Verifying Quantum Computation Comes From a Quantum Computer

(and not a simulator)

Richard Kent

Boston University

PY 536 - Quantum Computing

Abstract

Provided that it is currently possible to simulate error in the measurement of quantum circuits with few qubits on a classical computer, we seek to discern where the results originate from. First by looking into the technologies available today to simulate quantum computation on a classical computer then, run those same circuits on a quantum computer, and compare the results. Afterwards, evaluating the possibility to statistically project measurements of quantum circuits needing a reasonably large number of qubits.

1 Introduction

The resources required to simulate quantum hardware increases exponentially with the number of qubits. Perhaps there is a way to statistically speed up this process by simulating the probability distribution of quantum circuits. That is, analyzing the distribution of possible quantum states and determining if we can verifiably simulate the probability distribution (including simulating error and quantum noise). We first look into the technologies currently available to the general public, quantum computers to send programs to, and quantum simulators. Then, we create some quantum circuits to be run on a quantum computer and on a classical computer simulating a quantum computer. Finally, knowing the resulting states of a particular computation, we seek to generalize the possibility to simply emulate the probabilities of these outcomes (including error) and if pseudo-randomness is not enough, determining how to generate the entropy needed to simulate the distribution of states.

2 Quantum Programming

Today, there are various popular software development kits targeting a variety of programming languages for programming quantum circuits to run on quantum computers or simulators. Of these, IBM seems to have produced the most comprehensive set of tools. They have produced Qiskit which is an open-source quantum computing software development framework mostly targeting the python programming language. With Qiskit, one can create programs to run quantum circuits.

3 IBM Circuit Composer

Along with Qiskit, IBM also provides the browser based "Circuit Composer". One can create quantum circuits visually to be run on one of IBM's quantum computers or a simulator. Here is an example of a circuit for the Quantum Fourier Transform.

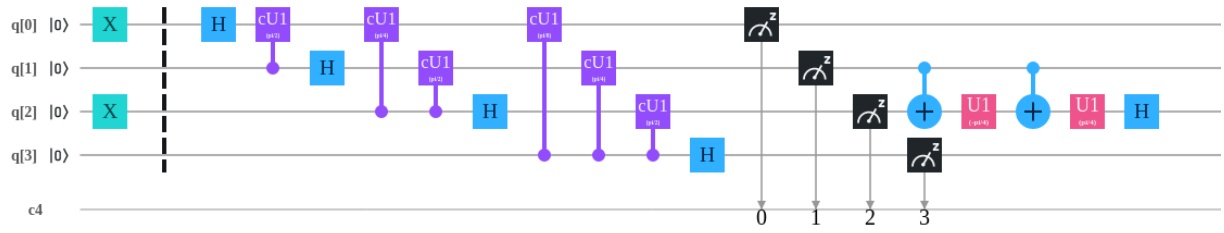


Figure 3.1 Quantum Fourier Transform

After running this circuit a histogram of the states will be output. Here is an example output. See the Figure 3.3 in the Figures section for the QASM of the QFT.

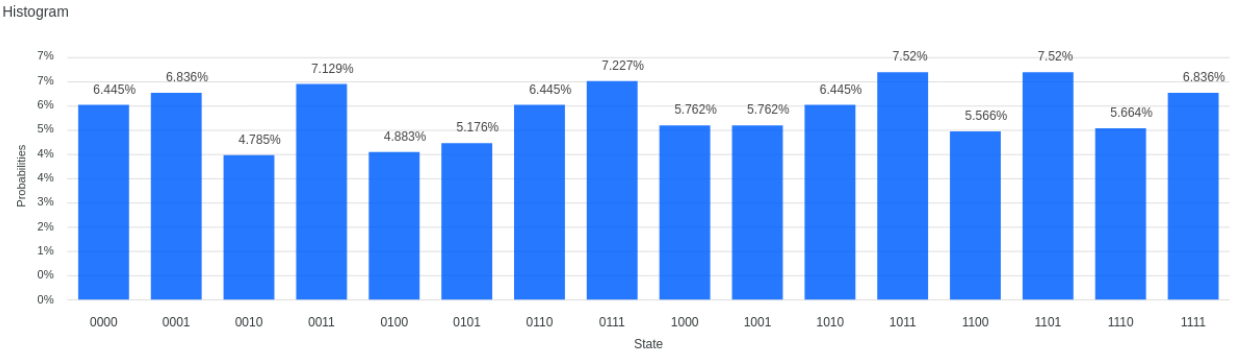


Figure 3.2 Quantum Simulator results for QFT

4 Open Quantum Assembly Language

IBM has also created Open Quantum Assembly Language (QASM) which is an intermediate representation for quantum instructions, inspired by assembly language of modern day computers. Here is an example QASM instructions of the Greenberger–Horne–Zeilinger state.

```
1   OPENQASM 2.0;
2   include "qelib1.inc";
3
4   qreg q[3];
5   creg c[3];
6
7   h q[0];
8   cx q[0],q[1];
9   cx q[1],q[2];
10
11  measure q -> c;
```

Figure 4.1 QASM Greenberger–Horne–Zeilinger state

These instructions correspond to this diagram constructed on the Circuit Composer.

