(continued from previous page)

```
>>> electric_potential
2*R.x**2*R.y
```

It is to be noted that `BaseScalar` instances can be used just like any other SymPy `Symbol`, except that they store the information about the coordinate system and axis they correspond to.

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.x`), you would use the `diff` method.

```
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> electric_potential = 2*R.x**2*R.y
>>> from sympy import diff
>>> diff(electric_potential, R.x)
4*R.x*R.y
```

It is worth noting that having a `BaseScalar` in the expression implies that a 'field' changes with position, in 3D space. Technically speaking, a simple `Expr` with no `BaseScalar` s is still a field, though constant.

Like scalar fields, vector fields that vary with position can also be constructed using `BaseScalar` s in the measure-number expressions.

```
>>> from sympy.vector import CoordSys3D
>>> R = CoordSys3D('R')
>>> v = R.x**2*R.i + 2*R.x*R.z*R.k
```

The Del operator

The Del, or 'Nabla' operator - written as ∇ is commonly known as the vector differential operator. Depending on its usage in a mathematical expression, it may denote the gradient of a scalar field, the divergence of a vector field, or the curl of a vector field.

Essentially, ∇ is not technically an 'operator', but a convenient mathematical notation to denote any one of the aforementioned field operations.

In *sympy.vector* (page 1425), ∇ has been implemented as the `Del()` class. The instance of this class is independent of coordinate system. Hence, the ∇ operator would be accessible as `Del()`.

Given below is an example of usage of the `Del()` class.

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> gradient_field = delop(C.x*C.y*C.z)
>>> gradient_field
(Derivative(C.x*C.y*C.z, C.x))*C.i + (Derivative(C.x*C.y*C.z, C.y))*C.j
+ (Derivative(C.x*C.y*C.z, C.z))*C.k
```

The above expression can be evaluated using the SymPy `doit()` routine.

```
>>> gradient_field.doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Usage of the ∇ notation in *sympy.vector* (page 1425) has been described in greater detail in the subsequent subsections.

Field operators and related functions

Here we describe some basic field-related functionality implemented in *sympy.vector* (page 1425).

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 $\mathbf{F}$.

Computing the curl of a vector field in *sympy.vector* (page 1425) can be accomplished in two ways.

One, by using the `Del()` class

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.cross(C.x*C.y*C.z*C.i).doit()
C.x*C.y*C.j + (-C.x*C.z)*C.k
>>> (delop ^ C.x*C.y*C.z*C.i).doit()
C.x*C.y*C.j + (-C.x*C.z)*C.k
```

Or by using the dedicated function

```
>>> from sympy.vector import curl
>>> curl(C.x*C.y*C.z*C.i)
C.x*C.y*C.j + (-C.x*C.z)*C.k
```

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.

Computing the divergence of a vector field in *sympy.vector* (page 1425) can be accomplished in two ways.

One, by using the `Del()` class

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.dot(C.x*C.y*C.z*(C.i + C.j + C.k)).doit()
C.x*C.y + C.x*C.z + C.y*C.z
>>> (delop & C.x*C.y*C.z*(C.i + C.j + C.k)).doit()
C.x*C.y + C.x*C.z + C.y*C.z
```

Or by using the dedicated function

```
>>> from sympy.vector import divergence
>>> divergence(C.x*C.y*C.z*(C.i + C.j + C.k))
C.x*C.y + C.x*C.z + C.y*C.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 axes respectively.

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

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

Computing the divergence of a vector field in *sympy.vector* (page 1425) can be accomplished in two ways.

One, by using the `Del()` class

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.gradient(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
>>> delop(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Or by using the dedicated function

```
>>> from sympy.vector import gradient
>>> gradient(C.x*C.y*C.z)
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

Directional Derivative

Apart from the above three common applications of ∇, it is also possible to compute the directional derivative of a field wrt a `Vector` in *sympy.vector* (page 1425).

By definition, the directional derivative of a field $\mathbf{F}$ along a vector v at point x represents the instantaneous rate of change of $\mathbf{F}$ moving through x with the velocity v. It is represented mathematically as: $(\vec{v} \cdot \nabla)\,\mathbf{F}(x)$.

Directional derivatives of vector and scalar fields can be computed in *sympy.vector* (page 1425) using the `Del()` class

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> vel = C.i + C.j + C.k
>>> scalar_field = C.x*C.y*C.z
>>> vector_field = C.x*C.y*C.z*C.i
>>> (vel.dot(delop))(scalar_field)
C.x*C.y + C.x*C.z + C.y*C.z
>>> (vel & delop)(vector_field)
(C.x*C.y + C.x*C.z + C.y*C.z)*C.i
```

Or by using the dedicated function

```
>>> from sympy.vector import directional_derivative
>>> directional_derivative(C.x*C.y*C.z, 3*C.i + 4*C.j + C.k)
C.x*C.y + 4*C.x*C.z + 3*C.y*C.z
```

Field operator in orthogonal curvilinear coordinate system

`vector` package supports calculation in different kind of orthogonal curvilinear coordinate system. To do that, scaling factor (also known as Lame coefficients) are used to express `curl`, `divergence` or `gradient` in desired type of coordinate system.

For example if we want to calculate `gradient` in cylindrical coordinate system all we need to do is to create proper coordinate system

```
>>> from sympy.vector import CoordSys3D
>>> c = CoordSys3D('c', transformation='cylindrical', variable_names=("r",
↪"theta", "z"))
>>> gradient(c.r*c.theta*c.z)
    c.theta*c.z*c.i + c.z*c.j + c.r*c.theta*c.k
```

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 travelled. 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.vector* (page 1425), the `is_conservative` function can be used.

```
>>> from sympy.vector import CoordSys3D, is_conservative
>>> R = CoordSys3D('R')
>>> field = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> is_conservative(field)
True
>>> curl(field)
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.vector* (page 1425), the `is_solenoidal` function can be used.

```
>>> from sympy.vector import CoordSys3D, is_solenoidal
>>> R = CoordSys3D('R')
>>> field = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> is_solenoidal(field)
True
>>> divergence(field)
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 `scalar_potential` function in *sympy.vector* (page 1425) 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.vector import CoordSys3D, scalar_potential
>>> R = CoordSys3D('R')
>>> conservative_field = 4*R.x*R.y*R.z*R.i + 2*R.x**2*R.z*R.j + 2*R.x**2*R.
↪y*R.k
>>> scalar_potential(conservative_field, R)
2*R.x**2*R.y*R.z
```

Providing a non-conservative vector field as an argument to `scalar_potential` 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.vector* (page 1425).

```
>>> from sympy.vector import CoordSys3D, Point
>>> from sympy.vector import scalar_potential_difference
>>> R = CoordSys3D('R')
>>> P = R.origin.locate_new('P', 1*R.i + 2*R.j + 3*R.k)
>>> vectfield = 4*R.x*R.y*R.i + 2*R.x**2*R.j
>>> scalar_potential_difference(vectfield, R, R.origin, P)
4
```

If provided with a scalar expression instead of a vector field, `scalar_potential_difference` returns the difference between the values of that scalar field at the two given points in space.

