References

[R675]

Dagger

Hermitian conjugation.

class sympy.physics.quantum.dagger.**Dagger**(*arg*)

General Hermitian conjugate operation.

Parameters

arg : Expr

The SymPy expression that we want to take the dagger of.

Explanation

Take the Hermetian conjugate of an argument [R676]. For matrices this operation is equivalent to transpose and complex conjugate [R677].

Examples

Daggering various quantum objects:

```
>>> from sympy.physics.quantum.dagger import Dagger
>>> from sympy.physics.quantum.state import Ket, Bra
>>> from sympy.physics.quantum.operator import Operator
>>> Dagger(Ket('psi'))
<psi|
>>> Dagger(Bra('phi'))
|phi>
>>> Dagger(Operator('A'))
Dagger(A)
```

Inner and outer products:

```
>>> from sympy.physics.quantum import InnerProduct, OuterProduct
>>> Dagger(InnerProduct(Bra('a'), Ket('b')))
<b|a>
>>> Dagger(OuterProduct(Ket('a'), Bra('b')))
|b><a|
```

Powers, sums and products:

```
>>> A = Operator('A')
>>> B = Operator('B')
>>> Dagger(A*B)
Dagger(B)*Dagger(A)
>>> Dagger(A+B)
Dagger(A) + Dagger(B)
```

(continues on next page)

(continued from previous page)

```
>>> Dagger(A**2)
Dagger(A)**2
```

Dagger also seamlessly handles complex numbers and matrices:

```
>>> from sympy import Matrix, I
>>> m = Matrix([[1,I],[2,I]])
>>> m
Matrix([
[1, I],
[2, I]])
>>> Dagger(m)
Matrix([
[ 1,  2],
[-I, -I]])
```

References

[R676], [R677]

Inner Product

Symbolic inner product.

class sympy.physics.quantum.innerproduct.**InnerProduct**(*bra*, *ket*)

An unevaluated inner product between a Bra and a Ket [1].

Parameters

bra : BraBase or subclass

The bra on the left side of the inner product.

ket : KetBase or subclass

The ket on the right side of the inner product.

Examples

Create an InnerProduct and check its properties:

```
>>> from sympy.physics.quantum import Bra, Ket
>>> b = Bra('b')
>>> k = Ket('k')
>>> ip = b*k
>>> ip
<b|k>
>>> ip.bra
<b|
>>> ip.ket
|k>
```

In simple products of kets and bras inner products will be automatically identified and created:

```
>>> b*k
<b|k>
```

But in more complex expressions, there is ambiguity in whether inner or outer products should be created:

```
>>> k*b*k*b
|k><b|*|k>*<b|
```

A user can force the creation of a inner products in a complex expression by using parentheses to group the bra and ket:

```
>>> k*(b*k)*b
<b|k>*|k>*<b|
```

Notice how the inner product <b|k> moved to the left of the expression because inner products are commutative complex numbers.

References

[R683]

Tensor Product

Abstract tensor product.

class sympy.physics.quantum.tensorproduct.**TensorProduct**(**args*)

The tensor product of two or more arguments.

For matrices, this uses `matrix_tensor_product` to compute the Kronecker or tensor product matrix. For other objects a symbolic `TensorProduct` instance is returned. The tensor product is a non-commutative multiplication that is used primarily with operators and states in quantum mechanics.

Currently, the tensor product distinguishes between commutative and non-commutative arguments. Commutative arguments are assumed to be scalars and are pulled out in front of the `TensorProduct`. Non-commutative arguments remain in the resulting `TensorProduct`.

Parameters

args : tuple

A sequence of the objects to take the tensor product of.

Examples

Start with a simple tensor product of SymPy matrices:

```
>>> from sympy import Matrix
>>> from sympy.physics.quantum import TensorProduct

>>> m1 = Matrix([[1,2],[3,4]])
>>> m2 = Matrix([[1,0],[0,1]])
>>> TensorProduct(m1, m2)
Matrix([
[1, 0, 2, 0],
[0, 1, 0, 2],
[3, 0, 4, 0],
[0, 3, 0, 4]])
>>> TensorProduct(m2, m1)
Matrix([
[1, 2, 0, 0],
[3, 4, 0, 0],
[0, 0, 1, 2],
[0, 0, 3, 4]])
```

We can also construct tensor products of non-commutative symbols:

```
>>> from sympy import Symbol
>>> A = Symbol('A',commutative=False)
>>> B = Symbol('B',commutative=False)
>>> tp = TensorProduct(A, B)
>>> tp
AxB
```

We can take the dagger of a tensor product (note the order does NOT reverse like the dagger of a normal product):

```
>>> from sympy.physics.quantum import Dagger
>>> Dagger(tp)
Dagger(A)xDagger(B)
```

Expand can be used to distribute a tensor product across addition:

```
>>> C = Symbol('C',commutative=False)
>>> tp = TensorProduct(A+B,C)
>>> tp
(A + B)xC
>>> tp.expand(tensorproduct=True)
AxC + BxC
```

sympy.physics.quantum.tensorproduct.**tensor_product_simp**(*e*, ***hints*)

Try to simplify and combine TensorProducts.

In general this will try to pull expressions inside of `TensorProducts`. It currently only works for relatively simple cases where the products have only scalars, raw `TensorProducts`, not `Add`, `Pow`, `Commutators` of `TensorProducts`. It is best to see what it does by showing examples.

Examples

```
>>> from sympy.physics.quantum import tensor_product_simp
>>> from sympy.physics.quantum import TensorProduct
>>> from sympy import Symbol
>>> A = Symbol('A',commutative=False)
>>> B = Symbol('B',commutative=False)
>>> C = Symbol('C',commutative=False)
>>> D = Symbol('D',commutative=False)
```

First see what happens to products of tensor products:

```
>>> e = TensorProduct(A,B)*TensorProduct(C,D)
>>> e
AxB*CxD
>>> tensor_product_simp(e)
(A*C)x(B*D)
```

This is the core logic of this function, and it works inside, powers, sums, commutators and anticommutators as well:

```
>>> tensor_product_simp(e**2)
(A*C)x(B*D)**2
```

States and Operators

Cartesian Operators and States

Operators and states for 1D cartesian position and momentum.

TODO:

- Add 3D classes to mappings in operatorset.py

class sympy.physics.quantum.cartesian.**PositionBra3D**(**args*, ***kwargs*)

3D cartesian position eigenbra

class sympy.physics.quantum.cartesian.**PositionKet3D**(**args*, ***kwargs*)

3D cartesian position eigenket

class sympy.physics.quantum.cartesian.**PositionState3D**(**args*, ***kwargs*)

Base class for 3D cartesian position eigenstates

property position_x

The x coordinate of the state

property position_y

The y coordinate of the state

property position_z

The z coordinate of the state

class sympy.physics.quantum.cartesian.**PxBra**(**args*, ***kwargs*)

1D cartesian momentum eigenbra.

property momentum
The momentum of the state.

class sympy.physics.quantum.cartesian.**PxKet**(**args*, ***kwargs*)
1D cartesian momentum eigenket.

property momentum
The momentum of the state.

class sympy.physics.quantum.cartesian.**PxOp**(**args*, ***kwargs*)
1D cartesian momentum operator.

class sympy.physics.quantum.cartesian.**XBra**(**args*, ***kwargs*)
1D cartesian position eigenbra.

property position
The position of the state.

class sympy.physics.quantum.cartesian.**XKet**(**args*, ***kwargs*)
1D cartesian position eigenket.

property position
The position of the state.

class sympy.physics.quantum.cartesian.**XOp**(**args*, ***kwargs*)
1D cartesian position operator.

class sympy.physics.quantum.cartesian.**YOp**(**args*, ***kwargs*)
Y cartesian coordinate operator (for 2D or 3D systems)

class sympy.physics.quantum.cartesian.**ZOp**(**args*, ***kwargs*)
Z cartesian coordinate operator (for 3D systems)

Hilbert Space

Hilbert spaces for quantum mechanics.

