Explanation

The index to substitute is the index with less information regarding fermi level. If indices contain the same information, 'a' is preferred before 'b'.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).killable_index
p
>>> KroneckerDelta(p, a).killable_index
p
>>> KroneckerDelta(i, j).killable_index
j
```

See also:

preferred_index (page 1557)

property preferred_index

Returns the index which is preferred to keep in the final expression.

Explanation

The preferred index is the index with more information regarding fermi level. If indices contain the same information, 'a' is preferred before 'b'.

Examples

```
>>> from sympy import KroneckerDelta, Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> j = Symbol('j', below_fermi=True)
>>> p = Symbol('p')
>>> KroneckerDelta(p, i).preferred_index
i
>>> KroneckerDelta(p, a).preferred_index
a
>>> KroneckerDelta(i, j).preferred_index
i
```

See also:

killable_index (page 1556)

class sympy.physics.secondquant.**NO**(*arg*)

This Object is used to represent normal ordering brackets.

i.e. {abcd} sometimes written :abcd:

Explanation

Applying the function NO(arg) to an argument means that all operators in the argument will be assumed to anticommute, and have vanishing contractions. This allows an immediate reordering to canonical form upon object creation.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import NO, F, Fd
>>> p,q = symbols('p,q')
>>> NO(Fd(p)*F(q))
NO(CreateFermion(p)*AnnihilateFermion(q))
>>> NO(F(q)*Fd(p))
-NO(CreateFermion(p)*AnnihilateFermion(q))
```

Note

If you want to generate a normal ordered equivalent of an expression, you should use the function wicks(). This class only indicates that all operators inside the brackets anticommute, and have vanishing contractions. Nothing more, nothing less.

doit(***hints*)

Either removes the brackets or enables complex computations in its arguments.

Examples

```
>>> from sympy.physics.secondquant import NO, Fd, F
>>> from textwrap import fill
>>> from sympy import symbols, Dummy
>>> p,q = symbols('p,q', cls=Dummy)
>>> print(fill(str(NO(Fd(p)*F(q)).doit())))
KroneckerDelta(_a, _p)*KroneckerDelta(_a,
_q)*CreateFermion(_a)*AnnihilateFermion(_a) + KroneckerDelta(_a,
_p)*KroneckerDelta(_i, _q)*CreateFermion(_a)*AnnihilateFermion(_i) -
KroneckerDelta(_a, _q)*KroneckerDelta(_i,
_p)*AnnihilateFermion(_a)*CreateFermion(_i) - KroneckerDelta(_i,
_p)*KroneckerDelta(_i, _q)*AnnihilateFermion(_i)*CreateFermion(_i)
```

get_subNO(*i*)

Returns a NO() without FermionicOperator at index i.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import F, NO
>>> p, q, r = symbols('p,q,r')
```

```
>>> NO(F(p)*F(q)*F(r)).get_subNO(1)
NO(AnnihilateFermion(p)*AnnihilateFermion(r))
```

property has_q_annihilators

Return 0 if the rightmost argument of the first argument is a not a q_annihilator, else 1 if it is above fermi or -1 if it is below fermi.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import NO, F, Fd
```

```
>>> a = symbols('a', above_fermi=True)
>>> i = symbols('i', below_fermi=True)
>>> NO(Fd(a)*Fd(i)).has_q_annihilators
-1
>>> NO(F(i)*F(a)).has_q_annihilators
1
>>> NO(Fd(a)*F(i)).has_q_annihilators
0
```

property has_q_creators

Return 0 if the leftmost argument of the first argument is a not a q_creator, else 1 if it is above fermi or -1 if it is below fermi.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import NO, F, Fd
```

```
>>> a = symbols('a', above_fermi=True)
>>> i = symbols('i', below_fermi=True)
>>> NO(Fd(a)*Fd(i)).has_q_creators
1
>>> NO(F(i)*F(a)).has_q_creators
-1
>>> NO(Fd(i)*F(a)).has_q_creators
0
```

iter_q_annihilators()

Iterates over the annihilation operators.

Examples

```
>>> from sympy import symbols
>>> i, j = symbols('i j', below_fermi=True)
>>> a, b = symbols('a b', above_fermi=True)
>>> from sympy.physics.secondquant import NO, F, Fd
>>> no = NO(Fd(a)*F(i)*F(b)*Fd(j))
```

```
>>> no.iter_q_creators()
<generator object... at 0x...>
>>> list(no.iter_q_creators())
[0, 1]
>>> list(no.iter_q_annihilators())
[3, 2]
```

iter_q_creators()

Iterates over the creation operators.

Examples

```
>>> from sympy import symbols
>>> i, j = symbols('i j', below_fermi=True)
>>> a, b = symbols('a b', above_fermi=True)
>>> from sympy.physics.secondquant import NO, F, Fd
>>> no = NO(Fd(a)*F(i)*F(b)*Fd(j))
```

```
>>> no.iter_q_creators()
<generator object... at 0x...>
>>> list(no.iter_q_creators())
[0, 1]
>>> list(no.iter_q_annihilators())
[3, 2]
```

class sympy.physics.secondquant.**PermutationOperator**(*i*, *j*)

Represents the index permutation operator P(ij).

P(ij)*f(i)*g(j) = f(i)*g(j) - f(j)*g(i)

get_permuted(*expr*)

Returns -expr with permuted indices.

Explanation

```
>>> from sympy import symbols, Function
>>> from sympy.physics.secondquant import PermutationOperator
>>> p,q = symbols('p,q')
>>> f = Function('f')
>>> PermutationOperator(p,q).get_permuted(f(p,q))
-f(q, p)
```

class sympy.physics.secondquant.**VarBosonicBasis**(*n_max*)

A single state, variable particle number basis set.

Examples

```
>>> from sympy.physics.secondquant import VarBosonicBasis
>>> b = VarBosonicBasis(5)
>>> b
[FockState((0,)), FockState((1,)), FockState((2,)),
 FockState((3,)), FockState((4,))]
```

index(*state*)

Returns the index of state in basis.

Examples

```
>>> from sympy.physics.secondquant import VarBosonicBasis
>>> b = VarBosonicBasis(3)
>>> state = b.state(1)
>>> b
[FockState((0,)), FockState((1,)), FockState((2,))]
>>> state
FockStateBosonKet((1,))
>>> b.index(state)
1
```

state(*i*)

The state of a single basis.

Examples

```
>>> from sympy.physics.secondquant import VarBosonicBasis
>>> b = VarBosonicBasis(5)
>>> b.state(3)
FockStateBosonKet((3,))
```

sympy.physics.secondquant.**apply_operators**(*e*)

Take a SymPy expression with operators and states and apply the operators.

Examples

```
>>> from sympy.physics.secondquant import apply_operators
>>> from sympy import sympify
>>> apply_operators(sympify(3)+4)
7
```

sympy.physics.secondquant.**contraction**(*a*, *b*)

Calculates contraction of Fermionic operators a and b.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import F, Fd, contraction
>>> p, q = symbols('p,q')
>>> a, b = symbols('a,b', above_fermi=True)
>>> i, j = symbols('i,j', below_fermi=True)
```

A contraction is non-zero only if a quasi-creator is to the right of a quasi-annihilator:

```
>>> contraction(F(a),Fd(b))
KroneckerDelta(a, b)
>>> contraction(Fd(i),F(j))
KroneckerDelta(i, j)
```

For general indices a non-zero result restricts the indices to below/above the fermi surface:

```
>>> contraction(Fd(p),F(q))
KroneckerDelta(_i, q)*KroneckerDelta(p, q)
>>> contraction(F(p),Fd(q))
KroneckerDelta(_a, q)*KroneckerDelta(p, q)
```

Two creators or two annihilators always vanishes:

```
>>> contraction(F(p),F(q))
0
>>> contraction(Fd(p),Fd(q))
0
```

sympy.physics.secondquant.**evaluate_deltas**(*e*)

We evaluate KroneckerDelta symbols in the expression assuming Einstein summation.

Explanation

If one index is repeated it is summed over and in effect substituted with the other one. If both indices are repeated we substitute according to what is the preferred index. this is determined by KroneckerDelta.preferred_index and KroneckerDelta.killable_index.

In case there are no possible substitutions or if a substitution would imply a loss of information, nothing is done.

In case an index appears in more than one KroneckerDelta, the resulting substitution depends on the order of the factors. Since the ordering is platform dependent, the literal expression resulting from this function may be hard to predict.

Examples

We assume the following:

```
>>> from sympy import symbols, Function, Dummy, KroneckerDelta
>>> from sympy.physics.secondquant import evaluate_deltas
>>> i,j = symbols('i j', below_fermi=True, cls=Dummy)
>>> a,b = symbols('a b', above_fermi=True, cls=Dummy)
>>> p,q = symbols('p q', cls=Dummy)
>>> f = Function('f')
>>> t = Function('t')
```

The order of preference for these indices according to KroneckerDelta is (a, b, i, j, p, q).

