(continued from previous page)

```
↪True))
>>> f = Wavefunction(g, x)
>>> f.norm
1
>>> f.is_normalized
True
>>> p = f.prob()
>>> p(0)
0
>>> p(L)
0
>>> p(0.5)
2
>>> p(0.85*L)
2*sin(0.85*pi)**2
>>> N(p(0.85*L))
0.412214747707527
```

Additionally, you can specify the bounds of the function and the indices in a more compact way:

```
>>> from sympy import symbols, pi, diff
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x,L', positive=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sqrt(2/L)*sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.norm
1
>>> f(L+1)
0
>>> f(L-1)
sqrt(2)*sin(pi*n*(L - 1)/L)/sqrt(L)
>>> f(-1)
0
>>> f(0.85)
sqrt(2)*sin(0.85*pi*n/L)/sqrt(L)
>>> f(0.85, n=1, L=1)
sqrt(2)*sin(0.85*pi)
>>> f.is_commutative
False
```

All arguments are automatically sympified, so you can define the variables as strings rather than symbols:

```
>>> expr = x**2
>>> f = Wavefunction(expr, 'x')
>>> type(f.variables[0])
<class 'sympy.core.symbol.Symbol'>
```

Derivatives of Wavefunctions will return Wavefunctions:

```
>>> diff(f, x)
Wavefunction(2*x, x)
```

property expr

Return the expression which is the functional form of the Wavefunction

Examples

```
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> f = Wavefunction(x**2, x)
>>> f.expr
x**2
```

property is_commutative

Override Function's is_commutative so that order is preserved in represented expressions

property is_normalized

Returns true if the Wavefunction is properly normalized

Examples

```
>>> from sympy import symbols, pi
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x,L', positive=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sqrt(2/L)*sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.is_normalized
True
```

property limits

Return the limits of the coordinates which the w.f. depends on If no limits are specified, defaults to `(-oo, oo)`.

Examples

```
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy import symbols
>>> x, y = symbols('x, y')
>>> f = Wavefunction(x**2, (x, 0, 1))
>>> f.limits
{x: (0, 1)}
>>> f = Wavefunction(x**2, x)
>>> f.limits
```

(continues on next page)

(continued from previous page)

```
{x: (-oo, oo)}
>>> f = Wavefunction(x**2 + y**2, x, (y, -1, 2))
>>> f.limits
{x: (-oo, oo), y: (-1, 2)}
```

property norm

Return the normalization of the specified functional form.

This function integrates over the coordinates of the Wavefunction, with the bounds specified.

Examples

```
>>> from sympy import symbols, pi
>>> from sympy.functions import sqrt, sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x,L', positive=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sqrt(2/L)*sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.norm
1
>>> g = sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.norm
sqrt(2)*sqrt(L)/2
```

normalize()

Return a normalized version of the Wavefunction

Examples

```
>>> from sympy import symbols, pi
>>> from sympy.functions import sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x = symbols('x', real=True)
>>> L = symbols('L', positive=True)
>>> n = symbols('n', integer=True, positive=True)
>>> g = sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.normalize()
Wavefunction(sqrt(2)*sin(pi*n*x/L)/sqrt(L), (x, 0, L))
```

prob()

Return the absolute magnitude of the w.f., $|\psi(x)|^2$

Examples

```
>>> from sympy import symbols, pi
>>> from sympy.functions import sin
>>> from sympy.physics.quantum.state import Wavefunction
>>> x, L = symbols('x,L', real=True)
>>> n = symbols('n', integer=True)
>>> g = sin(n*pi*x/L)
>>> f = Wavefunction(g, (x, 0, L))
>>> f.prob()
Wavefunction(sin(pi*n*x/L)**2, x)
```

property variables

Return the coordinates which the wavefunction depends on

Examples

```
>>> from sympy.physics.quantum.state import Wavefunction
>>> from sympy import symbols
>>> x,y = symbols('x,y')
>>> f = Wavefunction(x*y, x, y)
>>> f.variables
(x, y)
>>> g = Wavefunction(x*y, x)
>>> g.variables
(x,)
```


Quantum Computation

Circuit Plot

Matplotlib based plotting of quantum circuits.

Todo:

- Optimize printing of large circuits.
- Get this to work with single gates.
- Do a better job checking the form of circuits to make sure it is a Mul of Gates.
- Get multi-target gates plotting.
- Get initial and final states to plot.
- Get measurements to plot. Might need to rethink measurement as a gate issue.
- Get scale and figsize to be handled in a better way.
- Write some tests/examples!

class sympy.physics.quantum.circuitplot.**CircuitPlot**(*c*, *nqubits*, ***kwargs*)

A class for managing a circuit plot.

control_line(*gate_idx*, *min_wire*, *max_wire*)
Draw a vertical control line.

control_point(*gate_idx*, *wire_idx*)
Draw a control point.

not_point(*gate_idx*, *wire_idx*)
Draw a NOT gates as the circle with plus in the middle.

one_qubit_box(*t*, *gate_idx*, *wire_idx*)
Draw a box for a single qubit gate.

swap_point(*gate_idx*, *wire_idx*)
Draw a swap point as a cross.

two_qubit_box(*t*, *gate_idx*, *wire_idx*)
Draw a box for a two qubit gate. Does not work yet.

update(*kwargs*)
Load the kwargs into the instance dict.

sympy.physics.quantum.circuitplot.**CreateCGate**(*name*, *latexname=None*)
Use a lexical closure to make a controlled gate.

class sympy.physics.quantum.circuitplot.**Mx**(**args*, ***kwargs*)
Mock-up of an x measurement gate.

This is in circuitplot rather than gate.py because it's not a real gate, it just draws one.

class sympy.physics.quantum.circuitplot.**Mz**(**args*, ***kwargs*)
Mock-up of a z measurement gate.

This is in circuitplot rather than gate.py because it's not a real gate, it just draws one.

sympy.physics.quantum.circuitplot.**circuit_plot**(*c*, *nqubits*, ***kwargs*)
Draw the circuit diagram for the circuit with nqubits.

Parameters
c : circuit

The circuit to plot. Should be a product of Gate instances.

nqubits : int

The number of qubits to include in the circuit. Must be at least as big as the largest `min_qubits` of the gates.

sympy.physics.quantum.circuitplot.**labeller**(*n*, *symbol='q'*)
Autogenerate labels for wires of quantum circuits.

Parameters
n : int

number of qubits in the circuit.

symbol : string

A character string to precede all gate labels. E.g. 'q_0', 'q_1', etc.

>>> from sympy.physics.quantum.circuitplot import labeller

>>> labeller(2)

['q_1', 'q_0']

>>> labeller(3,'j')

['j_2', 'j_1', 'j_0']

Gates

An implementation of gates that act on qubits.

Gates are unitary operators that act on the space of qubits.

Medium Term Todo:

- Optimize Gate._apply_operators_Qubit to remove the creation of many intermediate Qubit objects.
- Add commutation relationships to all operators and use this in gate_sort.
- Fix gate_sort and gate_simp.
- Get multi-target UGates plotting properly.
- Get UGate to work with either sympy/numpy matrices and output either format. This should also use the matrix slots.

class sympy.physics.quantum.gate.**CGate**(**args*, ***kwargs*)

A general unitary gate with control qubits.

A general control gate applies a target gate to a set of targets if all of the control qubits have a particular values (set by `CGate.control_value`).