General examples of usage

This section details the solution of two basic problems in vector math/calculus using the *sympy.vector* (page 1425) package.

Quadrilateral problem

The Problem

OABC is any quadrilateral in 3D space. P is the midpoint of OA, Q is the midpoint of AB, R is the midpoint of BC and S is the midpoint of OC. Prove that PQ is parallel to SR

Solution

The solution to this problem demonstrates the usage of `Point`, and basic operations on `Vector`.

Define a coordinate system

```
>>> from sympy.vector import CoordSys3D
>>> Sys = CoordSys3D('Sys')
```

Define point O to be Sys' origin. We can do this without loss of generality

```
>>> O = Sys.origin
```

Define point A with respect to O

```
>>> from sympy import symbols
>>> a1, a2, a3 = symbols('a1 a2 a3')
>>> A = O.locate_new('A', a1*Sys.i + a2*Sys.j + a3*Sys.k)
```

Similarly define points B and C

```
>>> b1, b2, b3 = symbols('b1 b2 b3')
>>> B = O.locate_new('B', b1*Sys.i + b2*Sys.j + b3*Sys.k)
>>> c1, c2, c3 = symbols('c1 c2 c3')
>>> C = O.locate_new('C', c1*Sys.i + c2*Sys.j + c3*Sys.k)
```

P is the midpoint of OA. Lets locate it with respect to O (you could also define it with respect to A).

```
>>> P = O.locate_new('P', A.position_wrt(O) + (O.position_wrt(A) / 2))
```

Similarly define points Q, R and S as per the problem definitions.

```
>>> Q = A.locate_new('Q', B.position_wrt(A) / 2)
>>> R = B.locate_new('R', C.position_wrt(B) / 2)
>>> S = O.locate_new('R', C.position_wrt(O) / 2)
```

Now compute the vectors in the directions specified by PQ and SR.

```
>>> PQ = Q.position_wrt(P)
>>> SR = R.position_wrt(S)
```

Compute cross product

```
>>> PQ.cross(SR)
0
```

Since the cross product is a zero vector, the two vectors have to be parallel, thus proving that PQ || SR.

Third product rule for Del operator

See

The Problem

Prove the third rule - $\nabla \cdot (f\vec{v}) = f(\nabla \cdot \vec{v}) + \vec{v} \cdot (\nabla f)$

Solution

Start with a coordinate system

```
>>> from sympy.vector import CoordSys3D, Del
>>> delop = Del()
>>> C = CoordSys3D('C')
```

The scalar field f and the measure numbers of the vector field $\vec{v}$ are all functions of the coordinate variables of the coordinate system in general. Hence, define SymPy functions that way.

```
>>> from sympy import symbols, Function
>>> v1, v2, v3, f = symbols('v1 v2 v3 f', cls=Function)
```

v1, v2 and v3 are the X, Y and Z components of the vector field respectively.

Define the vector field as `vfield` and the scalar field as `sfield`.

```
>>> vfield = v1(C.x, C.y, C.z)*C.i + v2(C.x, C.y, C.z)*C.j + v3(C.x, C.y, C.
↪z)*C.k
>>> ffield = f(C.x, C.y, C.z)
```

Construct the expression for the LHS of the equation using `Del()`.

```
>>> lhs = (delop.dot(ffield * vfield)).doit()
```

Similarly, the RHS would be defined.

```
>>> rhs = ((vfield.dot(delop(ffield))) + (ffield * (delop.dot(vfield)))).
↪doit()
```

Now, to prove the product rule, we would just need to equate the expanded and simplified versions of the lhs and the rhs, so that the SymPy expressions match.

```
>>> lhs.expand().simplify() == rhs.expand().doit().simplify()
True
```

Thus, the general form of the third product rule mentioned above can be proven using *sympy.vector* (page 1425).

Applications of Vector Integrals

To integrate a scalar or vector field over a region, we have to first define a region. SymPy provides three methods for defining a region:

1. Using Parametric Equations with *ParametricRegion* (page 1461).
2. Using Implicit Equation with *ImplicitRegion* (page 1462).
3. Using objects of geometry module.

The *vector_integrate()* (page 1475) function is used to integrate scalar or vector field over any type of region. It automatically determines the type of integration (line, surface, or volume) depending on the nature of the object.

We define a coordinate system and make necesssary imports for examples.

```
>>> from sympy import sin, cos, exp, pi, symbols
>>> from sympy.vector import CoordSys3D, ParametricRegion, ImplicitRegion,␣
↪vector_integrate
>>> from sympy.abc import r, x, y, z, theta, phi
>>> C = CoordSys3D('C')
```

Calculation of Perimeter, Surface Area, and Volume

To calculate the perimeter of a circle, we need to define it. Let's define it using its parametric equation.

```
>>> param_circle = ParametricRegion((4*cos(theta), 4*sin(theta)), (theta, 0,␣
↪2*pi))
```

We can also define a circle using its implicit equation.

```
>>> implicit_circle = ImplicitRegion((x, y), x**2 + y**2 - 4)
```

The perimeter of a figure is equal to the absolute value of its integral over a unit scalar field.

```
>>> vector_integrate(1, param_circle)
8*pi
>>> vector_integrate(1, implicit_circle)
4*pi
```

Suppose a user wants to calculate the perimeter of a triangle. Determining the parametric representation of a triangle can be difficult. Instead, the user can use an object of *Polygon* (page 2273) class in the geometry module.

```
>>> from sympy.geometry import Point, Polygon
>>> triangle = Polygon(Point(1, 2), (3, 5), (1,6))
>>> vector_integrate(1, triangle)
sqrt(5) + sqrt(13) + 4
```

To define a solid sphere, we need to use three parameters (r, theta and phi). For *ParametricRegion* (page 1461) obextj, the order of limits determine the sign of the integral.

```
>>> solidsphere = ParametricRegion((r*sin(phi)*cos(theta),
↪r*sin(phi)*sin(theta), r*cos(phi)),
...                             (phi, 0, pi), (theta, 0, 2*pi), (r, 0, 3))
>>> vector_integrate(1, solidsphere)
36*pi
```

Calculation of mass of a body

Consider a triangular lamina R with vertices (0,0), (0, 5), (5,0) and with density $\rho(x, y) = xy\ kg/m^2$. Find the total mass.

```
>>> triangle = ParametricRegion((x, y), (x, 0, 5), (y, 0, 5 - x))
>>> vector_integrate(C.x*C.y, triangle)
625/24
```

Find the mass of a cylinder centered on the z-axis which has height h, radius a, and density $\rho = x^2 + y^2\ kg/m^2$.

```
>>> a, h = symbols('a h', positive=True)
>>> cylinder = ParametricRegion((r*cos(theta), r*sin(theta), z),
...                             (theta, 0, 2*pi), (z, 0, h), (r, 0, a))
```

(continues on next page)

(continued from previous page)

```
>>> vector_integrate(C.x**2 + C.y**2, cylinder)
pi*a**4*h/2
```

Calculation of Flux

1. Consider a region of space in which there is a constant vectorfield $E(x, y, z) = a\hat{\mathbf{k}}$. A hemisphere of radius r lies on the x-y plane. What is the flux of the field through the sphere?

```
>>> semisphere = ParametricRegion((r*sin(phi)*cos(theta),␣
↪r*sin(phi)*sin(theta), r*cos(phi)),\
...                                (phi, 0, pi/2), (theta, 0, 2*pi))
>>> flux = vector_integrate(a*C.k, semisphere)
>>> flux
pi*a*r**2
```

2. Consider a region of space in which there is a vector field $E(x, y, z) = x^2\hat{\mathbf{k}}$ above the x-y plane, and a field $E(x, y, z) = y^2\hat{\mathbf{k}}$ below the x-y plane. What is the flux of that vector field through a cube of side length L with its center at the origin?"

