Quantum Mechanics, Quantum Computation, and the Density Operator in SymPy

Addison Cugini

06/12/2011

Abstract

Because aspects of quantum mechanics are both difficult to understand and difficult algebraically, there is a need for software which symbolically simulates quantum mechanical phenomena. To accomplish this goal, code has been added to an open-source computer algebra system, called SymPy, which models the abstraction of Dirac notation and the density operator. Additionally, a quantum computer simulation has been built using this abstraction. This paper shall discuss the code that has been added as well as any relevant physics important to understanding the code. In particular, we shall focus on the density operator of statistical quantum mechanics using SymPy's density matrices as a guide.

1 Introduction

Students unfamiliar with quantum mechanics oftentimes find themselves getting lost in arithmetic and algebraic details and consequently have a hard time grasping the fundamental concepts behind the theory. For this reason, there is a need for software which simplifies the learning process by doing much of the meticulous algebra for the user. There is no doubt that a symbolic engine would be highly useful for attaining this goal. With this in mind, we have added code to an open source project called SymPy, which extends the Python programming language to handle symbolic computation. The code we have added includes both a base layer which implements abstract Dirac notation in SymPy, as well as specific sub-modules which model specific quantum systems.

This paper will discuss the basic quantum mechanics simulation code that was added to the Computer Algebra System (CAS) SymPy during the 2010 Google Summer of Code by Matt Curry and Dr Brian Granger. This includes a general introduction to the syntax and data structures of SymPy as well as a look at how this was used to implement Dirac notation. The code provides a notion of states, operators, commutators, anticommutators and many other quantum mechanical structures. These structures can be represented in a basis, as well as simplified using a function called `qapply`. In this way, the base layer can be used to create simulations of many quantum mechanical systems.

One such system which was implemented is a quantum computer. We shall look at how Dirac Notation was used to simulate the gate model of quantum computation. The quantum

computing module inherits much of its capabilities from the main quantum base layer. We shall look at how this occurs and what capabilities the module currently has.

Finally, we shall discuss the density operator of statistical quantum mechanics and how we recently extended the code to model its use. The code (and discussion of its meaning) was developed during Addison Cugini's senior project. We shall include a discussion of the important physics concepts which come from the study of the density operator. This includes a discussion of the Von Neumann entropy and how it relates to the Louiville Theorem. Additionally, we shall look at how the reduced density matrix of a subsystem is calculated and use this knowledge to inform our understanding of quantum decoherence and the "entropy of entanglement" measure of entanglement.

2 Python and SymPy

Python is a high level programming language that aims to be easy to use and easy to learn. To further this aim, the language supports multiple programming paradigms including functional, imperative and object oriented; both functions and classes are considered first class citizens. In contrast to Perl's credo "There is more than one way to do it", The Zen of Python asserts "There should be one–and preferably only one–obvious way to do it". In this way, Python aims to be easy to read and learn by providing only one obvious way to solve a particular task (Perl is infamous for requiring people to learn too many programming techniques).

Keeping with its simplicity, Python is an interactive, interpreted language, which means that code does not need to be compiled and can be typed directly into an interpreter. In addition to the basic command line Python interpreter, a third-party open-source package called IPython adds shell functionality. This includes basic functionality, such as changing directories, running files, listing files in a directory, as well as more advanced features such as tab completion and command history. The bleeding edge versions of IPython allow for rendering of the common typesetting languages LaTeX and HTML. These features create an environment that makes it easy to test code. All code examples contained in this document were created using this version of IPython.

Python has several modules which provide functionality that is useful for mathematics and scientific computing. Fast numerical algorithms on matrices and arrays are supplied by the Scipy and Numpy modules, which have run times nearly as fast as hand-written C/C++ code. These modules are competitors to Matlab with the advantage of using a much more manageable and scalable programming language. SymPy is a computer algebra system which hopes to compete with well established symbolic engines such as Mathematica, Maple or Maxima. Features include support for algebraic simplifications, ODE solving, integration (the Risch Algorithm is partially implemented), linear algebra, LaTeX printing of expressions and more.

What follows is a short demonstration of the general symbolic capabilities of SymPy rendered in the bleeding edge version of IPython. All code was attained by screen-grabbing the input-output of the IPython notebook; there is no magic and no trickery, the notebook actually renders LaTeX code. This example is done so that the reader can understand the clear syntax SymPy provides.