Parameters

label : tuple

The label in this case has the form (controls, gate), where controls is a tuple/list of control qubits (as ints) and gate is a `Gate` instance that is the target operator.

property controls

A tuple of control qubits.

decompose(***options*)

Decompose the controlled gate into CNOT and single qubits gates.

eval_controls(*qubit*)

Return True/False to indicate if the controls are satisfied.

property gate

The non-controlled gate that will be applied to the targets.

property min_qubits

The minimum number of qubits this gate needs to act on.

property nqubits

The total number of qubits this gate acts on.

For controlled gate subclasses this includes both target and control qubits, so that, for examples the CNOT gate acts on 2 qubits.

plot_gate(*circ_plot*, *gate_idx*)

Plot the controlled gate. If *simplify_cgate* is true, simplify C-X and C-Z gates into their more familiar forms.

property targets

A tuple of target qubits.

class sympy.physics.quantum.gate.**CGateS**(**args*, ***kwargs*)

Version of CGate that allows gate simplifications. I.e. cnot looks like an oplus, cphase has dots, etc.

sympy.physics.quantum.gate.**CNOT**

alias of *CNotGate* (page 1843)

class sympy.physics.quantum.gate.**CNotGate**(**args*, ***kwargs*)

Two qubit controlled-NOT.

This gate performs the NOT or X gate on the target qubit if the control qubits all have the value 1.

Parameters

label : tuple

A tuple of the form (control, target).

Examples

```
>>> from sympy.physics.quantum.gate import CNOT
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.qubit import Qubit
>>> c = CNOT(1,0)
>>> qapply(c*Qubit('10')) # note that qubits are indexed from right to␣
↪left
|11>
```

property controls

A tuple of control qubits.

property gate

The non-controlled gate that will be applied to the targets.

property min_qubits

The minimum number of qubits this gate needs to act on.

property targets

A tuple of target qubits.

class sympy.physics.quantum.gate.**Gate**(**args*, ***kwargs*)

Non-controlled unitary gate operator that acts on qubits.

This is a general abstract gate that needs to be subclassed to do anything useful.

Parameters

label : tuple, int

A list of the target qubits (as ints) that the gate will apply to.

get_target_matrix(*format='sympy'*)

The matrix represenation of the target part of the gate.

Parameters
format : str

The format string (‘sympy’,’numpy’, etc.)

property min_qubits

The minimum number of qubits this gate needs to act on.

property nqubits

The total number of qubits this gate acts on.

For controlled gate subclasses this includes both target and control qubits, so that, for examples the CNOT gate acts on 2 qubits.

property targets

A tuple of target qubits.

sympy.physics.quantum.gate.**H**

alias of *HadamardGate* (page 1844)

class sympy.physics.quantum.gate.**HadamardGate**(**args*, ***kwargs*)

The single qubit Hadamard gate.

Parameters
target : int

The target qubit this gate will apply to.

Examples

```
>>> from sympy import sqrt
>>> from sympy.physics.quantum.qubit import Qubit
>>> from sympy.physics.quantum.gate import HadamardGate
>>> from sympy.physics.quantum.qapply import qapply
>>> qapply(HadamardGate(0)*Qubit('1'))
sqrt(2)*|0>/2 - sqrt(2)*|1>/2
>>> # Hadamard on bell state, applied on 2 qubits.
>>> psi = 1/sqrt(2)*(Qubit('00')+Qubit('11'))
>>> qapply(HadamardGate(0)*HadamardGate(1)*psi)
sqrt(2)*|00>/2 + sqrt(2)*|11>/2
```

class sympy.physics.quantum.gate.**IdentityGate**(**args*, ***kwargs*)

The single qubit identity gate.

Parameters
target : int

The target qubit this gate will apply to.

class sympy.physics.quantum.gate.**OneQubitGate**(**args*, ***kwargs*)

A single qubit unitary gate base class.

sympy.physics.quantum.gate.**Phase**

alias of *PhaseGate* (page 1844)

class sympy.physics.quantum.gate.**PhaseGate**(**args*, ***kwargs*)

The single qubit phase, or S, gate.

This gate rotates the phase of the state by pi/2 if the state is |1> and does nothing if the state is |0>.

Parameters
target : int

The target qubit this gate will apply to.

sympy.physics.quantum.gate.**S**

alias of *PhaseGate* (page 1844)

sympy.physics.quantum.gate.**SWAP**

alias of *SwapGate* (page 1845)

class sympy.physics.quantum.gate.**SwapGate**(**args*, ***kwargs*)

Two qubit SWAP gate.

This gate swap the values of the two qubits.

Parameters
label : tuple

A tuple of the form (target1, target2).

decompose(***options*)

Decompose the SWAP gate into CNOT gates.

sympy.physics.quantum.gate.**T**

alias of *TGate* (page 1845)

class sympy.physics.quantum.gate.**TGate**(**args*, ***kwargs*)

The single qubit pi/8 gate.

This gate rotates the phase of the state by pi/4 if the state is |1> and does nothing if the state is |0>.

Parameters
target : int

The target qubit this gate will apply to.

class sympy.physics.quantum.gate.**TwoQubitGate**(**args*, ***kwargs*)

A two qubit unitary gate base class.

class sympy.physics.quantum.gate.**UGate**(**args*, ***kwargs*)

General gate specified by a set of targets and a target matrix.

Parameters
label : tuple

A tuple of the form (targets, U), where targets is a tuple of the target qubits and U is a unitary matrix with dimension of len(targets).

get_target_matrix(*format='sympy'*)

The matrix rep. of the target part of the gate.

Parameters
format : str

The format string (‘sympy’,’numpy’, etc.)

property targets

A tuple of target qubits.

sympy.physics.quantum.gate.**X**

alias of *XGate* (page 1846)

class sympy.physics.quantum.gate.**XGate**(**args*, ***kwargs*)

The single qubit X, or NOT, gate.

Parameters

target : int

The target qubit this gate will apply to.

sympy.physics.quantum.gate.**Y**

alias of *YGate* (page 1846)

class sympy.physics.quantum.gate.**YGate**(**args*, ***kwargs*)

The single qubit Y gate.

Parameters

target : int

The target qubit this gate will apply to.

sympy.physics.quantum.gate.**Z**

alias of *ZGate* (page 1846)

class sympy.physics.quantum.gate.**ZGate**(**args*, ***kwargs*)

The single qubit Z gate.

Parameters

target : int

The target qubit this gate will apply to.

sympy.physics.quantum.gate.**gate_simp**(*circuit*)

Simplifies gates symbolically

It first sorts gates using gate_sort. It then applies basic simplification rules to the circuit, e.g., XGate**2 = Identity

sympy.physics.quantum.gate.**gate_sort**(*circuit*)

Sorts the gates while keeping track of commutation relations

This function uses a bubble sort to rearrange the order of gate application. Keeps track of Quantum computations special commutation relations (e.g. things that apply to the same Qubit do not commute with each other)

circuit is the Mul of gates that are to be sorted.

sympy.physics.quantum.gate.**normalized**(*normalize*)

Set flag controlling normalization of Hadamard gates by $1/\sqrt{2}$.

This is a global setting that can be used to simplify the look of various expressions, by leaving off the leading $1/\sqrt{2}$ of the Hadamard gate.