The field is parallel to the z-axis so only the top and bottom face of the box will contribute to flux.

```
>>> L = symbols('L', positive=True)
>>> top_face = ParametricRegion((x, y, L/2), (x, -L/2, L/2), (y, -L/2, L/2))
>>> bottom_face = ParametricRegion((x, y, -L/2), (x, -L/2, L/2), (y, -L/2, L/
↪2))
>>> flux = vector_integrate(C.x**2*C.k, top_face) + vector_integrate(C.y**2*C.
↪k, bottom_face)
>>> flux
L**4/6
```

Verifying Stoke's Theorem

See https://en.wikipedia.org/wiki/Stokes%27_theorem

Example 1

```
>>> from sympy.vector import curl
>>> curve = ParametricRegion((cos(theta), sin(theta)), (theta, 0, pi/2))
>>> surface = ParametricRegion((r*cos(theta), r*sin(theta)), (r, 0, 1),␣
↪(theta, 0, pi/2))
>>> F = C.y*C.i + C.z*C.k + C.x*C.k
>>>
>>> vector_integrate(F, curve)
-pi/4
>>> vector_integrate(curl(F), surface)
-pi/4
```

Example 2

```
>>> circle = ParametricRegion((cos(theta), sin(theta), 1), (theta, 0,␣
↪2*pi))
>>> cone = ParametricRegion((r*cos(theta), r*sin(theta), r), (r, 0, 1),␣
↪(theta, 0, 2*pi))
>>> cone = ParametricRegion((r*cos(theta), r*sin(theta), r), (r, 0, 1),␣
↪(theta, 0, 2*pi))
>>> f = (-C.y**3/3 + sin(C.x))*C.i + (C.x**3/3 + cos(C.y))*C.j + C.x*C.
↪y*C.z*C.k
>>> vector_integrate(f,  circle)
pi/2
>>> vector_integrate(curl(f),  cone)
pi/2
```

Verifying Divergence Theorem

See https://en.wikipedia.org/wiki/Divergence_theorem

Example 1

```
>>> from sympy.vector import divergence
>>> sphere = ParametricRegion((4*sin(phi)*cos(theta),
↪4*sin(phi)*sin(theta), 4*cos(phi)),
...                          (phi, 0, pi), (theta, 0, 2*pi))
>>> solidsphere = ParametricRegion((r*sin(phi)*cos(theta),
↪r*sin(phi)*sin(theta), r*cos(phi)),
...     (r, 0, 4),(phi, 0, pi), (theta, 0, 2*pi))
>>> field = C.x**3*C.i + C.y**3*C.j + C.z**3*C.k
>>> vector_integrate(field, sphere)
12288*pi/5
>>> vector_integrate(divergence(field), solidsphere)
12288*pi/5
```

Example 2

```
>>> cube = ParametricRegion((x, y, z), (x, 0, 1), (y, 0, 1), (z, 0, 1))
>>> field = 2*C.x*C.y*C.i + 3*C.x*C.y*C.j + C.z*exp(C.x + C.y)*C.k
>>> vector_integrate(divergence(field), cube)
-E + 7/2 + E*(-1 + E)
```

Vector API

Essential Classes in sympy.vector (docstrings)

CoordSys3D

class sympy.vector.coordsysrect.**CoordSys3D**(*name*, *transformation=None*, *parent=None*, *location=None*, *rotation_matrix=None*, *vector_names=None*, *variable_names=None*)

Represents a coordinate system in 3-D space.

__init__(*name*, *location=None*, *rotation_matrix=None*, *parent=None*, *vector_names=None*, *variable_names=None*, *latex_vects=None*, *pretty_vects=None*, *latex_scalars=None*, *pretty_scalars=None*, *transformation=None*)

The orientation/location parameters are necessary if this system is being defined at a certain orientation or location wrt another.

Parameters

name : str

The name of the new CoordSys3D instance.

transformation : Lambda, Tuple, str

Transformation defined by transformation equations or chosen from predefined ones.

location : Vector

The position vector of the new system's origin wrt the parent instance.

rotation_matrix : SymPy ImmutableMatrix

The rotation matrix of the new coordinate system with respect to the parent. In other words, the output of new_system.rotation_matrix(parent).

parent : CoordSys3D

The coordinate system wrt which the orientation/location (or both) is being defined.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

create_new(*name*, *transformation*, *variable_names=None*, *vector_names=None*)

Returns a CoordSys3D which is connected to self by transformation.

Parameters

name : str

The name of the new CoordSys3D instance.

transformation : Lambda, Tuple, str

Transformation defined by transformation equations or chosen from predefined ones.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> a = CoordSys3D('a')
>>> b = a.create_new('b', transformation='spherical')
>>> b.transformation_to_parent()
(b.r*sin(b.theta)*cos(b.phi), b.r*sin(b.phi)*sin(b.theta), b.r*cos(b.
↪theta))
>>> b.transformation_from_parent()
(sqrt(a.x**2 + a.y**2 + a.z**2), acos(a.z/sqrt(a.x**2 + a.y**2 + a.
↪z**2)), atan2(a.y, a.x))
```

locate_new(*name*, *position*, *vector_names=None*, *variable_names=None*)

Returns a CoordSys3D with its origin located at the given position wrt this coordinate system's origin.

Parameters

name : str

The name of the new CoordSys3D instance.

position : Vector

The position vector of the new system's origin wrt this one.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> A = CoordSys3D('A')
>>> B = A.locate_new('B', 10 * A.i)
>>> B.origin.position_wrt(A.origin)
10*A.i
```

orient_new(*name*, *orienters*, *location=None*, *vector_names=None*, *variable_names=None*)

Creates a new CoordSys3D oriented in the user-specified way with respect to this system.

Please refer to the documentation of the orienter classes for more information about the orientation procedure.

Parameters

name : str

The name of the new CoordSys3D instance.

orienters : iterable/Orienter

An Orienter or an iterable of Orienters for orienting the new coordinate system. If an Orienter is provided, it is applied to get the new

system. If an iterable is provided, the orienters will be applied in the order in which they appear in the iterable.

