QCEmu: A Quantum Computing Simulator

Daniel Araújo de Medeiros

October 21, 2021

Contents

1 Introduction

This paper aims to present QCEmu, a simple quantum computer simulator developed as the final project for the course **FDD3280 - Quantum Computing for Computer Scientists** at KTH Royal Institute of Technology. I will discuss the implementation details, the domain-specific language from QCEmu while also showing application examples and finally some performance issues regarding the quantum simulator.

The source code for this work is available at Github and is fully public.

2 Implementation

2.1 First design

The first design attempt of QCEmu is contained at the "v1" folder on the repository and is currently obsolete/incomplete. However, it remains there as an interesting approach for the quantum computer simulator. This design was originally inspired by the Qiskit package from IBM.

Figure 1 illustrates the concept idea for this attempt. In a nutshell, each individual qubit would be an object of a qubit class. This class also contains the unitary gates to be applied to the state vector of this qubit (which has length 2) and read/write operations. Meanwhile, the qcircuit class would contain the state vector for all qubits, the multi-qubit gates (i.e., CNOT, CZ, etc), do all the visualization and, finally, also be the interface for the user.

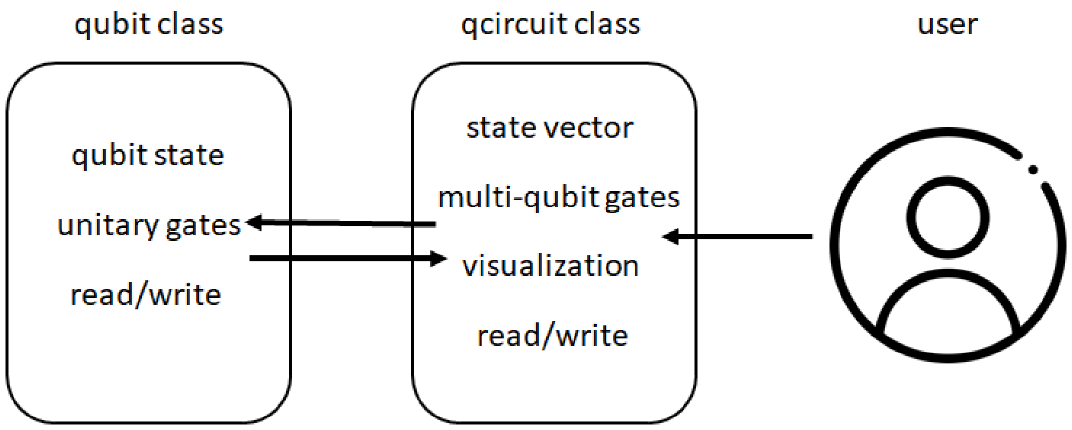

Figure 1: First design of the QCEmu.

While this approach seems to be valid in terms of performance (i.e., operating on single qubits is cheaper than operating in large state vectors), this design was dropped due to the fact of the added complexity when leading with quantum entanglement. As quantum separability is classed as an NP-hard problem, one would need to check conditions most of the time to know if the operation should be carried at the state vector or the individual qubit class. Those implementation details seem to be beyond the scope of this course.

2.2 Second design

As the first approach did not work well, the second attempt was to use the one shown during the course classes. Figure 2 illustrates the finished design as found in the repository. The user interacts with the main function, which is solely responsible for checking the input (a text file) and invoking the parser. The parser breaks the provided input line-by-line and analyzes each one of them according to the grammar described in Section 3.

Instead of using a qubit class as Version 1 did, this time the QuantumCircuit class performs all operations (regardless of uni-gate or multi-gate) directly at the state vector, without worrying about quantum entanglement. This also means that the vector size grows exponentially as the number of qubits also increases (given by the formula 2^{qubits}).