Trivial cases:

```
>>> evaluate_deltas(KroneckerDelta(i,j)*f(i))       # d_ij f(i) -> f(j)
f(_j)
>>> evaluate_deltas(KroneckerDelta(i,j)*f(j))       # d_ij f(j) -> f(i)
f(_i)
>>> evaluate_deltas(KroneckerDelta(i,p)*f(p))       # d_ip f(p) -> f(i)
f(_i)
>>> evaluate_deltas(KroneckerDelta(q,p)*f(p))       # d_qp f(p) -> f(q)
f(_q)
>>> evaluate_deltas(KroneckerDelta(q,p)*f(q))       # d_qp f(q) -> f(p)
f(_p)
```

More interesting cases:

```
>>> evaluate_deltas(KroneckerDelta(i,p)*t(a,i)*f(p,q))
f(_i, _q)*t(_a, _i)
>>> evaluate_deltas(KroneckerDelta(a,p)*t(a,i)*f(p,q))
f(_a, _q)*t(_a, _i)
>>> evaluate_deltas(KroneckerDelta(p,q)*f(p,q))
f(_p, _p)
```

Finally, here are some cases where nothing is done, because that would imply a loss of information:

```
>>> evaluate_deltas(KroneckerDelta(i,p)*f(q))
f(_q)*KroneckerDelta(_i, _p)
```

(continues on next page)

(continued from previous page)

```
>>> evaluate_deltas(KroneckerDelta(i,p)*f(i))
f(_i)*KroneckerDelta(_i, _p)
```

sympy.physics.secondquant.**matrix_rep**(*op*, *basis*)

Find the representation of an operator in a basis.

Examples

```
>>> from sympy.physics.secondquant import VarBosonicBasis, B, matrix_rep
>>> b = VarBosonicBasis(5)
>>> o = B(0)
>>> matrix_rep(o, b)
Matrix([
[0, 1,       0,       0, 0],
[0, 0, sqrt(2),       0, 0],
[0, 0,       0, sqrt(3), 0],
[0, 0,       0,       0, 2],
[0, 0,       0,       0, 0]])
```

sympy.physics.secondquant.**simplify_index_permutations**(*expr*, *permutation_operators*)

Performs simplification by introducing PermutationOperators where appropriate.

Explanation

Schematically:

[abij] - [abji] - [baij] + [baji] -> P(ab)*P(ij)*[abij]

permutation_operators is a list of PermutationOperators to consider.

If permutation_operators=[P(ab),P(ij)] we will try to introduce the permutation operators P(ij) and P(ab) in the expression. If there are other possible simplifications, we ignore them.

```
>>> from sympy import symbols, Function
>>> from sympy.physics.secondquant import simplify_index_permutations
>>> from sympy.physics.secondquant import PermutationOperator
>>> p,q,r,s = symbols('p,q,r,s')
>>> f = Function('f')
>>> g = Function('g')
```

```
>>> expr = f(p)*g(q) - f(q)*g(p); expr
f(p)*g(q) - f(q)*g(p)
>>> simplify_index_permutations(expr,[PermutationOperator(p,q)])
f(p)*g(q)*PermutationOperator(p, q)
```

```
>>> PermutList = [PermutationOperator(p,q),PermutationOperator(r,s)]
>>> expr = f(p,r)*g(q,s) - f(q,r)*g(p,s) + f(q,s)*g(p,r) - f(p,s)*g(q,r)
>>> simplify_index_permutations(expr,PermutList)
f(p, r)*g(q, s)*PermutationOperator(p, q)*PermutationOperator(r, s)
```

sympy.physics.secondquant.**substitute_dummies**(*expr*, *new_indices=False*, *pretty_indices={}*)

Collect terms by substitution of dummy variables.

Explanation

This routine allows simplification of Add expressions containing terms which differ only due to dummy variables.

The idea is to substitute all dummy variables consistently depending on the structure of the term. For each term, we obtain a sequence of all dummy variables, where the order is determined by the index range, what factors the index belongs to and its position in each factor. See _get_ordered_dummies() for more information about the sorting of dummies. The index sequence is then substituted consistently in each term.

Examples

```
>>> from sympy import symbols, Function, Dummy
>>> from sympy.physics.secondquant import substitute_dummies
>>> a,b,c,d = symbols('a b c d', above_fermi=True, cls=Dummy)
>>> i,j = symbols('i j', below_fermi=True, cls=Dummy)
>>> f = Function('f')
```

```
>>> expr = f(a,b) + f(c,d); expr
f(_a, _b) + f(_c, _d)
```

Since a, b, c and d are equivalent summation indices, the expression can be simplified to a single term (for which the dummy indices are still summed over)

```
>>> substitute_dummies(expr)
2*f(_a, _b)
```

Controlling output:

By default the dummy symbols that are already present in the expression will be reused in a different permutation. However, if new_indices=True, new dummies will be generated and inserted. The keyword 'pretty_indices' can be used to control this generation of new symbols.

By default the new dummies will be generated on the form i_1, i_2, a_1, etc. If you supply a dictionary with key:value pairs in the form:

{ index_group: string_of_letters }

The letters will be used as labels for the new dummy symbols. The index_groups must be one of 'above', 'below' or 'general'.

```
>>> expr = f(a,b,i,j)
>>> my_dummies = { 'above':'st', 'below':'uv' }
>>> substitute_dummies(expr, new_indices=True, pretty_indices=my_dummies)
f(_s, _t, _u, _v)
```

If we run out of letters, or if there is no keyword for some index_group the default dummy generator will be used as a fallback:

```
>>> p,q = symbols('p q', cls=Dummy)  # general indices
>>> expr = f(p,q)
>>> substitute_dummies(expr, new_indices=True, pretty_indices=my_dummies)
f(_p_0, _p_1)
```

sympy.physics.secondquant.**wicks**(*e*, ***kw_args*)

Returns the normal ordered equivalent of an expression using Wicks Theorem.

Examples

```
>>> from sympy import symbols, Dummy
>>> from sympy.physics.secondquant import wicks, F, Fd
>>> p, q, r = symbols('p,q,r')
>>> wicks(Fd(p)*F(q))
KroneckerDelta(_i, q)*KroneckerDelta(p, q) +␣
↪NO(CreateFermion(p)*AnnihilateFermion(q))
```

By default, the expression is expanded:

```
>>> wicks(F(p)*(F(q)+F(r)))
NO(AnnihilateFermion(p)*AnnihilateFermion(q)) +␣
↪NO(AnnihilateFermion(p)*AnnihilateFermion(r))
```

With the keyword 'keep_only_fully_contracted=True', only fully contracted terms are returned.

By request, the result can be simplified in the following order:

– KroneckerDelta functions are evaluated – Dummy variables are substituted consistently across terms

```
>>> p, q, r = symbols('p q r', cls=Dummy)
>>> wicks(Fd(p)*(F(q)+F(r)), keep_only_fully_contracted=True)
KroneckerDelta(_i, _q)*KroneckerDelta(_p, _q) + KroneckerDelta(_i, _
↪r)*KroneckerDelta(_p, _r)
```

Wigner Symbols

Wigner, Clebsch-Gordan, Racah, and Gaunt coefficients

Collection of functions for calculating Wigner 3j, 6j, 9j, Clebsch-Gordan, Racah as well as Gaunt coefficients exactly, all evaluating to a rational number times the square root of a rational number [Rasch03].

Please see the description of the individual functions for further details and examples.

References

Credits and Copyright

This code was taken from Sage with the permission of all authors:

https://groups.google.com/forum/#!topic/sage-devel/M4NZdu-7O38

Authors

- Jens Rasch (2009-03-24): initial version for Sage
- Jens Rasch (2009-05-31): updated to sage-4.0
- Oscar Gerardo Lazo Arjona (2017-06-18): added Wigner D matrices

Copyright (C) 2008 Jens Rasch <jyr2000@gmail.com>

`sympy.physics.wigner.`**`clebsch_gordan`**(*j_1*, *j_2*, *j_3*, *m_1*, *m_2*, *m_3*)

Calculates the Clebsch-Gordan coefficient. $\langle j_1 m_1\ j_2 m_2 | j_3 m_3 \rangle$.

The reference for this function is [Edmonds74].

Parameters

j_1, j_2, j_3, m_1, m_2, m_3 :

Integer or half integer.

Returns

Rational number times the square root of a rational number.

Examples

```
>>> from sympy import S
>>> from sympy.physics.wigner import clebsch_gordan
>>> clebsch_gordan(S(3)/2, S(1)/2, 2, S(3)/2, S(1)/2, 2)
1
>>> clebsch_gordan(S(3)/2, S(1)/2, 1, S(3)/2, -S(1)/2, 1)
sqrt(3)/2
>>> clebsch_gordan(S(3)/2, S(1)/2, 1, -S(1)/2, S(1)/2, 0)
-sqrt(2)/2
```

Notes

The Clebsch-Gordan coefficient will be evaluated via its relation to Wigner 3j symbols:

$$\langle j_1 m_1\ j_2 m_2 | j_3 m_3 \rangle = (-1)^{j_1 - j_2 + m_3} \sqrt{2 j_3 + 1}\, \mathrm{Wigner3j}(j_1, j_2, j_3, m_1, m_2, -m_3)$$

See also the documentation on Wigner 3j symbols which exhibit much higher symmetry relations than the Clebsch-Gordan coefficient.

Authors

- Jens Rasch (2009-03-24): initial version

sympy.physics.wigner.**dot_rot_grad_Ynm**(*j*, *p*, *l*, *m*, *theta*, *phi*)

Returns dot product of rotational gradients of spherical harmonics.