General CAS capabilities and syntax example in Sympy

Expressions are fully symbolic
(Operator overloading has been used to allow Sympy objects to be subjected to binary ops)

```
In [19]: expression = (x**2+y**2)/x; expression
```

Out[19]: $\frac{x^2 + y^2}{x}$

Capable of taking derivatives and integrals of expressions

```
In [20]: derivative = diff(expression, x); derivative
```

Out[20]: $2 - \frac{x^2 + y^2}{x^2}$

```
In [21]: integrate(derivative, x)
```

Out[21]: $x + \frac{y^2}{x}$

The code above shows the creation of an algebraic expression, $\frac{x^2+y^2}{x}$, using standard Python operators for division, addition, and powers. The code allows for taking derivatives of expressions using the `diff` function, which takes an algebraic expression as its first argument and the second argument is the variable in which the derivative was taken. The `integrate` function takes the same arguments and returns the integral of the input expression.

2.1 Data Structures in SymPy

SymPy's symbolic engine takes advantage of object orientation (inheritance in particular) to create an easily extensible code base. All classes derive features, such as the ability to compare itself with other objects, from methods in the `Basic` super-class. Objects which can be acted upon by algebraic operations gain this ability through a set of methods in a class called `Expr`. These `Expr` objects can be held in container objects (which also subclass `Expr`) such as `Mul`, `Add` and `Pow`; the container objects are instantiated using Python's operator overloading feature which allows the constructor of the container class to be called whenever the appropriate binary operator is used (`*` for `Mul`, `+` for `Add`, and `**` for `Pow`).

In this way, additional objects can be added by simply creating a subclass which inherits features from the `Expr` class. These subclasses get for free certain features such as the ability to be compared, multiplied, added, etc. Herein is how SymPy creates a maintainable, modular and therefore easy to extend environment. With the ability to inherit properties from higher classes, the amount of code required to develop, for example, a system which models quantum mechanics and Dirac notation decreases significantly.

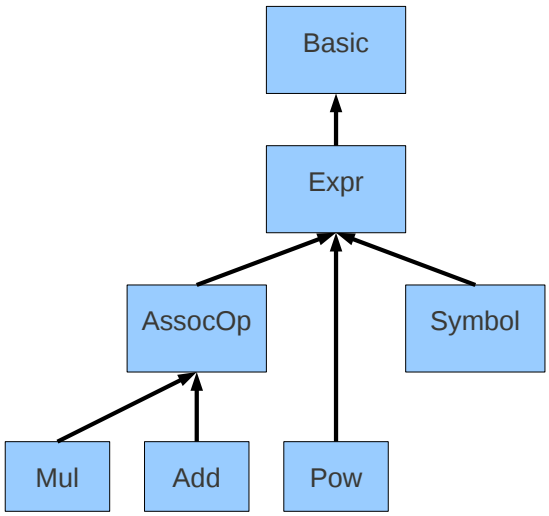


Figure 1: SymPy's inheritance diagram including some of the important classes.

3 General Quantum Mechanics in SymPy

SymPy has a base module which provides functionality for symbolic quantum mechanics in Dirac notation. Much of this functionality was added during the 2010 Google Summer of Code by Matt Curry and his advisor Dr. Brian Granger, with some help/input from Addison Cugini. The code has a notion of ket and bra states, the operators which act on these states, the Hilbert spaces to which states and operators belong, as well as inner, outer and tensor products. Representation of the these objects is done through the use of a `represent` function which goes through a `Mul`, `Add`, or `Pow` of expressions and represents them in the basis given as an input argument; if no basis is given, then the code uses a default basis. Code also exists within a function called `qapply` which can apply operators to states as well as evaluate inner products based on orthogonality rules. Using these bits of code as a base, a variety of quantum systems have been implemented including: Hilbert space rotations, second quantization, continuous Cartesian systems and quantum computation with more systems being added. In this way, the code has the ability to represent quantum systems in Dirac notation as well as preform many elementary operations on these quantum objects.

The base class for this module is `QExpr` which inherits from `Expr` and provides some features unique to quantum mechanics (such as Hilbert spaces and the ability to be represented). These abilities are inherited by all subclasses.

The following is a code example, which shows some of the capabilities of the module:

General Quantum

We can produce arbitrary states with no reference to what system it represents

```
In [7]:  superposition = Ket('psi')/sqrt(2) + Ket('phi')/sqrt(2); superposition
```

Out[7]: $\frac{1}{2}\sqrt{2}|\phi\rangle + \frac{1}{2}\sqrt{2}|\psi\rangle$

We can multiply states together, take their dagger and their inner product

```
In [10]: inner_prod = Dagger(superposition)*superposition; inner_prod
```

Out[10]: $\left(\frac{1}{2}\sqrt{2}\langle\phi| + \frac{1}{2}\sqrt{2}\langle\psi|\right)\left(\frac{1}{2}\sqrt{2}|\phi\rangle + \frac{1}{2}\sqrt{2}|\psi\rangle\right)$

(note that the code makes no assumptions about orthonormality)

```
In [12]: qapply(inner_prod)
```