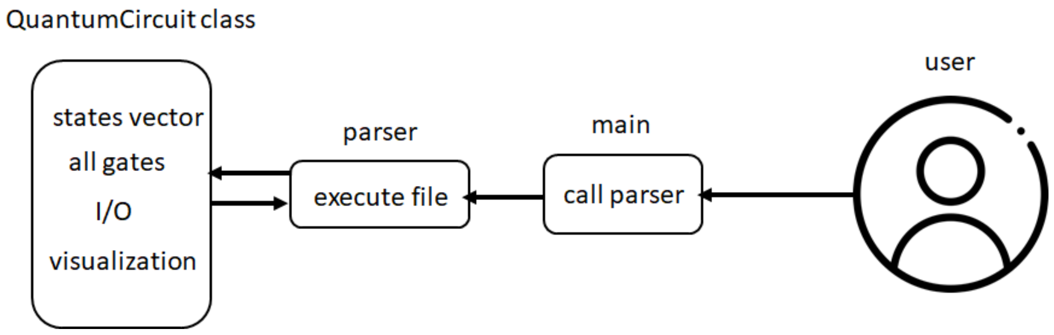

Figure 2: Second design of the QCEmu.

2.2.1 Unitary Gates

The unitary gates are fairly straight-forward to implement: in order to apply a gate into the q_n qubit, do a tensor product between identity matrices in the range of (0, n - 1). Then, do a tensor product between the previous result and the gate matrix, following for more tensor products with identities matrices within the range (n, number of qubits - 1). As example, in order to apply a unitary gate operation on the second qubit of a three qubit system, the mathematical formulation would be established as $I_1 \otimes G_{op} \otimes I_2$ Finally, a inner (dot) product should be applied between this resulting matrix and the state vector. This algorithm makes the code for unitary gates fairly reusable, as the only variable to be supplied is the gate matrix.

The implemented operations for unitary gates on QCEmu are Hadamard, NOT, ROTY, ROTZ, PHASE, T and $T^\dagger$ (TDG). As the matrix formulation for each one of them are easily available on the internet, only the results of their usage will be shown. In particular, Figure 3 starts from a 2 qubit system (4 states) setting and applies each operation in the order as displayed. Results were asserted based on the ones from the Quirk web simulator.

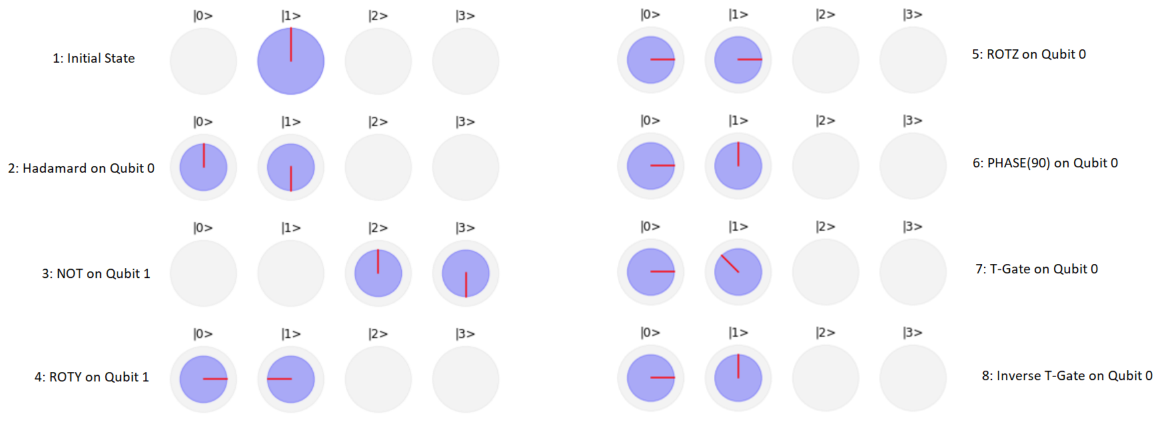

Figure 3: Multiple unitary gate operations.

2.2.2 Arithmetic Operators

QCEmu has two arithmetic operations, namely Increment and Decrement. As their names implies, their function is carrying the values of a state to the next one (or previous, in the case of decrement). Their implementation are rather straight-forward, mainly requiring a manipulation in the vector states.

