# Symbolic Quantum Circuit Simplification in SymPy

Matthew Curry

June 6, 2011

**Abstract**

In the field of quantum information science, one can design a series of quantum logic operations known as a circuit. Circuits are the basis for quantum computations in quantum computing. As circuits will most likely be designed from a logical standpoint, there could exist mathematical redundancies which will lead to a larger circuit than necessary. These redundancies are computationally expensive, and there is a need for them to be found and eliminated to simplify the circuit. We present our research on finding the rules for simplifying circuits and its implementation in SymPy.

## Part I
# Introduction

Python is an open-source, dynamic, high level programming language with emphasis on code readability. Python is easy to learn and allows one to create very advanced code that would otherwise take a long time to implement. Python has nice features such as fast standard libraries, extensive error handling, and wide usage. Although Python has extensive and highly optimized standard libraries, there are many other libraries being developed by the open-source community that add additional capabilities.

One of these libraries is called SymPy. SymPy is an open-source, symbolic mathematics library for Python. It aims to be a full-featured computer algebra system. Some of its features include code simplicity which leads it to be comprehensible and easily extensible, and it is written purely in Python and does not require external libraries. Its mathematical capabilities include basic arithmetic, algebra, calculus, etc. SymPy has specialized modules for special functions, limits, integration, quantum physics and more. Recently the quantum physics module has been expanded to include general, symbolic quantum mechanics capabilities and symbolic quantum computation capabilities. This expansion was done through two Google Summer of Code (GSoC) 2010 projects, one of which was my own. GSoC is a program that gives stipends to students who work on various open-source projects which they design.

SymPy's quantum module now supports Hilbert spaces, states, spin systems, bras, kets, inner/outer products, operators, commutators, anticommutators, daggers, etc. in their most abstract form. During GSoC 2010, I implemented most of this code along with my research adviser, Brian Granger. The following code example shows some of the basic symbolic quantum mechanics functionality in SymPy:

**Create symbolic states**

```
In [21]: state = Symbol('alpha')*Ket('psi') + Symbol('beta')*Ket('phi'); state
```

Out[21]: $\alpha|\psi\rangle + \beta|\phi\rangle$

**Create symbolic operators**

```
In [22]: A = Operator('A')
         B = Operator('B')
```

```
In [23]: expand((A+B)**2)
```

Out[23]: $AB + (A)^2 + BA + (B)^2$

**Carry out a symbolic commutator**

```
In [24]: comm = Commutator(A,B); comm
```

Out[24]: $[A, B]$

```
In [25]: comm.doit()
```

Out[25]: $AB - BA$

**Take the dagger**

```
In [26]: Dagger(comm.doit())
```

Out[26]: $-A^{\dagger}B^{\dagger} + B^{\dagger}A^{\dagger}$

Notice that the output is in LaTeX. This is done with a cutting edge version of IPython, an enhanced interactive Python shell. Abstract states print nicely in IPython. We see that operators do not commute, because the usual simplification after squaring the polynomial does not occur. The commutator stays as an abstract object until we tell it to carry itself out by calling `.doit()`. Even then, the operators are still abstract and do not simplify. And the dagger reverses the order of the operator multiplication. Everything in this example is in its most abstract form, and until we tell any of the objects what they are (e.g. creating an angular momentum operator or an energy eigenstate, etc.) they will not simplify. SymPy's quantum module now has many quantum computing features such as qubits, gates, circuit plotting, quantum Fourier transforms, etc.

A quantum computer is a computer that exploits the strange effects of quantum mechanics such as entanglement and superposition of quantum states to perform computations that would otherwise take much longer on a classical computer. One of the main differences between a classical computer and quantum computer is that they use bits and qubits respectively. A bit is a basic unit of information that can have one of two states at any given time (traditionally 0 or 1). A qubit can be in a state similar to 0 or 1 or any superposition of these states. In physics, a qubit is a two level quantum system such as a spin system with spin up and spin down states. The mathematics of such a system are states being represented by orthonormal vectors in Hilbert space, a complex vector space with inner products defined. The dimensionality of the Hilbert space grows exponentially with the number of qubits. And depending on the superposition and normalization coefficients (to keep them orthonormal), these vectors can “point” anywhere in their Hilbert space. Single qubit states can be represented in vector form as the following:

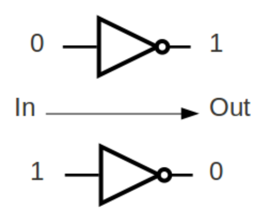
$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{1}$$

Notice that they are linearly independent column vectors.