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSys3D('N')
```

Using an AxisOrienter

```
>>> from sympy.vector import AxisOrienter
>>> axis_orienter = AxisOrienter(q1, N.i + 2 * N.j)
>>> A = N.orient_new('A', (axis_orienter, ))
```

Using a BodyOrienter

```
>>> from sympy.vector import BodyOrienter
>>> body_orienter = BodyOrienter(q1, q2, q3, '123')
>>> B = N.orient_new('B', (body_orienter, ))
```

Using a SpaceOrienter

```
>>> from sympy.vector import SpaceOrienter
>>> space_orienter = SpaceOrienter(q1, q2, q3, '312')
>>> C = N.orient_new('C', (space_orienter, ))
```

Using a QuaternionOrienter

```
>>> from sympy.vector import QuaternionOrienter
>>> q_orienter = QuaternionOrienter(q0, q1, q2, q3)
>>> D = N.orient_new('D', (q_orienter, ))
```

orient_new_axis(*name*, *angle*, *axis*, *location=None*, *vector_names=None*, *variable_names=None*)

Axis rotation is a rotation about an arbitrary axis by some angle. The angle is supplied as a SymPy expr scalar, and the axis is supplied as a Vector.

Parameters

name : string

The name of the new coordinate system

angle : Expr

The angle by which the new system is to be rotated

axis : Vector

The axis around which the rotation has to be performed

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = CoordSys3D('N')
>>> B = N.orient_new_axis('B', q1, N.i + 2 * N.j)
```

orient_new_body(*name*, *angle1*, *angle2*, *angle3*, *rotation_order*, *location=None*, *vector_names=None*, *variable_names=None*)

Body orientation takes this coordinate system through three successive simple rotations.

Body fixed rotations include both Euler Angles and Tait-Bryan Angles, see https://en.wikipedia.org/wiki/Euler_angles.

Parameters

name : string

The name of the new coordinate system

angle1, angle2, angle3 : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')
```

A 'Body' fixed rotation is described by three angles and three body-fixed rotation axes. To orient a coordinate system D with respect to N, each sequential rotation is always about the orthogonal unit vectors fixed to D. For example, a '123' rotation will specify rotations about N.i, then D.j, then D.k. (Initially, D.i is same as N.i) Therefore,

```
>>> D = N.orient_new_body('D', q1, q2, q3, '123')
```

is same as

```
>>> D = N.orient_new_axis('D', q1, N.i)
>>> D = D.orient_new_axis('D', q2, D.j)
>>> D = D.orient_new_axis('D', q3, D.k)
```

Acceptable rotation orders are of length 3, expressed in XYZ or 123, and cannot have a rotation about about an axis twice in a row.

```
>>> B = N.orient_new_body('B', q1, q2, q3, '123')
>>> B = N.orient_new_body('B', q1, q2, 0, 'ZXZ')
>>> B = N.orient_new_body('B', 0, 0, 0, 'XYX')
```

orient_new_quaternion(*name*, *q0*, *q1*, *q2*, *q3*, *location=None*, *vector_names=None*, *variable_names=None*)

Quaternion orientation orients the new CoordSys3D with Quaternions, defined as a finite rotation about lambda, a unit vector, by some amount theta.

This orientation 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)

Quaternion does not take in a rotation order.

Parameters

name : string

The name of the new coordinate system

q0, q1, q2, q3 : Expr

The quaternions to rotate the coordinate system by

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSys3D('N')
>>> B = N.orient_new_quaternion('B', q0, q1, q2, q3)
```

orient_new_space(*name*, *angle1*, *angle2*, *angle3*, *rotation_order*, *location=None*, *vector_names=None*, *variable_names=None*)

Space rotation is similar to Body rotation, but the rotations are applied in the opposite order.

Parameters

name : string

The name of the new coordinate system

angle1, angle2, angle3 : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

location : Vector(optional)

The location of the new coordinate system's origin wrt this system's origin. If not specified, the origins are taken to be coincident.

vector_names, variable_names : iterable(optional)

Iterables of 3 strings each, with custom names for base vectors and base scalars of the new system respectively. Used for simple str printing.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')
```

To orient a coordinate system D with respect to N, each sequential rotation is always about N's orthogonal unit vectors. For example, a '123' rotation will specify rotations about N.i, then N.j, then N.k. Therefore,

```
>>> D = N.orient_new_space('D', q1, q2, q3, '312')
```

is same as

```
>>> B = N.orient_new_axis('B', q1, N.i)
>>> C = B.orient_new_axis('C', q2, N.j)
>>> D = C.orient_new_axis('D', q3, N.k)
```

See also:

CoordSys3D.orient_new_body **(page 1451)**
method to orient via Euler angles

position_wrt(*other*)

Returns the position vector of the origin of this coordinate system with respect to another Point/CoordSys3D.

Parameters
other : Point/CoordSys3D

If other is a Point, the position of this system's origin wrt it is returned. If its an instance of CoordSyRect, the position wrt its origin is returned.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N1 = N.locate_new('N1', 10 * N.i)
>>> N.position_wrt(N1)
(-10)*N.i
```

rotation_matrix(*other*)

Returns the direction cosine matrix(DCM), also known as the 'rotation matrix' of this coordinate system with respect to another system.

If v_a is a vector defined in system 'A' (in matrix format) and v_b is the same vector defined in system 'B', then v_a = A.rotation_matrix(B) * v_b.

A SymPy Matrix is returned.

Parameters
other : CoordSys3D

The system which the DCM is generated to.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = CoordSys3D('N')
>>> A = N.orient_new_axis('A', q1, N.i)
>>> N.rotation_matrix(A)
Matrix([
[1,       0,        0],
```

(continues on next page)

(continued from previous page)

```
[0, cos(q1), -sin(q1)],
[0, sin(q1),  cos(q1)]])
```

scalar_map(*other*)

Returns a dictionary which expresses the coordinate variables (base scalars) of this frame in terms of the variables of otherframe.

Parameters

otherframe : CoordSys3D

The other system to map the variables to.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import Symbol
>>> A = CoordSys3D('A')
>>> q = Symbol('q')
>>> B = A.orient_new_axis('B', q, A.k)
>>> A.scalar_map(B)
{A.x: B.x*cos(q) - B.y*sin(q), A.y: B.x*sin(q) + B.y*cos(q), A.z: B.z}
```

Vector

class sympy.vector.vector.**Vector**(**args*)

Super class for all Vector classes. Ideally, neither this class nor any of its subclasses should be instantiated by the user.

property components

Returns the components of this vector in the form of a Python dictionary mapping BaseVector instances to the corresponding measure numbers.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> C = CoordSys3D('C')
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v.components
{C.i: 3, C.j: 4, C.k: 5}
```

cross(*other*)

Returns the cross product of this Vector with another Vector or Dyadic instance. The cross product is a Vector, if 'other' is a Vector. If 'other' is a Dyadic, this returns a Dyadic instance.

Parameters

other: Vector/Dyadic

The Vector or Dyadic we are crossing with.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> C = CoordSys3D('C')
>>> C.i.cross(C.j)
C.k
>>> C.i ^ C.i
0
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v ^ C.i
5*C.j + (-4)*C.k
>>> d = C.i.outer(C.i)
>>> C.j.cross(d)
(-1)*(C.k|C.i)
```

dot(*other*)

Returns the dot product of this Vector, either with another Vector, or a Dyadic, or a Del operator. If 'other' is a Vector, returns the dot product scalar (SymPy expression). If 'other' is a Dyadic, the dot product is returned as a Vector. If 'other' is an instance of Del, returns the directional derivative operator as a Python function. If this function is applied to a scalar expression, it returns the directional derivative of the scalar field wrt this Vector.

