Quantum Programming in Python

Quantum 1D Simple Harmonic Oscillator and Quantum Mapping Gate

A Senior Project

presented to

the Faculty of the Physics Department

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science

by

Matthew Hoff

March, 2013

Table of Contents

Abstract

A common problem when learning Quantum Mechanics is the complexity in the mathematical and physical concepts, which leads to difficulty in solving and understanding problems. Using programming languages like Python have become more and more prevalent in solving challenging physical systems. An open-source computer algebra system, SymPy, has been developed using Python to help solve these difficult systems. I have added code to the SymPy library for two different systems, a One-Dimensional Quantum Harmonic Oscillator and a Quantum Mapping Gate used in Quantum Computing.

Introduction

The goal of SymPy is "to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible" ("SymPy"). Through SymPy, I have submitted two projects to their library on GitHub, which "is a web-based hosting service for software development projects that use the Git revision control system" ("GitHub"). The first project is a Quantum Simple Harmonic Oscillator (QSHO), which is explained in a sample notebook using an IPython notebook. The goal of coding the QSHO is to allow others to learn how the simple harmonic oscillator is applied to a quantum system as well as allowing others to use components of the QSHO in other future projects. The second project is a Quantum Mapping Gate (QMG), which again is explained in greater detail using an IPython notebook. The QMG allows for custom creation of logic gates for quantum systems and can be used in addition to or instead of the current quantum gates. Both projects were coded using the Python language and have been added to the SymPy library.

Heading 1

One-Dimensional Quantum Simple Harmonic Oscillator

Imports

Before examining the Quantum 1D Simple Harmonic Oscillator, the relevant files need to be loaded to create simple harmonic oscillator states and operators.

In [1]:
```
%load_ext sympy.interactive.ipythonprinting
from sympy import Symbol, Integer
from sympy.physics.quantum import (Dagger,
                                   qapply,
                                   represent,
                                   InnerProduct,
                                   Commutator)
from sympy.physics.quantum.sho1d import (RaisingOp,
                                         LoweringOp,
                                         NumberOp,
                                         Hamiltonian,
                                         SHOKet,
                                         SHOBra)
```

Background

For a detailed background on the Quantum Simple Harmonic Oscillator consult Griffith's *Introduciton to Quantum Mechanics* or the Wikipedia page "Quantum Harmonic Oscillator"

Components

States

The Quantum 1D Simple Harmonic Oscillator is made up of states which can be expressed as bras and kets. And those states are acted on by different operators. Looking at the states, there are two types of states that can be made: a generic state 'n' or numerical states. Passing a string (i.e. 'n') to **SHOBra** or **SHOKet** creates a generic bra or ket state, respectively. And passing an integer to SHOBra or SHOKet creates a numerical bra or ket state.

SHOBra and SHOKet are passed the information from **SHOState** and from the Bra and Ket classes, respectively. SHOState contains the information to find the Hilbert Space of the state as well as the energy level. SHOState also has all the information and properties from State because State is passed into SHOState.

In [2]:
```
b = SHOBra('b')
b0 = SHOBra(0)
b1 = SHOBra(1)

k = SHOKet('k')
k0 = SHOKet(0)
k1 = SHOKet(1)
```

Printing

There are multiple printing methods in python: LaTeX, Pretty, repr, and srepr.

Bra

In [3]:
```
b
```

Out[3]: $\left\langle b\right|$

In [4]:
```
b0
```

Out[4]: $\left\langle 0\right|$

In [5]:
```
b1
```

Out[5]: $\left\langle 1\right|$

LaTex: Gives the printing for LaTeX typesetting

In [6]:
```
latex(b)
```

Out[6]:
```
{\left\langle b\right|}
```

In [7]:
```
latex(b0)
```

Out[7]:
```
{\left\langle 0\right|}
```

In [8]:
```
latex(b1)
```

Out[8]:
```
{\left\langle 1\right|}
```

Pretty

In [9]:
```
pprint(b)
```

```
⟨b|
```

In [10]:
```
pprint(b0)
```

```
⟨0|
```

In [11]:
```
pprint(b1)
```

```
⟨1|
```

repr

```
In [12]: repr(b)
```

```
Out[12]: <b|
```

```
In [13]: repr(b0)
```

```
Out[13]: <0|
```

```
In [14]: repr(b1)
```

```
Out[14]: <1|
```

srepr

```
In [15]: srepr(b)
```

```
Out[15]: SHOBra(Symbol('b'))
```

```
In [16]: srepr(b0)
```

```
Out[16]: SHOBra(Integer(0))
```

```
In [17]: srepr(b1)
```

```
Out[17]: SHOBra(Integer(1))
```

Ket

```
In [18]: k
```

Out[18]: $\left|k\right\rangle$

```
In [19]: k0
```

Out[19]: $\left|0\right\rangle$

```
In [20]: k1
```

Out[20]: $\left|1\right\rangle$

Latex: Gives the printing for LaTeX typesetting

```
In [21]: latex(k)
```

```
Out[21]: {\left|k\right\rangle
}
```

In [22]:
```
latex(k0)
```

Out[22]:
```
{\left|0\right\rangle
}
```

In [23]:
```
latex(k1)
```

Out[23]:
```
{\left|1\right\rangle
}
```

Pretty

In [24]:
```
pprint(k)
```

```
|k⟩
```

In [25]:
```
pprint(k0)
```

```
|0⟩
```

In [26]:
```
pprint(k1)
```

```
|1⟩
```

repr

In [27]:
```
repr(k)
```

Out[27]:
```
|k>
```

In [28]:
```
repr(k0)
```

Out[28]:
```
|0>
```

In [29]:
```
repr(k1)
```

Out[29]:
```
|1>
```

srepr

In [30]:
```
srepr(k)
```

Out[30]:
```
SHOKet(Symbol('k'))
```

In [31]:
```
srepr(k0)
```

Out[31]:
```
SHOKet(Integer(0))
```

In [32]:
```
srepr(k1)
```

Out[32]:
```
SHOKet(Integer(1))
```

Properites

Hilbert Space

In [33]:
```
b.hilbert_space
```
Out[33]: $\mathcal{C}^{\infty}$

In [34]:
```
b0.hilbert_space
```
Out[34]: $\mathcal{C}^{\infty}$

In [35]:
```
k.hilbert_space
```
Out[35]: $\mathcal{C}^{\infty}$

In [36]:
```
k0.hilbert_space
```
Out[36]: $\mathcal{C}^{\infty}$

Energy Level

In [37]:
```
b.n
```
Out[37]: b

In [38]:
```
b0.n
```
Out[38]: 0

In [39]:
```
b1.n
```
Out[39]: 1

In [40]:
```
k.n
```
Out[40]: k

In [41]:
```
k0.n
```
Out[41]: 0

In [42]:
```
k1.n
```
Out[42]: 1

Operators

The states are acted upon by operators. There are four operators that act on simple harmonic kets: **RaisingOp**, **LoweringOp**, **NumberOp**, and **Hamiltonian**. The operators are created by passing a string 'n' to the operator. They can also be printed in multiple ways, but only the raising operator has a distinct difference.

Each of the operators are passed the information and properties from **SHOOp**. SHOOp contains information on the hilbert space of the operators and how the arguments are evaluated. Each of these operators are limited to one argument. The

Operatos class is passed to SHOOp, which is in turn passed to each of the four quantum harmonic oscillators.

In [43]:
```
ad = RaisingOp('a')
a = LoweringOp('a')
N = NumberOp('N')
H = Hamiltonian('H')
```

Printing

RaisingOp

In [44]:
```
ad
```

Out[44]: $a^{\dagger}$

In [45]:
```
latex(ad)
```

Out[45]: `a^{\dag}`

In [46]:
```
pprint(ad)
```

```
 †
a
```

In [47]:
```
repr(ad)
```

Out[47]: `RaisingOp(a)`

In [48]:
```
srepr(ad)
```

Out[48]: `RaisingOp(Symbol('a'))`

LoweringOp

In [49]:
```
a
```

Out[49]: a

In [50]:
```
latex(a)
```

Out[50]: `a`

In [51]:
```
pprint(a)
```

```
a
```

In [52]:
```
repr(a)
```

Out[52]: `a`

In [53]:
```
srepr(a)
```

Out[53]: `LoweringOp(Symbol('a'))`

NumberOp

In [54]:
```
N
```
Out[54]: N

In [55]:
```
latex(N)
```
Out[55]:
```
N
```

In [56]:
```
pprint(N)
```
```
N
```

In [57]:
```
repr(N)
```
Out[57]:
```
N
```

In [58]:
```
srepr(N)
```
Out[58]:
```
NumberOp(Symbol('N'))
```

Hamiltonian

In [59]:
```
H
```
Out[59]: H

In [60]:
```
latex(H)
```
Out[60]:
```
H
```

In [61]:
```
pprint(H)
```
```
H
```

In [62]:
```
repr(H)
```
Out[62]:
```
H
```

In [63]:
```
srepr(H)
```
Out[63]:
```
Hamiltonian(Symbol('H'))
```

Properties

Hilbert Space

In [64]:
```
ad.hilbert_space
```
Out[64]: $\mathcal{C}^{\infty}$

In [65]:
```
a.hilbert_space
```
Out[65]: $\mathcal{C}^{\infty}$

In [66]:
```
N.hilbert_space
```
Out[66]: $\mathcal{C}^{\infty}$

In [67]:
```
H.hilbert_space
```
Out[67]: $\mathcal{C}^{\infty}$

Properties and Operations

There are a couple properties and operations of a quantum simple harmonic oscillator state that are defined. Taking the dagger of a bra returns the ket and vice versa. Using the property 'dual' returns the same value as taking the dagger. The property 'n' as seen in the State Section above returns the argument/energy level of the state, which is used when operators act on states.

