Second Quantization

Second quantization operators and states for bosons.

This follow the formulation of Fetter and Welecka, “Quantum Theory of Many-Particle Systems.”

class sympy.physics.secondquant.**Dagger**
Hermitian conjugate of creation/annihilation operators.

Examples

```
>>> from sympy import I
>>> from sympy.physics.secondquant import Dagger, B, Bd
>>> Dagger(2*I)
-2*I
>>> Dagger(B(0))
CreateBoson(0)
>>> Dagger(Bd(0))
AnnihilateBoson(0)
```

classmethod eval(arg)
Evaluates the Dagger instance.

Examples

```
>>> from sympy import I
>>> from sympy.physics.secondquant import Dagger, B, Bd
>>> Dagger(2*I)
-2*I
>>> Dagger(B(0))
CreateBoson(0)
>>> Dagger(Bd(0))
AnnihilateBoson(0)
```

The eval() method is called automatically.

class sympy.physics.secondquant.**KroneckerDelta**
The discrete, or Kronecker, delta function.

A function that takes in two integers i and j. It returns 0 if i and j are not equal or it returns 1 if i and j are equal.

Parameters **i** : Number, Symbol

The first index of the delta function.

j : Number, Symbol

The second index of the delta function.

See also:

eval (page 1519), sympy.functions.special.delta_functions.DiracDelta (page 498)

References

[R487]

Examples

A simple example with integer indices:

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> KroneckerDelta(1, 2)
0
>>> KroneckerDelta(3, 3)
1
```

Symbolic indices:

```
>>> from sympy.abc import i, j, k
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

classmethod eval(i, j)
Evaluates the discrete delta function.

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy.abc import i, j, k
```

```
>>> KroneckerDelta(i, j)
KroneckerDelta(i, j)
>>> KroneckerDelta(i, i)
1
>>> KroneckerDelta(i, i + 1)
0
>>> KroneckerDelta(i, i + 1 + k)
KroneckerDelta(i, i + k + 1)
```

indirect doctest

indices_contain_equal_information
Returns True if indices are either both above or below fermi.

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, q).indices_contain_equal_information
True
>>> KroneckerDelta(p, q+1).indices_contain_equal_information
True
>>> KroneckerDelta(i, p).indices_contain_equal_information
False
```

is_above_fermi

True if Delta can be non-zero above fermi

See also:

is_below_fermi (page 1520), is_only_below_fermi (page 1521), is_only_above_fermi (page 1521)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_above_fermi
True
>>> KroneckerDelta(p, i).is_above_fermi
False
>>> KroneckerDelta(p, q).is_above_fermi
True
```

is_below_fermi

True if Delta can be non-zero below fermi

See also:

is_above_fermi (page 1520), is_only_above_fermi (page 1521), is_only_below_fermi (page 1521)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_below_fermi
False
>>> KroneckerDelta(p, i).is_below_fermi
```

(continues on next page)

(continued from previous page)

```
True
>>> KroneckerDelta(p, q).is_below_fermi
True
```

is_only_above_fermi

True if Delta is restricted to above fermi

See also:

is_above_fermi (page 1520), is_below_fermi (page 1520), is_only_below_fermi (page 1521)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, a).is_only_above_fermi
True
>>> KroneckerDelta(p, q).is_only_above_fermi
False
>>> KroneckerDelta(p, i).is_only_above_fermi
False
```

is_only_below_fermi

True if Delta is restricted to below fermi

See also:

is_above_fermi (page 1520), is_below_fermi (page 1520), is_only_above_fermi (page 1521)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import Symbol
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
>>> q = Symbol('q')
>>> KroneckerDelta(p, i).is_only_below_fermi
True
>>> KroneckerDelta(p, q).is_only_below_fermi
False
>>> KroneckerDelta(p, a).is_only_below_fermi
False
```

killable_index

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

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

See also:

preferred_index (page 1522)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import 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
```

preferred_index

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

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

See also:

killable_index (page 1521)

Examples

```
>>> from sympy.functions.special.tensor_functions import KroneckerDelta
>>> from sympy import 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
```

class sympy.physics.secondquant.**AnnihilateBoson**

Bosonic annihilation operator.

Examples

```
>>> from sympy.physics.secondquant import B
>>> from sympy.abc import x
>>> B(x)
AnnihilateBoson(x)
```

apply_operator(state)
Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```
>>> from sympy.physics.secondquant import B, BKet
>>> from sympy.abc import x, y, n
>>> B(x).apply_operator(y)
y*AnnihilateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

class sympy.physics.secondquant.**CreateBoson**
Bosonic creation operator.

apply_operator(state)
Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y*CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

class sympy.physics.secondquant.**AnnihilateFermion**
Fermionic annihilation operator.

apply_operator(state)
Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y*CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

is_only_q_annihilator
Always destroy a quasi-particle? (annihilate hole or annihilate particle)

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> F(a).is_only_q_annihilator
True
>>> F(i).is_only_q_annihilator
False
>>> F(p).is_only_q_annihilator
False
```

is_only_q_creator

Always create a quasi-particle? (create hole or create particle)

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> F(a).is_only_q_creator
False
>>> F(i).is_only_q_creator
True
>>> F(p).is_only_q_creator
False
```

is_q_annihilator

Can we destroy a quasi-particle? (annihilate hole or annihilate particle) If so, would that be above or below the fermi surface?

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=1)
>>> i = Symbol('i', below_fermi=1)
>>> p = Symbol('p')
```

```
>>> F(a).is_q_annihilator
1
>>> F(i).is_q_annihilator
0
>>> F(p).is_q_annihilator
1
```

is_q_creator

Can we create a quasi-particle? (create hole or create particle) If so, would that be above or below the fermi surface?