Authors: * Brian Granger * Matt Curry

class sympy.physics.quantum.hilbert.**ComplexSpace**(*dimension*)
Finite dimensional Hilbert space of complex vectors.

The elements of this Hilbert space are n-dimensional complex valued vectors with the usual inner product that takes the complex conjugate of the vector on the right.

A classic example of this type of Hilbert space is spin-1/2, which is `ComplexSpace(2)`. Generalizing to spin-s, the space is `ComplexSpace(2*s+1)`. Quantum computing with N qubits is done with the direct product space `ComplexSpace(2)**N`.

Examples

```
>>> from sympy import symbols
>>> from sympy.physics.quantum.hilbert import ComplexSpace
>>> c1 = ComplexSpace(2)
>>> c1
C(2)
>>> c1.dimension
2
```

```
>>> n = symbols('n')
>>> c2 = ComplexSpace(n)
>>> c2
C(n)
>>> c2.dimension
n
```

class sympy.physics.quantum.hilbert.**DirectSumHilbertSpace**(**args*)

A direct sum of Hilbert spaces [R678].

This class uses the + operator to represent direct sums between different Hilbert spaces.

A `DirectSumHilbertSpace` object takes in an arbitrary number of `HilbertSpace` objects as its arguments. Also, addition of `HilbertSpace` objects will automatically return a direct sum object.

Examples

```
>>> from sympy.physics.quantum.hilbert import ComplexSpace, FockSpace
```

```
>>> c = ComplexSpace(2)
>>> f = FockSpace()
>>> hs = c+f
>>> hs
C(2)+F
>>> hs.dimension
oo
>>> list(hs.spaces)
[C(2), F]
```

References

[R678]

classmethod eval(*args*)

Evaluates the direct product.

property spaces

A tuple of the Hilbert spaces in this direct sum.

class sympy.physics.quantum.hilbert.**FockSpace**

The Hilbert space for second quantization.

Technically, this Hilbert space is a infinite direct sum of direct products of single particle Hilbert spaces [R679]. This is a mess, so we have a class to represent it directly.

Examples

```
>>> from sympy.physics.quantum.hilbert import FockSpace
>>> hs = FockSpace()
>>> hs
F
>>> hs.dimension
oo
```

References

[R679]

class sympy.physics.quantum.hilbert.**HilbertSpace**

An abstract Hilbert space for quantum mechanics.

In short, a Hilbert space is an abstract vector space that is complete with inner products defined [R680].

Examples

```
>>> from sympy.physics.quantum.hilbert import HilbertSpace
>>> hs = HilbertSpace()
>>> hs
H
```

References

[R680]

property dimension

Return the Hilbert dimension of the space.

class sympy.physics.quantum.hilbert.**L2**(*interval*)

The Hilbert space of square integrable functions on an interval.

An L2 object takes in a single SymPy Interval argument which represents the interval its functions (vectors) are defined on.

Examples

```
>>> from sympy import Interval, oo
>>> from sympy.physics.quantum.hilbert import L2
>>> hs = L2(Interval(0,oo))
>>> hs
L2(Interval(0, oo))
>>> hs.dimension
oo
>>> hs.interval
Interval(0, oo)
```

class sympy.physics.quantum.hilbert.**TensorPowerHilbertSpace**(**args*)

An exponentiated Hilbert space [R681].

Tensor powers (repeated tensor products) are represented by the operator ** Identical Hilbert spaces that are multiplied together will be automatically combined into a single tensor power object.

Any Hilbert space, product, or sum may be raised to a tensor power. The `TensorPowerHilbertSpace` takes two arguments: the Hilbert space; and the tensor power (number).

Examples

```
>>> from sympy.physics.quantum.hilbert import ComplexSpace, FockSpace
>>> from sympy import symbols
```

```
>>> n = symbols('n')
>>> c = ComplexSpace(2)
>>> hs = c**n
>>> hs
C(2)**n
>>> hs.dimension
2**n
```

```
>>> c = ComplexSpace(2)
>>> c*c
C(2)**2
>>> f = FockSpace()
>>> c*f*f
C(2)*F**2
```

References

[R681]

class sympy.physics.quantum.hilbert.**TensorProductHilbertSpace**(**args*)

A tensor product of Hilbert spaces [R682].

The tensor product between Hilbert spaces is represented by the operator * Products of the same Hilbert space will be combined into tensor powers.

A `TensorProductHilbertSpace` object takes in an arbitrary number of `HilbertSpace` objects as its arguments. In addition, multiplication of `HilbertSpace` objects will automatically return this tensor product object.

Examples

```
>>> from sympy.physics.quantum.hilbert import ComplexSpace, FockSpace
>>> from sympy import symbols
```

```
>>> c = ComplexSpace(2)
>>> f = FockSpace()
>>> hs = c*f
>>> hs
C(2)*F
>>> hs.dimension
oo
>>> hs.spaces
(C(2), F)
```

```
>>> c1 = ComplexSpace(2)
>>> n = symbols('n')
>>> c2 = ComplexSpace(n)
>>> hs = c1*c2
>>> hs
C(2)*C(n)
>>> hs.dimension
2*n
```

References

[R682]

classmethod eval(*args*)

Evaluates the direct product.

property spaces

A tuple of the Hilbert spaces in this tensor product.

Operator

Quantum mechanical operators.

TODO:

- Fix early 0 in apply_operators.
- Debug and test apply_operators.
- Get cse working with classes in this file.
- Doctests and documentation of special methods for InnerProduct, Commutator, Anti-Commutator, represent, apply_operators.

class sympy.physics.quantum.operator.**DifferentialOperator**(**args*, ***kwargs*)

An operator for representing the differential operator, i.e. d/dx

It is initialized by passing two arguments. The first is an arbitrary expression that involves a function, such as `Derivative(f(x), x)`. The second is the function (e.g. `f(x)`) which we are to replace with the `Wavefunction` that this `DifferentialOperator` is applied to.

Parameters

expr : Expr

The arbitrary expression which the appropriate Wavefunction is to be substituted into

func : Expr

A function (e.g. f(x)) which is to be replaced with the appropriate Wavefunction when this DifferentialOperator is applied

Examples

You can define a completely arbitrary expression and specify where the Wavefunction is to be substituted

```
>>> from sympy import Derivative, Function, Symbol
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy.physics.quantum.qapply import qapply
>>> f = Function('f')
>>> x = Symbol('x')
>>> d = DifferentialOperator(1/x*Derivative(f(x), x), f(x))
>>> w = Wavefunction(x**2, x)
>>> d.function
f(x)
>>> d.variables
(x,)
>>> qapply(d*w)
Wavefunction(2, x)
```

property expr

Returns the arbitrary expression which is to have the Wavefunction substituted into it

Examples

```
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy import Function, Symbol, Derivative
>>> x = Symbol('x')
>>> f = Function('f')
>>> d = DifferentialOperator(Derivative(f(x), x), f(x))
>>> d.expr
Derivative(f(x), x)
>>> y = Symbol('y')
>>> d = DifferentialOperator(Derivative(f(x, y), x) +
...                          Derivative(f(x, y), y), f(x, y))
>>> d.expr
Derivative(f(x, y), x) + Derivative(f(x, y), y)
```

property free_symbols

Return the free symbols of the expression.

property function

Returns the function which is to be replaced with the Wavefunction

Examples

```
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy import Function, Symbol, Derivative
>>> x = Symbol('x')
>>> f = Function('f')
>>> d = DifferentialOperator(Derivative(f(x), x), f(x))
>>> d.function
f(x)
>>> y = Symbol('y')
>>> d = DifferentialOperator(Derivative(f(x, y), x) +
...                          Derivative(f(x, y), y), f(x, y))
>>> d.function
f(x, y)
```

property variables

Returns the variables with which the function in the specified arbitrary expression is evaluated

Examples

```
>>> from sympy.physics.quantum.operator import DifferentialOperator
>>> from sympy import Symbol, Function, Derivative
>>> x = Symbol('x')
>>> f = Function('f')
>>> d = DifferentialOperator(1/x*Derivative(f(x), x), f(x))
>>> d.variables
(x,)
>>> y = Symbol('y')
```

(continues on next page)

(continued from previous page)

```
>>> d = DifferentialOperator(Derivative(f(x, y), x) +
...                          Derivative(f(x, y), y), f(x, y))
>>> d.variables
(x, y)
```

class sympy.physics.quantum.operator.**HermitianOperator**(**args*, ***kwargs*)

A Hermitian operator that satisfies H == Dagger(H).