Explanation

This function returns the right hand side of the following expression:

$$\vec{R}Y^p_j \cdot \vec{R}Y^m_l = (-1)^{m+p} \sum_{k=|l-j|}^{l+j} Y^{m+p}_k * \alpha_{l,m,j,p,k} * \frac{1}{2}(k^2 - j^2 - l^2 + k - j - l)$$

Arguments

j, p, l, m .... indices in spherical harmonics (expressions or integers) theta, phi .... angle arguments in spherical harmonics

Example

```
>>> from sympy import symbols
>>> from sympy.physics.wigner import dot_rot_grad_Ynm
>>> theta, phi = symbols("theta phi")
>>> dot_rot_grad_Ynm(3, 2, 2, 0, theta, phi).doit()
3*sqrt(55)*Ynm(5, 2, theta, phi)/(11*sqrt(pi))
```

sympy.physics.wigner.**gaunt**(*l_1*, *l_2*, *l_3*, *m_1*, *m_2*, *m_3*, *prec=None*)

Calculate the Gaunt coefficient.

Parameters

l_1, l_2, l_3, m_1, m_2, m_3 :

Integer.

prec - precision, default: ``None``.

Providing a precision can drastically speed up the calculation.

Returns

Rational number times the square root of a rational number

(if `prec=None`), or real number if a precision is given.

Explanation

The Gaunt coefficient is defined as the integral over three spherical harmonics:

$$\begin{aligned}\text{Gaunt}(l_1,l_2,l_3,m_1,m_2,m_3) &= \int Y_{l_1,m_1}(\Omega) Y_{l_2,m_2}(\Omega) Y_{l_3,m_3}(\Omega)\, d\Omega \\ &= \sqrt{\frac{(2l_1+1)(2l_2+1)(2l_3+1)}{4\pi}}\,\text{Wigner3j}(l_1,l_2,l_3,0,0,0)\,\text{Wigner3j}(l_1,l_2,l_3,m_1,m_2,m_3)\end{aligned}$$

Examples

```
>>> from sympy.physics.wigner import gaunt
>>> gaunt(1,0,1,1,0,-1)
-1/(2*sqrt(pi))
>>> gaunt(1000,1000,1200,9,3,-12).n(64)
0.00689500421922113448...
```

It is an error to use non-integer values for l and m:

```
sage: gaunt(1.2,0,1.2,0,0,0)
Traceback (most recent call last):
...
ValueError: l values must be integer
sage: gaunt(1,0,1,1.1,0,-1.1)
Traceback (most recent call last):
...
ValueError: m values must be integer
```


Notes

The Gaunt coefficient obeys the following symmetry rules:

- invariant under any permutation of the columns

$$\begin{aligned}Y(l_1,l_2,l_3,m_1,m_2,m_3) &= Y(l_3,l_1,l_2,m_3,m_1,m_2) \\ &= Y(l_2,l_3,l_1,m_2,m_3,m_1) \\ &= Y(l_3,l_2,l_1,m_3,m_2,m_1) \\ &= Y(l_1,l_3,l_2,m_1,m_3,m_2) \\ &= Y(l_2,l_1,l_3,m_2,m_1,m_3)\end{aligned}$$

- invariant under space inflection, i.e.

$$Y(l_1,l_2,l_3,m_1,m_2,m_3) = Y(l_1,l_2,l_3,-m_1,-m_2,-m_3)$$

- symmetric with respect to the 72 Regge symmetries as inherited for the $3j$ symbols [Regge58]
- zero for l_1, l_2, l_3 not fulfilling triangle relation
- zero for violating any one of the conditions: $l_1 \geq |m_1|$, $l_2 \geq |m_2|$, $l_3 \geq |m_3|$
- non-zero only for an even sum of the l_i, i.e. $L = l_1 + l_2 + l_3 = 2n$ for n in $\mathbb{N}$

Algorithms

This function uses the algorithm of [Liberatodebrito82] to calculate the value of the Gaunt coefficient exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

Authors

Jens Rasch (2009-03-24): initial version for Sage.

sympy.physics.wigner.**racah**(*aa*, *bb*, *cc*, *dd*, *ee*, *ff*, *prec=None*)

Calculate the Racah symbol $W(a,b,c,d;e,f)$.

Parameters

a, ..., f :

Integer or half integer.

prec :

Precision, default: `None`. Providing a precision can drastically speed up the calculation.

Returns

Rational number times the square root of a rational number

(if `prec=None`), or real number if a precision is given.

Examples

```
>>> from sympy.physics.wigner import racah
>>> racah(3,3,3,3,3,3)
-1/14
```

Notes

The Racah symbol is related to the Wigner 6j symbol:

$$\text{Wigner6j}(j_1,j_2,j_3,j_4,j_5,j_6) = (-1)^{j_1+j_2+j_4+j_5} W(j_1,j_2,j_5,j_4,j_3,j_6)$$

Please see the 6j symbol for its much richer symmetries and for additional properties.

Algorithm

This function uses the algorithm of [Edmonds74] to calculate the value of the 6j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

Authors

- Jens Rasch (2009-03-24): initial version

sympy.physics.wigner.**wigner_3j**(*j_1, j_2, j_3, m_1, m_2, m_3*)

Calculate the Wigner 3j symbol Wigner3j$(j_1, j_2, j_3, m_1, m_2, m_3)$.

Parameters

j_1, j_2, j_3, m_1, m_2, m_3 :

Integer or half integer.

Returns

Rational number times the square root of a rational number.

Examples

```
>>> from sympy.physics.wigner import wigner_3j
>>> wigner_3j(2, 6, 4, 0, 0, 0)
sqrt(715)/143
>>> wigner_3j(2, 6, 4, 0, 0, 1)
0
```

It is an error to have arguments that are not integer or half integer values:

```
sage: wigner_3j(2.1, 6, 4, 0, 0, 0)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer
sage: wigner_3j(2, 6, 4, 1, 0, -1.1)
Traceback (most recent call last):
...
ValueError: m values must be integer or half integer
```

Notes

The Wigner 3j symbol obeys the following symmetry rules:

- invariant under any permutation of the columns (with the exception of a sign change where $J := j_1 + j_2 + j_3$):

$$\begin{aligned}\text{Wigner3j}(j_1, j_2, j_3, m_1, m_2, m_3) &= \text{Wigner3j}(j_3, j_1, j_2, m_3, m_1, m_2) \\ &= \text{Wigner3j}(j_2, j_3, j_1, m_2, m_3, m_1) \\ &= (-1)^J \text{Wigner3j}(j_3, j_2, j_1, m_3, m_2, m_1) \\ &= (-1)^J \text{Wigner3j}(j_1, j_3, j_2, m_1, m_3, m_2) \\ &= (-1)^J \text{Wigner3j}(j_2, j_1, j_3, m_2, m_1, m_3)\end{aligned}$$

- invariant under space inflection, i.e.

$$\text{Wigner3j}(j_1, j_2, j_3, m_1, m_2, m_3) = (-1)^J \text{Wigner3j}(j_1, j_2, j_3, -m_1, -m_2, -m_3)$$

- symmetric with respect to the 72 additional symmetries based on the work by [Regge58]
- zero for j_1, j_2, j_3 not fulfilling triangle relation
- zero for $m_1 + m_2 + m_3 \neq 0$
- zero for violating any one of the conditions $j_1 \geq |m_1|$, $j_2 \geq |m_2|$, $j_3 \geq |m_3|$

Algorithm

This function uses the algorithm of [Edmonds74] to calculate the value of the 3j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

Authors

- Jens Rasch (2009-03-24): initial version

sympy.physics.wigner.**wigner_6j**(*j_1, j_2, j_3, j_4, j_5, j_6, prec=None*)

Calculate the Wigner 6j symbol Wigner6j$(j_1, j_2, j_3, j_4, j_5, j_6)$.

Parameters

j_1, ..., j_6 :

Integer or half integer.

prec :

Precision, default: None. Providing a precision can drastically speed up the calculation.

Returns

Rational number times the square root of a rational number

(if prec=None), or real number if a precision is given.

Examples

```
>>> from sympy.physics.wigner import wigner_6j
>>> wigner_6j(3,3,3,3,3,3)
-1/14
>>> wigner_6j(5,5,5,5,5,5)
1/52
```

It is an error to have arguments that are not integer or half integer values or do not fulfill the triangle relation:

```
sage: wigner_6j(2.5,2.5,2.5,2.5,2.5,2.5)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the␣
↪triangle relation
```

(continues on next page)

(continued from previous page)

```
sage: wigner_6j(0.5,0.5,1.1,0.5,0.5,1.1)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the␣
↪triangle relation
```

Notes

The Wigner 6j symbol is related to the Racah symbol but exhibits more symmetries as detailed below.

$$\text{Wigner6j}(j_1,j_2,j_3,j_4,j_5,j_6) = (-1)^{j_1+j_2+j_4+j_5} W(j_1,j_2,j_5,j_4,j_3,j_6)$$

The Wigner 6j symbol obeys the following symmetry rules:

- Wigner 6j symbols are left invariant under any permutation of the columns:

$$\begin{aligned}\text{Wigner6j}(j_1,j_2,j_3,j_4,j_5,j_6) &= \text{Wigner6j}(j_3,j_1,j_2,j_6,j_4,j_5)\\ &= \text{Wigner6j}(j_2,j_3,j_1,j_5,j_6,j_4)\\ &= \text{Wigner6j}(j_3,j_2,j_1,j_6,j_5,j_4)\\ &= \text{Wigner6j}(j_1,j_3,j_2,j_4,j_6,j_5)\\ &= \text{Wigner6j}(j_2,j_1,j_3,j_5,j_4,j_6)\end{aligned}$$

- They are invariant under the exchange of the upper and lower arguments in each of any two columns, i.e.