In [68]:
```
Dagger(b)
```
Out[68]: $|b\rangle$

In [69]:
```
Dagger(k)
```
Out[69]: $\langle k|$

In [70]:
```
Dagger(b0)
```
Out[70]: $|0\rangle$

Tests that the dagger of a bra is equal to the ket.

In [71]:
```
Dagger(b0) == k0
```
Out[71]: $True$

Tests that the dagger of a ket is equal to the bra

In [72]:
```
Dagger(k1) == b1
```
Out[72]: $True$

The energy level of the states must be the same for the dagger of the bra to equal the ket

In [73]:
```
Dagger(b1) == k0
```
Out[73]: $False$

Tests that dagger(ket) = ket.dual and dagger(bra) = bra.dual

In [74]:
```
k.dual
```

Out[74]: $\langle k|$

In [75]:
```
Dagger(k) == k.dual
```

Out[75]: $True$

The raising operator is the dagger of the lowering operator

In [76]:
```
Dagger(a)
```

Out[76]: $a^{\dagger}$

In [77]:
```
Dagger(ad)
```

Out[77]: a

In [78]:
```
Dagger(a) == ad
```

Out[78]: $True$

The operators can be expressed in terms of other operators. Aside from the operators stated above rewriting in terms of the position (*X*) and momentum operators (*Px*) is common. To rewrite the operators in terms of other operators, we pass a keyword that specifies which operators to rewrite in.

'xp' -- Position and Momentum Operators

'a' -- Raising and Lowering Operators

'H' -- Hamiltonian Operator

'N' -- Number Operator

In [79]:
```
ad.rewrite('xp')
```

Out[79]: $$\frac{\sqrt{2}(m\omega X - \imath Px)}{2\sqrt{\hbar}\sqrt{m\omega}}$$

In [80]:
```
a.rewrite('xp')
```

Out[80]: $$\frac{\sqrt{2}(m\omega X + \imath Px)}{2\sqrt{\hbar}\sqrt{m\omega}}$$

In [81]:
```
N.rewrite('xp')
```

Out[81]: $$-\frac{1}{2} + \frac{m^2\omega^2(X)^2 + (Px)^2}{2\hbar m\omega}$$

In [82]:
```
N.rewrite('a')
```
Out[82]: $a^{\dagger}a$

In [83]:
```
N.rewrite('H')
```
Out[83]: $-\frac{1}{2}+\frac{H}{\hbar\omega}$

In [84]:
```
H.rewrite('xp')
```
Out[84]: $\frac{m^2\omega^2(X)^2+(Px)^2}{2m}$

In [85]:
```
H.rewrite('a')
```
Out[85]: $\hbar\omega\left(\frac{1}{2}+a^{\dagger}a\right)$

In [86]:
```
H.rewrite('N')
```
Out[86]: $\hbar\omega\left(\frac{1}{2}+N\right)$

Operator Methods

Apply Operators to States: Each of the operators can act on kets using qapply.

The raising operator raises the value of the state by one as well as multiplies the state by the square root of the new state.

In [87]:
```
qapply(ad*k)
```
Out[87]: $\sqrt{k+1}|k+1\rangle$

Two numerical examples with the ground state and first excited state.

In [88]:
```
qapply(ad*k0)
```
Out[88]: $|1\rangle$

In [89]:
```
qapply(ad*k1)
```
Out[89]: $\sqrt{2}|2\rangle$

The lowering operator lowers the value of the state by one and multiples the state by the square root of the original state. When the lowering operator acts on the ground state it returns zero because the state cannot be lowered.

In [90]:
```
qapply(a*k)
```

Out[90]: $\sqrt{k}|k-1\rangle$

Two numerical examples with the ground state and first excited state.

In [91]:
```
qapply(a*k0)
```

Out[91]: 0

In [92]:
```
qapply(a*k1)
```

Out[92]: $|0\rangle$

The number operator is defined as the raising operator times the lowering operator. When the number operator acts on a ket it returns the same state multiplied by the value of the state. This can be checked by applying the lowering operator on a state then applying the raising operator to the result.

In [93]:
```
qapply(N*k)
```

Out[93]: $k|k\rangle$

In [94]:
```
qapply(N*k0)
```

Out[94]: 0

In [95]:
```
qapply(N*k1)
```

Out[95]: $|1\rangle$

In [96]:
```
result = qapply(a*k)
qapply(ad*result)
```

Out[96]: $k|k\rangle$

When the hamiltonian operator acts on a state it returns the energy of the state, which is equal to hbar*omega times the value of the state plus one half.

In [97]:
```
qapply(H*k)
```

Out[97]:

$$\hbar k\omega|k\rangle + \frac{1}{2}\hbar\omega|k\rangle$$

In [98]:
```
qapply(H*k0)
```

Out[98]:

$$\frac{1}{2}\hbar\omega|0\rangle$$

In [99]:
```
qapply(H*k1)
```

Out[99]:

$$\frac{3}{2}\hbar\omega|1\rangle$$

Commutators

A commutator is defined as [A, B] = A*B - B*A where A and B are both operators. Commutators are used to see if operators commute, which is an important property in quantum mechanics. If they commute it allows for rearranging the order operators act on states.

In [100]:
```
Commutator(ad,a).doit()
```
Out[100]: -1

In [101]:
```
Commutator(ad,N).doit()
```
Out[101]: $-a^{\dagger}$

In [102]:
```
Commutator(a,ad).doit()
```
Out[102]: 1

In [103]:
```
Commutator(a,N).doit()
```
Out[103]: a

Matrix Representation

The bras and kets can also be represented as a row or column vector, which are then used to create matrix representation of the different operators. The bras and kets must be numerical states rather than a generic *n* state

In [104]:
```
represent(b0)
```
Out[104]: $\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$

In [105]:
```
represent(k0)
```
Out[105]: $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

Because these vectors and matrices are mostly zeros there is a different way of creating and storing these vectors/matrices, that is to use the format scipy.sparse. The default format is sympy and another common format to use is numpy. Along with specifying the format in which the matrices are created, the dimension of the matrices can also be specified. A dimension of 4 is the default.

In [106]:
```
represent(k1, ndim=5, format='sympy')
```
Out[106]: $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

In [107]:
```
represent(k1, ndim=5, format='numpy')
```

Out[107]:
```
 [[
0.]
  [
1.]
  [
0.]
  [
0.]
 [
0.]]
```

In [108]:
```
represent(k1, ndim=5, format='scipy.sparse')
```

Out[108]:
```
  (1, 0)
1.0
```

Operators can be expressed as matrices using the vector representation of the bras and kets.
$\langle i|N|j\rangle$
The operator acts on the ket then the inner product of the bra and the new resulting ket is performed.

In [109]:
```
represent(ad, ndim=4, format='sympy')
```

Out[109]:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & \sqrt{2} & 0 & 0 \\ 0 & 0 & \sqrt{3} & 0 \end{bmatrix}$$

In [110]:
```
represent(ad, format='numpy')
```

Out[110]:
```
 [[ 0.          0.          0.
0.        ]
  [ 1.          0.          0.
0.        ]
  [ 0.          1.41421356  0.
0.        ]
 [ 0.          0.          1.73205081  0.
]]
```

In [111]:
```
represent(ad, format='scipy.sparse', spmatrix='lil')
```

Out[111]:
```
       (1, 0)
1.0
  (2, 1)1.41421356237
  (3, 2)1.73205080757
```

In [112]:
```
str(represent(ad, format='scipy.sparse', spmatrix='lil'))
```

Out[112]:
```
       (1, 0)
1.0
  (2, 1)1.41421356237
  (3, 2)1.73205080757
```

In [113]:
```
represent(a)
```

Out[113]:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \sqrt{2} & 0 \\ 0 & 0 & 0 & \sqrt{3} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

In [114]:
```
represent(N)
```

Out[114]:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

In [115]:
```
represent(H)
```

Out[115]:

$$\begin{bmatrix} \frac{1}{2}\hbar\omega & 0 & 0 & 0 \\ 0 & \frac{3}{2}\hbar\omega & 0 & 0 \\ 0 & 0 & \frac{5}{2}\hbar\omega & 0 \\ 0 & 0 & 0 & \frac{7}{2}\hbar\omega \end{bmatrix}$$

There are some interesting properties that we can test using the matrix representation, like the definition of the Number Operator.