Parameters

normalize : bool

Should the Hadamard gate include the $1/\sqrt{2}$ normalization factor? When True, the Hadamard gate will have the $1/\sqrt{2}$. When False, the Hadamard gate will not have this factor.

sympy.physics.quantum.gate.**random_circuit**(*ngates*, *nqubits*, *gate_space=(<class 'sympy.physics.quantum.gate.XGate'>, <class 'sympy.physics.quantum.gate.YGate'>, <class 'sympy.physics.quantum.gate.ZGate'>, <class 'sympy.physics.quantum.gate.PhaseGate'>, <class 'sympy.physics.quantum.gate.TGate'>, <class 'sympy.physics.quantum.gate.HadamardGate'>, <class 'sympy.physics.quantum.gate.CNotGate'>, <class 'sympy.physics.quantum.gate.SwapGate'>)*)

Return a random circuit of ngates and nqubits.

This uses an equally weighted sample of (X, Y, Z, S, T, H, CNOT, SWAP) gates.

Parameters

ngates : int

The number of gates in the circuit.

nqubits : int

The number of qubits in the circuit.

gate_space : tuple

A tuple of the gate classes that will be used in the circuit. Repeating gate classes multiple times in this tuple will increase the frequency they appear in the random circuit.

Grover's Algorithm

Grover's algorithm and helper functions.

Todo:

- W gate construction (or perhaps -W gate based on Mermin's book)
- Generalize the algorithm for an unknown function that returns 1 on multiple qubit states, not just one.
- Implement _represent_ZGate in OracleGate

class sympy.physics.quantum.grover.**OracleGate**(**args*, ***kwargs*)

A black box gate.

The gate marks the desired qubits of an unknown function by flipping the sign of the qubits. The unknown function returns true when it finds its desired qubits and false otherwise.

Parameters

qubits : int

Number of qubits.

oracle : callable

A callable function that returns a boolean on a computational basis.

Examples

Apply an Oracle gate that flips the sign of `|2>` on different qubits:

```
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.grover import OracleGate
>>> f = lambda qubits: qubits == IntQubit(2)
>>> v = OracleGate(2, f)
>>> qapply(v*IntQubit(2))
-|2>
>>> qapply(v*IntQubit(3))
|3>
```

property search_function

The unknown function that helps find the sought after qubits.

property targets

A tuple of target qubits.

class sympy.physics.quantum.grover.**WGate**(**args*, ***kwargs*)

General n qubit W Gate in Grover's algorithm.

The gate performs the operation `2|phi><phi| - 1` on some qubits. `|phi> = (tensor product of n Hadamards)*(|0> with n qubits)`

Parameters

nqubits : int

The number of qubits to operate on

sympy.physics.quantum.grover.**apply_grover**(*oracle*, *nqubits*, *iterations=None*)

Applies grover's algorithm.

Parameters

oracle : callable

The unknown callable function that returns true when applied to the desired qubits and false otherwise.

Returns

state : Expr

The resulting state after Grover's algorithm has been iterated.

Examples

Apply grover's algorithm to an even superposition of 2 qubits:

```
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.grover import apply_grover
>>> f = lambda qubits: qubits == IntQubit(2)
>>> qapply(apply_grover(f, 2))
|2>
```

sympy.physics.quantum.grover.**grover_iteration**(*qstate*, *oracle*)

Applies one application of the Oracle and W Gate, WV.

Parameters

qstate : Qubit

A superposition of qubits.

oracle : OracleGate

The black box operator that flips the sign of the desired basis qubits.

Returns

Qubit : The qubits after applying the Oracle and W gate.

Examples

Perform one iteration of grover's algorithm to see a phase change:

```
>>> from sympy.physics.quantum.qapply import qapply
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.grover import OracleGate
>>> from sympy.physics.quantum.grover import superposition_basis
>>> from sympy.physics.quantum.grover import grover_iteration
>>> numqubits = 2
>>> basis_states = superposition_basis(numqubits)
>>> f = lambda qubits: qubits == IntQubit(2)
>>> v = OracleGate(numqubits, f)
>>> qapply(grover_iteration(basis_states, v))
|2>
```

sympy.physics.quantum.grover.**superposition_basis**(*nqubits*)

Creates an equal superposition of the computational basis.

Parameters

nqubits : int

The number of qubits.

Returns

state : Qubit

An equal superposition of the computational basis with nqubits.

Examples

Create an equal superposition of 2 qubits:

```
>>> from sympy.physics.quantum.grover import superposition_basis
>>> superposition_basis(2)
|0>/2 + |1>/2 + |2>/2 + |3>/2
```

QFT

An implementation of qubits and gates acting on them.

Todo:

- Update docstrings.
- Update tests.
- Implement apply using decompose.
- Implement represent using decompose or something smarter. For this to work we first have to implement represent for SWAP.
- Decide if we want upper index to be inclusive in the constructor.
- Fix the printing of Rk gates in plotting.

class sympy.physics.quantum.qft.**IQFT**(**args*, ***kwargs*)

The inverse quantum Fourier transform.

decompose()

Decomposes IQFT into elementary gates.

class sympy.physics.quantum.qft.**QFT**(**args*, ***kwargs*)

The forward quantum Fourier transform.

decompose()

Decomposes QFT into elementary gates.

sympy.physics.quantum.qft.**Rk**

alias of *RkGate* (page 1850)

class sympy.physics.quantum.qft.**RkGate**(**args*)

This is the R_k gate of the QTF.

Qubit

Qubits for quantum computing.

Todo: * Finish implementing measurement logic. This should include POVM. * Update docstrings. * Update tests.

class sympy.physics.quantum.qubit.**IntQubit**(**args*, ***kwargs*)

A qubit ket that store integers as binary numbers in qubit values.

The differences between this class and `Qubit` are:

- The form of the constructor.

- The qubit values are printed as their corresponding integer, rather than the raw qubit values. The internal storage format of the qubit values in the same as `Qubit`.

Parameters

values : int, tuple

If a single argument, the integer we want to represent in the qubit values. This integer will be represented using the fewest possible number of qubits. If a pair of integers and the second value is more than one, the first integer gives the integer to represent in binary form and the second integer gives the number of qubits to use. List of zeros and ones is also accepted to generate qubit by bit pattern.

nqubits : int

The integer that represents the number of qubits. This number should be passed with keyword `nqubits=N`. You can use this in order to avoid ambiguity of Qubit-style tuple of bits. Please see the example below for more details.

Examples

Create a qubit for the integer 5:

```
>>> from sympy.physics.quantum.qubit import IntQubit
>>> from sympy.physics.quantum.qubit import Qubit
>>> q = IntQubit(5)
>>> q
|5>
```

We can also create an `IntQubit` by passing a `Qubit` instance.

```
>>> q = IntQubit(Qubit('101'))
>>> q
|5>
>>> q.as_int()
5
>>> q.nqubits
3
>>> q.qubit_values
(1, 0, 1)
```

We can go back to the regular qubit form.

```
>>> Qubit(q)
|101>
```

Please note that `IntQubit` also accepts a `Qubit`-style list of bits. So, the code below yields qubits 3, not a single bit 1.

```
>>> IntQubit(1, 1)
|3>
```

To avoid ambiguity, use `nqubits` parameter. Use of this keyword is recommended especially when you provide the values by variables.

```
>>> IntQubit(1, nqubits=1)
|1>
>>> a = 1
>>> IntQubit(a, nqubits=1)
|1>
```

class sympy.physics.quantum.qubit.**IntQubitBra**(**args*, ***kwargs*)

A qubit bra that store integers as binary numbers in qubit values.

class sympy.physics.quantum.qubit.**Qubit**(**args*, ***kwargs*)

A multi-qubit ket in the computational (z) basis.