Figure 4 shows a 3 qubit system (8 states) along with two increment operations and two decrement operations, which ends up making the last state equal to the initial one.

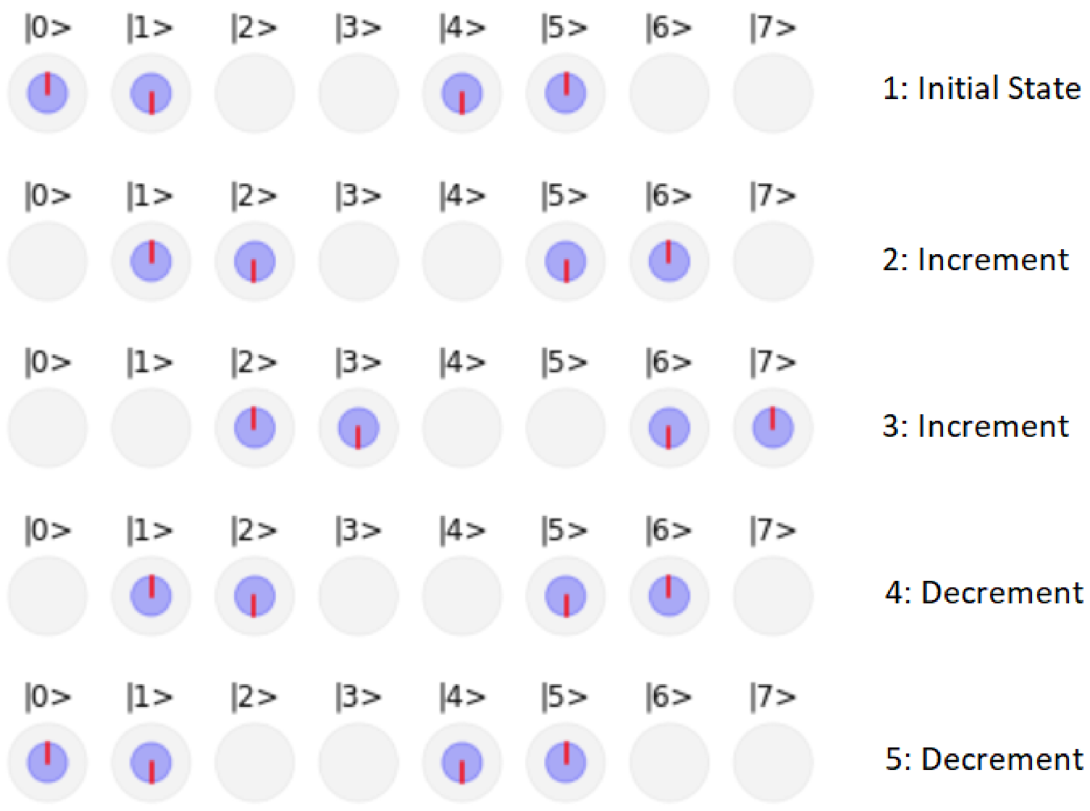

Figure 4: Performing increment and decrement operations.

2.2.3 Multi-qubit Gates

As for Multi-qubit gates, the Controled-NOT (CCNOT) and Toffoli gates are implemented. Results of CCNOT for two qubits are shown in Figure 5, where the control qubit is 1 and the target qubit is 0. The opposite implementation (target equals to 1 and control equals to zero) uses the previous CNOT between two Hadamard gates; the result is the same as the initial state in magnitude, but with the phases flipped.

For three qubits and over, the application is bugged. Three different implementations were tried: the one from Corbett, which is also used here for the two CNOTs, where the matrixes are hard-coded; the implementation from UCDavis, where the matrices were automatically generated; and another implementation found on Github that also creates gates automatically. However, the implementation does not work properly sometimes, mainly when using entangled states. In some cases, as seen in Figure 6, it works. The reason of this discrepancy should be investigated.