In [116]:
```
represent(N) == represent(ad) * represent(a)
```

Out[116]: $True$

For Additional Quantum Harmonic Oscillator Information

Griffiths, David J. Introduction to Quantum Mechanics. Upper Saddle River, NJ: Pearson Prentice Hall, 2005. Print.

http://en.wikipedia.org/wiki/Quantum_harmonic_oscillator

http://en.wikipedia.org/wiki/Harmonic_oscillator#Simple_harmonic_oscillator

In [ ]:
```

```

```python
"""Simple Harmonic Oscillator 1-Dimension"""

from sympy import sqrt, I, Symbol, Integer, S
from sympy.physics.quantum.constants import hbar
from sympy.physics.quantum.operator import Operator
from sympy.physics.quantum.state import Bra, Ket, State
from sympy.physics.quantum.qexpr import QExpr
from sympy.physics.quantum.cartesian import X, Px
from sympy.functions.special.tensor_functions import KroneckerDelta
from sympy.physics.quantum.hilbert import ComplexSpace
from sympy.physics.quantum.matrixutils import matrix_zeros
from sympy.physics.quantum.represent import represent

#------------------------------------------------------------------------------

class SHOOp(Operator):
    """A base class for the SHO Operators.

    We are limiting the number of arguments to be 1.

    """

    @classmethod
    def _eval_args(cls, args):
        args = QExpr._eval_args(args)
        if len(args) == 1:
            return args
        else:
            raise ValueError("Too many arguments")

    @classmethod
    def _eval_hilbert_space(cls, label):
        return ComplexSpace(S.Infinity)

class RaisingOp(SHOOp):
    """The Raising Operator or a^dagger.

    When a^dagger acts on a state it raises the state up by one. Taking
    the adjoint of a^dagger returns 'a', the Lowering Operator. a^dagger
    can be rewritten in terms of postion and momentum. We can represent
    a^dagger as a matrix, which will be its default basis.

    Parameters
    ==========

    args : tuple
        The list of numbers or parameters that uniquely specify the
        operator.

    Examples
    ========
```

```

    Create a Raising Operator and rewrite it in terms of positon and
    momentum, and show that taking its adjoint returns 'a':

        >>> from sympy.physics.quantum.sho1d import RaisingOp
        >>> from sympy.physics.quantum import Dagger

        >>> ad = RaisingOp('a')
        >>> ad.rewrite('xp').doit()
        sqrt(2)*(m*omega*X - I*Px)/(2*sqrt(hbar)*sqrt(m*omega))

        >>> Dagger(ad)
        a

    Taking the commutator of a^dagger with other Operators:

        >>> from sympy.physics.quantum import Commutator
        >>> from sympy.physics.quantum.sho1d import RaisingOp, LoweringOp
        >>> from sympy.physics.quantum.sho1d import NumberOp

        >>> ad = RaisingOp('a')
        >>> a = LoweringOp('a')
        >>> N = NumberOp('N')
        >>> Commutator(ad, a).doit()
        -1
        >>> Commutator(ad, N).doit()
        -RaisingOp(a)

    Apply a^dagger to a state:

        >>> from sympy.physics.quantum import qapply
        >>> from sympy.physics.quantum.sho1d import RaisingOp, SHOKet

        >>> ad = RaisingOp('a')
        >>> k = SHOKet('k')
        >>> qapply(ad*k)
        sqrt(k + 1)*|k + 1>

    Matrix Representation

        >>> from sympy.physics.quantum.sho1d import RaisingOp
        >>> from sympy.physics.quantum.represent import represent
        >>> ad = RaisingOp('a')
        >>> represent(ad, basis=N, ndim=4, format='sympy')
        [0,       0,       0, 0]
        [1,       0,       0, 0]
        [0, sqrt(2),       0, 0]
        [0,       0, sqrt(3), 0]

    """

    def _eval_rewrite_as_xp(self, *args):
```

```python
            return (Integer(1)/sqrt(Integer(2)*hbar*m*omega))*(
                Integer(-1)*I*Px + m*omega*X)

    def _eval_adjoint(self):
        return LoweringOp(*self.args)

    def _eval_commutator_LoweringOp(self, other):
        return Integer(-1)

    def _eval_commutator_NumberOp(self, other):
        return Integer(-1)*self

    def _apply_operator_SHOKet(self, ket):
        temp = ket.n + Integer(1)
        return sqrt(temp)*SHOKet(temp)

    def _represent_default_basis(self, **options):
        return self._represent_NumberOp(None, **options)

    def _represent_XOp(self, basis, **options):
        # This logic is good but the underlying positon
        # representation logic is broken.
        # temp = self.rewrite('xp').doit()
        # result = represent(temp, basis=X)
        # return result
        raise NotImplementedError('Position representation is not implemented')

    def _represent_NumberOp(self, basis, **options):
        ndim_info = options.get('ndim', 4)
        format = options.get('format','sympy')
        spmatrix = options.get('spmatrix', 'csr')
        matrix = matrix_zeros(ndim_info, ndim_info, **options)
        for i in range(ndim_info - 1):
            value = sqrt(i + 1)
            if format == 'scipy.sparse':
                value = float(value)
            matrix[i + 1, i] = value
        if format == 'scipy.sparse':
            matrix = matrix.tocsr()
        return matrix

    #--------------------------------------------------------------------------
    # Printing Methods
    #--------------------------------------------------------------------------

    def _print_contents(self, printer, *args):
        arg0 = printer._print(self.args[0], *args)
```

```
        return '%s(%s)' % (self.__class__.__name__, arg0)

    def _print_contents_pretty(self, printer, *args):
        from sympy.printing.pretty.stringpict import prettyForm
        pform = printer._print(self.args[0], *args)
        pform = pform**prettyForm(u'\u2020')
        return pform

    def _print_contents_latex(self, printer, *args):
        arg = printer._print(self.args[0])
        return '%s^{\\dag}' % arg

class LoweringOp(SHOOp):
    """The Lowering Operator or 'a'.

    When 'a' acts on a state it lowers the state up by one. Taking
    the adjoint of 'a' returns a^dagger, the Raising Operator. 'a'
    can be rewritten in terms of position and momentum. We can
    represent 'a' as a matrix, which will be its default basis.

    Parameters
    ==========

    args : tuple
        The list of numbers or parameters that uniquely specify the
        operator.

    Examples
    ========

    Create a Lowering Operator and rewrite it in terms of positon and
    momentum, and show that taking its adjoint returns a^dagger:

        >>> from sympy.physics.quantum.sho1d import LoweringOp
        >>> from sympy.physics.quantum import Dagger

        >>> a = LoweringOp('a')
        >>> a.rewrite('xp').doit()
        sqrt(2)*(m*omega*X + I*Px)/(2*sqrt(hbar)*sqrt(m*omega))

        >>> Dagger(a)
        RaisingOp(a)

    Taking the commutator of 'a' with other Operators:

        >>> from sympy.physics.quantum import Commutator
        >>> from sympy.physics.quantum.sho1d import LoweringOp, RaisingOp
        >>> from sympy.physics.quantum.sho1d import NumberOp

        >>> a = LoweringOp('a')
        >>> ad = RaisingOp('a')
        >>> N = NumberOp('N')
```

```
        >>> Commutator(a, ad).doit()
        1
        >>> Commutator(a, N).doit()
        a

    Apply 'a' to a state:

        >>> from sympy.physics.quantum import qapply
        >>> from sympy.physics.quantum.sho1d import LoweringOp, SHOKet

        >>> a = LoweringOp('a')
        >>> k = SHOKet('k')
        >>> qapply(a*k)
        sqrt(k)*|k - 1>

    Taking 'a' of the lowest state will return 0:

        >>> from sympy.physics.quantum import qapply
        >>> from sympy.physics.quantum.sho1d import LoweringOp, SHOKet

        >>> a = LoweringOp('a')
        >>> k = SHOKet(0)
        >>> qapply(a*k)
        0

    Matrix Representation

        >>> from sympy.physics.quantum.sho1d import LoweringOp
        >>> from sympy.physics.quantum.represent import represent
        >>> a = LoweringOp('a')
        >>> represent(a, basis=N, ndim=4, format='sympy')
        [0, 1,       0,       0]
        [0, 0, sqrt(2),       0]
        [0, 0,       0, sqrt(3)]
        [0, 0,       0,       0]

    """

    def _eval_rewrite_as_xp(self, *args):
        return (Integer(1)/sqrt(Integer(2)*hbar*m*omega))*(
            I*Px + m*omega*X)

    def _eval_adjoint(self):
        return RaisingOp(*self.args)

    def _eval_commutator_RaisingOp(self, other):
        return Integer(1)

    def _eval_commutator_NumberOp(self, other):
        return Integer(1)*self

    def _apply_operator_SHOKet(self, ket):
```

```python
        temp = ket.n - Integer(1)
        if ket.n == Integer(0):
            return Integer(0)
        else:
            return sqrt(ket.n)*SHOKet(temp)

    def _represent_default_basis(self, **options):
        return self._represent_NumberOp(None, **options)

    def _represent_XOp(self, basis, **options):
        # This logic is good but the underlying positon
        # representation logic is broken.
        # temp = self.rewrite('xp').doit()
        # result = represent(temp, basis=X)
        # return result
        raise NotImplementedError('Position representation is not implemented')

    def _represent_NumberOp(self, basis, **options):
        ndim_info = options.get('ndim', 4)
        format = options.get('format', 'sympy')
        spmatrix = options.get('spmatrix', 'csr')
        matrix = matrix_zeros(ndim_info, ndim_info, **options)
        for i in range(ndim_info - 1):
            value = sqrt(i + 1)
            if format == 'scipy.sparse':
                value = float(value)
            matrix[i,i + 1] = value
        if format == 'scipy.sparse':
            matrix = matrix.tocsr()
        return matrix


class NumberOp(SHOOp):
    """The Number Operator is simply a^dagger*a

    It is often useful to write a^dagger*a as simply the Number Operator
    because the Number Operator commutes with the Hamiltonian. And can be
    expressed using the Number Operator. Also the Number Operator can be
    applied to states. We can represent the Number Operator as a matrix,
    which will be its default basis.

    Parameters
    ==========

    args : tuple
        The list of numbers or parameters that uniquely specify the
        operator.

    Examples
    ========