We use the normal convention that the least significant qubit is on the right, so `|00001>` has a 1 in the least significant qubit.

Parameters

values : list, str

The qubit values as a list of ints ([0,0,0,1,1,]) or a string ('011').

Examples

Create a qubit in a couple of different ways and look at their attributes:

```
>>> from sympy.physics.quantum.qubit import Qubit
>>> Qubit(0,0,0)
|000>
>>> q = Qubit('0101')
>>> q
|0101>
```

```
>>> q.nqubits
4
>>> len(q)
4
>>> q.dimension
4
>>> q.qubit_values
(0, 1, 0, 1)
```

We can flip the value of an individual qubit:

```
>>> q.flip(1)
|0111>
```

We can take the dagger of a Qubit to get a bra:

```
>>> from sympy.physics.quantum.dagger import Dagger
>>> Dagger(q)
<0101|
>>> type(Dagger(q))
<class 'sympy.physics.quantum.qubit.QubitBra'>
```

Inner products work as expected:

```
>>> ip = Dagger(q)*q
>>> ip
<0101|0101>
>>> ip.doit()
1
```

class sympy.physics.quantum.qubit.**QubitBra**(**args*, ***kwargs*)

A multi-qubit bra in the computational (z) basis.

We use the normal convention that the least significant qubit is on the right, so `|00001>` has a 1 in the least significant qubit.

Parameters

values : list, str

The qubit values as a list of ints ([0,0,0,1,1,]) or a string ('011').

See also:

Qubit **(page 1852)**

Examples using qubits

sympy.physics.quantum.qubit.**matrix_to_density**(*mat*)

Works by finding the eigenvectors and eigenvalues of the matrix. We know we can decompose rho by doing: sum(EigenVal*|Eigenvect><Eigenvect|)

sympy.physics.quantum.qubit.**matrix_to_qubit**(*matrix*)

Convert from the matrix repr. to a sum of Qubit objects.

Parameters

matrix : Matrix, numpy.matrix, scipy.sparse

The matrix to build the Qubit representation of. This works with SymPy matrices, numpy matrices and scipy.sparse sparse matrices.

Examples

Represent a state and then go back to its qubit form:

```
>>> from sympy.physics.quantum.qubit import matrix_to_qubit, Qubit
>>> from sympy.physics.quantum.represent import represent
>>> q = Qubit('01')
>>> matrix_to_qubit(represent(q))
|01>
```

sympy.physics.quantum.qubit.**measure_all**(*qubit*, *format='sympy'*, *normalize=True*)

Perform an ensemble measurement of all qubits.

Parameters

qubit : Qubit, Add

The qubit to measure. This can be any Qubit or a linear combination of them.

format : str

The format of the intermediate matrices to use. Possible values are ('sympy','numpy','scipy.sparse'). Currently only 'sympy' is implemented.

Returns

result : list

A list that consists of primitive states and their probabilities.

Examples

```
>>> from sympy.physics.quantum.qubit import Qubit, measure_all
>>> from sympy.physics.quantum.gate import H
>>> from sympy.physics.quantum.qapply import qapply
```

```
>>> c = H(0)*H(1)*Qubit('00')
>>> c
H(0)*H(1)*|00>
>>> q = qapply(c)
>>> measure_all(q)
[(|00>, 1/4), (|01>, 1/4), (|10>, 1/4), (|11>, 1/4)]
```

sympy.physics.quantum.qubit.**measure_all_oneshot**(*qubit*, *format*='*sympy*')

Perform a oneshot ensemble measurement on all qubits.

A oneshot measurement is equivalent to performing a measurement on a quantum system. This type of measurement does not return the probabilities like an ensemble measurement does, but rather returns *one* of the possible resulting states. The exact state that is returned is determined by picking a state randomly according to the ensemble probabilities.

Parameters

qubits : Qubit

The qubit to measure. This can be any Qubit or a linear combination of them.

format : str

The format of the intermediate matrices to use. Possible values are ('sympy','numpy','scipy.sparse'). Currently only 'sympy' is implemented.

Returns

result : Qubit

The qubit that the system collapsed to upon measurement.

sympy.physics.quantum.qubit.**measure_partial**(*qubit*, *bits*, *format*='*sympy*', *normalize*=*True*)

Perform a partial ensemble measure on the specified qubits.

Parameters

qubits : Qubit

The qubit to measure. This can be any Qubit or a linear combination of them.

bits : tuple

The qubits to measure.

format : str

The format of the intermediate matrices to use. Possible values are ('sympy','numpy','scipy.sparse'). Currently only 'sympy' is implemented.

Returns

result : list

A list that consists of primitive states and their probabilities.

Examples

```
>>> from sympy.physics.quantum.qubit import Qubit, measure_partial
>>> from sympy.physics.quantum.gate import H
>>> from sympy.physics.quantum.qapply import qapply
```

```
>>> c = H(0)*H(1)*Qubit('00')
>>> c
H(0)*H(1)*|00>
>>> q = qapply(c)
>>> measure_partial(q, (0,))
[(sqrt(2)*|00>/2 + sqrt(2)*|10>/2, 1/2), (sqrt(2)*|01>/2 + sqrt(2)*|11>/
↪2, 1/2)]
```

sympy.physics.quantum.qubit.**measure_partial_oneshot**(*qubit*, *bits*, *format*='*sympy*')

Perform a partial oneshot measurement on the specified qubits.

A oneshot measurement is equivalent to performing a measurement on a quantum system. This type of measurement does not return the probabilities like an ensemble measurement does, but rather returns *one* of the possible resulting states. The exact state that is returned is determined by picking a state randomly according to the ensemble probabilities.

Parameters

qubits : Qubit

The qubit to measure. This can be any Qubit or a linear combination of them.

bits : tuple

The qubits to measure.

format : str

The format of the intermediate matrices to use. Possible values are ('sympy','numpy','scipy.sparse'). Currently only 'sympy' is implemented.

Returns

result : Qubit

The qubit that the system collapsed to upon measurement.

sympy.physics.quantum.qubit.**qubit_to_matrix**(*qubit*, *format='sympy'*)

Converts an Add/Mul of Qubit objects into it's matrix representation

This function is the inverse of `matrix_to_qubit` and is a shorthand for `represent(qubit)`.

Shor's Algorithm

Shor's algorithm and helper functions.

Todo:

- Get the CMod gate working again using the new Gate API.
- Fix everything.
- Update docstrings and reformat.

class sympy.physics.quantum.shor.**CMod**(**args*, ***kwargs*)

A controlled mod gate.

This is black box controlled Mod function for use by shor's algorithm. TODO: implement a decompose property that returns how to do this in terms of elementary gates

property N

N is the type of modular arithmetic we are doing.

property a

Base of the controlled mod function.

property t

Size of 1/2 input register. First 1/2 holds output.

sympy.physics.quantum.shor.**period_find**(*a*, *N*)

Finds the period of a in modulo N arithmetic

This is quantum part of Shor's algorithm. It takes two registers, puts first in superposition of states with Hadamards so: `|k>|0>` with k being all possible choices. It then does a controlled mod and a QFT to determine the order of a.

sympy.physics.quantum.shor.**shor**(*N*)

This function implements Shor's factoring algorithm on the Integer N

The algorithm starts by picking a random number (a) and seeing if it is coprime with N. If it is not, then the gcd of the two numbers is a factor and we are done. Otherwise, it begins the period_finding subroutine which finds the period of a in modulo N arithmetic. This period, if even, can be used to calculate factors by taking a**(r/2)-1 and a**(r/2)+1. These values are returned.