  $\text{Wigner6j}(j_1,j_2,j_3,j_4,j_5,j_6) = \text{Wigner6j}(j_1,j_5,j_6,j_4,j_2,j_3) = \text{Wigner6j}(j_4,j_2,j_6,j_1,j_5,j_3) = \text{Wigner6j}(j_4,j_5$

- additional 6 symmetries [Regge59] giving rise to 144 symmetries in total
- only non-zero if any triple of j's fulfill a triangle relation

Algorithm

This function uses the algorithm of [Edmonds74] to calculate the value of the 6j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

`sympy.physics.wigner.`**`wigner_9j`**(*j_1, j_2, j_3, j_4, j_5, j_6, j_7, j_8, j_9, prec=None*)

Calculate the Wigner 9j symbol $\text{Wigner9j}(j_1,j_2,j_3,j_4,j_5,j_6,j_7,j_8,j_9)$.

Parameters

j_1, ..., j_9 :

Integer or half integer.

prec : precision, default

`None`. Providing a precision can drastically speed up the calculation.

Returns

Rational number times the square root of a rational number

(if `prec=None`), or real number if a precision is given.

Examples

```
>>> from sympy.physics.wigner import wigner_9j
>>> wigner_9j(1,1,1, 1,1,1, 1,1,0, prec=64) # ==1/18
0.05555555...
```

```
>>> wigner_9j(1/2,1/2,0, 1/2,3/2,1, 0,1,1, prec=64) # ==1/6
0.1666666...
```

It is an error to have arguments that are not integer or half integer values or do not fulfill the triangle relation:

```
sage: wigner_9j(0.5,0.5,0.5, 0.5,0.5,0.5, 0.5,0.5,0.5,prec=64)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the␣
↪triangle relation
sage: wigner_9j(1,1,1, 0.5,1,1.5, 0.5,1,2.5,prec=64)
Traceback (most recent call last):
...
ValueError: j values must be integer or half integer and fulfill the␣
↪triangle relation
```

Algorithm

This function uses the algorithm of [Edmonds74] to calculate the value of the 3j symbol exactly. Note that the formula contains alternating sums over large factorials and is therefore unsuitable for finite precision arithmetic and only useful for a computer algebra system [Rasch03].

sympy.physics.wigner.**wigner_d**(*J*, *alpha*, *beta*, *gamma*)

Return the Wigner D matrix for angular momentum J.

Returns

A matrix representing the corresponding Euler angle rotation(in the basis of eigenvectors of J_z).

$$\mathcal{D}_{\alpha\beta\gamma} = \exp\left(\frac{i\alpha}{\hbar}J_z\right)\exp\left(\frac{i\beta}{\hbar}J_y\right)\exp\left(\frac{i\gamma}{\hbar}J_z\right)$$

The components are calculated using the general form [Edmonds74], equation 4.1.12.

Explanation

J :
An integer, half-integer, or SymPy symbol for the total angular momentum of the angular momentum space being rotated.

alpha, beta, gamma - Real numbers representing the Euler.
Angles of rotation about the so-called vertical, line of nodes, and figure axes. See [Edmonds74].

Examples

The simplest possible example:

```
>>> from sympy.physics.wigner import wigner_d
>>> from sympy import Integer, symbols, pprint
>>> half = 1/Integer(2)
>>> alpha, beta, gamma = symbols("alpha, beta, gamma", real=True)
>>> pprint(wigner_d(half, alpha, beta, gamma), use_unicode=True)
⎡  i·α  i·γ             i·α  -i·γ          ⎤
⎢  ───  ───             ───  ────          ⎥
⎢   2    2     ⎛β⎞       2     2      ⎛β⎞  ⎥
⎢ e   ·e   ·cos⎜─⎟     e   ·e     ·sin⎜─⎟  ⎥
⎢              ⎝2⎠                    ⎝2⎠  ⎥
⎢                                          ⎥
⎢  -i·α   i·γ           -i·α   -i·γ        ⎥
⎢  ────   ──            ────   ────        ⎥
⎢    2     2     ⎛β⎞      2      2     ⎛β⎞  ⎥
⎢-e     ·e   ·sin⎜─⎟   e     ·e    ·cos⎜─⎟  ⎥
⎣                ⎝2⎠                    ⎝2⎠  ⎦
```

sympy.physics.wigner.**wigner_d_small**(*J*, *beta*)
Return the small Wigner d matrix for angular momentum J.

Returns
A matrix representing the corresponding Euler angle rotation(in the basis of eigenvectors of J_z).

$$\lceil_\beta = \exp\left(\frac{i\beta}{\hbar} J_y\right)$$

The components are calculated using the general form [Edmonds74], equation 4.1.15.

Explanation

J
: [An integer, half-integer, or SymPy symbol for the total angular] momentum of the angular momentum space being rotated.

beta
: [A real number representing the Euler angle of rotation about] the so-called line of nodes. See [Edmonds74].

Examples

```
>>> from sympy import Integer, symbols, pi, pprint
>>> from sympy.physics.wigner import wigner_d_small
>>> half = 1/Integer(2)
>>> beta = symbols("beta", real=True)
>>> pprint(wigner_d_small(half, beta), use_unicode=True)
⎡   ⎛β⎞      ⎛β⎞⎤
⎢cos⎜─⎟   sin⎜─⎟⎥
⎢   ⎝2⎠      ⎝2⎠⎥
⎢               ⎥
⎢    ⎛β⎞     ⎛β⎞⎥
⎢-sin⎜─⎟  cos⎜─⎟⎥
⎣    ⎝2⎠     ⎝2⎠⎦
```

```
>>> pprint(wigner_d_small(2*half, beta), use_unicode=True)
⎡        2⎛β⎞                ⎛β⎞  ⎛β⎞          2⎛β⎞     ⎤
⎢     cos ⎜─⎟        √2⋅sin⎜─⎟⋅cos⎜─⎟        sin ⎜─⎟     ⎥
⎢         ⎝2⎠              ⎝2⎠  ⎝2⎠           ⎝2⎠     ⎥
⎢                                                     ⎥
⎢       ⎛β⎞    ⎛β⎞       2⎛β⎞      2⎛β⎞        ⎛β⎞    ⎛β⎞⎥
⎢-√2⋅sin⎜─⎟⋅cos⎜─⎟  - sin ⎜─⎟ + cos ⎜─⎟  √2⋅sin⎜─⎟⋅cos⎜─⎟⎥
⎢       ⎝2⎠    ⎝2⎠        ⎝2⎠       ⎝2⎠        ⎝2⎠    ⎝2⎠⎥
⎢                                                     ⎥
⎢        2⎛β⎞                ⎛β⎞  ⎛β⎞          2⎛β⎞     ⎥
⎢     sin ⎜─⎟       -√2⋅sin⎜─⎟⋅cos⎜─⎟       cos ⎜─⎟     ⎥
⎣         ⎝2⎠              ⎝2⎠  ⎝2⎠           ⎝2⎠     ⎦
```

From table 4 in [Edmonds74]

```
>>> pprint(wigner_d_small(half, beta).subs({beta:pi/2}), use_
↪unicode=True)
⎡ √2   √2⎤
⎢ ──   ──⎥
⎢ 2    2 ⎥
⎢        ⎥
⎢-√2   √2⎥
⎢────  ──⎥
⎣ 2    2 ⎦
```

```
>>> pprint(wigner_d_small(2*half, beta).subs({beta:pi/2}),
... use_unicode=True)
⎡        √2       ⎤
⎢1/2     ──   1/2 ⎥
⎢        2        ⎥
⎢                 ⎥
⎢-√2          √2  ⎥
⎢────    0    ──  ⎥
⎢ 2           2   ⎥
⎢                 ⎥
⎢       -√2       ⎥
⎢1/2    ────  1/2 ⎥
⎣        2        ⎦
```

```
>>> pprint(wigner_d_small(3*half, beta).subs({beta:pi/2}),
... use_unicode=True)
⎡ √2    √6    √6   √2⎤
⎢ ──    ──    ──   ──⎥
⎢ 4     4     4    4 ⎥
⎢                    ⎥
⎢-√6   -√2    √2   √6⎥
⎢────  ────   ──   ──⎥
⎢ 4     4     4    4 ⎥
⎢                    ⎥
⎢ √6   -√2   -√2   √6⎥
⎢ ──   ────  ────  ──⎥
⎢ 4     4     4    4 ⎥
⎢                    ⎥
⎢-√2    √6   -√6   √2⎥
⎢────   ──   ────  ──⎥
⎣ 4     4     4    4 ⎦
```

```
>>> pprint(wigner_d_small(4*half, beta).subs({beta:pi/2}),
... use_unicode=True)
⎡              √6             ⎤
⎢1/4   1/2     ──    1/2   1/4⎥
⎢              4              ⎥
⎢                             ⎥
⎢-1/2  -1/2    0     1/2   1/2⎥
⎢                             ⎥
⎢ √6                     √6   ⎥
⎢ ──     0    -1/2    0    ──  ⎥
⎢ 4                      4    ⎥
⎢                             ⎥
⎢-1/2  1/2     0     -1/2  1/2⎥
⎢                             ⎥
⎢              √6             ⎥
⎢1/4   -1/2    ──    -1/2  1/4⎥
⎣              4              ⎦
```

Unit systems

This module integrates unit systems into SymPy, allowing a user choose which system to use when doing their computations and providing utilities to display and convert units.