```

```
    Create a Number Operator and rewrite it in terms of the ladder
    operators, position and momentum operators, and Hamiltonian:

        >>> from sympy.physics.quantum.sho1d import NumberOp

        >>> N = NumberOp('N')
        >>> N.rewrite('a').doit()
        RaisingOp(a)*a
        >>> N.rewrite('xp').doit()
        -1/2 + (m**2*omega**2*X**2 + Px**2)/(2*hbar*m*omega)
        >>> N.rewrite('H').doit()
        -1/2 + H/(hbar*omega)

    Take the Commutator of the Number Operator with other Operators:

        >>> from sympy.physics.quantum import Commutator
        >>> from sympy.physics.quantum.sho1d import NumberOp, Hamiltonian
        >>> from sympy.physics.quantum.sho1d import RaisingOp, LoweringOp

        >>> N = NumberOp('N')
        >>> H = Hamiltonian('H')
        >>> ad = RaisingOp('a')
        >>> a = LoweringOp('a')
        >>> Commutator(N,H).doit()
        0
        >>> Commutator(N,ad).doit()
        RaisingOp(a)
        >>> Commutator(N,a).doit()
        -a

    Apply the Number Operator to a state:

        >>> from sympy.physics.quantum import qapply
        >>> from sympy.physics.quantum.sho1d import NumberOp, SHOKet

        >>> N = NumberOp('N')
        >>> k = SHOKet('k')
        >>> qapply(N*k)
        k*|k>

    Matrix Representation

        >>> from sympy.physics.quantum.sho1d import NumberOp
        >>> from sympy.physics.quantum.represent import represent
        >>> N = NumberOp('N')
        >>> represent(N, basis=N, ndim=4, format='sympy')
        [0, 0, 0, 0]
        [0, 1, 0, 0]
        [0, 0, 2, 0]
        [0, 0, 0, 3]

    """
```

```

    def _eval_rewrite_as_a(self, *args):
        return ad*a

    def _eval_rewrite_as_xp(self, *args):
        return (Integer(1)/(Integer(2)*m*hbar*omega))*(Px**2 + (
            m*omega*X)**2) - Integer(1)/Integer(2)

    def _eval_rewrite_as_H(self, *args):
        return H/(hbar*omega) - Integer(1)/Integer(2)

    def _apply_operator_SHOKet(self, ket):
        return ket.n*ket

    def _eval_commutator_Hamiltonian(self, other):
        return Integer(0)

    def _eval_commutator_RaisingOp(self, other):
        return other

    def _eval_commutator_LoweringOp(self, other):
        return Integer(-1)*other

    def _represent_default_basis(self, **options):
        return self._represent_NumberOp(None, **options)

    def _represent_XOp(self, basis, **options):
        # This logic is good but the underlying positon
        # representation logic is broken.
        # temp = self.rewrite('xp').doit()
        # result = represent(temp, basis=X)
        # return result
        raise NotImplementedError('Position representation is not implemented')

    def _represent_NumberOp(self, basis, **options):
        ndim_info = options.get('ndim', 4)
        format = options.get('format', 'sympy')
        spmatrix = options.get('spmatrix', 'csr')
        matrix = matrix_zeros(ndim_info, ndim_info, **options)
        for i in range(ndim_info):
            value = i
            if format == 'scipy.sparse':
                value = float(value)
            matrix[i,i] = value
        if format == 'scipy.sparse':
            matrix = matrix.tocsr()
        return matrix


class Hamiltonian(SHOOp):
    """The Hamiltonian Operator.
```

```
    The Hamiltonian is used to solve the time-independent Schrodinger
    equation. The Hamiltonian can be expressed using the ladder operators,
    as well as by position and momentum. We can represent the Hamiltonian
    Operator as a matrix, which will be its default basis.

    Parameters
    ==========

    args : tuple
        The list of numbers or parameters that uniquely specify the
        operator.

    Examples
    ========

    Create a Hamiltonian Operator and rewrite it in terms of the ladder
    operators, position and momentum, and the Number Operator:

        >>> from sympy.physics.quantum.sho1d import Hamiltonian

        >>> H = Hamiltonian('H')
        >>> H.rewrite('a').doit()
        hbar*omega*(1/2 + RaisingOp(a)*a)
        >>> H.rewrite('xp').doit()
        (m**2*omega**2*X**2 + Px**2)/(2*m)
        >>> H.rewrite('N').doit()
        hbar*omega*(1/2 + N)

    Take the Commutator of the Hamiltonian and the Number Operator:

        >>> from sympy.physics.quantum import Commutator
        >>> from sympy.physics.quantum.sho1d import Hamiltonian, NumberOp

        >>> H = Hamiltonian('H')
        >>> N = NumberOp('N')
        >>> Commutator(H,N).doit()
        0

    Apply the Hamiltonian Operator to a state:

        >>> from sympy.physics.quantum import qapply
        >>> from sympy.physics.quantum.sho1d import Hamiltonian, SHOKet

        >>> H = Hamiltonian('H')
        >>> k = SHOKet('k')
        >>> qapply(H*k)
        hbar*k*omega*|k> + hbar*omega*|k>/2

    Matrix Representation

        >>> from sympy.physics.quantum.sho1d import Hamiltonian
```

```
        >>> from sympy.physics.quantum.represent import represent

        >>> H = Hamiltonian('H')
        >>> represent(H, basis=N, ndim=4, format='sympy')
        [hbar*omega/2,              0,              0,              0]
        [           0, 3*hbar*omega/2,              0,              0]
        [           0,              0, 5*hbar*omega/2,              0]
        [           0,              0,              0, 7*hbar*omega/2]

    """

    def _eval_rewrite_as_a(self, *args):
        return hbar*omega*(ad*a + Integer(1)/Integer(2))

    def _eval_rewrite_as_xp(self, *args):
        return (Integer(1)/(Integer(2)*m))*(Px**2 + (m*omega*X)**2)

    def _eval_rewrite_as_N(self, *args):
        return hbar*omega*(N + Integer(1)/Integer(2))

    def _apply_operator_SHOKet(self, ket):
        return (hbar*omega*(ket.n + Integer(1)/Integer(2)))*ket

    def _eval_commutator_NumberOp(self, other):
        return Integer(0)

    def _represent_default_basis(self, **options):
        return self._represent_NumberOp(None, **options)

    def _represent_XOp(self, basis, **options):
        # This logic is good but the underlying positon
        # representation logic is broken.
        # temp = self.rewrite('xp').doit()
        # result = represent(temp, basis=X)
        # return result
        raise NotImplementedError('Position representation is not implemented')

    def _represent_NumberOp(self, basis, **options):
        ndim_info = options.get('ndim', 4)
        format = options.get('format', 'sympy')
        spmatrix = options.get('spmatrix', 'csr')
        matrix = matrix_zeros(ndim_info, ndim_info, **options)
        for i in range(ndim_info):
            value = i + Integer(1)/Integer(2)
            if format == 'scipy.sparse':
                value = float(value)
            matrix[i,i] = value
        if format == 'scipy.sparse':
            matirx = matrix.tocsr()
        return hbar*omega*matrix

```

```python
#--------------------------------------------------------------------------------

class SHOState(State):
    """State class for SHO states"""

    @classmethod
    def _eval_hilbert_space(cls, label):
        return ComplexSpace(S.Infinity)

    @property
    def n(self):
        return self.args[0]


class SHOKet(SHOState, Ket):
    """1D eigenket.

    Inherits from SHOState and Ket.

    Parameters
    ==========

    args : tuple
        The list of numbers or parameters that uniquely specify the ket
        This is usually its quantum numbers or its symbol.

    Examples
    ========

    Ket's know about their associated bra:

        >>> from sympy.physics.quantum.sho1d import SHOKet

        >>> k = SHOKet('k')
        >>> k.dual
        <k|
        >>> k.dual_class()
        <class 'sympy.physics.quantum.sho1d.SHOBra'>

    Take the Inner Product with a bra:

        >>> from sympy.physics.quantum import InnerProduct
        >>> from sympy.physics.quantum.sho1d import SHOKet, SHOBra

        >>> k = SHOKet('k')
        >>> b = SHOBra('b')
        >>> InnerProduct(b,k).doit()
        KroneckerDelta(k, b)

    Vector representation of a numerical state ket:

```

```
        >>> from sympy.physics.quantum.sho1d import SHOKet, NumberOp
        >>> from sympy.physics.quantum.represent import represent

        >>> k = SHOKet(3)
        >>> N = NumberOp('N')
        >>> represent(k, basis=N, ndim=4)
        [0]
        [0]
        [0]
        [1]

    """

    @classmethod
    def dual_class(self):
        return SHOBra

    def _eval_innerproduct_SHOBra(self, bra, **hints):
        result = KroneckerDelta(self.n, bra.n)
        return result

    def _represent_default_basis(self, **options):
        return self._represent_NumberOp(None, **options)

    def _represent_NumberOp(self, basis, **options):
        ndim_info = options.get('ndim', 4)
        format = options.get('format', 'sympy')
        options['spmatrix'] = 'lil'
        vector = matrix_zeros(ndim_info, 1, **options)
        if isinstance(self.n, Integer):
            if self.n >= ndim_info:
                return ValueError("N-Dimension too small")
            value = Integer(1)
            if format == 'scipy.sparse':
                vector[int(self.n), 0] = 1.0
                vector = vector.tocsr()
            elif format == 'numpy':
                vector[int(self.n), 0] = 1.0
            else:
                vector[self.n, 0] = Integer(1)
            return vector
        else:
            return ValueError("Not Numerical State")


class SHOBra(SHOState, Bra):
    """A time-independent Bra in SHO.

    Inherits from SHOState and Bra.

    Parameters
    ==========
```

```

    args : tuple
        The list of numbers or parameters that uniquely specify the ket
        This is usually its quantum numbers or its symbol.

    Examples
    ========

    Bra's know about their associated ket:

        >>> from sympy.physics.quantum.sho1d import SHOBra

        >>> b = SHOBra('b')
        >>> b.dual
        |b>
        >>> b.dual_class()
        <class 'sympy.physics.quantum.sho1d.SHOKet'>

    Vector representation of a numerical state bra:

        >>> from sympy.physics.quantum.sho1d import SHOBra, NumberOp
        >>> from sympy.physics.quantum.represent import represent

        >>> b = SHOBra(3)
        >>> N = NumberOp('N')
        >>> represent(b, basis=N, ndim=4)
        [0, 0, 0, 1]

    """

    @classmethod
    def dual_class(self):
        return SHOKet

    def _represent_default_basis(self, **options):
        return self._represent_NumberOp(None, **options)

    def _represent_NumberOp(self, basis, **options):
        ndim_info = options.get('ndim', 4)
        format = options.get('format', 'sympy')
        opitons['spmatrix'] = 'lil'
        vector = matrix_zeros(1, ndim_info, **options)
        if isinstance(self.n, Integer):
            if self.n >= ndim_info:
                return ValueError("N-Dimension too small")
            if format == 'scipy.sparse':
                vector[0, int(self.n)] = 1.0
                vector = vector.tocsr()
            elif format == 'numpy':
                vector[0, int(self.n)] = 1.0
            else:
                vector[0, self.n] = Integer(1)
```

```
            return vector
        else:
            return ValueError("Not Numerical State")


ad = RaisingOp('a')
a = LoweringOp('a')
H = Hamiltonian('H')
N = NumberOp('N')
omega = Symbol('omega')
m = Symbol('m')
```

```python
"""Tests for sho1d.py"""

from sympy import Integer, Symbol, sqrt, I, S
from sympy.physics.quantum import Dagger
from sympy.physics.quantum.constants import hbar
from sympy.physics.quantum import Commutator
from sympy.physics.quantum.qapply import qapply
from sympy.physics.quantum.innerproduct import InnerProduct
from sympy.physics.quantum.cartesian import X, Px
from sympy.functions.special.tensor_functions import KroneckerDelta
from sympy.physics.quantum.hilbert import ComplexSpace
from sympy.physics.quantum.represent import represent
from sympy.external import import_module
from sympy.utilities.pytest import skip

from sympy.physics.quantum.sho1d import (RaisingOp, LoweringOp,
                                        SHOKet, SHOBra,
                                        Hamiltonian, NumberOp)

ad = RaisingOp('a')
a = LoweringOp('a')
k = SHOKet('k')
kz = SHOKet(0)
kf = SHOKet(1)
k3 = SHOKet(3)
b = SHOBra('b')
b3 = SHOBra(3)
H = Hamiltonian('H')
N = NumberOp('N')
omega = Symbol('omega')
m = Symbol('m')
ndim = Integer(4)

np = import_module('numpy', min_python_version=(2, 6))
scipy = import_module('scipy', __import__kwargs={'fromlist': ['sparse']})

ad_rep_sympy = represent(ad, basis=N, ndim=4, format='sympy')
a_rep = represent(a, basis=N, ndim=4, format='sympy')
N_rep = represent(N, basis=N, ndim=4, format='sympy')
H_rep = represent(H, basis=N, ndim=4, format='sympy')
k3_rep = represent(k3, basis=N, ndim=4, format='sympy')
b3_rep = represent(b3, basis=N, ndim=4, format='sympy')

def test_RaisingOp():
    assert Dagger(ad) == a
    assert Commutator(ad, a).doit() == Integer(-1)
    assert Commutator(ad, N).doit() == Integer(-1)*ad
    assert qapply(ad*k) == (sqrt(k.n + 1)*SHOKet(k.n + 1)).expand()
    assert qapply(ad*kz) == (sqrt(kz.n + 1)*SHOKet(kz.n + 1)).expand()
    assert qapply(ad*kf) == (sqrt(kf.n + 1)*SHOKet(kf.n + 1)).expand()
    assert ad.rewrite('xp').doit() == \
        (Integer(1)/sqrt(Integer(2)*hbar*m*omega))*(Integer(-1)*I*Px +
```

```
m*omega*X)
    assert ad.hilbert_space == ComplexSpace(S.Infinity)
    for i in range(ndim - 1):
        assert ad_rep_sympy[i + 1,i] == sqrt(i + 1)

    if not np:
        skip("numpy not installed or Python too old.")

    ad_rep_numpy = represent(ad, basis=N, ndim=4, format='numpy')
    for i in range(ndim - 1):
        assert ad_rep_numpy[i + 1,i] == float(sqrt(i + 1))

    if not np:
        skip("numpy not installed or Python too old.")
    if not scipy:
        skip("scipy not installed.")
    else:
        sparse = scipy.sparse

    ad_rep_scipy = represent(ad, basis=N, ndim=4, format='scipy.sparse', spmatrix='lil')
    for i in range(ndim - 1):
        assert ad_rep_scipy[i + 1,i] == float(sqrt(i + 1))

    assert ad_rep_numpy.dtype == 'float64'
    assert ad_rep_scipy.dtype == 'float64'

def test_LoweringOp():
    assert Dagger(a) == ad
    assert Commutator(a, ad).doit() == Integer(1)
    assert Commutator(a, N).doit() == a
    assert qapply(a*k) == (sqrt(k.n)*SHOKet(k.n-Integer(1))).expand()
    assert qapply(a*kz) == Integer(0)
    assert qapply(a*kf) == (sqrt(kf.n)*SHOKet(kf.n-Integer(1))).expand()
    assert a.rewrite('xp').doit() == \
        (Integer(1)/sqrt(Integer(2)*hbar*m*omega))*(I*Px + m*omega*X)
    for i in range(ndim - 1):
        assert a_rep[i,i + 1] == sqrt(i + 1)

def test_NumberOp():
    assert Commutator(N, ad).doit() == ad
    assert Commutator(N, a).doit() == Integer(-1)*a
    assert Commutator(N, H).doit() == Integer(0)
    assert qapply(N*k) == (k.n*k).expand()
    assert N.rewrite('a').doit() == ad*a
    assert N.rewrite('xp').doit() == (Integer(1)/(Integer(2)*m*hbar*omega))*(
        Px**2 + (m*omega*X)**2) - Integer(1)/Integer(2)
    assert N.rewrite('H').doit() == H/(hbar*omega) - Integer(1)/Integer(2)
    for i in range(ndim):
        assert N_rep[i,i] == i
    assert N_rep == ad_rep_sympy*a_rep
```

```
def test_Hamiltonian():
    assert Commutator(H, N).doit() == Integer(0)
    assert qapply(H*k) == ((hbar*omega*(k.n + Integer(1)/Integer(2)))*k).expand()
    assert H.rewrite('a').doit() == hbar*omega*(ad*a + Integer(1)/Integer(2))
    assert H.rewrite('xp').doit() == \
        (Integer(1)/(Integer(2)*m))*(Px**2 + (m*omega*X)**2)
    assert H.rewrite('N').doit() == hbar*omega*(N + Integer(1)/Integer(2))
    for i in range(ndim):
        assert H_rep[i,i] == hbar*omega*(i + Integer(1)/Integer(2))

def test_SHOKet():
    assert SHOKet('k').dual_class() == SHOBra
    assert SHOBra('b').dual_class() == SHOKet
    assert InnerProduct(b,k).doit() == KroneckerDelta(k.n, b.n)
    assert k.hilbert_space == ComplexSpace(S.Infinity)
    assert k3_rep[k3.n, 0] == Integer(1)
    assert b3_rep[0, b3.n] == Integer(1)
```

Heading 1

Mapping Gate

Imports

Before examining the Mapping Gate, the relevant files need to be loaded.

In [1]:
```
%load_ext sympy.interactive.ipythonprinting
from sympy import Symbol, Integer, I
from sympy.core.containers import Dict
from sympy.physics.quantum import qapply, represent
from sympy.physics.quantum.qubit import Qubit
from sympy.physics.quantum.mappinggate import MappingGate
```

Theory/Background

Creating a **MappingGate** can be very useful in Quantum Computing. Normally one would have to use a combination of various quantum gates to get the desired input-output state pairings. The Mapping Gate allows for user provided initial and final state pairings for every state. Then a quantum gate that has the same pairings is created.

The Mapping Gate maps an initial qubits to scalars times final qubits. If no scalar is specified one is assumed. So there are two or three arguments for the mapping gate.

arg[0] = initial state

arg[1] = scalar or final state

arg[2] = final state or none

The qubits can either be strings or qubits. The resulting arguments of the MappingGate are converted to a sympy dictionary.