Analytic Solutions

Particle in a Box

1D quantum particle in a box.

class sympy.physics.quantum.piab.**PIABBra**(**args*, ***kwargs*)
Particle in a box eigenbra.

class sympy.physics.quantum.piab.**PIABHamiltonian**(**args*, ***kwargs*)
Particle in a box Hamiltonian operator.

class sympy.physics.quantum.piab.**PIABKet**(**args*, ***kwargs*)
Particle in a box eigenket.

Optics Module

Abstract

Contains docstrings of Physics-Optics module

Gaussian Optics

Gaussian optics.

The module implements:

- Ray transfer matrices for geometrical and gaussian optics.

  See RayTransferMatrix, GeometricRay and BeamParameter
- Conjugation relations for geometrical and gaussian optics.

  See geometric_conj*, gauss_conj and conjugate_gauss_beams

The conventions for the distances are as follows:

focal distance
positive for convergent lenses

object distance
positive for real objects

image distance
positive for real images

class sympy.physics.optics.gaussopt.**BeamParameter**(*wavelen*, *z*, *z_r=None*, *w=None*, *n=1*)
Representation for a gaussian ray in the Ray Transfer Matrix formalism.

Parameters
wavelen : the wavelength,

z : the distance to waist, and

w : the waist, or

z_r : the rayleigh range.

n : the refractive index of medium.

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.q
1 + 1.88679245283019*I*pi
```

```
>>> p.q.n()
1.0 + 5.92753330865999*I
>>> p.w_0.n()
0.00100000000000000
>>> p.z_r.n()
5.92753330865999
```

```
>>> from sympy.physics.optics import FreeSpace
>>> fs = FreeSpace(10)
>>> p1 = fs*p
>>> p.w.n()
0.00101413072159615
>>> p1.w.n()
0.00210803120913829
```

See also:

RayTransferMatrix (page 1863)

References

[R660], [R661]

property divergence

Half of the total angular spread.

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.divergence
0.00053/pi
```

property gouy

The Gouy phase.

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.gouy
atan(0.53/pi)
```

property q

The complex parameter representing the beam.

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.q
1 + 1.88679245283019*I*pi
```

property radius

The radius of curvature of the phase front.

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.radius
1 + 3.55998576005696*pi**2
```

property w

The radius of the beam w(z), at any position z along the beam. The beam radius at $1/e^2$ intensity (axial value).

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.w
0.001*sqrt(0.2809/pi**2 + 1)
```

See also:

w_0 **(page 1859)**
The minimal radius of beam.

property w_0

The minimal radius of beam at $1/e^2$ intensity (peak value).

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.w_0
0.00100000000000000
```

See also:

w **(page 1859)**
the beam radius at $1/e^2$ intensity (axial value).

property waist_approximation_limit

The minimal waist for which the gauss beam approximation is valid.

Explanation

The gauss beam is a solution to the paraxial equation. For curvatures that are too great it is not a valid approximation.

Examples

```
>>> from sympy.physics.optics import BeamParameter
>>> p = BeamParameter(530e-9, 1, w=1e-3)
>>> p.waist_approximation_limit
1.06e-6/pi
```

class sympy.physics.optics.gaussopt.**CurvedMirror**(*R*)

Ray Transfer Matrix for reflection from curved surface.

Parameters

R : radius of curvature (positive for concave)

Examples

```
>>> from sympy.physics.optics import CurvedMirror
>>> from sympy import symbols
>>> R = symbols('R')
>>> CurvedMirror(R)
Matrix([
[   1, 0],
[-2/R, 1]])
```

See also:

RayTransferMatrix (page 1863)

class sympy.physics.optics.gaussopt.**CurvedRefraction**(*R*, *n1*, *n2*)

Ray Transfer Matrix for refraction on curved interface.

Parameters

R :

Radius of curvature (positive for concave).

n1 :

Refractive index of one medium.

n2 :

Refractive index of other medium.

Examples

```
>>> from sympy.physics.optics import CurvedRefraction
>>> from sympy import symbols
>>> R, n1, n2 = symbols('R n1 n2')
>>> CurvedRefraction(R, n1, n2)
Matrix([
[               1,     0],
[(n1 - n2)/(R*n2), n1/n2]])
```

See also:

RayTransferMatrix (page 1863)

class sympy.physics.optics.gaussopt.**FlatMirror**

Ray Transfer Matrix for reflection.

Examples

```
>>> from sympy.physics.optics import FlatMirror
>>> FlatMirror()
Matrix([
[1, 0],
[0, 1]])
```

See also:

RayTransferMatrix (page 1863)

class sympy.physics.optics.gaussopt.**FlatRefraction**(*n1*, *n2*)

Ray Transfer Matrix for refraction.

Parameters

n1 :

Refractive index of one medium.

n2 :

Refractive index of other medium.

Examples

```
>>> from sympy.physics.optics import FlatRefraction
>>> from sympy import symbols
>>> n1, n2 = symbols('n1 n2')
>>> FlatRefraction(n1, n2)
Matrix([
[1,     0],
[0, n1/n2]])
```

See also:

RayTransferMatrix (page 1863)

class sympy.physics.optics.gaussopt.**FreeSpace**(*d*)

Ray Transfer Matrix for free space.

Parameters

distance

Examples

```
>>> from sympy.physics.optics import FreeSpace
>>> from sympy import symbols
>>> d = symbols('d')
>>> FreeSpace(d)
Matrix([
[1, d],
[0, 1]])
```

See also:

RayTransferMatrix (page 1863)

class sympy.physics.optics.gaussopt.**GeometricRay**(**args*)

Representation for a geometric ray in the Ray Transfer Matrix formalism.

Parameters

h : height, and

angle : angle, or

matrix : a 2x1 matrix (Matrix(2, 1, [height, angle]))

Examples

```
>>> from sympy.physics.optics import GeometricRay, FreeSpace
>>> from sympy import symbols, Matrix
>>> d, h, angle = symbols('d, h, angle')
```

```
>>> GeometricRay(h, angle)
Matrix([
[    h],
[angle]])
```

```
>>> FreeSpace(d)*GeometricRay(h, angle)
Matrix([
[angle*d + h],
[      angle]])
```

```
>>> GeometricRay( Matrix( ((h,), (angle,)) ) )
Matrix([
[    h],
[angle]])
```

See also:

RayTransferMatrix (page 1863)

property angle

The angle with the optical axis.

Examples

```
>>> from sympy.physics.optics import GeometricRay
>>> from sympy import symbols
>>> h, angle = symbols('h, angle')
>>> gRay = GeometricRay(h, angle)
>>> gRay.angle
angle
```

property height

The distance from the optical axis.

Examples

```
>>> from sympy.physics.optics import GeometricRay
>>> from sympy import symbols
>>> h, angle = symbols('h, angle')
>>> gRay = GeometricRay(h, angle)
>>> gRay.height
h
```

class sympy.physics.optics.gaussopt.**RayTransferMatrix**(**args*)

Base class for a Ray Transfer Matrix.

It should be used if there is not already a more specific subclass mentioned in See Also.

Parameters

parameters :

A, B, C and D or 2x2 matrix (Matrix(2, 2, [A, B, C, D]))

Examples

