Solovay-Kitaev's Algorithm

Operators Approximation

David Cardozo, Juanita Duque & Jonathan Niño

Quantum Computing 2015-1. Professor: César Galindo

Universidad de los Andes
Facultad de Ciencias

Abstract

In this work, we do a simple implementation of the Solovay-Kitaev's Algorithm using Python and Numpy for symbolic and numerical linear algebra evaluations, our discussion is mainly expository and it is mostly based on the C++ implementation in [**Harrow2001**], and the Python libraries in [**PaulPhamGit**].

Introduction

A model of quantum computation is based on quantum circuits, composed by unitary operators.
Any change over time can be expressed by an unitary matrix, without loss of generality we can normalize and suppose that this matrix has determinant equal to 1. The group of these matrices is named $SU(N)$, often called *the special unitary group*. Although $\mathrm{SU}(N)$ has nice properties, we cannot simulate exactly all the matrices in that group because $SU(N)$ has the cardinality of the continuum [**Nielsen2000**] and we can only simulate enumerable matrices in an exact manner.
A very good workaround is to use the Solovay's Kitaev's algorithm, which consists in approximating any matrix in $SU(N)$ with a group of given matrices with a distance between the operators to be less than a given $\epsilon > 0$. The algorithm works in polynomial time in classic computers and it is an important result for simulating a quantum computer or creating a quantum compiler.

Quantum Circuits

A quantum circuit over a basis $\mathcal{A}$ is a sequence $U_1[A_1], ..., U_L[A_L]$ where $\mathcal{A}$ is a fixed set of unitary operators, $U_i \in \mathcal{A}$ and A_i is and ordered set of qubits. It is the analogue of a circuit in classical computing and it is useful to represent quantum algorithms [**kitaev2002classical**].
A quantum circuit is built upon quantum gates. Formally quantum gates are operators over one or two qbits, for example the Hadamard gates or the Pauli transformations. The gates that are normally chosen are those that can be potentially implemented as a physical phenomenom. Every circuit is build by various combinations of tensorial products and multiplication of quantum gates.

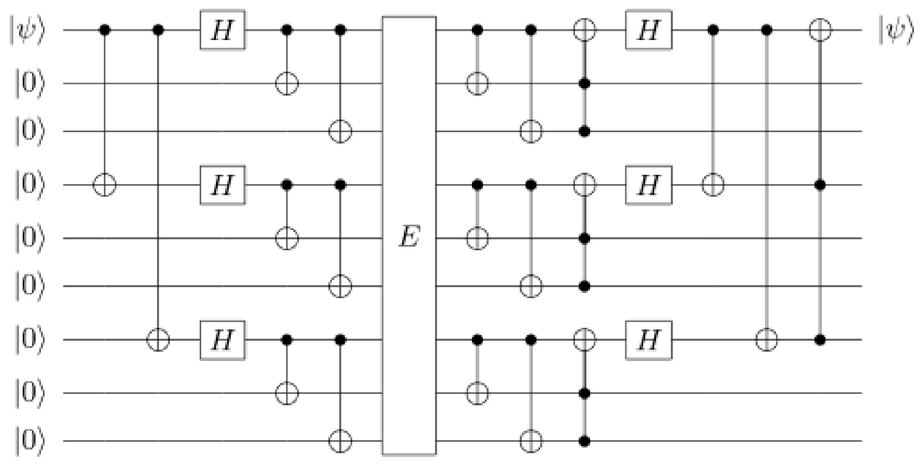

Figure 1: Example of a quantum circuit.

Solovay-Kitaev's Theorem

For any $A_1, ..., A_l \in SU(N)$ such that $\langle A_1, ..., A_l \rangle$ is dense in $SU(N)$, there exists a constant C and a procedure for approximating any $U \in SU(N)$ to precision ϵ with a string of $A_1, ..., A_l$ and their inverses of length no greater than $Clog^c(1/\epsilon)$, where $c \approx 4$ and C is independent of U and ϵ. This procedure can be implemented in polynomial time in $\log(1/\epsilon)$
As a result we have that for any family of universal gates, there exists a constant C such that any quantum circuit with n arbitrary gates can be constructed from fewer than $C_n \log^c(n) \log(1/\delta)$ universal gates if the probability of error is δ.