Parameters

args : tuple

The list of numbers or parameters that uniquely specify the operator. For time-dependent operators, this will include the time.

Examples

```
>>> from sympy.physics.quantum import Dagger, HermitianOperator
>>> H = HermitianOperator('H')
>>> Dagger(H)
H
```

class sympy.physics.quantum.operator.**IdentityOperator**(**args*, ***kwargs*)

An identity operator I that satisfies op * I == I * op == op for any operator op.

Parameters

N : Integer

Optional parameter that specifies the dimension of the Hilbert space of operator. This is used when generating a matrix representation.

Examples

```
>>> from sympy.physics.quantum import IdentityOperator
>>> IdentityOperator()
I
```

class sympy.physics.quantum.operator.**Operator**(**args*, ***kwargs*)

Base class for non-commuting quantum operators.

An operator maps between quantum states [R684]. In quantum mechanics, observables (including, but not limited to, measured physical values) are represented as Hermitian operators [R685].

Parameters

args : tuple

The list of numbers or parameters that uniquely specify the operator. For time-dependent operators, this will include the time.

Examples

Create an operator and examine its attributes:

```
>>> from sympy.physics.quantum import Operator
>>> from sympy import I
>>> A = Operator('A')
>>> A
A
>>> A.hilbert_space
H
>>> A.label
(A,)
>>> A.is_commutative
False
```

Create another operator and do some arithmetic operations:

```
>>> B = Operator('B')
>>> C = 2*A*A + I*B
>>> C
2*A**2 + I*B
```

Operators do not commute:

```
>>> A.is_commutative
False
>>> B.is_commutative
False
>>> A*B == B*A
False
```

Polymonials of operators respect the commutation properties:

```
>>> e = (A+B)**3
>>> e.expand()
A*B*A + A*B**2 + A**2*B + A**3 + B*A*B + B*A**2 + B**2*A + B**3
```

Operator inverses are handle symbolically:

```
>>> A.inv()
A**(-1)
>>> A*A.inv()
1
```

References

[R684], [R685]

class sympy.physics.quantum.operator.**OuterProduct**(**args*, ***old_assumptions*)

An unevaluated outer product between a ket and bra.

This constructs an outer product between any subclass of `KetBase` and `BraBase` as `|a><b|`. An `OuterProduct` inherits from Operator as they act as operators in quantum expressions. For reference see [R686].

Parameters

ket : KetBase

The ket on the left side of the outer product.

bar : BraBase

The bra on the right side of the outer product.

Examples

Create a simple outer product by hand and take its dagger:

```
>>> from sympy.physics.quantum import Ket, Bra, OuterProduct, Dagger
>>> from sympy.physics.quantum import Operator

>>> k = Ket('k')
>>> b = Bra('b')
>>> op = OuterProduct(k, b)
>>> op
|k><b|
>>> op.hilbert_space
H
>>> op.ket
|k>
>>> op.bra
<b|
>>> Dagger(op)
|b><k|
```

In simple products of kets and bras outer products will be automatically identified and created:

```
>>> k*b
|k><b|
```

But in more complex expressions, outer products are not automatically created:

```
>>> A = Operator('A')
>>> A*k*b
A*|k>*<b|
```

A user can force the creation of an outer product in a complex expression by using parentheses to group the ket and bra:

```
>>> A*(k*b)
A*|k><b|
```

References

[R686]

property bra

Return the bra on the right side of the outer product.

property ket

Return the ket on the left side of the outer product.

class sympy.physics.quantum.operator.**UnitaryOperator**(**args*, ***kwargs*)

A unitary operator that satisfies U*Dagger(U) == 1.

Parameters

args : tuple

The list of numbers or parameters that uniquely specify the operator. For time-dependent operators, this will include the time.

Examples

```
>>> from sympy.physics.quantum import Dagger, UnitaryOperator
>>> U = UnitaryOperator('U')
>>> U*Dagger(U)
1
```

Operator/State Helper Functions

A module for mapping operators to their corresponding eigenstates and vice versa

It contains a global dictionary with eigenstate-operator pairings. If a new state-operator pair is created, this dictionary should be updated as well.

It also contains functions operators_to_state and state_to_operators for mapping between the two. These can handle both classes and instances of operators and states. See the individual function descriptions for details.

TODO List: - Update the dictionary with a complete list of state-operator pairs

sympy.physics.quantum.operatorset.**operators_to_state**(*operators*, ***options*)

Returns the eigenstate of the given operator or set of operators

A global function for mapping operator classes to their associated states. It takes either an Operator or a set of operators and returns the state associated with these.

This function can handle both instances of a given operator or just the class itself (i.e. both XOp() and XOp)

There are multiple use cases to consider:

1) A class or set of classes is passed: First, we try to instantiate default instances for these operators. If this fails, then the class is simply returned. If we succeed in instantiating default instances, then we try to call state._operators_to_state on the operator instances. If this fails, the class is returned. Otherwise, the instance returned by _operators_to_state is returned.

2) An instance or set of instances is passed: In this case, state._operators_to_state is called on the instances passed. If this fails, a state class is returned. If the method returns an instance, that instance is returned.

In both cases, if the operator class or set does not exist in the state_mapping dictionary, None is returned.

Parameters

arg: Operator or set

The class or instance of the operator or set of operators to be mapped to a state

Examples

```
>>> from sympy.physics.quantum.cartesian import XOp, PxOp
>>> from sympy.physics.quantum.operatorset import operators_to_state
>>> from sympy.physics.quantum.operator import Operator
>>> operators_to_state(XOp)
|x>
>>> operators_to_state(XOp())
|x>
>>> operators_to_state(PxOp)
|px>
>>> operators_to_state(PxOp())
|px>
>>> operators_to_state(Operator)
|psi>
>>> operators_to_state(Operator())
|psi>
```

sympy.physics.quantum.operatorset.**state_to_operators**(*state*, ***options*)

Returns the operator or set of operators corresponding to the given eigenstate

A global function for mapping state classes to their associated operators or sets of operators. It takes either a state class or instance.

This function can handle both instances of a given state or just the class itself (i.e. both XKet() and XKet)

There are multiple use cases to consider:

1) A state class is passed: In this case, we first try instantiating a default instance of the class. If this succeeds, then we try to call state._state_to_operators on that instance. If the creation of the default instance or if the calling of _state_to_operators fails, then either an operator class or set of operator classes is returned. Otherwise, the appropriate operator instances are returned.

2) A state instance is returned: Here, state._state_to_operators is called for the instance. If this fails, then a class or set of operator classes is returned. Otherwise, the instances are returned.

In either case, if the state's class does not exist in state_mapping, None is returned.