The same happens with the CCNOT gate, also known as Toffoli. Two approaches to implement were attempted: the first one was the usage of the Toffoli matrix, but it doesn't appear to make any effect on the data. The second attempt was to implement Toffoli as a junction of CNOTs, T and $T^\dagger$ gates which also didn't work properly due to the bug on the CNOT implementation.

3 QCLang: A Domain-specific Language

QCLang is a procedural-style domain specific language for QCEmu. Its usage consists in the syntax of "COMMAND VALUE-1 VALUE-2", where VALUE-1 AND VALUE-2 are usually parameters related to qubits. Qubits are zero-indexed. All files should start with the command INIT, which express the number of qubits the circuit is supposed to have. The grammar for QLang can be seen in Table 3, while three application scripts can be seen in Section 4.

The implementation was done through the reading of each line and splitting them according to

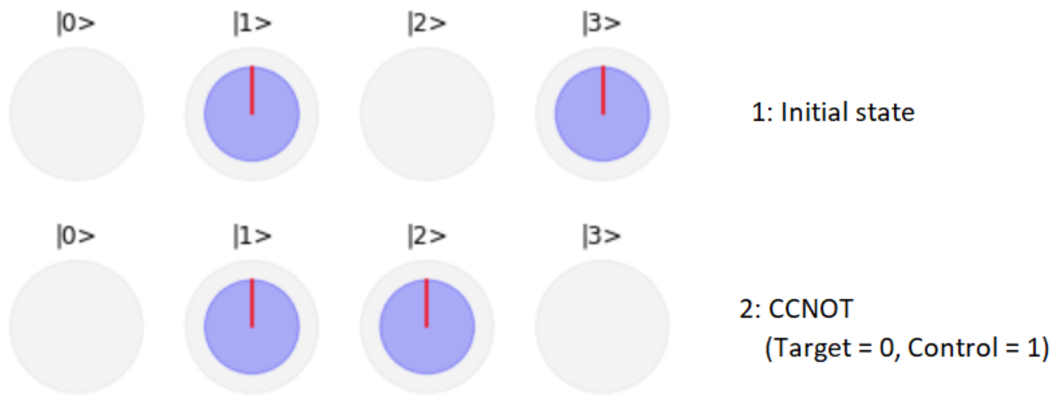

Figure 5: Application of CNOT for two qubits.

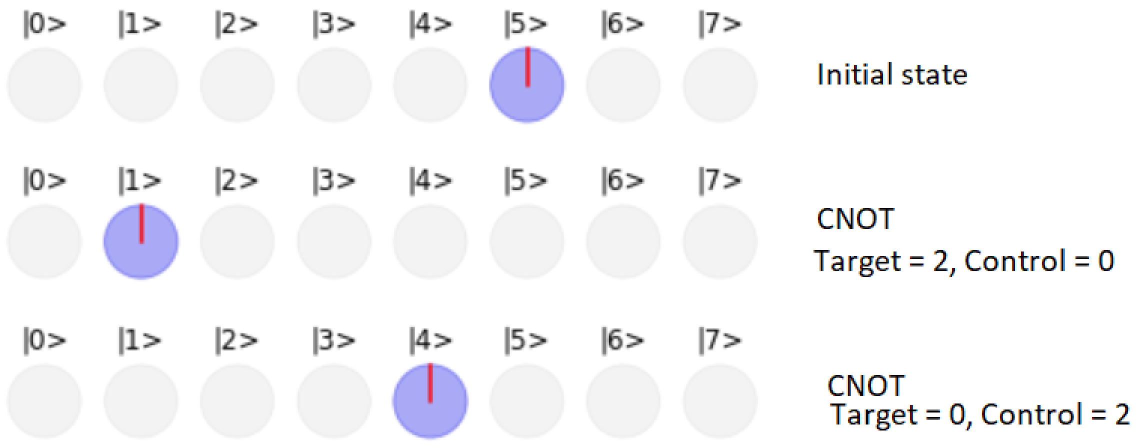

Figure 6: CNOT for three qubits.

the number of spaces. Since the grammar is simple (first object in the dictionary is always the gate, second and third are values), doing conditionals are fairly straightforward.