Unit systems are composed of units and constants, which are themselves described from dimensions and numbers, and possibly a prefix. Quantities are defined by their unit and their numerical value, with respect to the current system.

The main advantage of this implementation over the old unit module is that it divides the units in unit systems, so that the user can decide which units to use, instead of having all in the name space (for example astrophysicists can only use units with ua, Earth or Sun masses, the theoricists will use natural system, etc.). Moreover it allows a better control over the dimensions and conversions.

Ideas about future developments can be found on the Github wiki.

Philosophy behind unit systems

Dimensions

Introduction

At the root of unit systems are dimension systems, whose structure mainly determines the one of unit systems. Our definition could seem rough but they are largely sufficient for our purposes.

A dimension will be defined as a property which is measurable and assigned to a specific phenomenon. In this sense dimensions are different from pure numbers because they carry some extra-sense, and for this reason two different dimensions cannot be added. For example time or length are dimensions, but also any other things which has some sense for us, like angle, number of particles (moles...) or information (bits...).

From this point of view the only truly dimensionless quantity are pure numbers. The idea of being dimensionless is very system-dependent, as can be seen from the $(c, \hbar, G)$, in which all units appears to be dimensionless in the usual common sense. This is unavoidable for computability of generic unit systems (but at the end we can tell the program what is dimensionless).

Dimensions can be composed together by taking their product or their ratio (to be defined below). For example the velocity is defined as length divided by time, or we can see the length as velocity multiplied by time, depending of what we see as the more fundamental: in general we can select a set of base dimensions from which we can describe all the others.

Group structure

After this short introduction whose aim was to introduce the dimensions from an intuitive perspective, we describe the mathematical structure. A dimension system with n independent dimensions $\{d_i\}_{i=1,\dots,n}$ is described by a multiplicative group G:

- there an identity element 1 corresponding to pure numbers;
- the product $D_3 = D_1 D_2$ of two elements $D_1, D_2 \in G$ is also in G;
- any element $D \in G$ has an inverse $D^{-1} \in G$.

We denote

$$D^n = \underbrace{D \times \cdots \times D}_{n \text{ times}},$$

and by definition $D^0 = 1$. The $\{d_i\}_{i=1,\ldots,n}$ are called generators of the group since any element $D \in G$ can be expressed as the product of powers of the generators:

$$D = \prod_{i=1}^{n} d_i^{a_i}, \qquad a_i \in \mathbf{Z}.$$

The identity is given for $a_i = 0, \forall i$, while we recover the generator d_i for $a_i = 1, a_j = 0, \forall j \neq i$. This group has the following properties:

1. abelian, since the generator commutes, $[d_i, d_j] = 0$;
2. countable (infinite but discrete) since the elements are indexed by the powers of the generators[1].

One can change the dimension basis $\{d'_i\}_{i=1,\ldots,n}$ by taking some combination of the old generators:

$$d'_i = \prod_{j=1}^{n} d_j^{P_{ij}}.$$

Linear space representation

It is possible to use the linear space $\mathbf{Z}^n$ as a representation of the group since the power coefficients a_i carry all the information one needs (we do not distinguish between the element of the group and its representation):

$$(d_i)_j = \delta_{ij}, \qquad D = \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix}.$$

The change of basis to d'_i follows the usual rule of change of basis for linear space, the matrix being given by the coefficients P_{ij}, which are simply the coefficients of the new vectors in term of the old basis:

$$d'_i = P_{ij} d_j.$$

We will use this last solution in our algorithm.

An example

In order to illustrate all this formalism, we end this section with a specific example, the MKS system (m, kg, s) with dimensions (L: length, M: mass, T: time). They are represented as (we will always sort the vectors in alphabetic order)

$$L = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad M = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \qquad T = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

[1] In general we will consider only dimensions with a maximum coefficient, so we can only a truncation of the group; but this is not useful for the algorithm.

Other dimensions can be derived, for example velocity V or action A

$$V = LT^{-1}, \qquad A = ML^2T^{-2},$$
$$V = \begin{pmatrix} 1 \\ 0 \\ -1 \end{pmatrix}, \qquad A = \begin{pmatrix} 2 \\ 1 \\ -2 \end{pmatrix}.$$

We can change the basis to go to the natural system $(m, c, \hbar)$ with dimension (L: length, V: velocity, A: action)[2]. In this basis the generators are

$$A = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \qquad L = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \qquad V = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},$$

whereas the mass and time are given by

$$T = LV^{-1}, \qquad M = AV^{-2},$$
$$T = \begin{pmatrix} 0 \\ 1 \\ -1 \end{pmatrix}, \qquad M = \begin{pmatrix} 1 \\ 0 \\ -2 \end{pmatrix}.$$

Finally the inverse change of basis matrix P^{-1} is obtained by gluing the vectors expressed in the old basis:

$$P^{-1} = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 0 & 0 \\ -2 & 0 & -1 \end{pmatrix}.$$

To find the change of basis matrix we just have to take the inverse

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & -2 & -1 \end{pmatrix}.$$

Quantities

A quantity is defined by its name, dimension and factor to a canonical quantity of the same dimension. The canonical quantities are an internal reference of the units module and should not be relevant for end-users. Both units and physical constants are quantities.

Units

Units, such as meters, seconds and kilograms, are usually reference quantities chosen by men to refer to other quantities.

After defining several units of different dimensions we can form a unit system, which is basically a dimension system with a notion of scale.

[2] We anticipate a little by considering c and $\hbar$ as units and not as physical constants.

Constants

Physical constants are just quantities. They indicate that we used not to understand that two dimensions are in fact the same. For example, we see a velocity for the light different from 1 because we do not think that time is the same as space (which is normal because of our sense; but it is different at the fundamental level). For example, once there was the "heat constant" which allowed to convert between joules and calories since people did not know that heat was energy. As soon as they understood it they fixed this constant to 1 (this is a very schematic story).

We can interpret the fact that now we fix the value of fundamental constants in the SI as showing that they are units (and we use them to define the other usual units).

The need for a reference

It is not possible to define from scratch units and unit systems: one needs to define some references, and then build the rest over them. Said in another way, we need an origin for the scales of our units (i.e. a unit with factor 1), and to be sure that all units of a given dimension are defined consistently we need to use the same origin for all of them. This can happen if we want to use a derived unit as a base units in another system: we should not define it as having a scale 1, because, even if it is inconsistent inside the system, we could not convert to the first system since we have two different units (from our point of view) of same scale (which means they are equal for the computer).

We will say that the dimensions and scales defined outside systems are canonical, because we use them for all computations. On the other side the dimensions and scales obtained with reference to a system are called physical, because they ultimately carry a sense.

Let's use a concrete (and important) example: the case of the mass units. We would like to define the gram as the origin. We would like to define the gram as the canonical origin for the mass, so we assign it a scale 1. Then we can define a system (e.g. in chemistry) that take it as a base unit. The MKS system prefers to use the kilogram; a naive choice would be to attribute it a scale if 1 since it is a base, but we see that we could not convert to the chemistry system because g and kg have both been given the same factor. So we need to define kg as 1000 g, and only then use it as a base in MKS. But as soon as we ask the question "what is the factor of kg in MKS?", we get the answer 1, since it is a base unit.

Thus we will define all computations without referring to a system, and it is only at the end that we can plug the result into a system to give the context we are interested in.

Literature

Examples

In the following sections we give few examples of what can be done with this module.

Dimensional analysis

We will start from Newton's second law

$$ma = F$$

where m, a and F are the mass, the acceleration and the force respectively. Knowing the dimensions of m (M) and a (LT^{-2}), we will determine the dimension of F; obviously we will find that it is a force: MLT^{-2}.

From there we will use the expression of the gravitational force between the particle of mass m and the body of mass M, at a distance r

$$F = \frac{GmM}{r^2}$$

to determine the dimension of the Newton's constant G. The result should be $L^3M^{-1}T^{-2}$.

```
>>> from sympy import symbols
>>> from sympy.physics.units.systems import SI
>>> from sympy.physics.units import length, mass, acceleration, force
>>> from sympy.physics.units import gravitational_constant as G
>>> from sympy.physics.units.systems.si import dimsys_SI
>>> F = mass*acceleration
>>> F
Dimension(acceleration*mass)
>>> dimsys_SI.get_dimensional_dependencies(F)
{Dimension(length): 1, Dimension(mass): 1, Dimension(time): -2}
>>> dimsys_SI.get_dimensional_dependencies(force)
{Dimension(length): 1, Dimension(mass): 1, Dimension(time): -2}
```

Dimensions cannot compared directly, even if in the SI convention they are the same:

```
>>> F == force
False
```

Dimension system objects provide a way to test the equivalence of dimensions:

```
>>> dimsys_SI.equivalent_dims(F, force)
True
```

```
>>> m1, m2, r = symbols("m1 m2 r")
>>> grav_eq = G * m1 * m2 / r**2
>>> F2 = grav_eq.subs({m1: mass, m2: mass, r: length, G: G.dimension}
 ↪)
>>> F2
Dimension(mass*length*time**-2)
>>> F2.get_dimensional_dependencies()
{'length': 1, 'mass': 1, 'time': -2}
```

Note that one should first solve the equation, and then substitute with the dimensions.