Parameters

arg: StateBase class or instance (or subclasses)

The class or instance of the state to be mapped to an operator or set of operators

Examples

```
>>> from sympy.physics.quantum.cartesian import XKet, PxKet, XBra, PxBra
>>> from sympy.physics.quantum.operatorset import state_to_operators
>>> from sympy.physics.quantum.state import Ket, Bra
>>> state_to_operators(XKet)
X
>>> state_to_operators(XKet())
X
>>> state_to_operators(PxKet)
Px
>>> state_to_operators(PxKet())
Px
>>> state_to_operators(PxBra)
Px
>>> state_to_operators(XBra)
X
>>> state_to_operators(Ket)
0
>>> state_to_operators(Bra)
0
```


Qapply

Logic for applying operators to states.

Todo: * Sometimes the final result needs to be expanded, we should do this by hand.

`sympy.physics.quantum.qapply.`**`qapply`**(*e*, ***options*)

Apply operators to states in a quantum expression.

Parameters

e : Expr

The expression containing operators and states. This expression tree will be walked to find operators acting on states symbolically.

options : dict

A dict of key/value pairs that determine how the operator actions are carried out.

The following options are valid:

- `dagger`: try to apply Dagger operators to the left (default: False).
- `ip_doit`: call `.doit()` in inner products when they are encountered (default: True).

Returns
e : Expr

The original expression, but with the operators applied to states.

Examples

```
>>> from sympy.physics.quantum import qapply, Ket, Bra
>>> b = Bra('b')
>>> k = Ket('k')
>>> A = k * b
>>> A
|k><b|
>>> qapply(A * b.dual / (b * b.dual))
|k>
>>> qapply(k.dual * A / (k.dual * k), dagger=True)
<b|
>>> qapply(k.dual * A / (k.dual * k))
<k|*|k><b|/<k|k>
```

Represent

Logic for representing operators in state in various bases.

TODO:

- Get represent working with continuous hilbert spaces.
- Document default basis functionality.

`sympy.physics.quantum.represent.`**`enumerate_states`**(**args*, ***options*)

Returns instances of the given state with dummy indices appended

Operates in two different modes:

1. Two arguments are passed to it. The first is the base state which is to be indexed, and the second argument is a list of indices to append.
2. Three arguments are passed. The first is again the base state to be indexed. The second is the start index for counting. The final argument is the number of kets you wish to receive.

Tries to call state._enumerate_state. If this fails, returns an empty list

Parameters
args : list

See list of operation modes above for explanation

Examples

```
>>> from sympy.physics.quantum.cartesian import XBra, XKet
>>> from sympy.physics.quantum.represent import enumerate_states
>>> test = XKet('foo')
>>> enumerate_states(test, 1, 3)
[|foo_1>, |foo_2>, |foo_3>]
>>> test2 = XBra('bar')
>>> enumerate_states(test2, [4, 5, 10])
[<bar_4|, <bar_5|, <bar_10|]
```

sympy.physics.quantum.represent.**get_basis**(*expr*, *, *basis=None*, *replace_none=True*, ***options*)

Returns a basis state instance corresponding to the basis specified in options=s. If no basis is specified, the function tries to form a default basis state of the given expression.

There are three behaviors:

1. The basis specified in options is already an instance of StateBase. If this is the case, it is simply returned. If the class is specified but not an instance, a default instance is returned.
2. The basis specified is an operator or set of operators. If this is the case, the operator_to_state mapping method is used.
3. No basis is specified. If expr is a state, then a default instance of its class is returned. If expr is an operator, then it is mapped to the corresponding state. If it is neither, then we cannot obtain the basis state.

If the basis cannot be mapped, then it is not changed.

This will be called from within represent, and represent will only pass QExpr's.

TODO (?): Support for Muls and other types of expressions?

Parameters

expr : Operator or StateBase

Expression whose basis is sought

Examples

```
>>> from sympy.physics.quantum.represent import get_basis
>>> from sympy.physics.quantum.cartesian import XOp, XKet, PxOp, PxKet
>>> x = XKet()
>>> X = XOp()
>>> get_basis(x)
|x>
>>> get_basis(X)
|x>
>>> get_basis(x, basis=PxOp())
|px>
>>> get_basis(x, basis=PxKet)
|px>
```

sympy.physics.quantum.represent.**integrate_result**(*orig_expr, result, **options*)

Returns the result of integrating over any unities (|x><x|) in the given expression. Intended for integrating over the result of representations in continuous bases.

This function integrates over any unities that may have been inserted into the quantum expression and returns the result. It uses the interval of the Hilbert space of the basis state passed to it in order to figure out the limits of integration. The unities option must be specified for this to work.

Note: This is mostly used internally by represent(). Examples are given merely to show the use cases.

Parameters

orig_expr : quantum expression

The original expression which was to be represented

result: Expr

The resulting representation that we wish to integrate over

Examples

```
>>> from sympy import symbols, DiracDelta
>>> from sympy.physics.quantum.represent import integrate_result
>>> from sympy.physics.quantum.cartesian import XOp, XKet
>>> x_ket = XKet()
>>> X_op = XOp()
>>> x, x_1, x_2 = symbols('x, x_1, x_2')
>>> integrate_result(X_op*x_ket, x*DiracDelta(x-x_1)*DiracDelta(x_1-x_2))
x*DiracDelta(x - x_1)*DiracDelta(x_1 - x_2)
>>> integrate_result(X_op*x_ket, x*DiracDelta(x-x_1)*DiracDelta(x_1-x_2),
...     unities=[1])
x*DiracDelta(x - x_2)
```

sympy.physics.quantum.represent.**rep_expectation**(*expr, **options*)

Returns an <x'|A|x> type representation for the given operator.

Parameters

expr : Operator

Operator to be represented in the specified basis

Examples

```
>>> from sympy.physics.quantum.cartesian import XOp, PxOp, PxKet
>>> from sympy.physics.quantum.represent import rep_expectation
>>> rep_expectation(XOp())
x_1*DiracDelta(x_1 - x_2)
>>> rep_expectation(XOp(), basis=PxOp())
<px_2|*X*|px_1>
>>> rep_expectation(XOp(), basis=PxKet())
<px_2|*X*|px_1>
```

sympy.physics.quantum.represent.**rep_innerproduct**(*expr*, ***options*)

Returns an innerproduct like representation (e.g. <x'|x>) for the given state.

Attempts to calculate inner product with a bra from the specified basis. Should only be passed an instance of KetBase or BraBase

Parameters

expr : KetBase or BraBase

The expression to be represented

Examples

```
>>> from sympy.physics.quantum.represent import rep_innerproduct
>>> from sympy.physics.quantum.cartesian import XOp, XKet, PxOp, PxKet
>>> rep_innerproduct(XKet())
DiracDelta(x - x_1)
>>> rep_innerproduct(XKet(), basis=PxOp())
sqrt(2)*exp(-I*px_1*x/hbar)/(2*sqrt(hbar)*sqrt(pi))
>>> rep_innerproduct(PxKet(), basis=XOp())
sqrt(2)*exp(I*px*x_1/hbar)/(2*sqrt(hbar)*sqrt(pi))
```

sympy.physics.quantum.represent.**represent**(*expr*, ***options*)

Represent the quantum expression in the given basis.

In quantum mechanics abstract states and operators can be represented in various basis sets. Under this operation the follow transforms happen:

- Ket -> column vector or function
- Bra -> row vector of function
- Operator -> matrix or differential operator

This function is the top-level interface for this action.

This function walks the SymPy expression tree looking for `QExpr` instances that have a `_represent` method. This method is then called and the object is replaced by the representation returned by this method. By default, the `_represent` method will dispatch to other methods that handle the representation logic for a particular basis set. The naming convention for these methods is the following:

```
def _represent_FooBasis(self, e, basis, **options)
```

This function will have the logic for representing instances of its class in the basis set having a class named `FooBasis`.

Parameters

expr : Expr

The expression to represent.

basis : Operator, basis set

An object that contains the information about the basis set. If an operator is used, the basis is assumed to be the orthonormal eigenvectors of that operator. In general though, the basis argument can be any object that contains the basis set information.

options : dict

Key/value pairs of options that are passed to the underlying method that finds the representation. These options can be used to control how the representation is done. For example, this is where the size of the basis set would be set.