Out[12]: $\frac{1}{2}\langle\phi|\phi\rangle + \frac{1}{2}\langle\phi|\psi\rangle + \frac{1}{2}\langle\psi|\phi\rangle + \frac{1}{2}\langle\psi|\psi\rangle$

We can also form outer products of states with their daggers

```
In [14]: density = qapply(superposition*Dagger(superposition)); density
```

Out[14]: $\frac{1}{2}|\phi\rangle\langle\phi| + \frac{1}{2}|\phi\rangle\langle\psi| + \frac{1}{2}|\psi\rangle\langle\phi| + \frac{1}{2}|\psi\rangle\langle\psi|$

The above code produces an arbitrary superposition of `Ket` states by forming an algebraic expression of `Ket` objects. The `Ket` constructor takes as its argument the label that will identify the state. The dagger of this state is formed by calling the `Dagger` class's constructor which distributes the dagger operation on each ket, thus producing the superposition of bra states that we see. We use `qapply` to form the inner and outer products that we see in the last two outputs.

4 Quantum Computation in SymPy

Quantum computation is an emerging field which promises to exploit quantum effects (such as superpositions, entanglement, and interference) to create algorithms with faster asymptotic run times than what is possible with known classical algorithms. The field is of interest to individuals across many fields including mathematics, computer science and physics.

There is a need to provide software packages which simulate quantum computers for the purpose of teaching and research. For this reason, the general quantum package has been extended to include a quantum computer simulator. This includes a notion of `Qubit` objects (which are the states of interest to quantum computing), and the `Gate` objects which operate on the states. Many of the most well known quantum algorithms have been implemented using this extension including the quantum Fourier transform, Shor's order-finding algorithm, and Grover's search algorithm. With these example algorithms, someone new

to the field can quickly get a sense for how a quantum computer would exploit quantum effects to produce speed-ups over known classical algorithms. Additionally, researchers may be able to use the software to test new algorithms and gain intuition for how the algorithm functions.

As stated earlier, qubit objects are the state of interest for a quantum computer. In SymPy, these states are implemented by inheriting characteristics from the `State` superclass and then teaching the qubit object how to print and represent itself as a vector. `Gate` objects are implemented by inheriting properties from `Operator` and then teaching the gate class how to print, represent itself as a matrix, and apply itself to a qubit. Specific examples of quantum gates (e.g. `H`, `X`, `CNOT`, etc) are implemented by subclassing the generic `Gate` class and defining an associated matrix. The 'qapply' function works by looking at this matrix and applying it to a state without first representing the gate as a matrix in its full basis.

We can create a qubit state by instantiating the Qubit class

```
In [26]: qubit_state = Qubit('00'); qubit_state
```

Out[26]: $|00\rangle$

We can see that the qubit_state object is indeed an instance of a state.

```
In [27]: isinstance(qubit_state, State)
```

```
Out[27]: True
```

```
In [28]: gate_operator = HadamardGate(0); gate_operator
```

Out[28]: H_0

We can see that gate_operator is indeed an instance of a operator

```
In [29]: isinstance(gate_operator, Operator)
```

```
Out[29]: True
```

We can add together states to produce a superposition

```
In [30]: superposition = (Qubit('00') + Qubit('11'))/sqrt(2); superposition
```

Out[30]: $\frac{1}{2}\sqrt{2}(|00\rangle + |11\rangle)$

Likewise, we can have multiple gates operating in sequence

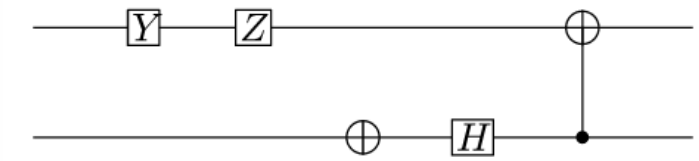
```
In [31]: circuit = CNOT(0,1)*H(0)*X(0)*Z(1)*Y(1); circuit
```

Out[31]: $CNOT_{0,1} H_0 X_0 Z_1 Y_1$

We can plot the sequence of gates using the circuit_plot function

```
In [32]: circuit_plot(circuit, nqubits=2)
```

```
Out[32]: <sympy.physics.quantum.circuitplot.CircuitPlot at 0x4442e90>
```



We can represent this gate in the default z-basis

```
In [33]: represent(gate_operator, nqubits=1)
```

Out[33]: $\begin{pmatrix} \frac{1}{2}\sqrt{2} & \frac{1}{2}\sqrt{2} \\ \frac{1}{2}\sqrt{2} & -\frac{1}{2}\sqrt{2} \end{pmatrix}$

We can apply the HadamardGate to the qubit state

```
In [34]: qapply(gate_operator*qubit_state)
```