Equation with quantities

Using Kepler's third law

$$\frac{T^2}{a^3} = \frac{4\pi^2}{GM}$$

we can find the Venus orbital period using the known values for the other variables (taken from Wikipedia). The result should be 224.701 days.

```
>>> from sympy import solve, symbols, pi, Eq
>>> from sympy.physics.units import Quantity, length, mass
>>> from sympy.physics.units import day, gravitational_constant as G
>>> from sympy.physics.units import meter, kilogram
>>> T = symbols("T")
>>> a = Quantity("venus_a")
```

Specify the dimension and scale in SI units:

```
>>> SI.set_quantity_dimension(a, length)
>>> SI.set_quantity_scale_factor(a, 108208000e3*meter)
```

Add the solar mass as quantity:

```
>>> M = Quantity("solar_mass")
>>> SI.set_quantity_dimension(M, mass)
>>> SI.set_quantity_scale_factor(M, 1.9891e30*kilogram)
```

Now Kepler's law:

```
>>> eq = Eq(T**2 / a**3, 4*pi**2 / G / M)
>>> eq
Eq(T**2/venus_a**3, 4*pi**2/(gravitational_constant*solar_mass))
>>> q = solve(eq, T)[1]
>>> q
2*pi*venus_a**(3/2)/(sqrt(gravitational_constant)*sqrt(solar_mass))
```

To convert to days, use the `convert_to` function (and possibly approximate the outcoming result):

```
>>> from sympy.physics.units import convert_to
>>> convert_to(q, day)
71.5112118495813*pi*day
>>> convert_to(q, day).n()
224.659097795948*day
```

We could also have the solar mass and the day as units coming from the astrophysical system, but we wanted to show how to create a unit that one needs.

We can see in this example that intermediate dimensions can be ill-defined, such as sqrt(G), but one should check that the final result - when all dimensions are combined - is well defined.

Dimensions and dimension systems

Definition of physical dimensions.

Unit systems will be constructed on top of these dimensions.

Most of the examples in the doc use MKS system and are presented from the computer point of view: from a human point, adding length to time is not legal in MKS but it is in natural system; for a computer in natural system there is no time dimension (but a velocity dimension instead) - in the basis - so the question of adding time to length has no meaning.

class sympy.physics.units.dimensions.**Dimension**(*name*, *symbol=None*)

This class represent the dimension of a physical quantities.

The `Dimension` constructor takes as parameters a name and an optional symbol.

For example, in classical mechanics we know that time is different from temperature and dimensions make this difference (but they do not provide any measure of these quantites.

```
>>> from sympy.physics.units import Dimension
>>> length = Dimension('length')
>>> length
Dimension(length)
>>> time = Dimension('time')
>>> time
Dimension(time)
```

Dimensions can be composed using multiplication, division and exponentiation (by a number) to give new dimensions. Addition and subtraction is defined only when the two objects are the same dimension.

```
>>> velocity = length / time
>>> velocity
Dimension(length/time)
```

It is possible to use a dimension system object to get the dimensionsal dependencies of a dimension, for example the dimension system used by the SI units convention can be used:

```
>>> from sympy.physics.units.systems.si import dimsys_SI
>>> dimsys_SI.get_dimensional_dependencies(velocity)
{Dimension(length, L): 1, Dimension(time, T): -1}
>>> length + length
Dimension(length)
>>> l2 = length**2
>>> l2
Dimension(length**2)
>>> dimsys_SI.get_dimensional_dependencies(l2)
{Dimension(length, L): 2}
```

has_integer_powers(*dim_sys*)

Check if the dimension object has only integer powers.

All the dimension powers should be integers, but rational powers may appear in intermediate steps. This method may be used to check that the final result is well-defined.

class sympy.physics.units.dimensions.**DimensionSystem**(*base_dims*, *derived_dims=()*, *dimensional_dependencies={}*)

DimensionSystem represents a coherent set of dimensions.

The constructor takes three parameters:

- base dimensions;
- derived dimensions: these are defined in terms of the base dimensions (for example velocity is defined from the division of length by time);
- dependency of dimensions: how the derived dimensions depend on the base dimensions.

Optionally either the `derived_dims` or the `dimensional_dependencies` may be omitted.

property can_transf_matrix

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Return the canonical transformation matrix from the canonical to the base dimension basis.

It is the inverse of the matrix computed with inv_can_transf_matrix().

property dim

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Give the dimension of the system.

That is return the number of dimensions forming the basis.

dim_can_vector(*dim*)

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Dimensional representation in terms of the canonical base dimensions.

dim_vector(*dim*)

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Vector representation in terms of the base dimensions.

property inv_can_transf_matrix

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Compute the inverse transformation matrix from the base to the canonical dimension basis.

It corresponds to the matrix where columns are the vector of base dimensions in canonical basis.

This matrix will almost never be used because dimensions are always defined with respect to the canonical basis, so no work has to be done to get them in this basis.

Nonetheless if this matrix is not square (or not invertible) it means that we have chosen a bad basis.

property is_consistent

Useless method, kept for compatibility with previous versions.

DO NOT USE.

Check if the system is well defined.

is_dimensionless(*dimension*)

Check if the dimension object really has a dimension.

A dimension should have at least one component with non-zero power.

property list_can_dims

Useless method, kept for compatibility with previous versions.

DO NOT USE.

List all canonical dimension names.

print_dim_base(*dim*)

Give the string expression of a dimension in term of the basis symbols.

Unit prefixes

Module defining unit prefixe class and some constants.

Constant dict for SI and binary prefixes are defined as PREFIXES and BIN_PREFIXES.

class sympy.physics.units.prefixes.**Prefix**(*name*, *abbrev*, *exponent*, *base=10*, *latex_repr=None*)

This class represent prefixes, with their name, symbol and factor.

Prefixes are used to create derived units from a given unit. They should always be encapsulated into units.

The factor is constructed from a base (default is 10) to some power, and it gives the total multiple or fraction. For example the kilometer km is constructed from the meter (factor 1) and the kilo (10 to the power 3, i.e. 1000). The base can be changed to allow e.g. binary prefixes.

A prefix multiplied by something will always return the product of this other object times the factor, except if the other object:

- is a prefix and they can be combined into a new prefix;
- defines multiplication with prefixes (which is the case for the Unit class).

Units and unit systems

Unit system for physical quantities; include definition of constants.

class sympy.physics.units.unitsystem.**UnitSystem**(*base_units*, *units=()*, *name=''*, *descr=''*, *dimension_system=None*, *derived_units: Dict[*Dimension *(page 1584)*, Quantity *(page 1587)]* *= {}*)

UnitSystem represents a coherent set of units.

A unit system is basically a dimension system with notions of scales. Many of the methods are defined in the same way.

It is much better if all base units have a symbol.

property dim

Give the dimension of the system.

That is return the number of units forming the basis.

extend(*base*, *units=()*, *name=''*, *description=''*, *dimension_system=None*, *derived_units: Dict[*Dimension *(page 1584)*, Quantity *(page 1587)]* *= {}*)

Extend the current system into a new one.

Take the base and normal units of the current system to merge them to the base and normal units given in argument. If not provided, name and description are overridden by empty strings.

get_units_non_prefixed() → Set[*Quantity* (page 1587)]

Return the units of the system that do not have a prefix.

property is_consistent

Check if the underlying dimension system is consistent.

Physical quantities

Physical quantities.

class sympy.physics.units.quantities.**Quantity**(*name*, *abbrev=None*, *dimension=None*, *scale_factor=None*, *latex_repr=None*, *pretty_unicode_repr=None*, *pretty_ascii_repr=None*, *mathml_presentation_repr=None*, *is_prefixed=False*, ***assumptions*)

Physical quantity: can be a unit of measure, a constant or a generic quantity.

property abbrev

Symbol representing the unit name.

Prepend the abbreviation with the prefix symbol if it is defines.

convert_to(*other*, *unit_system='SI'*)

Convert the quantity to another quantity of same dimensions.

Examples

```
>>> from sympy.physics.units import speed_of_light, meter, second
>>> speed_of_light
speed_of_light
>>> speed_of_light.convert_to(meter/second)
299792458*meter/second
```

```
>>> from sympy.physics.units import liter
>>> liter.convert_to(meter**3)
meter**3/1000
```

property free_symbols

Return free symbols from quantity.

property is_prefixed

Whether or not the quantity is prefixed. Eg. $kilogram$ is prefixed, but $gram$ is not.

property scale_factor

Overall magnitude of the quantity as compared to the canonical units.

set_global_relative_scale_factor(*scale_factor*, *reference_quantity*)

Setting a scale factor that is valid across all unit system.