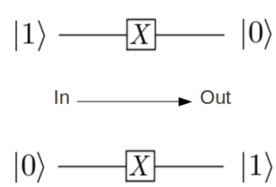
Logical operations can be performed on qubits similarly to how logical operations are performed on bits in a classical computer. The key difference is that operations on qubits correspond to rotation-like operations on vectors (states) in Hilbert space. Because rotations are reversible processes, these quantum operations are reversible. Not only are they reversible, but the rotations can be arbitrary as well. These operations are called quantum logic gates or gates. The dynamics of the use of the these gates is covered in the next section.

# Part II
# Theory

As we know, gates correspond to rotation-like operations in Hilbert space and are applied to qubits in order to perform computations. One gate that is found in both the classical and quantum computing architectures is the NOT gate, which takes a bit or qubit and flips it. Below is the classical example:




And now we see how the NOT gate is applied to a single qubit initially in the state $|1\rangle$ or $|0\rangle$:




Notice that the gate corresponding to a NOT in quantum computing is the X gate or Pauli X matrix:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{2}$$

Recall in (1) that qubits are represented as column matrices. So through basic matrix multiplication we see how the X gate flips a qubit:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{3}$$

And now if we add more qubits we get what is called a (quantum) register. The ordering of qubits is exactly analogous to binary numbers in classical computers. If we have two qubits that look like this:

$$|10\rangle \tag{4}$$

Then this corresponds to the binary number 10 or 2 in decimal. Some subtlety lies in this representation because what we really mean is:

$$|10\rangle = |1\rangle\,|0\rangle = |1\rangle \otimes |0\rangle \tag{5}$$

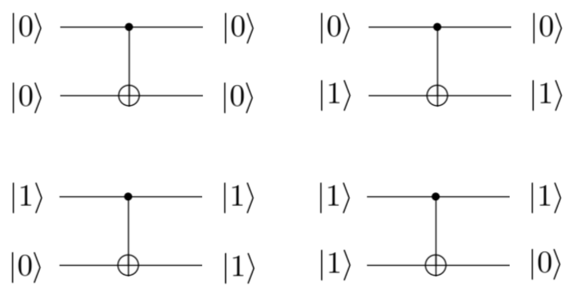
Where $\otimes$ is the tensor product between the states or qubits. It is important to note that this is not normal multiplication and properties in algebra such as distribution do not function the same way. The way these qubits and any more qubits will be represented on a quantum circuit is as follows:

$$\begin{array}{ccc} |1\rangle & \text{———————} & |1\rangle \\ |0\rangle & \text{———————} & |0\rangle \end{array}$$

Where the bottom "wire" represents the $0^{th}$ qubit and the wire above it represents the $1^{st}$ qubit (do not let the values of the qubits confuse you with their placeholder i.e. the $0^{th}$ qubit could be $|1\rangle$ or $|0\rangle$ and same with the $1^{st}$ qubit and the $2^{nd}$ qubit and so on). Now we can have an X gate acting on either qubit. In this case we shall have the X gate act on the $1^{st}$ qubit:

$$\begin{array}{ccc} |1\rangle & \text{———}\boxed{X}\text{———} & |0\rangle \\ |0\rangle & \text{———————} & |0\rangle \end{array}$$

So we have seen only the X gate so far, but there are a few more fundamental gates in quantum computing. They form a complete instruction set, which means that by themselves they can perform any quantum computation possible by (perhaps repeatedly) acting on a qubit register. They can also approximate any rotation of the qubit in its Hilbert space. These gates are the X, Y, Z, S, T, H, and CNOT gates. X, Y, and Z are the well known Pauli matrices of quantum mechanics while S is the square root of Z and T is the square root of S. H is the Hadamard gate which when applied to a qubit in a single state (i.e. $|0\rangle$ or $|1\rangle$) creates a superposition of states $|0\rangle \pm |1\rangle$ with a normalization constant. When applied to an even superposition of states, depending on the phase or negative sign in front of $|1\rangle$, it collapses the superposition to $|0\rangle$ or $|1\rangle$. And finally the only fundamental gate that inherently requires two qubits in order to work is the CNOT gate (or controlled NOT gate or controlled X gate). It requires a control qubit and a target qubit. The logic is simple: flip the target qubit (i.e. apply an X or NOT gate on the target qubit) if and only if the control qubit is $|1\rangle$. Consider the following four examples where the control qubit is the $1^{st}$ qubit and the target qubit is the $0^{th}$ qubit ($CNOT_{1,0}$):