There are multiple ways to specify the qubit pairings. All quantum gates have the property of being unitary, which means only half of the gate needs to be specified and the rest can be assumed and created to preserve the unitary property. When thinking about the gates as matrices, it is only required to specify half of the matrix. And if an initial state is not paired to a final state it will return itself like the Identity Gate.

The Mapping Gate can also take a Python or Sympy dictionary as its argument. The dictionary requires qubit objects rather than strings and the scalars are multiplied with the final states.

{Qubit('initial'):scalar*Qubit('final'), ...}

Creating the MappingGate

Specify all states

Here, all pairs are specified. There are 2^n number of pairs, where n is the number of qubits in each state.

In [2]:
```
M_all = MappingGate(('00', I, '11'), ('01', exp(I), '10'), ('10', exp(-I), '01'), ('11
```

In [3]:
```
M_all.args
```

Out[3]: $\left(\left\{ |00\rangle : \imath|11\rangle, \quad |01\rangle : e^{\imath}|10\rangle, \quad |10\rangle : e^{-\imath}|01\rangle, \quad |11\rangle : -\imath|00\rangle \right\} \right)$

Another useful way to express a quantum gate is using outer product representation. This takes the form:

ket(final)*bra(initial) + ...

To call this, we use rewrite and pass it a keyword, in this case 'op'.

In [4]:
```
M_all.rewrite('op')
```

Out[4]: $-\imath|00\rangle\langle 11| + e^{-\imath}|01\rangle\langle 10| + e^{\imath}|10\rangle\langle 01| + \imath|11\rangle\langle 00|$

Another common way of expressing a quantum gate is in matrix form. Any term in a matrix can be identified using a bra and a ket (i.e. (0,1) is the same as bra(0) ket(1)). Using this idea a quantum gate is created by inserting the outer product in between each identifying bra and ket for each term. If the qubits in the MappingGate return themselves the resulting matrix is the identity gate.

In [5]:
```
M_test = MappingGate(('0','0'), ('1','1'))
```

In [6]:
```
represent(M_test)
```

Out[6]: $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

Now taking a look at the more complex qubit mapping from above, it is clear that the outer product terms are directly related to the terms of the matrix.

In [7]:
```
represent(M_all)
```

Out[7]: $\begin{bmatrix} 0 & 0 & 0 & -\imath \\ 0 & 0 & e^{-\imath} & 0 \\ 0 & e^{\imath} & 0 & 0 \\ \imath & 0 & 0 & 0 \end{bmatrix}$

Specify only half of the states and relying on the unitary property of the gate.

Here, only half of the pairs are specified because the unitary property of the gate can fill the rest. Given that *initial* = *scalar* * *final*, this implies that *final* = *conjugate(scalar)* * *initial*. This is what allows for only mapping the upper triangle of the matrix. The idea of only mapping the one triangle will be more easily seen when the gate is represented in matrix form

In [8]:
```
M_half = MappingGate(('00', I, '11'), ('01', exp(I), '10'))
```

In [9]:
```
M_half.args
```
Out[9]: $\left(\left\{ |00\rangle : \imath|11\rangle, \quad |01\rangle : e^{\imath}|10\rangle, \quad |10\rangle : e^{-\imath}|01\rangle, \quad |11\rangle : -\imath|00\rangle \right\}\right)$

We can check that the mapping for this is the same as the full mapping

In [10]:
```
M_half.args == M_all.args
```
Out[10]: True

Let's look at the outerproduct representation, it should be the same as above in M_all

In [11]:
```
M_half.rewrite('op')
```
Out[11]: $-\imath|00\rangle\langle 11| + e^{-\imath}|01\rangle\langle 10| + e^{\imath}|10\rangle\langle 01| + \imath|11\rangle\langle 00|$

Using the matrix representation it is clear that we only mapped the terms at (3,0) and (2,1) and the terms at (0,3) and (1,2) are the conjugates of the mapped terms.

In [12]:
```
represent(M_half)
```
Out[12]: $\begin{bmatrix} 0 & 0 & 0 & -\imath \\ 0 & 0 & e^{-\imath} & 0 \\ 0 & e^{\imath} & 0 & 0 \\ \imath & 0 & 0 & 0 \end{bmatrix}$

Specify only some states

Here, only some states will be specified and their compliments. Any state not specified returns itself, but is not one of the arguments of MappingGate.

In [13]:
```
M_some = MappingGate(('00', I, '11'))
```

In [14]:
```
M_some.args
```
Out[14]: $\left(\left\{ |00\rangle : \imath|11\rangle, \quad |11\rangle : -\imath|00\rangle \right\}\right)$

When using the outer product representation it will be clear that non-specified states return themselves.

In [15]:
```
M_some.rewrite('op')
```
Out[15]: $|01\rangle\langle 01| + |10\rangle\langle 10| - \imath|00\rangle\langle 11| + \imath|11\rangle\langle 00|$

States that return themselves will yield a 1 along the diaganol like the identity matrix.

In [16]:
```
represent(M_some)
```
Out[16]: $\begin{bmatrix} 0 & 0 & 0 & -\imath \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \imath & 0 & 0 & 0 \end{bmatrix}$

Using a Python Dictionary

The MappingGate can also take dictionaries as its arguments. Passing a dictionary as the argument to MappingGate works exactly the same as seen above except for the required form of the dictionary. Where as above MappingGate accepts either strings or qubits, a dictionary must contain qubits. Again if not all states are specified they will return themselves and only half of the matrix needs to be mapped. MappingGate converts the python dictionary to a sympy dictionary.

In [17]:
```
d = dict({Qubit('00'):I*Qubit('11'), Qubit('01'):exp(I)*Qubit('10')})
M_python_dict = MappingGate(d)
```

In [18]:
```
M_python_dict.args
```

Out[18]: $\left(\left\{ |00\rangle : \imath|11\rangle, \quad |01\rangle : e^{\imath}|10\rangle, \quad |10\rangle : e^{-\imath}|01\rangle, \quad |11\rangle : -\imath|00\rangle \right\}\right)$

Check that the outer product and matrix are the same as the the previous gates.

In [19]:
```
M_python_dict.rewrite('op')
```

Out[19]: $-\imath|00\rangle\langle 11| + e^{-\imath}|01\rangle\langle 10| + e^{\imath}|10\rangle\langle 01| + \imath|11\rangle\langle 00|$

In [20]:
```
M_python_dict.rewrite('op') == M_half.rewrite('op') == M_all.rewrite('op')
```

Out[20]: $True$

In [21]:
```
represent(M_python_dict)
```

Out[21]:
$$\begin{bmatrix} 0 & 0 & 0 & -\imath \\ 0 & 0 & e^{-\imath} & 0 \\ 0 & e^{\imath} & 0 & 0 \\ \imath & 0 & 0 & 0 \end{bmatrix}$$

In [22]:
```
represent(M_python_dict) == represent(M_half) == represent(M_all)
```

Out[22]: $True$

Using a Sympy Dictionary

A sympy dictionary can also be used to specify the qubit mapping. It works the same as the python dictionary.

In [23]:
```
d = Dict({Qubit('00'):I*Qubit('11'), Qubit('01'):exp(I)*Qubit('10')})
M_sympy_dict = MappingGate(d)
```

In [24]:
```
M_sympy_dict.args
```

Out[24]: $\left(\left\{ |00\rangle : \imath|11\rangle, \quad |01\rangle : e^{\imath}|10\rangle, \quad |10\rangle : e^{-\imath}|01\rangle, \quad |11\rangle : -\imath|00\rangle \right\}\right)$

In [25]:
```
M_sympy_dict.rewrite('op')
```

Out[25]: $-\imath|00\rangle\langle 11| + e^{-\imath}|01\rangle\langle 10| + e^{\imath}|10\rangle\langle 01| + \imath|11\rangle\langle 00|$

In [26]:
```
represent(M_sympy_dict)
```

Out[26]:

$$\begin{bmatrix} 0 & 0 & 0 & -\imath \\ 0 & 0 & e^{-\imath} & 0 \\ 0 & e^{\imath} & 0 & 0 \\ \imath & 0 & 0 & 0 \end{bmatrix}$$

Examples

Create an arbitrary qubit mapping and pass it to MappingGate, then check some of its properties

In [27]:
```
M = MappingGate(('000', -1, '001'), ('010', I, '101'), ('100', I*exp(I), '111'))
```

Check the arguments and outer product representation make sure there are 8 terms.

In [28]:
```
M.args
```

Out[28]: $\left(\left\{ |000\rangle : -|001\rangle, \quad |001\rangle : -|000\rangle, \quad |010\rangle : \imath|101\rangle, \quad |100\rangle : \imath e^{\imath}|111\rangle, \quad |101\rangle : -\imath|010\rangle, \quad |111\rangle : -\imath\right.\right.$

In [29]:
```
M.rewrite('op')
```

Out[29]: $|011\rangle\langle 011| + |110\rangle\langle 110| - |000\rangle\langle 001| - |001\rangle\langle 000| - \imath|010\rangle\langle 101| - \imath e^{-\imath}|100\rangle\langle 111| + \imath|101\rangle\langle 010| + \imath e^{\imath}|1$

Using hilbert_space checks the hilbert space of the gate.

In [30]:
```
M.hilbert_space
```

Out[30]: $\mathcal{C}^{2^{\otimes 3}}$

There are three ways to get the final state from the initial state: get_final_state, mapping, and qapply. get_final_state takes either strings or qubits where both mapping and qapply require qubits.