Conversion between quantities

Several methods to simplify expressions involving unit objects.

sympy.physics.units.util.**convert_to**(*expr*, *target_units*, *unit_system='SI'*)

Convert `expr` to the same expression with all of its units and quantities represented as factors of `target_units`, whenever the dimension is compatible.

`target_units` may be a single unit/quantity, or a collection of units/quantities.

Examples

```
>>> from sympy.physics.units import speed_of_light, meter, gram, second,␣
↪day
>>> from sympy.physics.units import mile, newton, kilogram, atomic_mass_
↪constant
>>> from sympy.physics.units import kilometer, centimeter
>>> from sympy.physics.units import gravitational_constant, hbar
>>> from sympy.physics.units import convert_to
>>> convert_to(mile, kilometer)
25146*kilometer/15625
>>> convert_to(mile, kilometer).n()
1.609344*kilometer
>>> convert_to(speed_of_light, meter/second)
299792458*meter/second
>>> convert_to(day, second)
86400*second
```

(continues on next page)

(continued from previous page)

```
>>> 3*newton
3*newton
>>> convert_to(3*newton, kilogram*meter/second**2)
3*kilogram*meter/second**2
>>> convert_to(atomic_mass_constant, gram)
1.660539060e-24*gram
```

Conversion to multiple units:

```
>>> convert_to(speed_of_light, [meter, second])
299792458*meter/second
>>> convert_to(3*newton, [centimeter, gram, second])
300000*centimeter*gram/second**2
```

Conversion to Planck units:

```
>>> convert_to(atomic_mass_constant, [gravitational_constant, speed_of_
↪light, hbar]).n()
7.62963087839509e-20*hbar**0.5*speed_of_light**0.5/gravitational_
↪constant**0.5
```

High energy physics

Abstract

Contains docstrings for methods in high energy physics.

Gamma matrices

Module to handle gamma matrices expressed as tensor objects.

Examples

```
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G,␣
↪LorentzIndex
>>> from sympy.tensor.tensor import tensor_indices
>>> i = tensor_indices('i', LorentzIndex)
>>> G(i)
GammaMatrix(i)
```

Note that there is already an instance of GammaMatrixHead in four dimensions: GammaMatrix, which is simply declare as

```
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix
>>> from sympy.tensor.tensor import tensor_indices
>>> i = tensor_indices('i', LorentzIndex)
```

(continues on next page)

(continued from previous page)

```
>>> GammaMatrix(i)
GammaMatrix(i)
```

To access the metric tensor

```
>>> LorentzIndex.metric
metric(LorentzIndex,LorentzIndex)
```

sympy.physics.hep.gamma_matrices.**extract_type_tens**(*expression*, *component*)

Extract from a `TensExpr` all tensors with *component*.

Returns two tensor expressions:

- the first contains all `Tensor` of having *component*.
- the second contains all remaining.

sympy.physics.hep.gamma_matrices.**gamma_trace**(*t*)

trace of a single line of gamma matrices

Examples

```
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G,
↪   gamma_trace, LorentzIndex
>>> from sympy.tensor.tensor import tensor_indices, tensor_heads
>>> p, q = tensor_heads('p, q', [LorentzIndex])
>>> i0,i1,i2,i3,i4,i5 = tensor_indices('i0:6', LorentzIndex)
>>> ps = p(i0)*G(-i0)
>>> qs = q(i0)*G(-i0)
>>> gamma_trace(G(i0)*G(i1))
4*metric(i0, i1)
>>> gamma_trace(ps*ps) - 4*p(i0)*p(-i0)
0
>>> gamma_trace(ps*qs + ps*ps) - 4*p(i0)*p(-i0) - 4*p(i0)*q(-i0)
0
```

sympy.physics.hep.gamma_matrices.**kahane_simplify**(*expression*)

This function cancels contracted elements in a product of four dimensional gamma matrices, resulting in an expression equal to the given one, without the contracted gamma matrices.

Parameters

`expression` the tensor expression containing the gamma matrices to simplify.

Notes

If spinor indices are given, the matrices must be given in the order given in the product.

Algorithm

The idea behind the algorithm is to use some well-known identities, i.e., for contractions enclosing an even number of γ matrices

$\gamma^\mu \gamma_{a_1} \cdots \gamma_{a_{2N}} \gamma_\mu = 2(\gamma_{a_{2N}} \gamma_{a_1} \cdots \gamma_{a_{2N-1}} + \gamma_{a_{2N-1}} \cdots \gamma_{a_1} \gamma_{a_{2N}})$

for an odd number of γ matrices

$\gamma^\mu \gamma_{a_1} \cdots \gamma_{a_{2N+1}} \gamma_\mu = -2\gamma_{a_{2N+1}} \gamma_{a_{2N}} \cdots \gamma_{a_1}$

Instead of repeatedly applying these identities to cancel out all contracted indices, it is possible to recognize the links that would result from such an operation, the problem is thus reduced to a simple rearrangement of free gamma matrices.

Examples

When using, always remember that the original expression coefficient has to be handled separately

```
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G,␣
↪LorentzIndex
>>> from sympy.physics.hep.gamma_matrices import kahane_simplify
>>> from sympy.tensor.tensor import tensor_indices
>>> i0, i1, i2 = tensor_indices('i0:3', LorentzIndex)
>>> ta = G(i0)*G(-i0)
>>> kahane_simplify(ta)
Matrix([
[4, 0, 0, 0],
[0, 4, 0, 0],
[0, 0, 4, 0],
[0, 0, 0, 4]])
>>> tb = G(i0)*G(i1)*G(-i0)
>>> kahane_simplify(tb)
-2*GammaMatrix(i1)
>>> t = G(i0)*G(-i0)
>>> kahane_simplify(t)
Matrix([
[4, 0, 0, 0],
[0, 4, 0, 0],
[0, 0, 4, 0],
[0, 0, 0, 4]])
>>> t = G(i0)*G(-i0)
>>> kahane_simplify(t)
Matrix([
[4, 0, 0, 0],
[0, 4, 0, 0],
[0, 0, 4, 0],
[0, 0, 0, 4]])
```

If there are no contractions, the same expression is returned

```
>>> tc = G(i0)*G(i1)
>>> kahane_simplify(tc)
GammaMatrix(i0)*GammaMatrix(i1)
```

References

[1] Algorithm for Reducing Contracted Products of gamma Matrices, Joseph Kahane, Journal of Mathematical Physics, Vol. 9, No. 10, October 1968.

sympy.physics.hep.gamma_matrices.**simplify_gpgp**(*ex*, *sort=True*)

simplify products `G(i)*p(-i)*G(j)*p(-j) -> p(i)*p(-i)`

Examples

```
>>> from sympy.physics.hep.gamma_matrices import GammaMatrix as G,
↪  LorentzIndex, simplify_gpgp
>>> from sympy.tensor.tensor import tensor_indices, tensor_heads
>>> p, q = tensor_heads('p, q', [LorentzIndex])
>>> i0,i1,i2,i3,i4,i5 = tensor_indices('i0:6', LorentzIndex)
>>> ps = p(i0)*G(-i0)
>>> qs = q(i0)*G(-i0)
>>> simplify_gpgp(ps*qs*qs)
GammaMatrix(-L_0)*p(L_0)*q(L_1)*q(-L_1)
```

The Physics Vector Module

Abstract

In this documentation the components of the sympy.physics.vector module have been discussed. *sympy.physics.vector* (page 1592) has been written to facilitate the operations pertaining to 3-dimensional vectors, as functions of time or otherwise, in *sympy.physics* (page 1533).

References for Physics/Vector

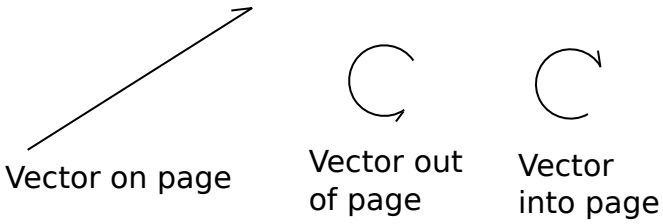
Guide to Vector

Vector & ReferenceFrame

In *sympy.physics.vector* (page 1592), vectors and reference frames are the "building blocks" of dynamic systems. This document will describe these mathematically and describe how to use them with this module's code.

Vector

A vector is a geometric object that has a magnitude (or length) and a direction. Vectors in 3-space are often represented on paper as:



Vector Algebra

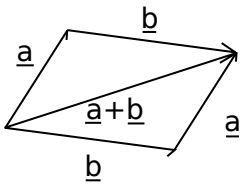
Vector algebra is the first topic to be discussed.

Two vectors are said to be equal if and only if (iff) they have the same magnitude and orientation.

Vector Operations

Multiple algebraic operations can be done with vectors: addition between vectors, scalar multiplication, and vector multiplication.

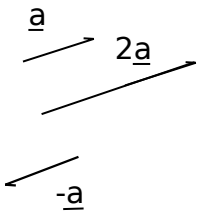
Vector addition as based on the parallelogram law.



Vector addition is also commutative:

$$\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}$$
$$(\mathbf{a} + \mathbf{b}) + \mathbf{c} = \mathbf{a} + (\mathbf{b} + \mathbf{c})$$

Scalar multiplication is the product of a vector and a scalar; the result is a vector with the same orientation but whose magnitude is scaled by the scalar. Note that multiplication by -1 is equivalent to rotating the vector by 180 degrees about an arbitrary axis in the plane perpendicular to the vector.