One of the central results of quantum computation, the Solovay-Kitaev (SK) algorithm for the approximation of $SU(d)$ gates by a fixed, finite basis $\mathcal{B}$. Although many recent results have surpassed SK in terms of efficiency, many of them have, at their core, the base-level approximation of SK. Moreover, many techniques for analyzing and understanding quantum compilers were first developed for SK, which continues to be the best way to learn them. Finally, the overall structure of the original SK algorithm is so simple, it is surprising that it works so well. The essential structure of SK is to recursively generate nets of unitary operators with successively finer precision. At any given level of recursion, the input gate is divided up into two halves which can be approximated with less precision, but whose errors cancel out when the approximated halves are recombined. Our pseudo-code in Figure 2 and explanation follows the exposition in [**Dawson2005**] and [**Harrow2001**].
Since the recursion must eventually bottom out, we must precompute some sequences of gates from $\mathcal{B}$ up to length l_0. This is the classical preprocessing step which requires upfront storage space for this coarsest-grained net, where each sequence is no more than ϵ_0 from its nearest neighbor. According to [**Dawson2005**] the values of $l_0 = 16$ and $\epsilon_0 = 0.14$ using operator norm distance is sufficient for most applications. This step can be done offline and reused across multiple runs of the compiler, assuming $\mathcal{B}$ for your quantum computer doesn't change.
The BASIC-APPROX function below does a lookup (e.g. using some kd-tree search maneuvers through higher-dimensional vector spaces) using this ϵ_0-net, and all higher recursive calls to SK are effectively constructing finer ϵ-nets "on the fly" as needed.
The FACTOR function performs a balanced group commutator decomposition, $U = ABA^\dagger B^\dagger$, and then recursively approximates the A and B operators, again using SK. We denote by $\tilde{U}_i$ the approximation of U using i levels of SK recursion. When they are multiplied together again, along with their inverses, their errors (which go like ϵ) are symmetric and cancel out in such a way that the resulting product U has errors which go like ϵ^2, using the properties of the balanced group commutator. In this manner, we can eventually sharpen our desired error down to any value. A geometric decomposition is used in the Dawson-Nielsen implementation [**Dawson2005**], while one based on a Baker-Campbell-Hausdorff approximation is used in the Harrow implementation [**Harrow2001**] following Nielsen and Chuang [**Nielsen2000**]. It is not known which method converges more quickly to a desired gate in general.

Pseudocode

The pseudocode in which the implementation was based is the following

FUNCTION $\tilde{U}_i \leftarrow \mathrm{SK}(U, i)$
if $i = 0$ **then**
  $\tilde{U}_i \leftarrow \text{BASIC-APPROX}(U)$
else
  $\tilde{U}_{i-1} \leftarrow \mathrm{SK}(U, i-1)$
  $A, B \leftarrow \text{FACTOR}(U\tilde{U}_{i-1}^\dagger)$
  $\tilde{A}_{i-1} \leftarrow \mathrm{SK}(A, i-1)$
  $\tilde{B}_{i-1} \leftarrow \mathrm{SK}(B, i-1)$
  $\tilde{U}_i \leftarrow \tilde{A}_{i-1}\tilde{B}_{i-1}\tilde{A}_{i-1}^\dagger\tilde{B}_{i-1}^\dagger\tilde{U}_{i-1}$
end if
return $\tilde{U}_i$

Figure 2: Pseudo-code for the Solovay-Kitaev algorithm [**Dawson2005**].

Implementation

As mentioned above the implementation of the algorithm was develop in Python using a previous work by Paul Pham. The program shows the exact sequences of matrices used to approximate the unitary operator and gives the distance between the approximation and the original matrix. An improvement made was the use of a k-d-tree to optimize the search time of the original algorithm. The algorithm is performed within polynomial time.

Observations and Conclusions

We often have a very small trace distance in our calculations of the approximation (of the order ≈ 0.23), which as expected, is very good for applications and future work. On the other hand, the use of Python for this implementations decreases the speed considerably compared with the C++ implementation. We don't see this implementation feasible for working with $SU(4)$ or $SU(8)$, mainly because generating the sequences associated to these groups, generates an 'out of memory' situation and the process takes an extremely long time.