In [31]:
```
M.get_final_state('100')
```

Out[31]: $\imath e^{\imath}|111\rangle$

In [32]:
```
M.mapping[Qubit('100')]
```

Out[32]: $\imath e^{\imath}|111\rangle$

In [33]:
```
qapply(M*Qubit('100'))
```

Out[33]: $\imath e^{\imath}|111\rangle$

The MappingGate can also act on individual qubit states or multiple qubit states.

In [34]:
```
q1 = Qubit('000')
q2 = Qubit('110') + Qubit('100')
q3 = Qubit('101') + Qubit('000') + Qubit('100')
```

In [35]:
```
qapply(M*q1)
```

Out[35]: $-|001\rangle$

In [36]:
```
qapply(M*q2)
```

Out[36]: $|110\rangle + \imath e^{\imath}|111\rangle$

In [37]:
```
qapply(M*q3)
```

Out[37]: $-|001\rangle - \imath|010\rangle + \imath e^{\imath}|111\rangle$

There are three formats of the matrix representation that can be used, sympy-default, numpy, and scipy.sparse. For large matrices (i.e. large number of qubits) it is common to use the scipy.sparse format.

In [38]:
```
represent(M, format='sympy')
```

Out[38]:

$$\begin{bmatrix} 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\imath & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\imath e^{-\imath} \\ 0 & 0 & \imath & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \imath e^{\imath} & 0 & 0 & 0 \end{bmatrix}$$

In [39]:
```
represent(M, format='numpy')
```

Out[39]:
```
[[ 0.00000000+0.j          -1.00000000+0.j
0.00000000+0.j
   0.00000000+0.j           0.00000000+0.j
0.00000000+0.j
           0.00000000+0.j           0.00000000+0.j
]
 [-1.00000000+0.j           0.00000000+0.j
0.00000000+0.j
   0.00000000+0.j           0.00000000+0.j
0.00000000+0.j
           0.00000000+0.j           0.00000000+0.j
]
 [ 0.00000000+0.j           0.00000000+0.j
0.00000000+0.j
   0.00000000+0.j           0.00000000+0.j
0.00000000-1.j
           0.00000000+0.j           0.00000000+0.j
]
 [ 0.00000000+0.j           0.00000000+0.j
0.00000000+0.j
   1.00000000+0.j           0.00000000+0.j
0.00000000+0.j
           0.00000000+0.j           0.00000000+0.j
]
 [ 0.00000000+0.j           0.00000000+0.j
0.00000000+0.j
   0.00000000+0.j           0.00000000+0.j
0.00000000+0.j
           0.00000000+0.j
-0.84147098-0.54030231j]
 [ 0.00000000+0.j           0.00000000+0.j
0.00000000+1.j
   0.00000000+0.j           0.00000000+0.j
0.00000000+0.j
           0.00000000+0.j           0.00000000+0.j
]
 [ 0.00000000+0.j           0.00000000+0.j
0.00000000+0.j
   0.00000000+0.j           0.00000000+0.j
0.00000000+0.j
           1.00000000+0.j           0.00000000+0.j
]
 [ 0.00000000+0.j           0.00000000+0.j
0.00000000+0.j
   0.00000000+0.j          -0.84147098+0.54030231j
0.00000000+0.j
          0.00000000+0.j           0.00000000+0.j
]]
```

In [40]:
```python
represent(M, format='scipy.sparse')
```

Out[40]:
```
                  (0, 1.0) (-1+0j)
                  (1, 0.0) (-1+0j)
                    (2, 5.0)
-1j
                   (3, 3)   (1+0j)
  (4, 7.0)         (-0.841470984808-0.540302305868j)
                    (5, 2.0)
1j
                   (6, 6)   (1+0j)
  (7, 4.0)         (-0.841470984808+0.540302305868j)
```

Example with nqubits = 5

The matrix representation of a quantum gate is a 2^n by 2^n matrix, so even for relatively small states there is a lot of data stored. An example with states of 5 qubits should use scipy.sparse representation rather than the default sympy representation.

In [41]:
```python
M_large = MappingGate(('00000', I, '00001'), ('00010', '11111'))
```

In [42]:
```python
represent(M_large)
```

Out[42]:

$$\begin{bmatrix}
0 & -i & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
i & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Notice that essentially the entire matrix is zeros with ones along the diagonal. Using the scipy.sparse format here will condense the result.

In [43]:
```python
represent(M_large, format='scipy.sparse')
```

Out[43]:
```
      (0, 1.0)	-1j
      (1, 0.0)	1j
    (2, 31.0)	(1+0j)
    (3, 3)
(1+0j)
    (4, 4)
(1+0j)
    (5, 5)
(1+0j)
    (6, 6)
(1+0j)
    (7, 7)
(1+0j)
    (8, 8)
(1+0j)
    (9, 9)
(1+0j)
    (10, 10)	(1+0j)
    (11, 11)	(1+0j)
    (12, 12)	(1+0j)
    (13, 13)	(1+0j)
    (14, 14)	(1+0j)
    (15, 15)	(1+0j)
    (16, 16)	(1+0j)
    (17, 17)	(1+0j)
    (18, 18)	(1+0j)
    (19, 19)	(1+0j)
    (20, 20)	(1+0j)
    (21, 21)	(1+0j)
    (22, 22)	(1+0j)
    (23, 23)	(1+0j)
    (24, 24)	(1+0j)
    (25, 25)	(1+0j)
    (26, 26)	(1+0j)
    (27, 27)	(1+0j)
    (28, 28)	(1+0j)
    (29, 29)	(1+0j)
    (30, 30)	(1+0j)
    (31, 2.0)	(1+0j)
```

Creating Quantum Gates

MappingGate can be used to create any of the common quantum gates by specifying the same mappings. Let's create the ZGate, XGate, and YGate.

ZGate

In [44]:
```python
from sympy.physics.quantum.gate import ZGate
```

In [45]:
```python
Z = ZGate(0)
M_Z = MappingGate(('0', '0'), ('1', -1, '1'))
```

In [46]:
```
represent(Z, nqubits=1)
```

Out[46]: $\begin{bmatrix}1 & 0\\0 & -1\end{bmatrix}$

In [47]:
```
represent(M_Z)
```

Out[47]: $\begin{bmatrix}1 & 0\\0 & -1\end{bmatrix}$

In [48]:
```
represent(Z, nqubits=1) == represent(M_Z)
```

Out[48]: $True$

XGate

In [49]:
```
from sympy.physics.quantum.gate import XGate
```

In [50]:
```
X = XGate(0)
M_X = MappingGate(('0', '1'))
```

In [51]:
```
represent(X, nqubits=1)
```

Out[51]: $\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$

In [52]:
```
represent(M_X)
```

Out[52]: $\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$

In [53]:
```
represent(X, nqubits=1) == represent(M_X)
```

Out[53]: $True$

In [54]:
```
from sympy.physics.quantum.gate import YGate
```

In [55]:
```
Y = YGate(0)
M_Y = MappingGate(('0', I, '1'))
```

In [56]:
```
represent(Y, nqubits=1)
```

Out[56]: $\begin{bmatrix}0 & -\imath\\\imath & 0\end{bmatrix}$

In [57]:
```
represent(M_Y)
```

Out[57]:
$$\begin{bmatrix} 0 & -\imath \\ \imath & 0 \end{bmatrix}$$

In [58]:
```
represent(Y, nqubits=1) == represent(M_Y)
```

Out[58]: *True*

For Additional Quantum Gate Information

Griffiths, David J. Introduction to Quantum Mechanics. Upper Saddle River, NJ: Pearson Prentice Hall, 2005. Print.