4 Applications

4.1 SWAP Algorithm

The SWAP algorithm shown here uses two qubits. Results in Figure 7 are obtained after running Algorithm 1 in the QCEmu. As expected, the $|01>$ state is transformed into the $|10>$ state after the SWAP.

Algorithm 1 SWAP algorithm with 2 qubits

```
1: INIT 2
2: WRITE 10
3: CX 1 0
4: CX 0 1
5: CX 1 0
6: BLOCH
```

Command	VAL-1	VAL-2	Comment
INIT	X		Instantiate a Quantum Circuit with X qubits.
H	X		Performs Hadamard operation on the X-th qubit.
NOT	X		Performs NOT (ROTX) operation on the X-th qubit.
ROTY	X		Performs the ROTY operation on the X-th qubit.
ROTZ	X		Performs the ROTZ operation on the X-th qubit.
PHASE	X	Y	Performs the phase operation of Y-angle (degrees) on the X-th qubit.
T	X		Performs the T-GATE operation on the X-th qubit.
TDG	X		Performes the T-DAGGER GATE operation on the X-th qubit.
CX	X	Y	Controlled NOT operation where the X-th qubit is the control and the Y-th qubit act as target.
READ			Save 1000 quantum reads as a bar graph.
WRITE	X		Instantiate the X-th state (binary).
BLOCH			Saves state matrix as bloch circles.
#			Commentary (line is skiped).

Table 1: QCLang grammar.

4.2 Deutsch's Algorithm

Through the usage of QCLang, we also implement Deutsch algorithm as shown in Algorithm 2. It consists solely on a CNOT enclosed by Hadamard functions. Results are shown in Figure 4.2, which output the expected state in the literature.

Algorithm 2 Deutsch's algorithm with 2 qubits

```
1: INIT 2
2: WRITE 1
3: H 0
4: H 1
5: CX 0 1
6: H 0
7: H 1
8: BLOCH
```

4.3 Grover's Algorithm

As explained in Section 2.2.3, the Toffoli gate is currently buggy hence why it isn't possible to show the results of the Grover's Algorithm. However, it is also implemented in QCLang for whenever that issue is solved. This Grover's algorithm (3-qubits) uses the CCNOT gate as a basis for the construction of the CCZ gate. The expected outcome would be a high amplitude on the state —111¿.

5 Performance

As one would expect, the performance in this type of implementation is heavily constrained by the size of the state vector, which is given by the formula 2^{qbits}. A timer was added before and after the

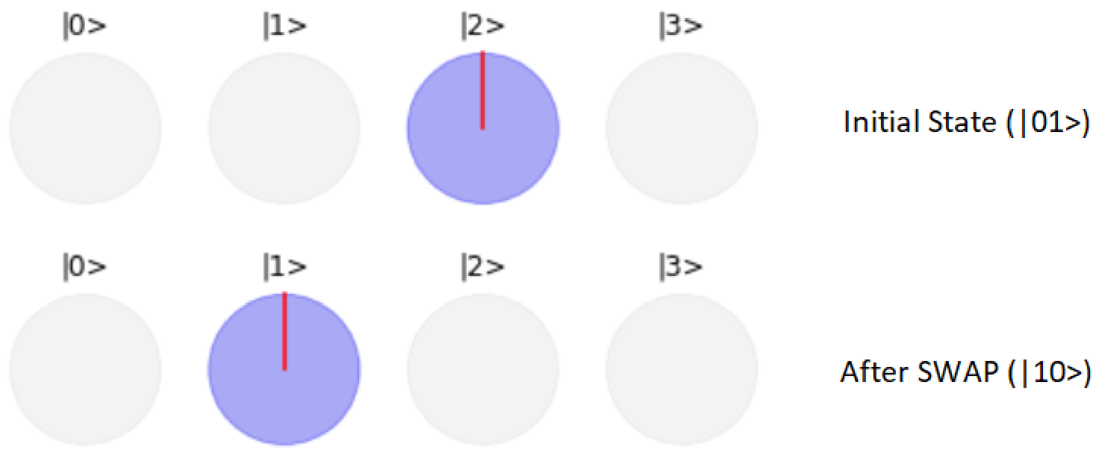

Figure 7: SWAP algorithm for two qubits.

execution of the Hadamard function - which includes lots of tensor products with identities matrices - in order to check how long it would take.