```
>>> from sympy.physics.optics import RayTransferMatrix, ThinLens
>>> from sympy import Symbol, Matrix
```

```
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat
Matrix([
[1, 2],
[3, 4]])
```

```
>>> RayTransferMatrix(Matrix([[1, 2], [3, 4]]))
Matrix([
[1, 2],
[3, 4]])
```

```
>>> mat.A
1
```

```
>>> f = Symbol('f')
>>> lens = ThinLens(f)
>>> lens
Matrix([
[   1, 0],
[-1/f, 1]])
```

```
>>> lens.C
-1/f
```

See also:

GeometricRay (page 1862), *BeamParameter* (page 1857), *FreeSpace* (page 1862), *FlatRefraction* (page 1861), *CurvedRefraction* (page 1860), *FlatMirror* (page 1861), *CurvedMirror* (page 1860), *ThinLens* (page 1865)

References

[R662]

property A

The A parameter of the Matrix.

Examples

```
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.A
1
```

property B

The B parameter of the Matrix.

Examples

```
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.B
2
```

property C

The C parameter of the Matrix.

Examples

```
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.C
3
```

property D

The D parameter of the Matrix.

Examples

```
>>> from sympy.physics.optics import RayTransferMatrix
>>> mat = RayTransferMatrix(1, 2, 3, 4)
>>> mat.D
4
```

class sympy.physics.optics.gaussopt.**ThinLens**(*f*)

Ray Transfer Matrix for a thin lens.

Parameters

f :

The focal distance.

Examples

```
>>> from sympy.physics.optics import ThinLens
>>> from sympy import symbols
>>> f = symbols('f')
>>> ThinLens(f)
Matrix([
[   1, 0],
[-1/f, 1]])
```

See also:

RayTransferMatrix (page 1863)

sympy.physics.optics.gaussopt.**conjugate_gauss_beams**(*wavelen*, *waist_in*, *waist_out*, ***kwargs*)

Find the optical setup conjugating the object/image waists.

Parameters

wavelen :

The wavelength of the beam.

waist_in and waist_out :

The waists to be conjugated.

f :

The focal distance of the element used in the conjugation.

Returns

a tuple containing (s_in, s_out, f)

s_in :

The distance before the optical element.

s_out :

The distance after the optical element.

f :

The focal distance of the optical element.

Examples

```
>>> from sympy.physics.optics import conjugate_gauss_beams
>>> from sympy import symbols, factor
>>> l, w_i, w_o, f = symbols('l w_i w_o f')
```

```
>>> conjugate_gauss_beams(l, w_i, w_o, f=f)[0]
f*(1 - sqrt(w_i**2/w_o**2 - pi**2*w_i**4/(f**2*l**2)))
```

```
>>> factor(conjugate_gauss_beams(l, w_i, w_o, f=f)[1])
f*w_o**2*(w_i**2/w_o**2 - sqrt(w_i**2/w_o**2 -
          pi**2*w_i**4/(f**2*l**2)))/w_i**2
```

```
>>> conjugate_gauss_beams(l, w_i, w_o, f=f)[2]
f
```

sympy.physics.optics.gaussopt.**gaussian_conj**(*s_in*, *z_r_in*, *f*)

Conjugation relation for gaussian beams.

Parameters

s_in :

The distance to optical element from the waist.

z_r_in :

The rayleigh range of the incident beam.

f :

The focal length of the optical element.

Returns

a tuple containing (s_out, z_r_out, m)

s_out :

The distance between the new waist and the optical element.

z_r_out :

The rayleigh range of the emergent beam.

m :

The ration between the new and the old waists.

Examples

```
>>> from sympy.physics.optics import gaussian_conj
>>> from sympy import symbols
>>> s_in, z_r_in, f = symbols('s_in z_r_in f')
```

```
>>> gaussian_conj(s_in, z_r_in, f)[0]
1/(-1/(s_in + z_r_in**2/(-f + s_in)) + 1/f)
```

```
>>> gaussian_conj(s_in, z_r_in, f)[1]
z_r_in/(1 - s_in**2/f**2 + z_r_in**2/f**2)
```

```
>>> gaussian_conj(s_in, z_r_in, f)[2]
1/sqrt(1 - s_in**2/f**2 + z_r_in**2/f**2)
```

sympy.physics.optics.gaussopt.**geometric_conj_ab**(*a*, *b*)

Conjugation relation for geometrical beams under paraxial conditions.

Explanation

Takes the distances to the optical element and returns the needed focal distance.

Examples

```
>>> from sympy.physics.optics import geometric_conj_ab
>>> from sympy import symbols
>>> a, b = symbols('a b')
>>> geometric_conj_ab(a, b)
a*b/(a + b)
```

See also:

geometric_conj_af (page 1868), *geometric_conj_bf* (page 1868)

sympy.physics.optics.gaussopt.**geometric_conj_af**(*a*, *f*)

Conjugation relation for geometrical beams under paraxial conditions.

Explanation

Takes the object distance (for geometric_conj_af) or the image distance (for geometric_conj_bf) to the optical element and the focal distance. Then it returns the other distance needed for conjugation.

Examples

```
>>> from sympy.physics.optics.gaussopt import geometric_conj_af,␣
↪geometric_conj_bf
>>> from sympy import symbols
>>> a, b, f = symbols('a b f')
>>> geometric_conj_af(a, f)
a*f/(a - f)
>>> geometric_conj_bf(b, f)
b*f/(b - f)
```

See also:

geometric_conj_ab (page 1867)

sympy.physics.optics.gaussopt.**geometric_conj_bf**(*a*, *f*)

Conjugation relation for geometrical beams under paraxial conditions.

Explanation

Takes the object distance (for geometric_conj_af) or the image distance (for geometric_conj_bf) to the optical element and the focal distance. Then it returns the other distance needed for conjugation.

Examples

```
>>> from sympy.physics.optics.gaussopt import geometric_conj_af,␣
↪geometric_conj_bf
>>> from sympy import symbols
>>> a, b, f = symbols('a b f')
>>> geometric_conj_af(a, f)
a*f/(a - f)
>>> geometric_conj_bf(b, f)
b*f/(b - f)
```

See also:

geometric_conj_ab (page 1867)

sympy.physics.optics.gaussopt.**rayleigh2waist**(*z_r*, *wavelen*)

Calculate the waist from the rayleigh range of a gaussian beam.

Examples

```
>>> from sympy.physics.optics import rayleigh2waist
>>> from sympy import symbols
>>> z_r, wavelen = symbols('z_r wavelen')
>>> rayleigh2waist(z_r, wavelen)
sqrt(wavelen*z_r)/sqrt(pi)
```

See also:

waist2rayleigh (page 1869), *BeamParameter* (page 1857)

sympy.physics.optics.gaussopt.**waist2rayleigh**(*w*, *wavelen*, *n=1*)

Calculate the rayleigh range from the waist of a gaussian beam.

Examples

```
>>> from sympy.physics.optics import waist2rayleigh
>>> from sympy import symbols
>>> w, wavelen = symbols('w wavelen')
>>> waist2rayleigh(w, wavelen)
pi*w**2/wavelen
```

See also:

rayleigh2waist (page 1869), *BeamParameter* (page 1857)

Medium

Contains

- Medium

class sympy.physics.optics.medium.**Medium**(*name*, *permittivity=None*, *permeability=None*, *n=None*)

This class represents an optical medium. The prime reason to implement this is to facilitate refraction, Fermat's principle, etc.

Parameters

name: string

The display name of the Medium.

permittivity: Sympifyable

Electric permittivity of the space.

permeability: Sympifyable

Magnetic permeability of the space.

n: Sympifyable

Index of refraction of the medium.

Explanation

An optical medium is a material through which electromagnetic waves propagate. The permittivity and permeability of the medium define how electromagnetic waves propagate in it.

Examples