Returns

e : Expr

The SymPy expression of the represented quantum expression.

Examples

Here we subclass `Operator` and `Ket` to create the z-spin operator and its spin 1/2 up eigenstate. By defining the `_represent_SzOp` method, the ket can be represented in the z-spin basis.

```
>>> from sympy.physics.quantum import Operator, represent, Ket
>>> from sympy import Matrix
```

```
>>> class SzUpKet(Ket):
...     def _represent_SzOp(self, basis, **options):
...         return Matrix([1,0])
...
>>> class SzOp(Operator):
...     pass
...
>>> sz = SzOp('Sz')
>>> up = SzUpKet('up')
>>> represent(up, basis=sz)
Matrix([
[1],
[0]])
```

Here we see an example of representations in a continuous basis. We see that the result of representing various combinations of cartesian position operators and kets give us continuous expressions involving DiracDelta functions.

```
>>> from sympy.physics.quantum.cartesian import XOp, XKet, XBra
>>> X = XOp()
>>> x = XKet()
>>> y = XBra('y')
>>> represent(X*x)
x*DiracDelta(x - x_2)
>>> represent(X*x*y)
x*DiracDelta(x - x_3)*DiracDelta(x_1 - y)
```

Spin

Quantum mechanical angular momemtum.

class sympy.physics.quantum.spin.**J2Op**(**args*, ***kwargs*)

The J^2 operator.

class sympy.physics.quantum.spin.**JxBra**(*j*, *m*)

Eigenbra of Jx.

See JzKet for the usage of spin eigenstates.

See also:

JzKet **(page 1821)**
Usage of spin states

class sympy.physics.quantum.spin.**JxBraCoupled**(*j*, *m*, *jn*, **jcoupling*)

Coupled eigenbra of Jx.

See JzKetCoupled for the usage of coupled spin eigenstates.

See also:

JzKetCoupled **(page 1823)**
Usage of coupled spin states

class sympy.physics.quantum.spin.**JxKet**(*j*, *m*)

Eigenket of Jx.

See JzKet for the usage of spin eigenstates.

See also:

JzKet **(page 1821)**
Usage of spin states

class sympy.physics.quantum.spin.**JxKetCoupled**(*j*, *m*, *jn*, **jcoupling*)

Coupled eigenket of Jx.

See JzKetCoupled for the usage of coupled spin eigenstates.

See also:

JzKetCoupled **(page 1823)**
Usage of coupled spin states

class sympy.physics.quantum.spin.**JyBra**(*j*, *m*)

Eigenbra of Jy.

See JzKet for the usage of spin eigenstates.

See also:

JzKet **(page 1821)**
Usage of spin states

class sympy.physics.quantum.spin.**JyBraCoupled**(*j*, *m*, *jn*, **jcoupling*)

Coupled eigenbra of Jy.

See JzKetCoupled for the usage of coupled spin eigenstates.

See also:

JzKetCoupled **(page 1823)**
Usage of coupled spin states

class sympy.physics.quantum.spin.**JyKet**(*j*, *m*)

Eigenket of Jy.

See JzKet for the usage of spin eigenstates.

See also:

JzKet **(page 1821)**
Usage of spin states

class sympy.physics.quantum.spin.**JyKetCoupled**(*j*, *m*, *jn*, **jcoupling*)

Coupled eigenket of Jy.

See JzKetCoupled for the usage of coupled spin eigenstates.

See also:

JzKetCoupled **(page 1823)**
Usage of coupled spin states

class sympy.physics.quantum.spin.**JzBra**(*j*, *m*)

Eigenbra of Jz.

See the JzKet for the usage of spin eigenstates.

See also:

JzKet **(page 1821)**
Usage of spin states

class sympy.physics.quantum.spin.**JzBraCoupled**(*j*, *m*, *jn*, **jcoupling*)

Coupled eigenbra of Jz.

See the JzKetCoupled for the usage of coupled spin eigenstates.

See also:

JzKetCoupled **(page 1823)**
Usage of coupled spin states

class sympy.physics.quantum.spin.**JzKet**(*j*, *m*)

Eigenket of Jz.

Spin state which is an eigenstate of the Jz operator. Uncoupled states, that is states representing the interaction of multiple separate spin states, are defined as a tensor product of states.

Parameters
j : Number, Symbol

Total spin angular momentum

m : Number, Symbol

Eigenvalue of the Jz spin operator

Examples

Normal States:

Defining simple spin states, both numerical and symbolic:

```
>>> from sympy.physics.quantum.spin import JzKet, JxKet
>>> from sympy import symbols
>>> JzKet(1, 0)
|1,0>
>>> j, m = symbols('j m')
>>> JzKet(j, m)
|j,m>
```

Rewriting the JzKet in terms of eigenkets of the Jx operator: Note: that the resulting eigenstates are JxKet's

```
>>> JzKet(1,1).rewrite("Jx")
|1,-1>/2 - sqrt(2)*|1,0>/2 + |1,1>/2
```

Get the vector representation of a state in terms of the basis elements of the Jx operator:

```
>>> from sympy.physics.quantum.represent import represent
>>> from sympy.physics.quantum.spin import Jx, Jz
>>> represent(JzKet(1,-1), basis=Jx)
Matrix([
[      1/2],
[sqrt(2)/2],
[      1/2]])
```

Apply innerproducts between states:

```
>>> from sympy.physics.quantum.innerproduct import InnerProduct
>>> from sympy.physics.quantum.spin import JxBra
>>> i = InnerProduct(JxBra(1,1), JzKet(1,1))
>>> i
<1,1|1,1>
>>> i.doit()
1/2
```

Uncoupled States:

Define an uncoupled state as a TensorProduct between two Jz eigenkets:

```
>>> from sympy.physics.quantum.tensorproduct import TensorProduct
>>> j1,m1,j2,m2 = symbols('j1 m1 j2 m2')
>>> TensorProduct(JzKet(1,0), JzKet(1,1))
|1,0>x|1,1>
>>> TensorProduct(JzKet(j1,m1), JzKet(j2,m2))
|j1,m1>x|j2,m2>
```

A TensorProduct can be rewritten, in which case the eigenstates that make up the tensor product is rewritten to the new basis:

```
>>> TensorProduct(JzKet(1,1),JxKet(1,1)).rewrite('Jz')
|1,1>x|1,-1>/2 + sqrt(2)*|1,1>x|1,0>/2 + |1,1>x|1,1>/2
```

The represent method for TensorProduct's gives the vector representation of the state. Note that the state in the product basis is the equivalent of the tensor product of the vector representation of the component eigenstates:

```
>>> represent(TensorProduct(JzKet(1,0),JzKet(1,1)))
Matrix([
[0],
[0],
[0],
[1],
[0],
[0],
[0],
[0],
[0]])
>>> represent(TensorProduct(JzKet(1,1),JxKet(1,1)), basis=Jz)
Matrix([
[      1/2],
[sqrt(2)/2],
[      1/2],
[        0],
[        0],
[        0],
[        0],
[        0],
[        0]])
```

See also:

JzKetCoupled **(page 1823)**
Coupled eigenstates

sympy.physics.quantum.tensorproduct.TensorProduct **(page 1799)**
Used to specify uncoupled states

uncouple **(page 1830)**
Uncouples states given coupling parameters

couple **(page 1829)**
Couples uncoupled states

class sympy.physics.quantum.spin.**JzKetCoupled**(*j, m, jn, *jcoupling*)

Coupled eigenket of Jz

Spin state that is an eigenket of Jz which represents the coupling of separate spin spaces.

The arguments for creating instances of JzKetCoupled are `j`, `m`, `jn` and an optional `jcoupling` argument. The `j` and `m` options are the total angular momentum quantum numbers, as used for normal states (e.g. JzKet).