If the control and target qubit were swapped, then the CNOT gate would appear upside-down relative to how it was previously shown and perform accordingly.

There are some properties of these gates which we will now examine. First of all, these gates are known as operators in quantum mechanics. They are unitary operators which means they preserve the inner product (or "length") of the states they act on. Also, their dagger or Hermitian conjugate is their inverse:

$$S^{\dagger}S = 1 \tag{6}$$

X, Y, Z, H and CNOT are Hermitian operators which means they are their own inverse or equivalently they and their Hermitian conjugates are equal:

$$Y^{\dagger} = Y \tag{7}$$

So any of these squared Hermitian operators are equal to the identity operator:

$$Z^2 = 1 \tag{8}$$

In their matrix form, these are unitary matrices which are symmetric about the diagonal. Note that all gates are unitary but not necessarily Hermitian.

SymPy's quantum computation module can symbolically handle all of what has been discussed so far. This module was written by Addison Cugini during GSoC 2010. The following code example shows some if the modules features:

**Create a qubit**

```
In [12]: qubit=Qubit('0'); qubit
```

Out[12]: $|0\rangle$

**Flip the qubit with an X gate**

(we are using the function "qapply" to apply the X gate on the 0th bit)

```
In [13]: qapply(X(0)*qubit)
```

Out[13]: $|1\rangle$

**Create a qubit register**

```
In [14]: register=Qubit('11'); register
```

Out[14]: $|11\rangle$

**Apply a CNOT gate**

```
In [15]: qapply(CNOT(1,0)*register)
```

Out[15]: $|10\rangle$

With our knowledge of gates and qubits, we can now see how a quantum computation works on a rudimentary scale. A quantum computation is basically a long series of the fundamental gates acting on a set of qubits or quantum register (the subscripts represent which qubit the gate is acting on i.e. $X_i$ is acting on the $i^{th}$ qubit):

$$(\dots X_i CNOT_{j,i} Z_j \dots) |\dots 11101101101 \dots\rangle \tag{9}$$

The series of gates multiplied together is called a circuit. If we consider them as matrices, then in a sense we could multiply all of the matrices and end up with a single matrix, a single arbitrary unitary transformation:

$$(\dots X_i CNOT_{j,i} Z_j \dots) = U \tag{10}$$

But from a logical standpoint quantum computations cannot (usually) be easily designed with only one unitary transformation in mind from the start. Instead, we have to compile or decompose such a transformation using the fundamental quantum logic gates. Moreover, in the experimental implementation of a quantum computer, a gate represents a specific physical (quantum mechanical) process - shining a laser on a trapped rubidium atom or subjecting an atom to a magnetic field for a duration of time. If we designed quantum computations as single arbitrary unitary transformations every time, then a new physical process would need to be designed/implemented every time as well. This would be very time consuming! Therefore, we need a set of agreed upon physical processes that can reproduce (or at least approximate) any arbitrary process - the fundamental gates:

$$U = (\dots X_i CNOT_{j,i} Z_j \dots) \tag{11}$$

Conveniently, there is the Solovay-Kitaev algorithm that approximates any unitary transformation by decomposing it into the fundamental gates.

So quantum computations are strings of fundamental gates multiplied together (called circuits) acting on qubits which represent a series of physical processes governed by quantum mechanics. It turns out that there are certain mathematical relationships with the fundamental gates that allow us to simplify these quantum computations/circuits. This project sought to optimize circuits by using these simplification relations as well as decompose arbitrary gates into a finite amount to fundamental gates (not an approximation).