```
>>> from sympy.abc import epsilon, mu
>>> from sympy.physics.optics import Medium
>>> m1 = Medium('m1')
>>> m2 = Medium('m2', epsilon, mu)
>>> m1.intrinsic_impedance
149896229*pi*kilogram*meter**2/(1250000*ampere**2*second**3)
>>> m2.refractive_index
299792458*meter*sqrt(epsilon*mu)/second
```

References

[R663]

property refractive_index

Returns refractive index of the medium.

Examples

```
>>> from sympy.physics.optics import Medium
>>> m = Medium('m')
>>> m.refractive_index
1
```

property speed

Returns speed of the electromagnetic wave travelling in the medium.

Examples

```
>>> from sympy.physics.optics import Medium
>>> m = Medium('m')
>>> m.speed
299792458*meter/second
>>> m2 = Medium('m2', n=1)
>>> m.speed == m2.speed
True
```


Polarization

The module implements routines to model the polarization of optical fields and can be used to calculate the effects of polarization optical elements on the fields.

- Jones vectors.
- Stokes vectors.
- Jones matrices.
- Mueller matrices.

Examples

We calculate a generic Jones vector:

```
>>> from sympy import symbols, pprint, zeros, simplify
>>> from sympy.physics.optics.polarization import (jones_vector, stokes_
↪vector,
...     half_wave_retarder, polarizing_beam_splitter, jones_2_stokes)
```

```
>>> psi, chi, p, I0 = symbols("psi, chi, p, I0", real=True)
>>> x0 = jones_vector(psi, chi)
>>> pprint(x0, use_unicode=True)
⎡-i·sin(χ)·sin(ψ) + cos(χ)·cos(ψ)⎤
⎢                                ⎥
⎣i·sin(χ)·cos(ψ) + sin(ψ)·cos(χ) ⎦
```

And the more general Stokes vector: >>> s0 = stokes_vector(psi, chi, p, I0) >>> pprint(s0, use_unicode=True) ⎡ I_0 ⎤ ⎢ ⎥ ⎢I_0·p·cos(2·χ)·cos(2·ψ)⎥ ⎢ ⎥ ⎢I_0·p·sin(2·ψ)·cos(2·χ)⎥ ⎢ ⎥ ⎣ I_0·p·sin(2·χ) ⎦

We calculate how the Jones vector is modified by a half-wave plate: >>> alpha = symbols("alpha", real=True) >>> HWP = half_wave_retarder(alpha) >>> x1 = simplify(HWP*x0)

We calculate the very common operation of passing a beam through a half-wave plate and then through a polarizing beam-splitter. We do this by putting this Jones vector as the first entry of a two-Jones-vector state that is transformed by a 4x4 Jones matrix modelling the polarizing beam-splitter to get the transmitted and reflected Jones vectors:

```
>>> PBS = polarizing_beam_splitter()
>>> X1 = zeros(4, 1)
>>> X1[:2, :] = x1
>>> X2 = PBS*X1
>>> transmitted_port = X2[:2, :]
>>> reflected_port = X2[2:, :]
```

This allows us to calculate how the power in both ports depends on the initial polarization:

```
>>> transmitted_power = jones_2_stokes(transmitted_port)[0]
>>> reflected_power = jones_2_stokes(reflected_port)[0]
>>> print(transmitted_power)
cos(-2*alpha + chi + psi)**2/2 + cos(2*alpha + chi - psi)**2/2
```

```
>>> print(reflected_power)
sin(-2*alpha + chi + psi)**2/2 + sin(2*alpha + chi - psi)**2/2
```

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

References

sympy.physics.optics.polarization.**half_wave_retarder**(*theta*)

A half-wave retarder Jones matrix at angle *theta*.

Parameters

``**theta**`` : numeric type or SymPy Symbol

The angle of the fast axis relative to the horizontal plane.

Returns

SymPy Matrix

A Jones matrix representing the retarder.

Examples

A generic half-wave plate.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import half_wave_retarder
>>> theta= symbols("theta", real=True)
>>> HWP = half_wave_retarder(theta)
>>> pprint(HWP, use_unicode=True)
⎡   ⎛     2         2   ⎞                          ⎤
⎢-i·⎝- sin (θ) + cos (θ)⎠    -2·i·sin(θ)·cos(θ)   ⎥
⎢                                                 ⎥
⎢                              ⎛   2         2   ⎞⎥
⎣   -2·i·sin(θ)·cos(θ)     -i·⎝sin (θ) - cos (θ)⎠⎦
```

sympy.physics.optics.polarization.**jones_2_stokes**(*e*)

Return the Stokes vector for a Jones vector e.

Parameters

``**e**`` : SymPy Matrix

A Jones vector.

Returns

SymPy Matrix

A Jones vector.

Examples

The axes on the Poincaré sphere.

```
>>> from sympy import pprint, pi
>>> from sympy.physics.optics.polarization import jones_vector
>>> from sympy.physics.optics.polarization import jones_2_stokes
>>> H = jones_vector(0, 0)
>>> V = jones_vector(pi/2, 0)
>>> D = jones_vector(pi/4, 0)
>>> A = jones_vector(-pi/4, 0)
>>> R = jones_vector(0, pi/4)
>>> L = jones_vector(0, -pi/4)
>>> pprint([jones_2_stokes(e) for e in [H, V, D, A, R, L]],
...           use_unicode=True)
⎡⎡1⎤  ⎡1 ⎤  ⎡1⎤  ⎡1 ⎤  ⎡1⎤  ⎡1 ⎤⎤
⎢⎢ ⎥  ⎢  ⎥  ⎢ ⎥  ⎢  ⎥  ⎢ ⎥  ⎢  ⎥⎥
⎢⎢1⎥  ⎢-1⎥  ⎢0⎥  ⎢0 ⎥  ⎢0⎥  ⎢0 ⎥⎥
⎢⎢ ⎥, ⎢  ⎥, ⎢ ⎥, ⎢  ⎥, ⎢ ⎥, ⎢  ⎥⎥
⎢⎢0⎥  ⎢0 ⎥  ⎢1⎥  ⎢-1⎥  ⎢0⎥  ⎢0 ⎥⎥
⎢⎢ ⎥  ⎢  ⎥  ⎢ ⎥  ⎢  ⎥  ⎢ ⎥  ⎢  ⎥⎥
⎣⎣0⎦  ⎣0 ⎦  ⎣0⎦  ⎣0 ⎦  ⎣1⎦  ⎣-1⎦⎦
```

sympy.physics.optics.polarization.**jones_vector**(*psi*, *chi*)

A Jones vector corresponding to a polarization ellipse with psi tilt, and chi circularity.

Parameters

``**psi**`` : numeric type or SymPy Symbol

The tilt of the polarization relative to the x axis.

``**chi**`` : numeric type or SymPy Symbol

The angle adjacent to the mayor axis of the polarization ellipse.

Returns

Matrix :

A Jones vector.