http://en.wikipedia.org/wiki/Quantum_gate

In [ ]:
```
```

```
"""Mapping Quantum Gates

TODO:
- Enable sparse mappings for large numbers of qubits
"""

from sympy import Integer, conjugate, Add, Mul
from sympy.core.containers import Dict
from sympy.physics.quantum import Dagger
from sympy.physics.quantum.gate import Gate
from sympy.physics.quantum.qubit import Qubit, IntQubit
from sympy.physics.quantum.matrixutils import matrix_eye
from sympy.physics.quantum.qexpr import split_qexpr_parts
from sympy.physics.quantum.hilbert import ComplexSpace

#--------------------------------------------------------------------------


class MappingGate(Gate):
    """

    Parameters
    ==========

    args : tuple, dict

        arg[0] = initial state
        arg[1] = scalar or final state
        arg[2] = None or final state

        The list of initial state and final state pairs. The final states a
        multiplied by some scalar. If no scalar is given, 1 is assumed. Onl
        supply the qubit mapping for half of the matrix representation of t
        Since a quantum gate is required to be unitary, the other half is c
        to ensure it is unitary. Can pass either a python or sympy dictiona
        already has the qubit mappings.

    Examples
    ========

    Creating a Mapping Gate and checking its arguments and properties. Gett
    state from initial state.

        >>> from sympy.physics.quantum.mappinggate import MappingGate
        >>> from sympy.physics.quantum.qubit import Qubit
        >>> from sympy import I
        >>> M = MappingGate(('00',I,'11'), ('01','10'), ('10','01'), ('11',
        >>> M.args
        ({|00>: I*|11>, |01>: |10>, |10>: |01>, |11>: -I*|00>},)
        >>> M.nqubits
        2
        >>> M.mapping[Qubit('00')]
```

```
        I*|11>
        >>> M.get_final_state('00')
        I*|11>

    Create a Mapping Gate by only giving half of the initial and final stat
    the resulting arguments are the same as the example above. Also passing
    or sympy dictionary to MappingGate can have the same result.

        >>> from sympy.physics.quantum.mappinggate import MappingGate
        >>> from sympy import I
        >>> M = MappingGate(('00', I, '11'), ('01', '10'))
        >>> M.args
        ({|00>: I*|11>, |01>: |10>, |10>: |01>, |11>: -I*|00>},)
        >>> d = dict({Qubit('00'):I*Qubit('11'), Qubit('01'):Qubit('10')})
        >>> M_dict = MappingGate(d)
        >>> M.args
        ({|00>: I*|11>, |01>: |10>, |10>: |01>, |11>: -I*|00>},)

    Using qapply on initial states returns the final states.

        >>> from sympy.physics.quantum.mappinggate import MappingGate
        >>> from sympy import I
        >>> from sympy.physics.quantum.qapply import qapply
        >>> from sympy.physics.quantum.qubit import Qubit
        >>> M = MappingGate(('00', I, '11'), ('01', '10'))
        >>> q = Qubit('00') + Qubit('01')
        >>> qapply(M*q)
        |10> + I*|11>

    The MappingGate can be rewritten as an outer product of states. We will
    examples: one where all four states are given and one where only one st
    given. If not all initial states are specified they return themselves a
    states.

        >>> from sympy.physics.quantum.mappinggate import MappingGate
        >>> from sympy import I
        >>> M = MappingGate(('00', I, '11'), ('01', '10'))
        >>> M.rewrite('op')
        |01><10| + |10><01| - I*|00>*<11| + I*|11>*<00|
        >>> M = MappingGate(('00', -1, '00'))
        >>> M.rewrite('op')
        |01><01| + |10><10| + |11><11| - |00>*<00|

    The MappingGate is also expressed as a matrix where the rows and column
    represent the Qubits.

        >>> from sympy.physics.quantum.mappinggate import MappingGate
        >>> from sympy.physics.quantum.represent import represent
        >>> from sympy import I
        >>> M = MappingGate(('00', I, '11'), ('01', '10'))
        >>> represent(M)
        [0, 0, 0, -I]
```

```
        [0, 0, 1,  0]
        [0, 1, 0,  0]
        [I, 0, 0,  0]

    """

    @classmethod
    def _eval_args(cls, args):
        if len(args) == 1 and isinstance(args[0], (dict, Dict)):
            temp = {}
            for i, f in args[0].items():
                terms = split_qexpr_parts(f)
                if len(terms[1]) == 0:
                    temp[f] = i
                else:
                    temp[terms[1][0]] = conjugate(Mul(*terms[0]))*i
                temp[i] = f
            new_args = Dict(temp)
        else:
            temp = {}
            for arg in args:
                i = Qubit(arg[0])
                if len(arg) == 2:
                    scalar = Integer(1)
                    f = Qubit(arg[1])
                elif len(arg) == 3:
                    scalar = arg[1]
                    f = Qubit(arg[2])
                else:
                    raise ValueError('Too many scalar arguments')
                if i.nqubits != f.nqubits:
                    raise ValueError('Number of qubits for each state do nc
                temp[f] = conjugate(scalar)*i
                temp[i] = scalar*f
            new_args = Dict(temp)
        return (new_args,)

    @classmethod
    def _eval_hilbert_space(cls, args):
        return ComplexSpace(2)**args[0].keys()[0].nqubits

    @property
    def mapping(self):
        return self.args[0]

    @property
    def nqubits(self):
        """Gives the dimension of the matrix representation"""
        return self.args[0].keys()[0].nqubits

    def get_final_state(self, qubit):
        """Returns the final state for a given initial state, if initial st
```

```
        not mapped to a final state the initial state is returned."""
        i = Qubit(qubit)
        return self.mapping.get(i, i)

    def _apply_operator_Qubit(self, qubit):
        return self.get_final_state(qubit)

    def _eval_rewrite_as_op(self, *args):
        terms = []
        for i in range(2**self.nqubits):
            initial = Qubit(IntQubit(i, self.nqubits))
            fin = self.get_final_state(initial)
            terms.append(fin*Dagger(initial))
        return Add(*terms)

    def _represent_default_basis(self, **options):
        return self._represent_ZGate(None, **options)

    def _represent_ZGate(self, basis, **options):
        format = options.get('format','sympy')
        matrix = matrix_eye(2**self.nqubits, **options)
        for i, f in self.mapping.items():
            col = IntQubit(i).as_int()
            terms = split_qexpr_parts(f)
            if len(terms[1]) == 0:
                row = IntQubit(*terms[0]).as_int()
                scalar = Integer(1)
            else:
                row = IntQubit(*terms[1]).as_int()
                scalar = Mul(*terms[0])
            if format == 'scipy.sparse':
                matrix = matrix.tolil()
                col = float(col)
                row = float(row)
                scalar = complex(scalar)
                matrix[col, col] = 0.0
                matrix[row, col] = scalar
            elif format == 'numpy':
                scalar = complex(scalar)
                matrix[col, col] = 0.0
                matrix[row, col] = scalar
            else:
                matrix[col, col] = Integer(0)
                matrix[row, col] = scalar
        return matrix
```

```python
"""Tests for mappinggate.py"""

from sympy import I, Integer, Mul, Add
from sympy.physics.quantum import Dagger
from sympy.physics.quantum.qapply import qapply
from sympy.physics.quantum.represent import represent
from sympy.physics.quantum.qexpr import split_qexpr_parts
from sympy.physics.quantum.hilbert import ComplexSpace
from sympy.physics.quantum.qubit import Qubit, IntQubit
from sympy.physics.quantum.mappinggate import MappingGate
from sympy.external import import_module
from sympy.utilities.pytest import skip

np = import_module('numpy', min_python_version=(2, 6))
scipy = import_module('scipy', __import__kwargs={'fromlist': ['sparse']})

# All 3 ways produce same Qubit Mappings
M = MappingGate(('00', I, '11'), ('01', '10'), ('10', '01'), ('11', -I,
'00'))
M_half = MappingGate(('00', I, '11'), ('01', '10'))
d = dict({Qubit('00'):I*Qubit('11'), Qubit('01'):Qubit('10')})
M_dict = MappingGate(d)

M_rep = represent(M, format='sympy')

def test_MappingGate():
    assert M.get_final_state('00') == I*Qubit('11')
    assert M.mapping[Qubit('00')] == I*Qubit('11')
    assert qapply(M*Qubit('01')) == Qubit('10')
    assert M.hilbert_space == ComplexSpace(2)**M.nqubits
    # Shows same qubit mappings
    assert M.args == M_half.args
    assert M.args == M_dict.args

    terms = []
    for i in range(2**M.nqubits):
        initial = Qubit(IntQubit(i, M.nqubits))
        fin = M.get_final_state(initial)
        terms.append(fin*Dagger(initial))
    result = Add(*terms)
    assert M.rewrite('op') == result

    for i, f in M.mapping.items():
        col = IntQubit(i).as_int()
        terms = split_qexpr_parts(f)
        if len(terms[1]) == 0:
            row = IntQubit(*terms[0]).as_int()
            scalar = Integer(1)
        else:
            row = IntQubit(*terms[1]).as_int()
            scalar = Mul(*terms[0])
        assert M_rep[row, col] == scalar
```

```python

    if not np:
        skip("numpy not installed or Python too old.")

    M_rep_numpy = represent(M, format='numpy')
    for i, f in M.mapping.items():
        col = IntQubit(i).as_int()
        terms = split_qexpr_parts(f)
        if len(terms[1]) == 0:
            row = IntQubit(*terms[0]).as_int()
            scalar = Integer(1)
        else:
            row = IntQubit(*terms[1]).as_int()
            scalar = Mul(*terms[0])
        assert M_rep_numpy[row, col] == complex(scalar)

    if not np:
        skip("numpy not installed or Python too old.")
    if not scipy:
        skip("scipy not installed.")
    else:
        sparse = scipy.sparse

    M_rep_scipy = represent(M, format='scipy.sparse')
    for i, f in M.mapping.items():
        col = IntQubit(i).as_int()
        terms = split_qexpr_parts(f)
        if len(terms[1]) == 0:
            row = IntQubit(*terms[0]).as_int()
            scalar = Integer(1)
        else:
            row = IntQubit(*terms[1]).as_int()
            scalar = Mul(*terms[0])
        col = float(col)
        row = float(row)
        scalar = complex(scalar)
        assert M_rep_scipy[row, col] == scalar

```

Links

The following are the addresses of the two Quantum projects on the GitHub website in the SymPy directory.

Quantum Simple Harmonic Oscillator
https://github.com/sympy/sympy/blob/master/sympy/physics/quantum/sho1d.py

Quantum Mapping Gate
https://github.com/sympy/sympy/blob/master/sympy/physics/quantum/mappinggate.py

References

"GitHub." <http://en.wikipedia.org/wiki/GitHub>.

Griffiths, David J. Introduction to Quantum Mechanics. Upper Saddle River, NJ: Pearson Prentice Hall, 2005. Print.

"SymPy." SymPy. Web. 15 Mar. 2013. <http://sympy.org/en/index.html>.