These simplification relations are:

- Hermitian gates square to the identity gate.
  - So in a circuit, two adjacent gates that are identical and Hermitian may simply be ignored or removed from the quantum computation. This removal shortens the circuit and reduces the computational cost of the circuit (less physical processes):

$$X_i Z_i Z_i Y_i \Rightarrow X_i Y_i \tag{12}$$

- Commutation relations.
  - Certain gates commute (adjacently swap) with each other if they have the right characteristics.
  - All gates mutually commute if they act on entirely different qubits:

$$Z_i X_j = X_j Z_i \tag{13}$$

  - The Z, S, and T gates all mutually commute even if they act on the same qubit:

$$Z_i S_i = S_i Z_i \tag{14}$$

  - Z, S, and T gates also commute with any controlled gate if they act on the control qubit:

$$Z_j CNOT_{j,i} = CNOT_{j,i} Z_j \tag{15}$$

  - Any gate commutes with its respective controlled gate if it acts on the target qubit (controlled-Y $\equiv CY$):

$$Y_i CY_{j,i} = CY_{j,i} Y_i \tag{16}$$

Indeed, swapping gates around until they square to the identity gate can lead to more simplified circuits, but more advanced simplification techniques exist.

Interestingly, certain nontrivial sequences of gates multiplied together equal the identity gate. We will refer to these as gate rules from now on. Gate rules allow us to search through a circuit looking for the sequence of gates and replace it with the identity gate. This works fine provided we find the sequence we are looking for but we can get more useful gate relationships out of this knowledge. Take for example the actual gate rule:

$$H_i X_i H_i Z_i = 1 \tag{17}$$

We can search for that sequence of gates and replace them with the identity gate or conversely we can insert that sequence of gates anywhere in a circuit:

$$Y_i S_j H_i X_i H_i Z_i X_i \Rightarrow Y_i S_j X_i \tag{18}$$

But now let's multiply both right sides (side multiplication must be specified for non-commutative math) of both sides of (17) by $Z_i$:

$$H_i X_i H_i = Z_i \tag{19}$$

It looks as though there is an entirely new gate rule that could be used to simplify a circuit, but in reality this does not provide us with more information. Instead, it can still be used to simplify a circuit, but it is a transformation of a gate rule - a gate identity. As before we can look through the circuit for the sequence of gates on the LHS (left-hand side) of (19) and replace them with $Z_i$:

$$Y_i S_j H_i X_i H_i X_i \Rightarrow Y_i S_j Z_i X_i \tag{20}$$

Or we can look for $Z_i$ and replace it with the gates on the LHS of (19) (in this case we take advantage of Hermitian squaring as well):

$$H_i Z_i \Rightarrow H_i H_i X_i H_i \Rightarrow X_i H_i \tag{21}$$

We can keep going and right multiply each side of (19) by $H_i$ and arrive at a new gate identity. Or we can left multiply each side at any time as well. The following procedure leads us to find many gate identities (gate(s) = gate(s)) within one gate rule (gates = identity).

The goal of this project was to find gate rules through brute-force searching and create logic in SymPy that can dynamically generate gate identities from these fundamental rules for use in circuit optimization.

## Part III

# Results

The results discovered from brute-force searching have been fascinating to a degree. I designed and wrote the code used for this searching. Only two qubit space (the fundamental gates acting on two qubits) was examined so far. First, we will see how the brute-force searching worked.

As we know, gates can be represented by matrices. It turns out that two qubits can be represented by a column matrix that has four rows (as opposed to two rows for a single qubit, see (1)):

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, |10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, |11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{22}$$

Therefore, gates will have to be represented by 4x4 matrices even if they only act on one qubit for the matrix dimensions to match. So our possibilities for gates in this space are H, X, Y, Z, S, T acting on either the $0^{th}$ or $1^{st}$ qubit and CNOT being controlled by the $1^{st}$ qubit and targeting the $0^{th}$ qubit or vice versa. We see there are 14 total possibilities for gates in two qubit space, the simplest nontrivial qubit space. Our goal is to find gate rules which are sequences of gates multiplied together that equal the identity gate. Obviously none of the gates themselves equal the identity gate, so we must examine circuits comprised of two or more gates to see if they are a gate rule.