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import F
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> F(a).is_q_creator
0
>>> F(i).is_q_creator
-1
>>> F(p).is_q_creator
-1
```

class sympy.physics.secondquant.**CreateFermion**

Fermionic creation operator.

apply_operator(state)

Apply state to self if self is not symbolic and state is a FockStateKet, else multiply self by state.

Examples

```
>>> from sympy.physics.secondquant import B, Dagger, BKet
>>> from sympy.abc import x, y, n
>>> Dagger(B(x)).apply_operator(y)
y*CreateBoson(x)
>>> B(0).apply_operator(BKet((n,)))
sqrt(n)*FockStateBosonKet((n - 1,))
```

is_only_q_annihilator

Always destroy a quasi-particle? (annihilate hole or annihilate particle)

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_only_q_annihilator
False
>>> Fd(i).is_only_q_annihilator
True
>>> Fd(p).is_only_q_annihilator
False
```

is_only_q_creator

Always create a quasi-particle? (create hole or create particle)

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_only_q_creator
True
>>> Fd(i).is_only_q_creator
False
>>> Fd(p).is_only_q_creator
False
```

is_q_annihilator

Can we destroy a quasi-particle? (annihilate hole or annihilate particle) If so, would that be above or below the fermi surface?

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=1)
```

(continues on next page)

(continued from previous page)

```
>>> i = Symbol('i', below_fermi=1)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_q_annihilator
0
>>> Fd(i).is_q_annihilator
-1
>>> Fd(p).is_q_annihilator
-1
```

is_q_creator

Can we create a quasi-particle? (create hole or create particle) If so, would that be above or below the fermi surface?

```
>>> from sympy import Symbol
>>> from sympy.physics.secondquant import Fd
>>> a = Symbol('a', above_fermi=True)
>>> i = Symbol('i', below_fermi=True)
>>> p = Symbol('p')
```

```
>>> Fd(a).is_q_creator
1
>>> Fd(i).is_q_creator
0
>>> Fd(p).is_q_creator
1
```

class sympy.physics.secondquant.**FockState**

Many particle Fock state with a sequence of occupation numbers.

Anywhere you can have a FockState, you can also have S.Zero. All code must check for this!

Base class to represent FockStates.

class sympy.physics.secondquant.**FockStateBra**

Representation of a bra.

class sympy.physics.secondquant.**FockStateKet**

Representation of a ket.

class sympy.physics.secondquant.**FockStateBosonKet**

Many particle Fock state with a sequence of occupation numbers.

Occupation numbers can be any integer >= 0.

Examples

```
>>> from sympy.physics.secondquant import BKet
>>> BKet([1, 2])
FockStateBosonKet((1, 2))
```

class sympy.physics.secondquant.**FockStateBosonBra**

Describes a collection of BosonBra particles.

Examples

```
>>> from sympy.physics.secondquant import BBra
>>> BBra([1, 2])
FockStateBosonBra((1, 2))
```

sympy.physics.secondquant.**BBra**
alias of sympy.physics.secondquant.FockStateBosonBra (page 1526)

sympy.physics.secondquant.**BKet**
alias of sympy.physics.secondquant.FockStateBosonKet (page 1526)

sympy.physics.secondquant.**FBra**
alias of sympy.physics.secondquant.FockStateFermionBra

sympy.physics.secondquant.**FKet**
alias of sympy.physics.secondquant.FockStateFermionKet

sympy.physics.secondquant.**F**
alias of sympy.physics.secondquant.AnnihilateFermion (page 1523)

sympy.physics.secondquant.**Fd**
alias of sympy.physics.secondquant.CreateFermion (page 1524)

sympy.physics.secondquant.**B**
alias of sympy.physics.secondquant.AnnihilateBoson (page 1522)

sympy.physics.secondquant.**Bd**
alias of sympy.physics.secondquant.CreateBoson (page 1523)

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
```

class sympy.physics.secondquant.**InnerProduct**
An unevaluated inner product between a bra and ket.

Currently this class just reduces things to a product of Kronecker Deltas. In the future, we could introduce abstract states like |a> and |b>, and leave the inner product unevaluated as <a|b>.

bra
Returns the bra part of the state

ket
Returns the ket part of the state

class sympy.physics.secondquant.**BosonicBasis**
Base class for a basis set of bosonic Fock states.

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,))
```

class sympy.physics.secondquant.**FixedBosonicBasis**(n_particles, n_levels)
Fixed particle number basis set.