Parameters

other: Vector/Dyadic/Del

The Vector or Dyadic we are dotting with, or a Del operator .

Examples

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> C.i.dot(C.j)
0
>>> C.i & C.i
1
>>> v = 3*C.i + 4*C.j + 5*C.k
>>> v.dot(C.k)
5
>>> (C.i & delop)(C.x*C.y*C.z)
C.y*C.z
>>> d = C.i.outer(C.i)
>>> C.i.dot(d)
C.i
```

magnitude()

Returns the magnitude of this vector.

normalize()

Returns the normalized version of this vector.

outer(*other*)

Returns the outer product of this vector with another, in the form of a Dyadic instance.

Parameters

other : Vector

The Vector with respect to which the outer product is to be computed.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> N.i.outer(N.j)
(N.i|N.j)
```

projection(*other*, *scalar=False*)

Returns the vector or scalar projection of the 'other' on 'self'.

Examples

```
>>> from sympy.vector.coordsysrect import CoordSys3D
>>> C = CoordSys3D('C')
>>> i, j, k = C.base_vectors()
>>> v1 = i + j + k
>>> v2 = 3*i + 4*j
>>> v1.projection(v2)
7/3*C.i + 7/3*C.j + 7/3*C.k
>>> v1.projection(v2, scalar=True)
7/3
```

separate()

The constituents of this vector in different coordinate systems, as per its definition.

Returns a dict mapping each CoordSys3D to the corresponding constituent Vector.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> R1 = CoordSys3D('R1')
>>> R2 = CoordSys3D('R2')
>>> v = R1.i + R2.i
>>> v.separate() == {R1: R1.i, R2: R2.i}
True
```

to_matrix(*system*)

Returns the matrix form of this vector with respect to the specified coordinate system.

Parameters

system : CoordSys3D

The system wrt which the matrix form is to be computed

Examples

```
>>> from sympy.vector import CoordSys3D
>>> C = CoordSys3D('C')
>>> from sympy.abc import a, b, c
>>> v = a*C.i + b*C.j + c*C.k
>>> v.to_matrix(C)
Matrix([
[a],
[b],
[c]])
```

Dyadic

class sympy.vector.dyadic.**Dyadic**(**args*)

Super class for all Dyadic-classes.

References

[R978], [R979]

property components

Returns the components of this dyadic in the form of a Python dictionary mapping BaseDyadic instances to the corresponding measure numbers.

cross(*other*)

Returns the cross product between this Dyadic, and a Vector, as a Vector instance.

Parameters

other : Vector

The Vector that we are crossing this Dyadic with

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> d = N.i.outer(N.i)
>>> d.cross(N.j)
(N.i|N.k)
```

dot(*other*)

Returns the dot product(also called inner product) of this Dyadic, with another Dyadic or Vector. If 'other' is a Dyadic, this returns a Dyadic. Else, it returns a Vector (unless an error is encountered).

Parameters

other : Dyadic/Vector

The other Dyadic or Vector to take the inner product with

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> D1 = N.i.outer(N.j)
>>> D2 = N.j.outer(N.j)
>>> D1.dot(D2)
(N.i|N.j)
>>> D1.dot(N.j)
N.i
```

to_matrix(*system*, *second_system=None*)

Returns the matrix form of the dyadic with respect to one or two coordinate systems.

Parameters

system : CoordSys3D

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

second_system : CoordSys3D, optional, default=None

The coordinate system that the columns of the matrix correspond to.

Examples

```
>>> from sympy.vector import CoordSys3D
>>> N = CoordSys3D('N')
>>> v = N.i + 2*N.j
>>> d = v.outer(N.i)
>>> d.to_matrix(N)
Matrix([
[1, 0, 0],
[2, 0, 0],
[0, 0, 0]])
>>> from sympy import Symbol
>>> q = Symbol('q')
>>> P = N.orient_new_axis('P', q, N.k)
>>> d.to_matrix(N, P)
Matrix([
[  cos(q),   -sin(q), 0],
[2*cos(q), -2*sin(q), 0],
[       0,         0, 0]])
```

Del

class sympy.vector.deloperator.**Del**

Represents the vector differential operator, usually represented in mathematical expressions as the 'nabla' symbol.

cross(*vect*, *doit=False*)

Represents the cross product between this operator and a given vector - equal to the curl of the vector field.

Parameters

vect : Vector

The vector whose curl is to be calculated.

doit : bool

If True, the result is returned after calling .doit() on each component. Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> v = C.x*C.y*C.z * (C.i + C.j + C.k)
>>> delop.cross(v, doit = True)
(-C.x*C.y + C.x*C.z)*C.i + (C.x*C.y - C.y*C.z)*C.j +
    (-C.x*C.z + C.y*C.z)*C.k
>>> (delop ^ C.i).doit()
0
```

dot(*vect*, *doit=False*)

Represents the dot product between this operator and a given vector - equal to the divergence of the vector field.

Parameters

vect : Vector

The vector whose divergence is to be calculated.

doit : bool

If True, the result is returned after calling .doit() on each component. Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, Del
>>> delop = Del()
>>> C = CoordSys3D('C')
>>> delop.dot(C.x*C.i)
Derivative(C.x, C.x)
>>> v = C.x*C.y*C.z * (C.i + C.j + C.k)
>>> (delop & v).doit()
C.x*C.y + C.x*C.z + C.y*C.z
```

gradient(*scalar_field*, *doit=False*)

Returns the gradient of the given scalar field, as a Vector instance.

Parameters

scalar_field : SymPy expression

The scalar field to calculate the gradient of.

doit : bool

If True, the result is returned after calling .doit() on each component. Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, Del
>>> C = CoordSys3D('C')
>>> delop = Del()
>>> delop.gradient(9)
0
>>> delop(C.x*C.y*C.z).doit()
C.y*C.z*C.i + C.x*C.z*C.j + C.x*C.y*C.k
```

ParametricRegion

class sympy.vector.parametricregion.**ParametricRegion**(*definition*, **bounds*)

Represents a parametric region in space.

Parameters

definition : tuple to define base scalars in terms of parameters.

bounds : Parameter or a tuple of length 3 to define parameter and corresponding lower and upper bound.