Out[34]: $\frac{1}{2}\sqrt{2}|00\rangle + \frac{1}{2}\sqrt{2}|01\rangle$

We can also intermix easy to read Python code with Sympy objects
We can create a superposition of states using a 'for-loop'
(Note the way that we can instantiate the qubit class with integers)

```
In [35]: superposition = 0
         for i in xrange(8):
             superposition += (-1)**i*IntQubit(i,3)
         superposition
```

Out[35]: $-|1\rangle - |3\rangle - |5\rangle - |7\rangle + |0\rangle + |2\rangle + |4\rangle + |6\rangle$

The code above shows Sympy's general quantum computational capabilities. As in the general quantum example, we create Qubit states and Qubit superpositions as well as Gate operators. We can see from the results of Python's 'isinstance' function that Qubit and Gate are indeed subclasses of State and Operator respectively. In the case of the Qubit class, the argument passed to the constructor determines in what state the individual qubits are; Qubit('00') produces a ket where both the 0-th and 1st bit are set to zero. By multiplying a sequence of gates together, we produce a quantum circuit. This circuit is printed using the 'circuit_plot' function, which takes a quantum circuit as the first argument and the number of qubits that it acts on as the second. Likewise, we represent the circuit as a matrix using the 'represent' function which takes the same arguments as the 'circuit_plot' function. We then use of 'qapply' to apply a Hadamard gate to the state $|00\rangle$ which produces a superposition of states. We then use the for-loop syntax to produce a superposition with alternating phases on odd and even kets. The IntQubit class, which is a subclass of Qubit, displays itself in decimal form rather than binary.

4.1 The Quantum Fourier Transform

The quantum Fourier transform (QFT) is a key part of Shor's number factoring algorithm, which provides an exponential speedup for the problem of finding the order of a number. The QFT acts exactly like a discrete Fourier transform except that it does not apply to an incoming array of data, but the state vector of a given system. There is a known short sequence of quantum gates (whose length grows polynomially in the number of qubits it applies to) which implements this transformation. To get a sense of how this works, it would be helpful to see the SymPy module in action:

```
In [13]: QFT(0,3).decompose()
```

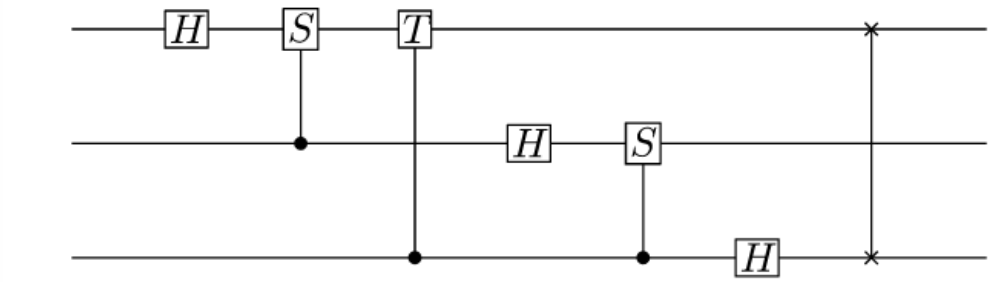
Out[13]: $SWAP_{0,2}H_0C_0(S_1)H_1C_0(T_2)C_1(S_2)H_2$

```
In [14]: IQFT(0,3).decompose()
```

Out[14]: $H_2C_1\left(R_{2,-2}\right)C_0\left(R_{2,-3}\right)H_1C_0\left(R_{1,-2}\right)H_0SWAP_{0,2}$

```
In [15]: CircuitPlot(QFT(0,3).decompose(), nqubits=3)
```

Out[15]: <sympy.physics.quantum.circuitplot.CircuitPlot at 0x3da2d90>




```
In [16]: represent(Fourier(0,3)*4/sqrt(2), nqubits=3)
```

Out[16]:
$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega \end{pmatrix}$$

The `QFT` class is called with the first argument of the `QFT` class being the first bit it applies to and the second argument is the number of bits to which it applies. The `QFT` class and `IQFT` classes are created which act on the 0th through 2nd bit, and then the decompose method is called which displays the transform in terms of elementary gates.