The other required parameter in `jn`, which is a tuple defining the j_n angular momentum quantum numbers of the product spaces. So for example, if a state represented the coupling of the product basis state $|j_1, m_1\rangle \times |j_2, m_2\rangle$, the `jn` for this state would be `(j1, j2)`.

The final option is `jcoupling`, which is used to define how the spaces specified by `jn` are coupled, which includes both the order these spaces are coupled together and the quantum numbers that arise from these couplings. The `jcoupling` parameter itself is a list of lists, such that each of the sublists defines a single coupling between the spin spaces. If there are N coupled angular momentum spaces, that is `jn` has N elements, then there must be N-1 sublists. Each of these sublists making up the `jcoupling` parameter have length 3. The first two elements are the indices of the product spaces that are considered to be coupled together. For example, if we want to couple j_1 and j_4, the indices would be 1 and 4. If a state has already been coupled, it is referenced by the smallest index that is coupled, so if j_2 and j_4 has already been coupled to some j_{24}, then this value can be coupled by referencing it with index 2. The final element of the sublist is the quantum number of the coupled state. So putting everything together, into a valid sublist for `jcoupling`, if j_1 and j_2 are coupled to an angular momentum space with quantum number j_{12} with the value `j12`, the sublist would be `(1,2,j12)`, N-1 of these sublists are used in the list for `jcoupling`.

Note the `jcoupling` parameter is optional, if it is not specified, the default coupling is taken. This default value is to coupled the spaces in order and take the quantum number of the coupling to be the maximum value. For example, if the spin spaces are j_1, j_2, j_3, j_4, then the default coupling couples j_1 and j_2 to $j_{12} = j_1 + j_2$, then, j_{12} and j_3 are coupled to $j_{123} = j_{12} + j_3$, and finally j_{123} and j_4 to $j = j_{123} + j_4$. The jcoupling value that would correspond to this is:

```
((1,2,j1+j2),(1,3,j1+j2+j3))
```

Parameters

args : tuple

The arguments that must be passed are `j`, `m`, `jn`, and `jcoupling`. The `j` value is the total angular momentum. The `m` value is the eigenvalue of the Jz spin operator. The `jn` list are the `j` values of argular momentum spaces coupled together. The `jcoupling` parameter is an optional parameter defining how the spaces are coupled together. See the above description for how these coupling parameters are defined.

Examples

Defining simple spin states, both numerical and symbolic:

```
>>> from sympy.physics.quantum.spin import JzKetCoupled
>>> from sympy import symbols
>>> JzKetCoupled(1, 0, (1, 1))
|1,0,j1=1,j2=1>
>>> j, m, j1, j2 = symbols('j m j1 j2')
>>> JzKetCoupled(j, m, (j1, j2))
|j,m,j1=j1,j2=j2>
```

Defining coupled spin states for more than 2 coupled spaces with various coupling parameters:

```
>>> JzKetCoupled(2, 1, (1, 1, 1))
|2,1,j1=1,j2=1,j3=1,j(1,2)=2>
>>> JzKetCoupled(2, 1, (1, 1, 1), ((1,2,2),(1,3,2)) )
|2,1,j1=1,j2=1,j3=1,j(1,2)=2>
>>> JzKetCoupled(2, 1, (1, 1, 1), ((2,3,1),(1,2,2)) )
|2,1,j1=1,j2=1,j3=1,j(2,3)=1>
```

Rewriting the JzKetCoupled in terms of eigenkets of the Jx operator: Note: that the resulting eigenstates are JxKetCoupled

```
>>> JzKetCoupled(1,1,(1,1)).rewrite("Jx")
|1,-1,j1=1,j2=1>/2 - sqrt(2)*|1,0,j1=1,j2=1>/2 + |1,1,j1=1,j2=1>/2
```

The rewrite method can be used to convert a coupled state to an uncoupled state. This is done by passing coupled=False to the rewrite function:

```
>>> JzKetCoupled(1, 0, (1, 1)).rewrite('Jz', coupled=False)
-sqrt(2)*|1,-1>x|1,1>/2 + sqrt(2)*|1,1>x|1,-1>/2
```

Get the vector representation of a state in terms of the basis elements of the Jx operator:

```
>>> from sympy.physics.quantum.represent import represent
>>> from sympy.physics.quantum.spin import Jx
>>> from sympy import S
>>> represent(JzKetCoupled(1,-1,(S(1)/2,S(1)/2)), basis=Jx)
Matrix([
[        0],
[      1/2],
[sqrt(2)/2],
[      1/2]])
```

See also:

***JzKet* (page 1821)**
Normal spin eigenstates

***uncouple* (page 1830)**
Uncoupling of coupling spin states

***couple* (page 1829)**
Coupling of uncoupled spin states

class sympy.physics.quantum.spin.**JzOp**(**args*, ***kwargs*)

The Jz operator.

class sympy.physics.quantum.spin.**Rotation**(**args*, ***kwargs*)

Wigner D operator in terms of Euler angles.

Defines the rotation operator in terms of the Euler angles defined by the z-y-z convention for a passive transformation. That is the coordinate axes are rotated first about the z-axis, giving the new x'-y'-z' axes. Then this new coordinate system is rotated about the new y'-axis, giving new x''-y''-z'' axes. Then this new coordinate system is rotated about the z''-axis. Conventions follow those laid out in [R687].

Parameters
alpha : Number, Symbol

First Euler Angle

beta : Number, Symbol

Second Euler angle

gamma : Number, Symbol

Third Euler angle

Examples

A simple example rotation operator:

```
>>> from sympy import pi
>>> from sympy.physics.quantum.spin import Rotation
>>> Rotation(pi, 0, pi/2)
R(pi,0,pi/2)
```

With symbolic Euler angles and calculating the inverse rotation operator:

```
>>> from sympy import symbols
>>> a, b, c = symbols('a b c')
>>> Rotation(a, b, c)
R(a,b,c)
>>> Rotation(a, b, c).inverse()
R(-c,-b,-a)
```

See also:

***WignerD* (page 1828)**
Symbolic Wigner-D function

***D* (page 1826)**
Wigner-D function

***d* (page 1827)**
Wigner small-d function

References

[R687]

classmethod D(*j, m, mp, alpha, beta, gamma*)

Wigner D-function.

Returns an instance of the WignerD class corresponding to the Wigner-D function specified by the parameters.

Parameters

j : Number

Total angular momentum

m : Number

Eigenvalue of angular momentum along axis after rotation

mp : Number

Eigenvalue of angular momentum along rotated axis

alpha : Number, Symbol

First Euler angle of rotation

beta : Number, Symbol

Second Euler angle of rotation

gamma : Number, Symbol

Third Euler angle of rotation

Examples

Return the Wigner-D matrix element for a defined rotation, both numerical and symbolic:

```
>>> from sympy.physics.quantum.spin import Rotation
>>> from sympy import pi, symbols
>>> alpha, beta, gamma = symbols('alpha beta gamma')
>>> Rotation.D(1, 1, 0,pi, pi/2,-pi)
WignerD(1, 1, 0, pi, pi/2, -pi)
```

See also:

WignerD **(page 1828)**
Symbolic Wigner-D function

classmethod d(*j*, *m*, *mp*, *beta*)

Wigner small-d function.

Returns an instance of the WignerD class corresponding to the Wigner-D function specified by the parameters with the alpha and gamma angles given as 0.

