The symbolic simulation and manipulation of quantum circuits

Addison Cugini
Apple Computer, Cupertino, CA

Matt Curry
Department of Physics, University of New Mexico, Albuquerque, NM 87131, USA

Raymond Wong
Department of Computer Science, California Polytechnic State University, San Luis Obispo, CA 93407, USA

Austen Greene and Brian E. Granger*
Department of Physics, California Polytechnic State University, San Luis Obispo, CA 93407, USA
(Dated: December 7, 2011)

The simulation of quantum computers and quantum information systems on classical computers is an important part of quantum information science. With such simulations, algorithms can be developed and tested, experiments can be validated and insight can be gained about the foundations of quantum information. We have developed an open source software package for simulating quantum computers symbolically using a computer algebra system, SymPy. The symbolic manipulation of gates and circuits has many advantages over the traditional numerical approach where gates and qubits are represented as large matrices and vectors. In this work, we introduce the software, describe its features through examples and outline the advantages of this approach.

I. INTRODUCTION

II. FOUNDATIONS

A. SymPy

SymPy [1] is a Python library for symbolic mathematics. It aims 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 is written entirely in Python and does not require any external libraries.

SymPy was started in 2005 by Ondřej Čertík and it now has an active community of over 100 developers contributing to it. SymPy is BSD licensed and anyone can easily contribute new features and algorithms by submitting a GitHub pull request. SymPy has been periodically participating in the Google Summer of Code (25 students in 5 years) and the quantum computing module has started as one such project as well.

In this paragraph we will give an overview of SymPy's capabilities, probably in the form of a bulleted list.

In this paragraph we will describe the latex printing support in SymPy and how that can be used in conjunction with SymPy Live or IPython to get nicely formatted output. We should emphasize the fact that all latex formulas in this paper were autogenerated by SymPy.

We now show a few examples of SymPy's capabilities to show its usefulness as a general computer algebra system.

In SymPy symbols have to be created explicitely. Here we create the symbols (x, y, z):

```
x, y, z = symbols("x y z")
```

Once these symbols have been created, symbolic expressions can be created and manipulated. Here is an example involving a rational number, π, i, e and basic arithmetic operations:

```
Rational(3,2)*pi+exp(I*x) / (x**2+y)
```

$$\frac{3}{2}\pi + \frac{e^{ix}}{x^2+y}$$

Expressions can be evaluated numerically using SymPy's arbitrary precision arithmetic algorithms:

```
exp(pi*sqrt(163)).evalf(35)
```

$$262537412640768743.99999999999925007$$

Here is an example of expanding and refactoring a polynomial:

```
((x+y)**2*(x+1)).expand()
```

$$x^3 + 2x^2y + x^2 + xy^2 + 2xy + y^2$$

```
factor(_)
```

$$(x+1)\,(x+y)^2$$

Derivatives can be taken of expressions:

* bgranger@calpoly.edu

```
diff(cos(x**2)**2/(1+x), x)
```

$$-4\frac{x\sin\left(x^2\right)\cos\left(x^2\right)}{x+1}-\frac{\cos^2\left(x^2\right)}{\left(x+1\right)^2}$$

We emphasize that these examples are merely representative of its vast capabilities. We encourage the reader to consult the SymPy documentation for a more complete set of examples.

Python is an object oriented language and SymPy leverages that to build reusable objects for symbolic mathematics. Internally, SymPy creates instances of these objects and stores them in an expression tree. In this example, we construct a simple expression and then display its expression tree and SymPy objects using the `srepr` function:

```
e = x + 2*y
srepr(e)
```

```
Add(Symbol('x'), Mul(Integer(2), Symbol('y')))
```

This combination of object orientation and expression trees makes SymPy easily extensible; once new objects are created they can immediately be used in arbitrary expressions.

B. Symbolic Dirac notation

Dirac's bra-ket notation has become the standard way of expressing the mathematics of quantum mechanics. This notation is concise, powerful and flexible. We have implemented this notation in SymPy in its most general form.