Examples

```
>>> from sympy import cos, sin, pi
>>> from sympy.abc import r, theta, t, a, b, x, y
>>> from sympy.vector import ParametricRegion
```

```
>>> ParametricRegion((t, t**2), (t, -1, 2))
ParametricRegion((t, t**2), (t, -1, 2))
>>> ParametricRegion((x, y), (x, 3, 4), (y, 5, 6))
ParametricRegion((x, y), (x, 3, 4), (y, 5, 6))
>>> ParametricRegion((r*cos(theta), r*sin(theta)), (r, -2, 2), (theta, 0,
↪ pi))
ParametricRegion((r*cos(theta), r*sin(theta)), (r, -2, 2), (theta, 0,␣
↪pi))
>>> ParametricRegion((a*cos(t), b*sin(t)), t)
ParametricRegion((a*cos(t), b*sin(t)), t)
```

```
>>> circle = ParametricRegion((r*cos(theta), r*sin(theta)), r, (theta, 0,
↪ pi))
>>> circle.parameters
(r, theta)
>>> circle.definition
(r*cos(theta), r*sin(theta))
>>> circle.limits
{theta: (0, pi)}
```

Dimension of a parametric region determines whether a region is a curve, surface or volume region. It does not represent its dimensions in space.

```
>>> circle.dimensions
1
```

ImplicitRegion

class sympy.vector.implicitregion.**ImplicitRegion**(*variables*, *equation*)

Represents an implicit region in space.

Parameters

variables : tuple to map variables in implicit equation to base scalars.

equation : An expression or Eq denoting the implicit equation of the region.

Examples

```
>>> from sympy import Eq
>>> from sympy.abc import x, y, z, t
>>> from sympy.vector import ImplicitRegion
```

```
>>> ImplicitRegion((x, y), x**2 + y**2 - 4)
ImplicitRegion((x, y), x**2 + y**2 - 4)
>>> ImplicitRegion((x, y), Eq(y*x, 1))
ImplicitRegion((x, y), x*y - 1)
```

```
>>> parabola = ImplicitRegion((x, y), y**2 - 4*x)
>>> parabola.degree
2
>>> parabola.equation
-4*x + y**2
>>> parabola.rational_parametrization(t)
(4/t**2, 4/t)
```

```
>>> r = ImplicitRegion((x, y, z), Eq(z, x**2 + y**2))
>>> r.variables
(x, y, z)
>>> r.singular_points()
EmptySet
>>> r.regular_point()
(-10, -10, 200)
```

multiplicity(*point*)

Returns the multiplicity of a singular point on the region.

A singular point (x,y) of region is said to be of multiplicity m if all the partial derivatives off to order m - 1 vanish there.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.vector import ImplicitRegion
>>> I = ImplicitRegion((x, y, z), x**2 + y**3 - z**4)
>>> I.singular_points()
{(0, 0, 0)}
>>> I.multiplicity((0, 0, 0))
2
```

rational_parametrization(*parameters=('t', 's'), reg_point=None*)

Returns the rational parametrization of implict region.

Examples

```
>>> from sympy import Eq
>>> from sympy.abc import x, y, z, s, t
>>> from sympy.vector import ImplicitRegion
```

```
>>> parabola = ImplicitRegion((x, y), y**2 - 4*x)
>>> parabola.rational_parametrization()
(4/t**2, 4/t)
```

```
>>> circle = ImplicitRegion((x, y), Eq(x**2 + y**2, 4))
>>> circle.rational_parametrization()
(4*t/(t**2 + 1), 4*t**2/(t**2 + 1) - 2)
```

```
>>> I = ImplicitRegion((x, y), x**3 + x**2 - y**2)
>>> I.rational_parametrization()
(t**2 - 1, t*(t**2 - 1))
```

```
>>> cubic_curve = ImplicitRegion((x, y), x**3 + x**2 - y**2)
>>> cubic_curve.rational_parametrization(parameters=(t))
(t**2 - 1, t*(t**2 - 1))
```

```
>>> sphere = ImplicitRegion((x, y, z), x**2 + y**2 + z**2 - 4)
>>> sphere.rational_parametrization(parameters=(t, s))
(-2 + 4/(s**2 + t**2 + 1), 4*s/(s**2 + t**2 + 1), 4*t/(s**2 + t**2 +␣
↪1))
```

For some conics, regular_points() is unable to find a point on curve. To calulcate the parametric representation in such cases, user need to determine a point on the region and pass it using reg_point.

```
>>> c = ImplicitRegion((x, y), (x  - 1/2)**2 + (y)**2 - (1/4)**2)
>>> c.rational_parametrization(reg_point=(3/4, 0))
(0.75 - 0.5/(t**2 + 1), -0.5*t/(t**2 + 1))
```

References

- Christoph M. Hoffmann, “Conversion Methods between Parametric and Implicit Curves and Surfaces”, Purdue e-Pubs, 1990. Available: https://docs.lib.purdue.edu/cgi/viewcontent.cgi?article=1827&context=cstech

regular_point()

Returns a point on the implicit region.

Examples

```
>>> from sympy.abc import x, y, z
>>> from sympy.vector import ImplicitRegion
>>> circle = ImplicitRegion((x, y), (x + 2)**2 + (y - 3)**2 - 16)
>>> circle.regular_point()
(-2, -1)
>>> parabola = ImplicitRegion((x, y), x**2 - 4*y)
>>> parabola.regular_point()
(0, 0)
>>> r = ImplicitRegion((x, y, z), (x + y + z)**4)
>>> r.regular_point()
(-10, -10, 20)
```

References

- Erik Hillgarter, "Rational Points on Conics", Diploma Thesis, RISC-Linz, J. Kepler Universitat Linz, 1996. Availaible: https://www3.risc.jku.at/publications/download/risc_1355/Rational%20Points%20on%20Conics.pdf

singular_points()

Returns a set of singular points of the region.

The singular points are those points on the region where all partial derivatives vanish.

Examples

```
>>> from sympy.abc import x, y
>>> from sympy.vector import ImplicitRegion
>>> I = ImplicitRegion((x, y), (y-1)**2 -x**3 + 2*x**2 -x)
>>> I.singular_points()
{(1, 1)}
```

ParametricIntegral

class sympy.vector.integrals.**ParametricIntegral**(*field*, *parametricregion*)

Represents integral of a scalar or vector field over a Parametric Region

Examples

```
>>> from sympy import cos, sin, pi
>>> from sympy.vector import CoordSys3D, ParametricRegion,␣
↪ParametricIntegral
>>> from sympy.abc import r, t, theta, phi
```

```
>>> C = CoordSys3D('C')
>>> curve = ParametricRegion((3*t - 2, t + 1), (t, 1, 2))
>>> ParametricIntegral(C.x, curve)
5*sqrt(10)/2
>>> length = ParametricIntegral(1, curve)
>>> length
sqrt(10)
>>> semisphere = ParametricRegion((2*sin(phi)*cos(theta),␣
↪2*sin(phi)*sin(theta), 2*cos(phi)),                              (theta,␣
↪0, 2*pi), (phi, 0, pi/2))
>>> ParametricIntegral(C.z, semisphere)
8*pi
```

```
>>> ParametricIntegral(C.j + C.k, ParametricRegion((r*cos(theta),␣
↪r*sin(theta)), r, theta))
0
```

Orienter classes (docstrings)

Orienter

class sympy.vector.orienters.**Orienter**(**args*)

Super-class for all orienter classes.

rotation_matrix()

The rotation matrix corresponding to this orienter instance.

AxisOrienter

class sympy.vector.orienters.**AxisOrienter**(*angle*, *axis*)

Class to denote an axis orienter.

__init__(*angle*, *axis*)

Axis rotation is a rotation about an arbitrary axis by some angle. The angle is supplied as a SymPy expr scalar, and the axis is supplied as a Vector.

Parameters

angle : Expr

The angle by which the new system is to be rotated

axis : Vector

The axis around which the rotation has to be performed

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q1 = symbols('q1')
>>> N = CoordSys3D('N')
>>> from sympy.vector import AxisOrienter
>>> orienter = AxisOrienter(q1, N.i + 2 * N.j)
>>> B = N.orient_new('B', (orienter, ))
```

rotation_matrix(*system*)

The rotation matrix corresponding to this orienter instance.