Parameters

j : Number

Total angular momentum

m : Number

Eigenvalue of angular momentum along axis after rotation

mp : Number

Eigenvalue of angular momentum along rotated axis

beta : Number, Symbol

Second Euler angle of rotation

Examples

Return the Wigner-D matrix element for a defined rotation, both numerical and symbolic:

```
>>> from sympy.physics.quantum.spin import Rotation
>>> from sympy import pi, symbols
>>> beta = symbols('beta')
>>> Rotation.d(1, 1, 0, pi/2)
WignerD(1, 1, 0, 0, pi/2, 0)
```

See also:

***WignerD* (page 1828)**
Symbolic Wigner-D function

class sympy.physics.quantum.spin.**WignerD**(**args*, ***hints*)

Wigner-D function

The Wigner D-function gives the matrix elements of the rotation operator in the jm-representation. For the Euler angles α, β, γ, the D-function is defined such that:

$$< j, m|\mathcal{R}(\alpha, \beta, \gamma)|j', m' >= \delta_{jj'} D(j, m, m', \alpha, \beta, \gamma)$$

Where the rotation operator is as defined by the Rotation class [R688].

The Wigner D-function defined in this way gives:

$$D(j, m, m', \alpha, \beta, \gamma) = e^{-im\alpha} d(j, m, m', \beta) e^{-im'\gamma}$$

Where d is the Wigner small-d function, which is given by Rotation.d.

The Wigner small-d function gives the component of the Wigner D-function that is determined by the second Euler angle. That is the Wigner D-function is:

$$D(j, m, m', \alpha, \beta, \gamma) = e^{-im\alpha} d(j, m, m', \beta) e^{-im'\gamma}$$

Where d is the small-d function. The Wigner D-function is given by Rotation.D.

Note that to evaluate the D-function, the j, m and mp parameters must be integer or half integer numbers.

Parameters

j : Number

Total angular momentum

m : Number

Eigenvalue of angular momentum along axis after rotation

mp : Number

Eigenvalue of angular momentum along rotated axis

alpha : Number, Symbol

First Euler angle of rotation

beta : Number, Symbol

Second Euler angle of rotation

gamma : Number, Symbol

Third Euler angle of rotation

Examples

Evaluate the Wigner-D matrix elements of a simple rotation:

```
>>> from sympy.physics.quantum.spin import Rotation
>>> from sympy import pi
>>> rot = Rotation.D(1, 1, 0, pi, pi/2, 0)
>>> rot
WignerD(1, 1, 0, pi, pi/2, 0)
>>> rot.doit()
sqrt(2)/2
```

Evaluate the Wigner-d matrix elements of a simple rotation

```
>>> rot = Rotation.d(1, 1, 0, pi/2)
>>> rot
WignerD(1, 1, 0, 0, pi/2, 0)
>>> rot.doit()
-sqrt(2)/2
```

See also:

Rotation **(page 1825)**
Rotation operator

References

[R688]

sympy.physics.quantum.spin.**couple**(*expr*, *jcoupling_list=None*)

Couple a tensor product of spin states

This function can be used to couple an uncoupled tensor product of spin states. All of the eigenstates to be coupled must be of the same class. It will return a linear combination of eigenstates that are subclasses of CoupledSpinState determined by Clebsch-Gordan angular momentum coupling coefficients.

Parameters

expr : Expr

An expression involving TensorProducts of spin states to be coupled. Each state must be a subclass of SpinState and they all must be the same class.

jcoupling_list : list or tuple

Elements of this list are sub-lists of length 2 specifying the order of the coupling of the spin spaces. The length of this must be N-1, where N is the number of states in the tensor product to be coupled. The elements of this sublist are the same as the first two elements of each sublist in the `jcoupling` parameter defined for JzKetCoupled. If this

parameter is not specified, the default value is taken, which couples the first and second product basis spaces, then couples this new coupled space to the third product space, etc

Examples

Couple a tensor product of numerical states for two spaces:

```
>>> from sympy.physics.quantum.spin import JzKet, couple
>>> from sympy.physics.quantum.tensorproduct import TensorProduct
>>> couple(TensorProduct(JzKet(1,0), JzKet(1,1)))
-sqrt(2)*|1,1,j1=1,j2=1>/2 + sqrt(2)*|2,1,j1=1,j2=1>/2
```

Numerical coupling of three spaces using the default coupling method, i.e. first and second spaces couple, then this couples to the third space:

```
>>> couple(TensorProduct(JzKet(1,1), JzKet(1,1), JzKet(1,0)))
sqrt(6)*|2,2,j1=1,j2=1,j3=1,j(1,2)=2>/3 + sqrt(3)*|3,2,j1=1,j2=1,j3=1,
 ↪j(1,2)=2>/3
```

Perform this same coupling, but we define the coupling to first couple the first and third spaces:

```
>>> couple(TensorProduct(JzKet(1,1), JzKet(1,1), JzKet(1,0)), ((1,3),(1,
 ↪2)) )
sqrt(2)*|2,2,j1=1,j2=1,j3=1,j(1,3)=1>/2 - sqrt(6)*|2,2,j1=1,j2=1,j3=1,
 ↪j(1,3)=2>/6 + sqrt(3)*|3,2,j1=1,j2=1,j3=1,j(1,3)=2>/3
```

Couple a tensor product of symbolic states:

```
>>> from sympy import symbols
>>> j1,m1,j2,m2 = symbols('j1 m1 j2 m2')
>>> couple(TensorProduct(JzKet(j1,m1), JzKet(j2,m2)))
Sum(CG(j1, m1, j2, m2, j, m1 + m2)*|j,m1 + m2,j1=j1,j2=j2>, (j, m1 + m2,␣
 ↪j1 + j2))
```

sympy.physics.quantum.spin.**uncouple**(*expr*, *jn=None*, *jcoupling_list=None*)

Uncouple a coupled spin state

Gives the uncoupled representation of a coupled spin state. Arguments must be either a spin state that is a subclass of CoupledSpinState or a spin state that is a subclass of SpinState and an array giving the j values of the spaces that are to be coupled

Parameters

expr : Expr

The expression containing states that are to be coupled. If the states are a subclass of SpinState, the `jn` and `jcoupling` parameters must be defined. If the states are a subclass of CoupledSpinState, `jn` and `jcoupling` will be taken from the state.

jn : list or tuple

The list of the j-values that are coupled. If state is a CoupledSpinState, this parameter is ignored. This must be defined if state is not

a subclass of CoupledSpinState. The syntax of this parameter is the same as the `jn` parameter of JzKetCoupled.

jcoupling_list : list or tuple

The list defining how the j-values are coupled together. If state is a CoupledSpinState, this parameter is ignored. This must be defined if state is not a subclass of CoupledSpinState. The syntax of this parameter is the same as the `jcoupling` parameter of JzKetCoupled.

Examples

Uncouple a numerical state using a CoupledSpinState state:

```
>>> from sympy.physics.quantum.spin import JzKetCoupled, uncouple
>>> from sympy import S
>>> uncouple(JzKetCoupled(1, 0, (S(1)/2, S(1)/2)))
sqrt(2)*|1/2,-1/2>x|1/2,1/2>/2 + sqrt(2)*|1/2,1/2>x|1/2,-1/2>/2
```

Perform the same calculation using a SpinState state:

```
>>> from sympy.physics.quantum.spin import JzKet
>>> uncouple(JzKet(1, 0), (S(1)/2, S(1)/2))
sqrt(2)*|1/2,-1/2>x|1/2,1/2>/2 + sqrt(2)*|1/2,1/2>x|1/2,-1/2>/2
```

Uncouple a numerical state of three coupled spaces using a CoupledSpinState state:

```
>>> uncouple(JzKetCoupled(1, 1, (1, 1, 1), ((1,3,1),(1,2,1)) ))
|1,-1>x|1,1>x|1,1>/2 - |1,0>x|1,0>x|1,1>/2 + |1,1>x|1,0>x|1,0>/2 - |1,1>
↪x|1,1>x|1,-1>/2
```

Perform the same calculation using a SpinState state:

```
>>> uncouple(JzKet(1, 1), (1, 1, 1), ((1,3,1),(1,2,1)) )
|1,-1>x|1,1>x|1,1>/2 - |1,0>x|1,0>x|1,1>/2 + |1,1>x|1,0>x|1,0>/2 - |1,1>
↪x|1,1>x|1,-1>/2
```

Uncouple a symbolic state using a CoupledSpinState state:

```
>>> from sympy import symbols
>>> j,m,j1,j2 = symbols('j m j1 j2')
>>> uncouple(JzKetCoupled(j, m, (j1, j2)))
Sum(CG(j1, m1, j2, m2, j, m)*|j1,m1>x|j2,m2>, (m1, -j1, j1), (m2, -j2,␣
↪j2))
```

Perform the same calculation using a SpinState state

```
>>> uncouple(JzKet(j, m), (j1, j2))
Sum(CG(j1, m1, j2, m2, j, m)*|j1,m1>x|j2,m2>, (m1, -j1, j1), (m2, -j2,␣
↪j2))
```

State

Dirac notation for states.

class sympy.physics.quantum.state.**Bra**(**args*, ***kwargs*)

A general time-independent Bra in quantum mechanics.