To get an idea of the relationship between the QFT and the DFT,
let's use the QFT in a way that someone knowledgeable in Fourier transforms would understand.
In this case, let's apply the QFT to a state which oscillates 4 times across the domain

```
In [18]: periodic_state = (IntQubit(0,3)-IntQubit(1,3)+IntQubit(2,3)-IntQubit(3,3)
          + IntQubit(4,3) - IntQubit(5,3) + IntQubit(6,3) - IntQubit(7,3))/(2*sqrt(2));
         periodic_state
```

Out[18]: $$\frac{1}{4}\sqrt{2}(-|1\rangle - |3\rangle - |5\rangle - |7\rangle + |0\rangle + |2\rangle + |4\rangle + |6\rangle)$$

Let's see that this state does have the expect vector representation

```
In [19]: represent(periodic_state, nqubits=3)
```

Out[19]: $$\begin{pmatrix} \frac{1}{4}\sqrt{2} \\ -\frac{1}{4}\sqrt{2} \\ \frac{1}{4}\sqrt{2} \\ -\frac{1}{4}\sqrt{2} \\ \frac{1}{4}\sqrt{2} \\ -\frac{1}{4}\sqrt{2} \\ \frac{1}{4}\sqrt{2} \\ -\frac{1}{4}\sqrt{2} \end{pmatrix}$$

Applying the QFT to this state returns the expected frequency of 4

```
In [20]: qapply(QFT(0,3).decompose()*periodic_state)
```

Out[20]: $|4\rangle$

The above code shows us actually applying the QFT to a state. In particular, we apply the QFT to three qubits which oscillates with a frequency of 4 cycles across the domain. One would expect that the frequency of this state would be 4; this indeed occurs when we apply the QFT to the state. In this way, we can see that a QFT is really not all that different from a standard DFT.

5 Density Operator Formalism in SymPy

In real situations found in the lab, an experimentalist cannot know with certainty what state a system is in. If the quantum state is not completely known, that is if there is a classical uncertainty as to what state a system is in, the density operator (or density matrix) formalism more naturally describes the system. In this way, the density operator is useful for modeling systems where both classical uncertainty and quantum superpositions are present. This is useful for real systems in which the whole history of a system may be impossible to account for. Despite the clear use of density matrices, undergraduate students of quantum mechanics tend to focus on how a well known state vector of a system evolves. For this reason, it is useful to add code which would model the density matrix.

5.1 Density Operator Defined

If a system is in a state $|\psi_n\rangle$ with a certain classical probability W_n, such that

$$\sum_n W_n = 1$$

(That is the system is normalized such that it must be in one of the ensemble of given states). Then the density operator which defines the system is defined by:

$$\rho = \sum_n W_n |\psi_n\rangle\langle\psi_n|.$$

Note that the states $|\psi_n\rangle$ need not be orthogonal. In a discrete basis with basis set $|\phi_1\rangle$, $|\phi_2\rangle$, ...,$|\phi_m\rangle$, we can define $a_i^n = \langle\phi_i|\psi_n\rangle$ and $a_j^n = \langle\phi_j|\psi_n\rangle$. The density operator takes the form of a matrix where the i-th and j-th matrix element is determined by the equation:

$$\langle\phi_i|\rho|\phi_j\rangle = \sum_n W_n a_i^n a_j^{n*}$$

(Blum, page 43)

5.2 Density Operators in SymPy

In SymPy, the density operator is implemented using the class `Density`, a subclass of `QExpr`. The Density class provides some functionality for any quantum mechanical system, but has some extra features which only work for Qubit objects.

The Density class constructor takes in a state and it's associated classical probability
These states can be completely general, or belong to a particular system

```
In [2]: Density([Ket('psi'), .5], [Ket('phi'), .5])
```

Out[2]: $0.5|\psi\rangle\langle\psi| + 0.5|\phi\rangle\langle\phi|$

From now on, we will look at the density matrix associated with tensor products of two state systems.
We can create a superposition of states and see the matrix representation using the represent function

```
In [14]: superposition_state = Density([(Qubit('00')+Qubit('01')+Qubit('10')+Qubit('11'))/2, 1]);
         superposition_state
```

Out[14]: $\frac{1}{4}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)(\langle 00| + \langle 01| + \langle 10| + \langle 11|)$

```
In [4]: superposition_matrix = represent(superposition_state,nqubits=2); superposition_matrix
```

Out[4]:
$$\begin{pmatrix} \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \end{pmatrix}$$

The matrix_to_density function allows the conversion of a Hermetian matrix representing a density operator into a instance of Density class

```
In [5]: matrix_to_density(superposition_matrix)
```

Out[5]: $(|00\rangle + |01\rangle + |10\rangle + |11\rangle)(\langle 00| + \langle 01| + \langle 10| + \langle 11|)$

We form a `Density` object made up of general ket states. The constructor for `Density` takes in a sequence of lists containing a state with its associated statistical weight. We then represent the density matrix using the `represent` function which takes in the same arguments as before. The `matrix_to_density` function takes in a matrix representation of a density operator and returns the Dirac notation representation (more on how this is done can be found in the next section).