A unit vector is simply a vector whose magnitude is equal to 1. Given any vector **v** we can define a unit vector as:

$$\hat{\mathbf{n}}_{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\|}$$

Note that every vector can be written as the product of a scalar and unit vector.

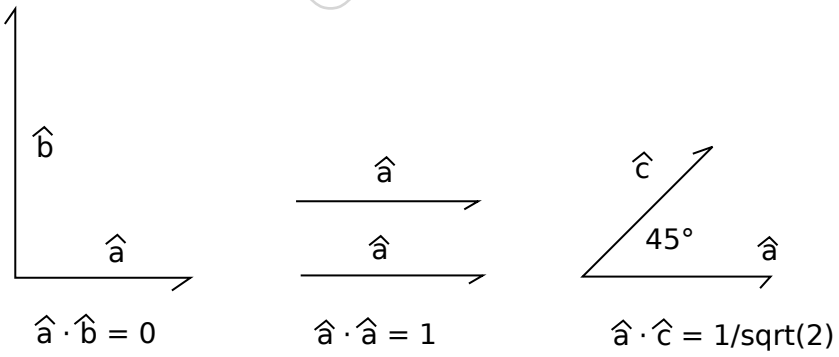
Three vector products are implemented in *sympy.physics.vector* (page 1592): the dot product, the cross product, and the outer product.

The dot product operation maps two vectors to a scalar. It is defined as:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\|\mathbf{b}\| \cos(\theta)$$

where θ is the angle between **a** and **b**.

The dot product of two unit vectors represent the magnitude of the common direction; for other vectors, it is the product of the magnitude of the common direction and the two vectors' magnitudes. The dot product of two perpendicular is zero. The figure below shows some examples:



The dot product is commutative:

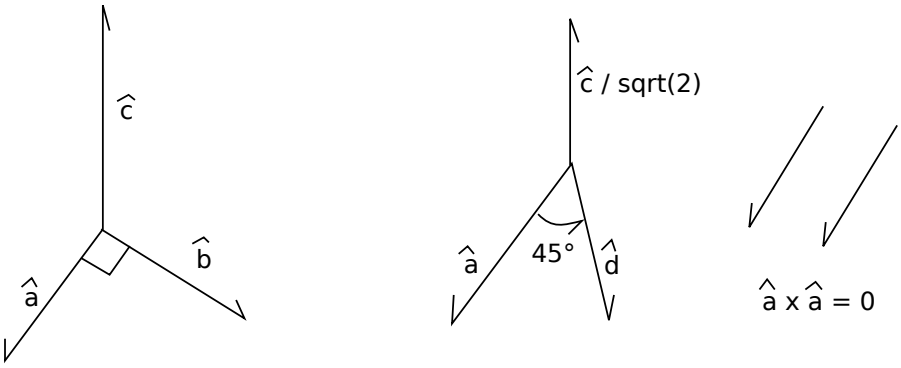
$$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$$

The cross product vector multiplication operation of two vectors returns a vector:

$$\mathbf{a} \times \mathbf{b} = \mathbf{c}$$

The vector **c** has the following properties: it's orientation is perpendicular to both **a** and **b**, it's magnitude is defined as $\|\mathbf{c}\| = \|\mathbf{a}\|\|\mathbf{b}\| \sin(\theta)$ (where θ is the angle between **a** and **b**), and

has a sense defined by using the right hand rule between $\|\mathbf{a}\|\|\mathbf{b}\|$. The figure below shows this:



The cross product has the following properties:

It is not commutative:

$$\mathbf{a} \times \mathbf{b} \neq \mathbf{b} \times \mathbf{a}$$
$$\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$$

and not associative:

$$(\mathbf{a} \times \mathbf{b}) \times \mathbf{c} \neq \mathbf{a} \times (\mathbf{b} \times \mathbf{c})$$

Two parallel vectors will have a zero cross product.

The outer product between two vectors will not be not be discussed here, but instead in the inertia section (that is where it is used). Other useful vector properties and relationships are:

$$\alpha(\mathbf{a} + \mathbf{b}) = \alpha\mathbf{a} + \alpha\mathbf{b}$$
$$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}$$
$$\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{b}$$

$(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c}$ gives the scalar triple product.

$\mathbf{a} \times (\mathbf{b} \cdot \mathbf{c})$ does not work, as you cannot cross a vector and a scalar.

$$(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} = \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$$
$$(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} = (\mathbf{b} \times \mathbf{c}) \cdot \mathbf{a} = (\mathbf{c} \times \mathbf{a}) \cdot \mathbf{b}$$
$$(\mathbf{a} \times \mathbf{b}) \times \mathbf{c} = \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) - \mathbf{a}(\mathbf{b} \cdot \mathbf{c})$$
$$\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) - \mathbf{c}(\mathbf{a} \cdot \mathbf{b})$$

Alternative Representation

If we have three non-coplanar unit vectors $\hat{\mathbf{n}}_\mathbf{x}, \hat{\mathbf{n}}_\mathbf{y}, \hat{\mathbf{n}}_\mathbf{z}$, we can represent any vector $\mathbf{a}$ as $\mathbf{a} = a_x\hat{\mathbf{n}}_\mathbf{x} + a_y\hat{\mathbf{n}}_\mathbf{y} + a_z\hat{\mathbf{n}}_\mathbf{z}$. In this situation $\hat{\mathbf{n}}_\mathbf{x}, \hat{\mathbf{n}}_\mathbf{y}, \hat{\mathbf{n}}_\mathbf{z}$ are referred to as a basis. a_x, a_y, a_z are called the measure numbers. Usually the unit vectors are mutually perpendicular, in which case we can refer to them as an orthonormal basis, and they are usually right-handed.

To test equality between two vectors, now we can do the following. With vectors:

$$\mathbf{a} = a_x\hat{\mathbf{n}}_\mathbf{x} + a_y\hat{\mathbf{n}}_\mathbf{y} + a_z\hat{\mathbf{n}}_\mathbf{z}$$
$$\mathbf{b} = b_x\hat{\mathbf{n}}_\mathbf{x} + b_y\hat{\mathbf{n}}_\mathbf{y} + b_z\hat{\mathbf{n}}_\mathbf{z}$$

We can claim equality if: $a_x = b_x, a_y = b_y, a_z = b_z$.

Vector addition is then represented, for the same two vectors, as:

$$\mathbf{a} + \mathbf{b} = (a_x + b_x)\hat{\mathbf{n}}_\mathbf{x} + (a_y + b_y)\hat{\mathbf{n}}_\mathbf{y} + (a_z + b_z)\hat{\mathbf{n}}_\mathbf{z}$$

Multiplication operations are now defined as:

$$\alpha\mathbf{b} = \alpha b_x\hat{\mathbf{n}}_\mathbf{x} + \alpha b_y\hat{\mathbf{n}}_\mathbf{y} + \alpha b_z\hat{\mathbf{n}}_\mathbf{z}$$
$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z$$
$$\mathbf{a} \times \mathbf{b} = \det \begin{bmatrix} \hat{\mathbf{n}}_\mathbf{x} & \hat{\mathbf{n}}_\mathbf{y} & \hat{\mathbf{n}}_\mathbf{z} \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix}$$
$$(\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c} = \det \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{bmatrix}$$

To write a vector in a given basis, we can do the follow:

$$\mathbf{a} = (\mathbf{a} \cdot \hat{\mathbf{n}}_\mathbf{x})\hat{\mathbf{n}}_\mathbf{x} + (\mathbf{a} \cdot \hat{\mathbf{n}}_\mathbf{y})\hat{\mathbf{n}}_\mathbf{y} + (\mathbf{a} \cdot \hat{\mathbf{n}}_\mathbf{z})\hat{\mathbf{n}}_\mathbf{z}$$

Examples

Some numeric examples of these operations follow:

$$\mathbf{a} = \hat{\mathbf{n}}_\mathbf{x} + 5\hat{\mathbf{n}}_\mathbf{y}$$
$$\mathbf{b} = \hat{\mathbf{n}}_\mathbf{y} + \alpha\hat{\mathbf{n}}_\mathbf{z}$$
$$\mathbf{a} + \mathbf{b} = \hat{\mathbf{n}}_\mathbf{x} + 6\hat{\mathbf{n}}_\mathbf{y} + \alpha\hat{\mathbf{n}}_\mathbf{z}$$
$$\mathbf{a} \cdot \mathbf{b} = 5$$
$$\mathbf{a} \cdot \hat{\mathbf{n}}_\mathbf{y} = 5$$
$$\mathbf{a} \cdot \hat{\mathbf{n}}_\mathbf{z} = 0$$
$$\mathbf{a} \times \mathbf{b} = 5\alpha\hat{\mathbf{n}}_\mathbf{x} - \alpha\hat{\mathbf{n}}_\mathbf{y} + \hat{\mathbf{n}}_\mathbf{z}$$
$$\mathbf{b} \times \mathbf{a} = -5\alpha\hat{\mathbf{n}}_\mathbf{x} + \alpha\hat{\mathbf{n}}_\mathbf{y} - \hat{\mathbf{n}}_\mathbf{z}$$