Examples

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

index(state)
Returns the index of state in basis.

Examples

```
>>> from sympy.physics.secondquant import FixedBosonicBasis
>>> b = FixedBosonicBasis(2, 3)
>>> b.index(b.state(3))
3
```

state(i)
Returns the state that lies at index i of the basis

Examples

```
>>> from sympy.physics.secondquant import FixedBosonicBasis
>>> b = FixedBosonicBasis(2, 3)
>>> b.state(3)
FockStateBosonKet((1, 0, 1))
```

class sympy.physics.secondquant.**Commutator**
The Commutator: [A, B] = A*B - B*A

The arguments are ordered according to .__cmp__()

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import Commutator
>>> A, B = symbols('A,B', commutative=False)
>>> Commutator(B, A)
-Commutator(A, B)
```

Evaluate the commutator with .doit()

```
>>> comm = Commutator(A,B); comm
Commutator(A, B)
>>> comm.doit()
A*B - B*A
```

For two second quantization operators the commutator is evaluated immediately:

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

```
>>> Commutator(Fd(a),Fd(i))
2*NO(CreateFermion(a)*CreateFermion(i))
```

But for more complicated expressions, the evaluation is triggered by a call to .doit()

```
>>> comm = Commutator(Fd(p)*Fd(q),F(i)); comm
Commutator(CreateFermion(p)*CreateFermion(q), AnnihilateFermion(i))
>>> comm.doit(wicks=True)
-KroneckerDelta(i, p)*CreateFermion(q) +
 KroneckerDelta(i, q)*CreateFermion(p)
```

doit(**hints)
Enables the computation of complex expressions.

Examples

```
>>> from sympy.physics.secondquant import Commutator, F, Fd
>>> from sympy import symbols
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> c = Commutator(Fd(a)*F(i),Fd(b)*F(j))
>>> c.doit(wicks=True)
0
```

classmethod eval(a, b)

The Commutator [A,B] is on canonical form if A < B.

Examples

```
>>> from sympy.physics.secondquant import Commutator, F, Fd
>>> from sympy.abc import x
>>> c1 = Commutator(F(x), Fd(x))
>>> c2 = Commutator(Fd(x), F(x))
>>> Commutator.eval(c1, c2)
0
```

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.**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.**wicks**(e, **kw_args)

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

Examples

```
>>> from sympy import symbols, Function, Dummy
>>> from sympy.physics.secondquant import wicks, F, Fd, NO
>>> p,q,r = symbols('p,q,r')
>>> wicks(Fd(p)*F(q))  # doctest: +SKIP
d(p, q)*d(q, _i) + NO(CreateFermion(p)*AnnihilateFermion(q))
```

By default, the expression is expanded:

```
>>> wicks(F(p)*(F(q)+F(r))) # doctest: +SKIP
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) # doctest: +SKIP
KroneckerDelta(_i, _q)*KroneckerDelta(
    _p, _q) + KroneckerDelta(_i, _r)*KroneckerDelta(_p, _r)
```

class sympy.physics.secondquant.**NO**

This Object is used to represent normal ordering brackets.

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

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.

```
>>> 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(**kw_args)
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)  # doctest: +SKIP
NO(AnnihilateFermion(p)*AnnihilateFermion(r))
```

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
```

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           #doctest: +SKIP
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]
```

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

We evaluate KroneckerDelta symbols in the expression assuming Einstein summation.

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

class sympy.physics.secondquant.**AntiSymmetricTensor**

Stores upper and lower indices in separate Tuple's.

Each group of indices is assumed to be antisymmetric.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i j', below_fermi=True)
>>> a, b = symbols('a b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (i, a), (b, j))
-AntiSymmetricTensor(v, (a, i), (b, j))
```

As you can see, the indices are automatically sorted to a canonical form.

doit(**kw_args)

Returns self.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j)).doit()
AntiSymmetricTensor(v, (a, i), (b, j))
```

lower

Returns the lower indices.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (a, i), (b, j)).lower
(b, j)
```

symbol
Returns the symbol of the tensor.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (a, i), (b, j)).symbol
v
```

upper
Returns the upper indices.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.secondquant import AntiSymmetricTensor
>>> i, j = symbols('i,j', below_fermi=True)
>>> a, b = symbols('a,b', above_fermi=True)
>>> AntiSymmetricTensor('v', (a, i), (b, j))
AntiSymmetricTensor(v, (a, i), (b, j))
>>> AntiSymmetricTensor('v', (a, i), (b, j)).upper
(a, i)
```

sympy.physics.secondquant.**substitute_dummies**(expr, new_indices=False, pretty_indices={})
Collect terms by substitution of dummy variables.

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 inforation 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)
```

class sympy.physics.secondquant.**PermutationOperator**

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.

```
>>> 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)
```

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

Performs simplification by introducing PermutationOperators where appropriate.

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)
```

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

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 $\langle j_1 m_1\ j_2 m_2 | j_3 m_3 \rangle$.