1. Bras and kets

Symbolic kets can be created using the `Ket` class as seen here:

```
phi, psi = Ket('phi'), Ket('psi')
```

These ket instances are fully symbolic and behave exactly like the corresponding mathematical entities. For example, one can form a linear combination using addition and scalar multiplication:

```
alpha = Symbol('alpha', complex=True)
beta = Symbol('beta', complex=True)
state = alpha*psi + beta*phi; state
```

$$\alpha|\psi\rangle+\beta|\phi\rangle$$

Bras can be created using the `Bra` class directly or by using the `Dagger` class on an expression involving kets:

```
ip = Dagger(state)*state; ip
```

$$\left(\overline{\alpha}\langle\psi|+\overline{\beta}\langle\phi|\right)\left(\alpha|\psi\rangle+\beta|\phi\rangle\right)$$

Because this is a standard SymPy expression, we can use standard SymPy functions and methods for manipulating expression. Here we use expand to multiply this expression out, followed by `qapply` which identifies inner and outer products in an expression.

```
qapply(expand(ip))
```

$$\alpha\overline{\alpha}\,\langle\psi\,|\psi\rangle+\alpha\overline{\beta}\,\langle\phi\,|\psi\rangle+\beta\overline{\alpha}\,\langle\psi\,|\phi\rangle+\beta\overline{\beta}\,\langle\phi\,|\phi\rangle$$

2. Operators

SymPy also has a full set of classes for handling symbolic operators. Here we create three operators, one of which is hermitian:

```
A = Operator('A')
B = Operator('B')
C = HermitianOperator('C')
```

When used in arithmetic expressions SymPy knows that operators do not commute under multiplication/composition as is seen by expanding a polynomial of operators:

```
expand((A+B)**2)
```

$$AB+\left(A\right)^2+BA+\left(B\right)^2$$

Commutators of operators can also be created:

```
comm = Commutator(A*B,B+C); comm
```

$$[AB,C+B]$$

The `expand` function has custom logic for expanding commutators using standard commutator relations:

```
comm.expand(commutator=True)
```

$$-\left[C,A\right]B+\left[A,B\right]B-A\left[C,B\right]$$

Any commutator can be performed ($[A,B]\rightarrow AB-BA$) using the `doit` method:

```
_.doit().expand()
```

$$-CAB+ABC+A\left(B\right)^2-BAB$$

The `Dagger` class also works with operators and is aware of the properties of unitary and hermitian operators:

```
Dagger(_)
```

$$-B^\dagger A^\dagger B^\dagger-B^\dagger A^\dagger C+\left(B^\dagger\right)^2A^\dagger+CB^\dagger A^\dagger$$

3. Tensor products

Symbolic tensor products of operators and states can also be created and manipulated:

```
op = TensorProduct(A,B+C)
state = TensorProduct(psi,phi)
op*state
```

$$A \otimes (C + B) \left|\psi\right\rangle \otimes \left|\phi\right\rangle$$

Once a tensor product has been created, it can be simplified,

```
tensor_product_simp(_)
```

$$(A|\psi\rangle) \otimes ((C + B) \left|\phi\right\rangle)$$

and expanded:

```
expand(_)
```

$$(A|\psi\rangle) \otimes (C|\phi\rangle + B|\phi\rangle)$$

As we have developed the Dirac notation in SymPy, we have tried to balance mathematical formality with ease of use. For example, all bras, kets and operators internally track what Hilbert space they are associated with, but we do not require users to deal with this layer unless they need to.

An important aspect of quantum mechanics is the representation of operators and states in different bases. We have developed a framework that allows any operator or state to declare its representation in any number of different bases. This logic is handled through a single `represent` function that handles both discrete and continuous Hilbert spaces. Its usage will be illustrated below for qubits and gates.

We have also implemented completely general logic for declaring how operators act on states, which is handled through the `qapply` function ("quantum apply"). We emphasize that the application of operators to states is fully symbolic and does not in any way depend on representing those operators or states in a particular basis. In the context of quantum computing, this means that we can apply gates to qubits without representing them as large matrices.

The symbolic Dirac notation built into SymPy provides a solid foundation, upon which a wide range of quantum mechanical systems can be implemented. In addition to the quantum computing capabilities described below, we have also implemented quantum angular momentum and the second quantized approach to many body quantum theory using it. We now turn to describe the details of how we have used it to build a general tool for simulating quantum computers.

III. SYMBOLIC COMPUTING COMPUTING

A. Qubits

B. Gates

C. Density matrix

IV. ALGORITHMS AND EXAMPLES

A. Quantum Fourier transform

B. Grover's Algorithm

C. Teleportation

D. Dense coding

E. Heisenberg limited measurement

F. Quantum error correction

[1] SymPy Development Team, *SymPy: Python library for symbolic mathematics* (2011).
[2] L. K. Grover, Am. J. Phys **69**, 769 (2001).