We can very easily see the difference between a classical mixture and a superposition of states by representing its density matrix

```
In [7]: classical_mixture = Density([Qubit('00'), .5], [Qubit('11'),.5]); classical_mixture
```

Out[7]: $0.5|00\rangle\langle 00| + 0.5|11\rangle\langle 11|$

```
In [8]: represent(classical_mixture, nqubits=2)
```

Out[8]:
$$\begin{pmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.5 \end{pmatrix}$$

We can operate on states using qapply just as we did with pure states

```
In [10]: hadamarded_state = qapply(classical_mixture.operate_on(HadamardGate(0)*HadamardGate(1))); hadamarded_state
```

Out[10]: $0.5\left(\frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle\right)\left(\frac{1}{2}\langle 00| + \frac{1}{2}\langle 01| + \frac{1}{2}\langle 10| + \frac{1}{2}\langle 11|\right) + 0.5\left(\frac{1}{2}|00\rangle + \frac{1}{2}|11\rangle - \frac{1}{2}|01\rangle - \frac{1}{2}|10\rangle\right)\left(\frac{1}{2}\langle 00| + \frac{1}{2}\langle 11| - \frac{1}{2}\langle 01| - \frac{1}{2}\langle 10|\right)$

```
In [11]: represent(hadamarded_state,nqubits=2)
```

Out[11]:
$$\begin{pmatrix} 0.25 & 0 & 0 & 0.25 \\ 0 & 0.25 & 0.25 & 0 \\ 0 & 0.25 & 0.25 & 0 \\ 0.25 & 0 & 0 & 0.25 \end{pmatrix}$$

We use the same set of functions on a mixed state as we did on the pure state. The Density matrix has an 'operate_on' method which applies a sequence of gates to the density operator.

5.3 Matrix to Dirac Notation for a Density Operator

As seen in the previous segment of code, it is possible to take in a density matrix which represents the state of a system and determine a representation as a series of outer products of states using the eigenvalues and eigenvectors of the matrix. For a density matrix, ρ, with eigenvalues λ_n and associated eigenvectors $|\psi_n\rangle$, the matrix can be represented as

$$\rho = \sum_n \lambda_n |\psi_n\rangle\langle\psi_n|.$$

For a non-pure state the Dirac notation representation is not unique, so the returned result might not be the same set of states as one may expect. In SymPy, this is implemented in a separate function "matrix_to_density". So far, this function only works on Density matrices composed of qubits, but the mathematics described for doing this is general for any discrete system.

5.4 Von Neumann Entropy

An important concept which can be easily determined from the density operator formalism is the entropy of a system. In rough terms, the entropy of a system measures the amount of uncertainty we have about what state the system is in.

The entropy of a system can be determined by taking the eigenvalues λ_n of the density operator ρ and preforming the operation:

$$S = -\sum_n \lambda_n log(\lambda_n).$$

It is easy to show (see appendix) that when the system is in a pure state, the only non-zero eigenvalue of the density matrix is $\lambda = 1$. Plugging this eigenvalue into our equation for the Von Neumann entropy, we see that a pure state has zero entropy. This matches with our intuition as a perfectly well known system (a pure state) should not have any entropy.

In SymPy, the Von Neumann entropy is implemented for discrete systems using a method within the Density class. This method determines the eigenvalues of the density matrix by representing the operator as a matrix and making calls to the SymPy function which determines eigenvalues. The eigenvalues are then plugged into the equation for the Von Neumann entropy. This can be seen in the following code example:

Von Neumann Entropy

We can call the entropy method of a pure state density matrix to find that it does not have any entropy, as expected.

```
In [16]: pure_state = Density([(Qubit('00')+Qubit('01')+Qubit('10')+Qubit('11'))/2, 1]); pure_state
```

Out[16]: $\frac{1}{4}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)(\langle 00| + \langle 01| + \langle 10| + \langle 11|)$

```
In [18]: round(pure_state.entropy(nqubits=2),3)
Out[18]: 0.0
```

A non-pure state system will have a non-zero entropy

```
In [19]: classical_mixture = Density([Qubit('00'), .5], [Qubit('11'),.5]); classical_mixture
```

Out[19]: $0.5|00\rangle\langle 00| + 0.5|11\rangle\langle 11|$

```
In [21]: classical_mixture.entropy(nqubits=2)
Out[21]: 0.693147180559945
```

In the above example, we calculate the entropy for both a pure and mixed state system using the density operator's entropy method. The pure state system has zero entropy and the mixed state has non-zero entropy as expected.