Inherits from State and BraBase. A Bra is the dual of a Ket [R689]. This class and its subclasses will be the main classes that users will use for expressing Bras in Dirac notation.

Parameters

args : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time.

Examples

Create a simple Bra and look at its properties:

```
>>> from sympy.physics.quantum import Bra
>>> from sympy import symbols, I
>>> b = Bra('psi')
>>> b
<psi|
>>> b.hilbert_space
H
>>> b.is_commutative
False
```

Bra's know about their dual Ket's:

```
>>> b.dual
|psi>
>>> b.dual_class()
<class 'sympy.physics.quantum.state.Ket'>
```

Like Kets, Bras can have compound labels and be manipulated in a similar manner:

```
>>> n, m = symbols('n,m')
>>> b = Bra(n,m) - I*Bra(m,n)
>>> b
-I*<mn| + <nm|
```

Symbols in a Bra can be substituted using `.subs`:

```
>>> b.subs(n,m)
<mm| - I*<mm|
```

References

[R689]

class sympy.physics.quantum.state.**BraBase**(**args*, ***kwargs*)

Base class for Bras.

This class defines the dual property and the brackets for printing. This is an abstract base class and you should not instantiate it directly, instead use Bra.

class sympy.physics.quantum.state.**Ket**(**args*, ***kwargs*)

A general time-independent Ket in quantum mechanics.

Inherits from State and KetBase. This class should be used as the base class for all physical, time-independent Kets in a system. This class and its subclasses will be the main classes that users will use for expressing Kets in Dirac notation [R690].

Parameters

args : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time.

Examples

Create a simple Ket and looking at its properties:

```
>>> from sympy.physics.quantum import Ket
>>> from sympy import symbols, I
>>> k = Ket('psi')
>>> k
|psi>
>>> k.hilbert_space
H
>>> k.is_commutative
False
>>> k.label
(psi,)
```

Ket's know about their associated bra:

```
>>> k.dual
<psi|
>>> k.dual_class()
<class 'sympy.physics.quantum.state.Bra'>
```

Take a linear combination of two kets:

```
>>> k0 = Ket(0)
>>> k1 = Ket(1)
>>> 2*I*k0 - 4*k1
2*I*|0> - 4*|1>
```

Compound labels are passed as tuples:

```
>>> n, m = symbols('n,m')
>>> k = Ket(n,m)
>>> k
|nm>
```

References

[R690]

class sympy.physics.quantum.state.**KetBase**(**args*, ***kwargs*)

Base class for Kets.

This class defines the dual property and the brackets for printing. This is an abstract base class and you should not instantiate it directly, instead use Ket.

class sympy.physics.quantum.state.**OrthogonalBra**(**args*, ***kwargs*)

Orthogonal Bra in quantum mechanics.

class sympy.physics.quantum.state.**OrthogonalKet**(**args*, ***kwargs*)

Orthogonal Ket in quantum mechanics.

The inner product of two states with different labels will give zero, states with the same label will give one.

```
>>> from sympy.physics.quantum import OrthogonalBra, OrthogonalKet
>>> from sympy.abc import m, n
>>> (OrthogonalBra(n)*OrthogonalKet(n)).doit()
1
>>> (OrthogonalBra(n)*OrthogonalKet(n+1)).doit()
0
>>> (OrthogonalBra(n)*OrthogonalKet(m)).doit()
<n|m>
```

class sympy.physics.quantum.state.**OrthogonalState**(**args*, ***kwargs*)

General abstract quantum state used as a base class for Ket and Bra.

class sympy.physics.quantum.state.**State**(**args*, ***kwargs*)

General abstract quantum state used as a base class for Ket and Bra.

class sympy.physics.quantum.state.**StateBase**(**args*, ***kwargs*)

Abstract base class for general abstract states in quantum mechanics.

All other state classes defined will need to inherit from this class. It carries the basic structure for all other states such as dual, _eval_adjoint and label.

This is an abstract base class and you should not instantiate it directly, instead use State.

property dual

Return the dual state of this one.

classmethod dual_class()

Return the class used to construct the dual.

property operators

Return the operator(s) that this state is an eigenstate of

class sympy.physics.quantum.state.**TimeDepBra**(**args*, ***kwargs*)

General time-dependent Bra in quantum mechanics.

This inherits from TimeDepState and BraBase and is the main class that should be used for Bras that vary with time. Its dual is a TimeDepBra.

Parameters

args : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time as the final argument.

Examples

```
>>> from sympy.physics.quantum import TimeDepBra
>>> b = TimeDepBra('psi', 't')
>>> b
<psi;t|
>>> b.time
t
>>> b.label
(psi,)
>>> b.hilbert_space
H
>>> b.dual
|psi;t>
```

class sympy.physics.quantum.state.**TimeDepKet**(**args*, ***kwargs*)

General time-dependent Ket in quantum mechanics.

This inherits from `TimeDepState` and `KetBase` and is the main class that should be used for Kets that vary with time. Its dual is a `TimeDepBra`.

Parameters

args : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time as the final argument.

Examples

Create a TimeDepKet and look at its attributes:

```
>>> from sympy.physics.quantum import TimeDepKet
>>> k = TimeDepKet('psi', 't')
>>> k
|psi;t>
>>> k.time
t
>>> k.label
(psi,)
```

(continues on next page)

(continued from previous page)

```
>>> k.hilbert_space
H
```

TimeDepKets know about their dual bra:

```
>>> k.dual
<psi;t|
>>> k.dual_class()
<class 'sympy.physics.quantum.state.TimeDepBra'>
```

class sympy.physics.quantum.state.**TimeDepState**(**args*, ***kwargs*)

Base class for a general time-dependent quantum state.

This class is used as a base class for any time-dependent state. The main difference between this class and the time-independent state is that this class takes a second argument that is the time in addition to the usual label argument.

Parameters

args : tuple

The list of numbers or parameters that uniquely specify the ket. This will usually be its symbol or its quantum numbers. For time-dependent state, this will include the time as the final argument.

property label

The label of the state.

property time

The time of the state.

class sympy.physics.quantum.state.**Wavefunction**(**args*)

Class for representations in continuous bases

This class takes an expression and coordinates in its constructor. It can be used to easily calculate normalizations and probabilities.

Parameters

expr : Expr

The expression representing the functional form of the w.f.

coords : Symbol or tuple

The coordinates to be integrated over, and their bounds

Examples

Particle in a box, specifying bounds in the more primitive way of using Piecewise:

```
>>> from sympy import Symbol, Piecewise, pi, N
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x = Symbol('x', real=True)
>>> n = 1
>>> L = 1
>>> g = Piecewise((0, x < 0), (0, x > L), (sqrt(2//L)*sin(n*pi*x/L),␣
```

(continues on next page)