Most of the executions up to 10 qubits takes in between a few nanoseconds to almost 1 second. However, when the number of 14 qubits was reached, the experiment couldn't be carried since the size of the data couldn't be allocated to the available RAM on the system. This implies that normal desktop systems have issues emulating a large number of qubits due to the memory. However, it should also be suspected that the application might be compute-bound as the number of matrix operations starts to increase dramatically (even if most of them are just zeros and ones, due to the identities).

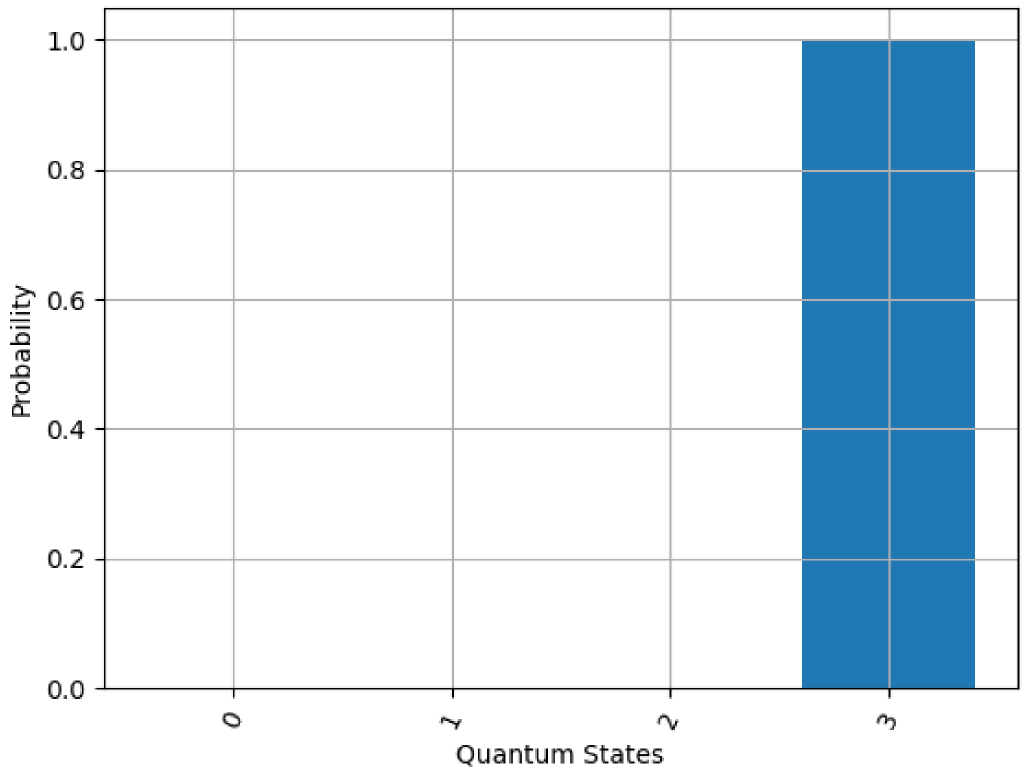

Figure 8: Deutsch's algorithm for two qubits.

Algorithm 3 Grover's algorithm with 3-qubits

```
1: INIT 3
2: WRITE 101
3: H 0
4: H 1
5: H 2
6: H 2
7: CCX 0 1 2
8: H 2
9: H 0
10: H 1
11: H 2
12: NOT 0
13: NOT 1
14: NOT 2
15: H 2
16: CCX 0 1 2
17: H 2
18: NOT 0
19: NOT 1
20: NOT 2
21: H 0
22: H 1
```