A student of classical statistical mechanics would no doubt be aware of the Louisville theorem which establishes that any system evolving under a linear Hamiltonian will maintain the same area of uncertainty in phase space. There is an equivalent form of this theorem for density operators which states that the uncertainty about the state of the system remains the same when evolving under unitary transformations. In other words, a pure state remains pure and a mixed state doesn't become any more or less mixed as it is operated on. One way this mathematically manifests itself is that the entropy of a state remains the same under unitary evolution. Unitary operations are isentropic. This is seen below:

Note that the entropy of a state does not change as you operate on in with linear gates
This is due to the Louiville equation for statistical mechanics (Blum 55-57)

```
In [23]: hadamarded_state = qapply(classical_mixture.operate_on(HadamardGate(0)*HadamardGate(1))); hadamarded_state
```

Out[23]: $0.5\left(\frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle\right)\left(\frac{1}{2}\langle 00| + \frac{1}{2}\langle 01| + \frac{1}{2}\langle 10| + \frac{1}{2}\langle 11|\right) + 0.5\left(\frac{1}{2}|00\rangle + \frac{1}{2}|11\rangle - \frac{1}{2}|01\rangle - \frac{1}{2}|10\rangle\right)\left(\frac{1}{2}\langle 00| + \frac{1}{2}\langle 11| - \frac{1}{2}\langle 01| - \frac{1}{2}\langle 10|\right)$

```
In [24]: hadamarded_state.entropy(nqubits=2)
Out[24]: 0.693147180559945
```

We can see that the entropy has remained the same after applying gates to a state.

5.5 The Reduced Density Matrix

Any student of quantum mechanics is aware of the definition of an entangled state as a state for which the constituent subsystems cannot be factored out. That is a state $|\Psi\rangle$ with constituent systems $|\theta\rangle$ and $|\phi\rangle$ is entangled (inseparable) if and only if it is impossible to write $|\psi\rangle = |\phi\rangle \otimes |\theta\rangle$. If this condition is not met, then the system is said to be in a inseparable or entangled state. For this entangled state, it is impossible to describe the entangled subsystem on its own without the loss of information. If however we wish to provide a statistical account of the state of a subsystem, the density operator provides a useful framework in the form of an object called the reduced density matrix.

Consider the arbitrary state $|\Phi\rangle$ with subsystems $|\phi\rangle$ and $|\theta\rangle$. Now, say that we only really care about the state of $|\phi\rangle$. We can describe the state of the subsystem by the reduced density ρ_r, where the elements of the matrix are determined by

$$\langle \theta\phi_j|\rho_r|\theta\phi_{j'}\rangle = \sum_i \langle \theta_i\phi_j|\rho_r|\theta_i\phi_{j'}\rangle.$$

Alternatively, we can describe the reduced density matrix in term of something called a partial trace

$$\rho_r = tr_\phi |\Phi\rangle\langle\Phi|.$$

The sum describing the reduced density matrix is doubtlessly a pain to calculate (determining the reduced density matrix for even a Bell state requires the addition of eight numbers, and it grows exponentially in the number of qubits). Fortunately, I have added code to the density operator module which can calculate the reduced density matrix of a tensor product of two-state systems (i.e. qubits) by specifying which bits are to be considered the unobserved bits (This method could be generalized for any discrete basis, but it would be more difficult to code as I could not exploit bit twiddling in the implementation). This can be seen below:

Reduced Density Matrix

```
In [2]: Bell_state = Density([(Qubit('00') + Qubit('11'))/sqrt(2), 1]); Bell_state
```

Out[2]: $0.5(|00\rangle + |11\rangle)(\langle 00| + \langle 11|)$

```
In [3]: represent(Bell_state, nqubits=2)
```

Out[3]:
$$\begin{pmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0.5 & 0 & 0 & 0.5 \end{pmatrix}$$

```
In [4]: round(Bell_state.entropy(nqubits=2),5)
```

Out[4]: 0.0

```
In [5]: reduced_Bell = reduced_density(Bell_state, 0, nqubits=2); reduced_Bell
```

Out[5]:
$$\begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}$$

```
In [7]: entropy(reduced_Bell, nqubits=2)
```

Out[7]: 0.693147180559945

The operation seen above is interesting in that we went irreversibly from a pure entangled Bell state with zero entropy to a mixed state with non-zero entropy. In this way, the partial trace of an entangled state can be thought of as us 'forgetting' information about one of the qubits and thus increasing the entropy. This will generally occur whenever a partial trace is preformed on two entangled states. It is thus apparent that the Louisville theorem discussed in the previous section does not generally apply to the partial trace operation as entropy is not conserved.

This shows us that a pure entangled state can appear to be a mixed unentangled state when we do not have information about one of the entangled systems. This concept sets up the framework for our understanding of decoherence. Decoherence is what occurs when a quantum superposition interacts with (and becomes entangled with) an unknown environment; this entanglement results in what appears to be the non-unitary transformation of quantum superpositions into classical mixtures. Mathematically, this manifests itself as a spontaneous diagonalization of the density matrix. Decoherence forms the classical/quantum barrier as this is where the weird effects of superposition and interference come to die. (the reader should note that coherence theory does not by itself solve the so called 'measurement problem' of quantum mechanics because measurement results in a definite state rather than a mixed state; we are still left with the question of why and when the quantum state collapsed into a pure definite state. For one interpretation of decoherence see Rolland Omnes, The Interpretation of Quantum Mechanics).