Examples

The axes on the Poincaré sphere.

```
>>> from sympy import pprint, symbols, pi
>>> from sympy.physics.optics.polarization import jones_vector
>>> psi, chi = symbols("psi, chi", real=True)
```

A general Jones vector. >>> pprint(jones_vector(psi, chi), use_unicode=True) ⎡-i·sin(χ)·sin(ψ) + cos(χ)·cos(ψ)⎤ ⎢ ⎥ ⎣i·sin(χ)·cos(ψ) + sin(ψ)·cos(χ) ⎦

Horizontal polarization. >>> pprint(jones_vector(0, 0), use_unicode=True) ⎡1⎤ ⎢ ⎥ ⎣0⎦

Vertical polarization. >>> pprint(jones_vector(pi/2, 0), use_unicode=True) ⎡0⎤ ⎢ ⎥ ⎣1⎦

Diagonal polarization. >>> pprint(jones_vector(pi/4, 0), use_unicode=True) ⎡√2⎤ ⎢──⎥ ⎢2 ⎥ ⎢ ⎥ ⎢√2⎥ ⎢──⎥ ⎣2 ⎦

Anti-diagonal polarization. >>> pprint(jones_vector(-pi/4, 0), use_unicode=True) ⎡ √2 ⎤ ⎢ ── ⎥ ⎢ 2 ⎥ ⎢ ⎥ ⎢-√2 ⎥ ⎢─────⎥ ⎣ 2 ⎦

Right-hand circular polarization. >>> pprint(jones_vector(0, pi/4), use_unicode=True) ⎡ √2 ⎤ ⎢ ── ⎥ ⎢ 2 ⎥ ⎢ ⎥ ⎢√2·i⎥ ⎢─────⎥ ⎣ 2 ⎦

Left-hand circular polarization. >>> pprint(jones_vector(0, -pi/4), use_unicode=True) ⎡ √2 ⎤ ⎢ ── ⎥ ⎢ 2 ⎥ ⎢ ⎥ ⎢-√2·i ⎥ ⎢──────⎥ ⎣ 2 ⎦

`sympy.physics.optics.polarization.`**`linear_polarizer`**(*theta=0*)

A linear polarizer Jones matrix with transmission axis at an angle `theta`.

Parameters

``**theta**`` : numeric type or SymPy Symbol

The angle of the transmission axis relative to the horizontal plane.

Returns

SymPy Matrix

A Jones matrix representing the polarizer.

Examples

A generic polarizer.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import linear_polarizer
>>> theta = symbols("theta", real=True)
>>> J = linear_polarizer(theta)
>>> pprint(J, use_unicode=True)
⎡      2                    ⎤
⎢   cos (θ)     sin(θ)·cos(θ)⎥
⎢                            ⎥
⎢                     2      ⎥
⎣sin(θ)·cos(θ)     sin (θ)   ⎦
```

sympy.physics.optics.polarization.**mueller_matrix**(*J*)

The Mueller matrix corresponding to Jones matrix J.

Parameters

``J`` : SymPy Matrix

A Jones matrix.

Returns

SymPy Matrix

The corresponding Mueller matrix.

Examples

Generic optical components.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import (mueller_matrix,
...     linear_polarizer, half_wave_retarder, quarter_wave_retarder)
>>> theta = symbols("theta", real=True)
```

A linear_polarizer >>> pprint(mueller_matrix(linear_polarizer(theta)), use_unicode=True) ⎡ cos(2·θ) sin(2·θ) ⎤ ⎢ 1/2 ──────── ──────── 0⎥ ⎢ 2 2 ⎥ ⎢ ⎥ ⎢cos(2·θ) cos(4·θ) 1 sin(4·θ) ⎥ ⎢──────── ──────── + − ──────── 0⎥ ⎢ 2 4 4 4 ⎥ ⎢ ⎥ ⎢sin(2·θ) sin(4·θ) 1 cos(4·θ) ⎥ ⎢──────── ──────── − - ──────── 0⎥ ⎢ 2 4 4 4 ⎥ ⎢ ⎥ ⎣ 0 0 0 0⎦

A half-wave plate >>> pprint(mueller_matrix(half_wave_retarder(theta)), use_unicode=True) ⎡1 0 0 0 ⎤ ⎢ ⎥ ⎢ 4 2 ⎥ ⎢0 8·sin (θ) - 8·sin (θ) + 1 sin(4·θ) 0 ⎥ ⎢ 4 2 ⎥ ⎢0 sin(4·θ) - 8·sin (θ) + 8·sin (θ) - 1 0 ⎥ ⎢ ⎥ ⎣0 0 0 -1⎦

A quarter-wave plate >>> pprint(mueller_matrix(quarter_wave_retarder(theta)), use_unicode=True) ⎡1 0 0 0 ⎤ ⎢ ⎥ ⎢ cos(4·θ) 1 sin(4·θ) ⎥ ⎢0 ──────── + − ──────── ⎥ ⎢ 2 2 2 -sin(2·θ)⎥ ⎢ ⎥ ⎢ sin(4·θ) 1 cos(4·θ) ⎥ ⎢0 ──────── − - ──────── cos(2·θ) ⎥ ⎢ 2 2 2 ⎥ ⎢ ⎥ ⎣0 sin(2·θ) -cos(2·θ) 0 ⎦

sympy.physics.optics.polarization.**phase_retarder**(*theta=0*, *delta=0*)

A phase retarder Jones matrix with retardance $delta$ at angle $theta$.

Parameters

``**theta**`` : numeric type or SymPy Symbol

The angle of the fast axis relative to the horizontal plane.

``**delta**`` : numeric type or SymPy Symbol

The phase difference between the fast and slow axes of the transmitted light.

Returns

SymPy Matrix :

A Jones matrix representing the retarder.

Examples

A generic retarder.

```
>>> from sympy import pprint, symbols
>>> from sympy.physics.optics.polarization import phase_retarder
>>> theta, delta = symbols("theta, delta", real=True)
>>> R = phase_retarder(theta, delta)
>>> pprint(R, use_unicode=True)
⎡                            -i·δ                -i·δ                ⎤
⎢                            ─────               ─────               ⎥
⎢⎛ i·δ    2         2   ⎞    2    ⎛     i·δ⎞    2                    ⎥
⎢⎝ℯ   ·sin (θ) + cos (θ)⎠·ℯ       ⎝1 - ℯ   ⎠·ℯ     ·sin(θ)·cos(θ)⎥
⎢                                                                    ⎥
⎢             -i·δ                                          -i·δ     ⎥
⎢             ─────                                         ─────    ⎥
⎢⎛     i·δ⎞    2                  ⎛ i·δ    2         2   ⎞    2     ⎥
⎣⎝1 - ℯ   ⎠·ℯ     ·sin(θ)·cos(θ)   ⎝ℯ   ·cos (θ) + sin (θ)⎠·ℯ      ⎦
```

sympy.physics.optics.polarization.**polarizing_beam_splitter**(*Tp=1*, *Rs=1*, *Ts=0*, *Rp=0*, *phia=0*, *phib=0*)

A polarizing beam splitter Jones matrix at angle $theta$.

Parameters

``**J**`` : SymPy Matrix

A Jones matrix.

``**Tp**`` : numeric type or SymPy Symbol

The transmissivity of the P-polarized component.

``**Rs**`` : numeric type or SymPy Symbol

The reflectivity of the S-polarized component.

``**Ts**`` : numeric type or SymPy Symbol

The transmissivity of the S-polarized component.

``**Rp**`` : numeric type or SymPy Symbol

The reflectivity of the P-polarized component.