Parameters

system : CoordSys3D

The coordinate system wrt which the rotation matrix is to be computed

BodyOrienter

class sympy.vector.orienters.**BodyOrienter**(*angle1*, *angle2*, *angle3*, *rot_order*)

Class to denote a body-orienter.

__init__(*angle1*, *angle2*, *angle3*, *rot_order*)

Body orientation takes this coordinate system through three successive simple rotations.

Body fixed rotations include both Euler Angles and Tait-Bryan Angles, see https://en.wikipedia.org/wiki/Euler_angles.

Parameters

angle1, angle2, angle3 : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

Examples

```
>>> from sympy.vector import CoordSys3D, BodyOrienter
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')
```

A 'Body' fixed rotation is described by three angles and three body-fixed rotation axes. To orient a coordinate system D with respect to N, each sequential rotation is always about the orthogonal unit vectors fixed to D. For example, a '123' rotation will specify rotations about N.i, then D.j, then D.k. (Initially, D.i is same as N.i) Therefore,

```
>>> body_orienter = BodyOrienter(q1, q2, q3, '123')
>>> D = N.orient_new('D', (body_orienter, ))
```

is same as

```
>>> from sympy.vector import AxisOrienter
>>> axis_orienter1 = AxisOrienter(q1, N.i)
>>> D = N.orient_new('D', (axis_orienter1, ))
>>> axis_orienter2 = AxisOrienter(q2, D.j)
>>> D = D.orient_new('D', (axis_orienter2, ))
>>> axis_orienter3 = AxisOrienter(q3, D.k)
>>> D = D.orient_new('D', (axis_orienter3, ))
```

Acceptable rotation orders are of length 3, expressed in XYZ or 123, and cannot have a rotation about about an axis twice in a row.

```
>>> body_orienter1 = BodyOrienter(q1, q2, q3, '123')
>>> body_orienter2 = BodyOrienter(q1, q2, 0, 'ZXZ')
>>> body_orienter3 = BodyOrienter(0, 0, 0, 'XYX')
```

SpaceOrienter

class sympy.vector.orienters.**SpaceOrienter**(*angle1*, *angle2*, *angle3*, *rot_order*)

Class to denote a space-orienter.

__init__(*angle1*, *angle2*, *angle3*, *rot_order*)

Space rotation is similar to Body rotation, but the rotations are applied in the opposite order.

Parameters

angle1, angle2, angle3 : Expr

Three successive angles to rotate the coordinate system by

rotation_order : string

String defining the order of axes for rotation

Examples

```
>>> from sympy.vector import CoordSys3D, SpaceOrienter
>>> from sympy import symbols
>>> q1, q2, q3 = symbols('q1 q2 q3')
>>> N = CoordSys3D('N')
```

To orient a coordinate system D with respect to N, each sequential rotation is always about N's orthogonal unit vectors. For example, a '123' rotation will specify rotations about N.i, then N.j, then N.k. Therefore,

```
>>> space_orienter = SpaceOrienter(q1, q2, q3, '312')
>>> D = N.orient_new('D', (space_orienter, ))
```

is same as

```
>>> from sympy.vector import AxisOrienter
>>> axis_orienter1 = AxisOrienter(q1, N.i)
>>> B = N.orient_new('B', (axis_orienter1, ))
>>> axis_orienter2 = AxisOrienter(q2, N.j)
>>> C = B.orient_new('C', (axis_orienter2, ))
>>> axis_orienter3 = AxisOrienter(q3, N.k)
>>> D = C.orient_new('C', (axis_orienter3, ))
```

See also:

BodyOrienter **(page 1467)**
Orienter to orient systems wrt Euler angles.

QuaternionOrienter

class sympy.vector.orienters.**QuaternionOrienter**(*q0*, *q1*, *q2*, *q3*)

Class to denote a quaternion-orienter.

__init__(*angle1*, *angle2*, *angle3*, *rot_order*)

Quaternion orientation orients the new CoordSys3D with Quaternions, defined as a finite rotation about lambda, a unit vector, by some amount theta.

This orientation 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)

Quaternion does not take in a rotation order.

Parameters

q0, q1, q2, q3 : Expr

The quaternions to rotate the coordinate system by

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import symbols
>>> q0, q1, q2, q3 = symbols('q0 q1 q2 q3')
>>> N = CoordSys3D('N')
>>> from sympy.vector import QuaternionOrienter
>>> q_orienter = QuaternionOrienter(q0, q1, q2, q3)
>>> B = N.orient_new('B', (q_orienter, ))
```

Essential Functions in sympy.vector (docstrings)

matrix_to_vector

sympy.vector.**matrix_to_vector**(*matrix*, *system*)

Converts a vector in matrix form to a Vector instance.

It is assumed that the elements of the Matrix represent the measure numbers of the components of the vector along basis vectors of 'system'.

Parameters

matrix : SymPy Matrix, Dimensions: (3, 1)

The matrix to be converted to a vector

system : CoordSys3D

The coordinate system the vector is to be defined in

Examples

```
>>> from sympy import ImmutableMatrix as Matrix
>>> m = Matrix([1, 2, 3])
>>> from sympy.vector import CoordSys3D, matrix_to_vector
>>> C = CoordSys3D('C')
>>> v = matrix_to_vector(m, C)
>>> v
C.i + 2*C.j + 3*C.k
>>> v.to_matrix(C) == m
True
```

express

sympy.vector.**express**(*expr*, *system*, *system2=None*, *variables=False*)

Global function for 'express' functionality.

Re-expresses a Vector, Dyadic or scalar(sympyfiable) in the given coordinate system.

If 'variables' is True, then the coordinate variables (base scalars) of other coordinate systems present in the vector/scalar field or dyadic are also substituted in terms of the base scalars of the given system.

Parameters

expr : Vector/Dyadic/scalar(sympyfiable)

The expression to re-express in CoordSys3D 'system'

system: CoordSys3D

The coordinate system the expr is to be expressed in

system2: CoordSys3D

The other coordinate system required for re-expression (only for a Dyadic Expr)

variables : boolean

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

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy import Symbol, cos, sin
>>> N = CoordSys3D('N')
>>> q = Symbol('q')
>>> B = N.orient_new_axis('B', q, N.k)
>>> from sympy.vector import express
>>> express(B.i, N)
(cos(q))*N.i + (sin(q))*N.j
>>> express(N.x, B, variables=True)
B.x*cos(q) - B.y*sin(q)
>>> d = N.i.outer(N.i)
>>> express(d, B, N) == (cos(q))*(B.i|N.i) + (-sin(q))*(B.j|N.i)
True
```

curl

sympy.vector.**curl**(*vect*, *doit=True*)

Returns the curl of a vector field computed wrt the base scalars of the given coordinate system.