5.6 Entropy of Entanglement

A wise and observant reader may have noticed that there is a very strong correlation between the entropy of the reduced density matrix and the entanglement of the system. After all,

the maximally entangled Bell state also produced a mixed state of maximal entropy under the partial trace operation. This observation leads to the definition of an entanglement measure called the 'entropy of entanglement'. This measure defines the entanglement of two subsystems of the state$|\Psi\rangle$, A and B, to be the entropy of the partial trace of the system. Thus:

$$Entanglement(|\Psi\rangle) = S(tr_A|\Psi\rangle\langle\Psi|) = S(tr_B|\Psi\rangle\langle\Psi|)$$

We have thus added a small bit of functionality to the code which determines the entropy of entanglement between any specified qubit and the system as a whole. This is seen below:

Entropy of Entanglement

Here, we determine the entropy of entanglement between the 0th and 1st bit of the Bell State

```
In [8]: Bell_state = Density([(Qubit('00') + Qubit('11'))/sqrt(2), 1]); Bell_state
```

Out[8]: $0.5(|00\rangle + |11\rangle)(\langle 00| + \langle 11|)$

```
In [9]: entropy_of_entanglement(Bell_state, 1, nqubits=2)
```

Out[9]: 0.693147180559945

The input arguments to the `entropy_of_entanglement` are the same as those used for the `reduced_density` function. We can see that the entropy of entanglement for the Bell state is .69. Right now, the code only allows for the calculation of the entanglement for a two state system, but there is no fundamental reason that this could not be done for any discrete system.

6 Conclusions and Future Directions

SymPy's quantum code base so far has built into it a notion of Dirac notation, the gate model of quantum computation, and the density matrix. The quantum computational module has all of the structures important to the gate model of quantum computation, and as such can simulate quantum algorithms. The density matrix object has the capability to be represented, acted upon by operators, determine the entropy of a state, and determine the entropy of entanglement. In this way, the code is quite capable of modeling a wide range of systems.

We can see from the examples that the code greatly eases the algebraic and arithmetic difficulty in simulating quantum mechanical systems. The code therefore could be used as a very powerful teaching and research tool. The added code doubtlessly has potential for modeling certain quantum phenomena. However, because the code is in its infancy, very few simulations of physical systems have been written using this framework. When more examples using this code are written, the module will be of great use for teaching quantum mechanics and quantum information concepts. Additionally, researchers will be able to simulate quantum phenomena where symbolic manipulations are necessary. In this way, the tool will be useful for both educators and researchers.

References

[1] Blum, Karl. Density Matrix Theory and Applications. Plenum Press, New York, 1996.

[2] "Entropy of Entanglement", Quantiki. Web. Accessed 1 Jun 2011. http://www.quantiki.org/wiki/Entropy_of_entanglement.

[3] Nielsen, Micheal. Chuang, Isaac. Quantum Computation and Quantum Information. Cambridge University Press, New York, 1996.

[4] Mermin, David. Quantum Computer Science. Cambridge University Press, New York, 2007.

[5] Omnes, Rolland. The Interpretation of Quantum Mechanics. Princeton University Press, New Jersey, 1994.

Further information and downloads for SymPy can be found at SymPy.org

Appendix A

Simple proof that 1 is the only non-zero eigenvalue for a pure state:

Suppose we have a generic pure state density matrix defined by $\rho = |\psi\rangle\langle\psi|$

The eigenvalue problem for ρ is $\rho|\phi\rangle = \lambda|\phi\rangle$

It is clear that $|\phi\rangle = |\psi\rangle$ is a solution for this eigenvector problem as $\rho|\psi\rangle = |\psi\rangle\langle\psi|\psi\rangle = |\psi\rangle$ (since the inner product $\langle\psi|\psi\rangle = 1$). Note that 1 is the eigenvalue associated with the eigenvector $|\psi\rangle$. Since we know that $\sum_n \lambda_n = 1$ for a normalized density matrix, the other eigenvalues must be zero. QED.

The Zen of Python

> Beautiful is better than ugly.
>
> Explicit is better than implicit.
>
> Simple is better than complex.
>
> Complex is better than complicated.
>
> Flat is better than nested.
>
> Sparse is better than dense.
>
> Readability counts.
>
> Special cases aren't special enough to break the rules.
>
> Although practicality beats purity.
>
> Errors should never pass silently. Unless explicitly silenced.
>
> In the face of ambiguity, refuse the temptation to guess.
>
> There should be one– and preferably only one –obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea – let's do more of those!
–The Zen of Python