frame that this reference frame will be rotated relative to.

numbers : 4-tuple of sympifiable

The four quaternion scalar numbers as defined above: `q0`, `q1`, `q2`, `q3`.

Warns

UserWarning

If the orientation creates a kinematic loop.

Examples

Setup variables for the examples:

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

Set the orientation:

```
>>> B.orient_quaternion(N, (q0, q1, q2, q3))
>>> B.dcm(N)
Matrix([
[q0**2 + q1**2 - q2**2 - q3**2,             2*q0*q3 + 2*q1*q2,
 ↪     -2*q0*q2 + 2*q1*q3],
[            -2*q0*q3 + 2*q1*q2, q0**2 - q1**2 + q2**2 - q3**2,
 ↪      2*q0*q1 + 2*q2*q3],
[             2*q0*q2 + 2*q1*q3,            -2*q0*q1 + 2*q2*q3, q0**2 -
 ↪ q1**2 - q2**2 + q3**2]])
```

orient_space_fixed(*parent*, *angles*, *rotation_order*)

Rotates this reference frame relative to the parent reference frame by right hand rotating through three successive space fixed simple axis rotations. Each subsequent axis of rotation is about the “space fixed” unit vectors of the parent reference frame.

Parameters

parent : ReferenceFrame

Reference frame that this reference frame will be rotated relative to.

angles : 3-tuple of sympifiable

Three angles in radians used for the successive rotations.

rotation_order : 3 character string or 3 digit integer

Order of the rotations about the parent reference frame’s unit vectors. The order can be specified by the strings `'XZX'`, `'131'`, or the integer `131`. There are 12 unique valid rotation orders.

Warns

UserWarning