For a circuit of a given length, we want to look at all possible combinations of gates that it can contain. The most intuitive and systematic method of doing this is to simply consider the gates as base-14 digits (remember there are 14 total gate possibilities in two qubit space) and the circuit length as the number of digits in a base-14 number. The Python code written for this task uses SymPy to count in base-14 through all circuit possibilities with the gates being matrices. The code simply multiplies the matrices in the circuit each time it counts and then compares the result to the identity matrix. No advanced logic is used in narrowing down the search space - this is the brute-force method. This method is reasonable for two qubit space, because we are only looking for rules in circuits up to about six or seven gates long. But even in this search space the amount of circuit possibilities scales like $14^N$ where N is the number of gates in the circuit!

The only two-gate rules found were the trivial Hermitian squaring identities. There were no three-gate rules, but one four-gate rule was found. We saw it earlier when we learned that a gate rule generates numerous gate identities:

$$H_iX_iH_iZ_i = 1 \tag{23}$$

Four five-gate rules were found:

$$CNOT_{j,i}X_jCNOT_{j,i}X_jX_i = 1 \tag{24}$$

$$CNOT_{j,i}Y_jCNOT_{j,i}Y_jX_i = 1 \tag{25}$$

$$CNOT_{j,i}Y_iCNOT_{j,i}Y_iZ_j = 1 \tag{26}$$

$$CNOT_{j,i}Z_iCNOT_{j,i}Z_iZ_j = 1 \tag{27}$$

And two six-gate rules were found:

$$CNOT_{j,i}H_iH_jCNOT_{i,j}H_jH_i = 1 \tag{28}$$

$$CNOT_{j,i}Y_jZ_iCNOT_{j,i}X_jY_i = 1 \tag{29}$$

It is interesting to note how sparse some of these search spaces are in terms of gate rules (i.e. there are only four five-gate rules in all 537,824 circuit possibilities for five gate circuits).

# Part IV
# Code

Gate rules are wonderful mathematical truths in and of themselves, and we have found a number of them at this point. But there are so many gate identities within a gate rule that we need some practical way to access them. Writing code that generates all gate identities and effectively searches through them is necessary for circuit optimization. I designed an algorithm and wrote code that does this. The way the code currently works is by cyclically permuting (changing the order of) a rule several times and each time looking for desired gate combinations on both sides. Keep in mind that this algorithm works for rules containing only Hermitian gates. Let us take a closer look at the algorithm.

Look at the classic gate rule:

$$H_iX_iH_iZ_i = 1 \tag{30}$$

If we right multiply by $Z_i$ and then $H_i$ we get:

$$H_iX_i = Z_iH_i \tag{31}$$

Notice that the ordering on the LHS of (31) stays the same as the first two gates in (30), but the RHS of (31) is now reversed relative to the last two gates in (30). From a programming standpoint, if we were looking for equivalencies to $H_iX_i$, we could simply look through the gate rules for this sequence and return the rest of the sequence reversed. Also, observe what happens when we right multiply (30) by $Z_i$ and then left multiply by $Z_i$:

$$Z_iH_iX_iH_i = 1 \tag{32}$$

Even though this looks like a new gate rule, no new information is obtained. Therefore, this is only a permutation of a rule. We can keep permuting (three times in this case) to effectively generate all cyclic permutations of the rule. And we can even look for reversed gate sequences and return the rest of the sequence unreversed (e.g. if we are looking for $H_iZ_i$, we actually search for $Z_iH_i$ in (32), find it on the left side and return $X_iH_i$ unreversed). With this algorithm, all gate identities will be found in a gate rule.

Here are some examples of finding gate identities with the code (note that the quantum computing module does not currently handle symbolic indices, so 1 and 0 are used in place of j and i respectively):

**Find circuits equivalent to the Hadamard gate**

```
In [3]:  h0 = match_gate_rules(H(0))
         for rule in h0:
             display(Eq(H(0),rule))
```

$H_0 = X_0 H_0 Z_0$
$H_0 = Z_0 H_0 X_0$
$H_0 = H_1 CNOT_{0,1} H_1 H_0 CNOT_{1,0}$
$H_0 = CNOT_{1,0} H_0 H_1 CNOT_{0,1} H_1$