Parameters

vect : Vector

The vector operand

doit : bool

If True, the result is returned after calling .doit() on each component. Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, curl
>>> R = CoordSys3D('R')
>>> v1 = R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> curl(v1)
0
>>> v2 = R.x*R.y*R.z*R.i
>>> curl(v2)
R.x*R.y*R.j + (-R.x*R.z)*R.k
```

divergence

sympy.vector.**divergence**(*vect*, *doit=True*)

Returns the divergence of a vector field computed wrt the base scalars of the given coordinate system.

Parameters

vector : Vector

The vector operand

doit : bool

If True, the result is returned after calling .doit() on each component.
Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, divergence
>>> R = CoordSys3D('R')
>>> v1 = R.x*R.y*R.z * (R.i+R.j+R.k)
```

```
>>> divergence(v1)
R.x*R.y + R.x*R.z + R.y*R.z
>>> v2 = 2*R.y*R.z*R.j
>>> divergence(v2)
2*R.z
```

gradient

sympy.vector.**gradient**(*scalar_field*, *doit=True*)

Returns the vector gradient of a scalar field computed wrt the base scalars of the given coordinate system.

Parameters

scalar_field : SymPy Expr

The scalar field to compute the gradient of

doit : bool

If True, the result is returned after calling .doit() on each component.
Else, the returned expression contains Derivative instances

Examples

```
>>> from sympy.vector import CoordSys3D, gradient
>>> R = CoordSys3D('R')
>>> s1 = R.x*R.y*R.z
>>> gradient(s1)
R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k
>>> s2 = 5*R.x**2*R.z
>>> gradient(s2)
10*R.x*R.z*R.i + 5*R.x**2*R.k
```

is_conservative

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

Checks if a field is conservative.

Parameters

field : Vector

The field to check for conservative property

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy.vector import is_conservative
>>> R = CoordSys3D('R')
>>> is_conservative(R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k)
True
>>> is_conservative(R.z*R.j)
False
```

is_solenoidal

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

Checks if a field is solenoidal.

Parameters

field : Vector

The field to check for solenoidal property

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy.vector import is_solenoidal
>>> R = CoordSys3D('R')
>>> is_solenoidal(R.y*R.z*R.i + R.x*R.z*R.j + R.x*R.y*R.k)
True
>>> is_solenoidal(R.y * R.j)
False
```

scalar_potential

sympy.vector.**scalar_potential**(*field*, *coord_sys*)

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

Parameters

field : Vector

The vector field whose scalar potential function is to be calculated

coord_sys : CoordSys3D

The coordinate system to do the calculation in

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy.vector import scalar_potential, gradient
>>> R = CoordSys3D('R')
>>> scalar_potential(R.k, R) == R.z
True
>>> scalar_field = 2*R.x**2*R.y*R.z
>>> grad_field = gradient(scalar_field)
>>> scalar_potential(grad_field, R)
2*R.x**2*R.y*R.z
```

scalar_potential_difference

sympy.vector.**scalar_potential_difference**(*field*, *coord_sys*, *point1*, *point2*)

Returns the scalar potential difference between two points in a certain coordinate system, 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 point2) - (potential at point1)

The position vectors of the two Points are calculated wrt the origin of the coordinate system provided.

Parameters

field : Vector/Expr

The field to calculate wrt

coord_sys : CoordSys3D

The coordinate system to do the calculations in

point1 : Point

The initial Point in given coordinate system

position2 : Point

The second Point in the given coordinate system

Examples

```
>>> from sympy.vector import CoordSys3D
>>> from sympy.vector import scalar_potential_difference
>>> R = CoordSys3D('R')
>>> P = R.origin.locate_new('P', R.x*R.i + R.y*R.j + R.z*R.k)
>>> vectfield = 4*R.x*R.y*R.i + 2*R.x**2*R.j
>>> scalar_potential_difference(vectfield, R, R.origin, P)
2*R.x**2*R.y
>>> Q = R.origin.locate_new('O', 3*R.i + R.j + 2*R.k)
>>> scalar_potential_difference(vectfield, R, P, Q)
-2*R.x**2*R.y + 18
```

vector_integrate

sympy.vector.integrals.**vector_integrate**(*field*, **region*)

Compute the integral of a vector/scalar field over a a region or a set of parameters.

Examples

```
>>> from sympy.vector import CoordSys3D, ParametricRegion, vector_
↪integrate
>>> from sympy.abc import x, y, t
>>> C = CoordSys3D('C')
```

```
>>> region = ParametricRegion((t, t**2), (t, 1, 5))
>>> vector_integrate(C.x*C.i, region)
12
```

Integrals over some objects of geometry module can also be calculated.

```
>>> from sympy.geometry import Point, Circle, Triangle
>>> c = Circle(Point(0, 2), 5)
>>> vector_integrate(C.x**2 + C.y**2, c)
290*pi
```

(continues on next page)

(continued from previous page)

```
>>> triangle = Triangle(Point(-2, 3), Point(2, 3), Point(0, 5))
>>> vector_integrate(3*C.x**2*C.y*C.i + C.j, triangle)
-8
```

Integrals over some simple implicit regions can be computed. But in most cases, it takes too long to compute over them. This is due to the expressions of parametric representation becoming large.

```
>>> from sympy.vector import ImplicitRegion
>>> c2 = ImplicitRegion((x, y), (x - 2)**2 + (y - 1)**2 - 9)
>>> vector_integrate(1, c2)
6*pi
```

Integral of fields with respect to base scalars:

```
>>> vector_integrate(12*C.y**3, (C.y, 1, 3))
240
>>> vector_integrate(C.x**2*C.z, C.x)
C.x**3*C.z/3
>>> vector_integrate(C.x*C.i - C.y*C.k, C.x)
(Integral(C.x, C.x))*C.i + (Integral(-C.y, C.x))*C.k
>>> _.doit()
C.x**2/2*C.i + (-C.x*C.y)*C.k
```


References for Vector

5.8.5 Number Theory

Contents

Ntheory Class Reference

class sympy.ntheory.generate.**Sieve**

An infinite list of prime numbers, implemented as a dynamically growing sieve of Eratosthenes. When a lookup is requested involving an odd number that has not been sieved, the sieve is automatically extended up to that number.

Examples

```
>>> from sympy import sieve
>>> sieve._reset() # this line for doctest only
>>> 25 in sieve
False
>>> sieve._list
array('l', [2, 3, 5, 7, 11, 13, 17, 19, 23])
```

extend(*n*)

Grow the sieve to cover all primes <= n (a real number).