**Find circuits equivalent to the circuit Z(0)*Y(1)**

```
In [12]: z0 = match_gate_rules(Z(0)*Y(1)); display(Eq((Z(0)*Y(1)),z0[0]))
```

$Z_0 Y_1 = CNOT_{1,0} X_1 Y_0 CNOT_{1,0}$

**Find circuits equivalent to the X gate**

```
In [13]: x0 = match_gate_rules(X(0))
         for rule in x0:
             display(Eq(X(0),rule))
```

$X_0 = H_0 Z_0 H_0$
$X_0 = CNOT_{1,0} X_1 CNOT_{1,0} X_1$
$X_0 = X_1 CNOT_{1,0} X_1 CNOT_{1,0}$
$X_0 = CNOT_{1,0} Y_1 CNOT_{1,0} Y_1$
$X_0 = Y_1 CNOT_{1,0} Y_1 CNOT_{1,0}$

**Find circuits equivalent to the circuit CNOT(1,0)*Y(0)*CNOT(1,0)**

```
In [16]: cyc = match_gate_rules(CNOT(1,0)*Y(0)*CNOT(1,0))
         for rule in cyc:
             display(Eq(CNOT(1,0)*Y(0)*CNOT(1,0),rule))
```

$CNOT_{1,0} Y_0 CNOT_{1,0} = Y_0 Z_1$
$CNOT_{1,0} Y_0 CNOT_{1,0} = Z_1 Y_0$

In the previous example we use the function `match_gate_rules()`, which takes in a gate or a circuit as its argument, searches through all known gate identities for equivalent gates or circuits, and returns the result(s) as a list. The results in this example are printed in LaTeX as input equals result for simplicity.

# Part V
# Conclusion and Future Directions

We now have a basic way to search for gate identities, and we have many gate rules to search through. The developmental structure for creating gate simplification logic has three levels to it:

1. The lowest level is finding the gate rules themselves. We only used the most brute-force, systematic way of finding them due to time constraints.

2. The second level is creating a method for searching through the gate rules for gate identities. This method (`match_gate_rules()` in SymPy) has a core structure built as of now, but there are several cases that it cannot handle which will be discussed soon.

3. The highest level of logic is determining ways to simplify circuits using gate simplification relations and the gate identity searching function. This level is the most difficult because it will not be trivial to know when to replace gates with other gates or to commute them and so on.

The work done on this project does not stop here. There exists higher dimensional qubit spaces (such as three qubit spaces and so on) to search through for gate rules. And not to mention better ways of searching such as caching matrix multiplications and applying circuit simplification logic (when it exists) to the results. The code created for finding the gate identities is not able to handle gate rules with non-Hermitian gates in it (S and T). This is simply do to the fact that the cyclic permutation method does not work anymore because it relied on the gates squaring to identity. A more encompassing approach will solve this problem. And finally no code exists yet that can actually look at a circuit, apply gate identities or simplification relations, and simplify the circuit. This is surely at least another senior project in and of itself.

## References


[1] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, New York, 2000.

[2] D. Mermin, *Quantum Computer Science*, Cambridge University Press, New York, 2007.

[3] D. Griffiths, *Introduction to Quantum Mechanics*, Pearson Education, 2005.

[4] J. Taylor, C. Zafiratos, and M. Dubson, *Modern Physics for Scientists and Engineers*, Prentice Hall, New Jersey, 2004.

[5] M. Sedlak and M. Plesch, "Towards Optimization of Quantum Circuits," *Central European Journal of Physics*, Springer-Verlag, Berlin, Germany, 2007, pp. 128-134.

[6] G. Cybenko, "Reducing Quantum Computations to Elementary Unitary Operators," *Computing in Science & Engineering*, IEEE, 2001, pp. 27-32.

[7] N. Scott and G. Dueck, "Pairwise Decomposition of Toffoli Gates in a Quantum Circuit," *GLSVLSI'08*, ACM, 2008, pp. 231-235.

[8] V. Shende and I Markov, "On the CNOT-cost of Toffoli gates," *Quant.Inf.Comp. 2008*

[9] *Python Programming Language – Official Website*, Web. Accessed 06 June 2011, http://www.python.org.

[10] *SymPy*, Web. Accessed 06 June 2011, http://www.sympy.org.