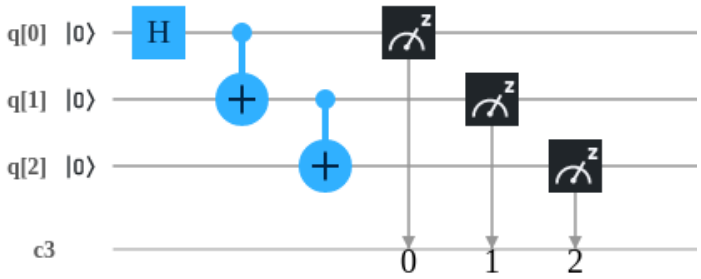


Figure 4.2 Greenberger–Horne–Zeilinger state

Now for the results, we first look at the histogram generated by the simulator.

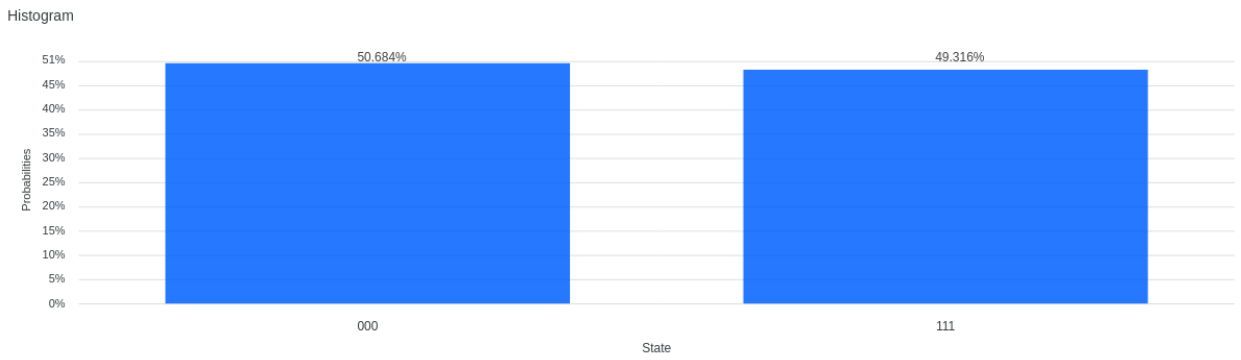


Figure 4.3 IBM Q Experience - Simulator

And here is the histogram generated by one of IBM's quantum computers.

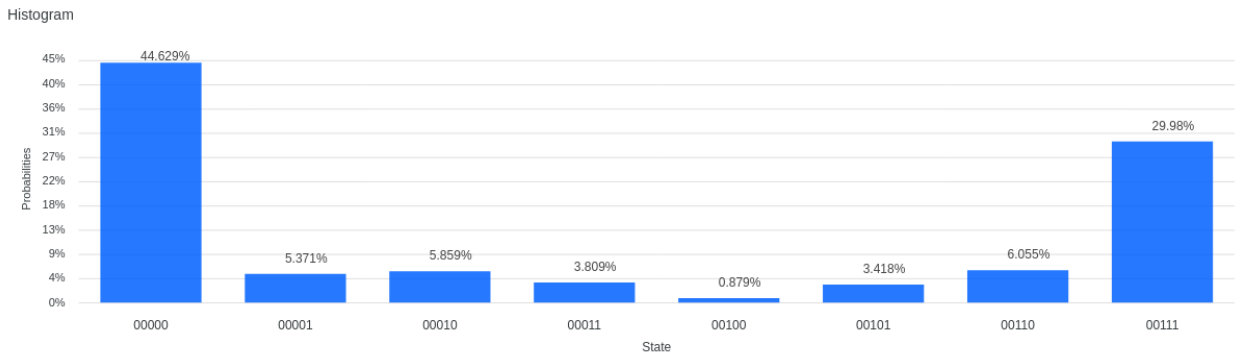
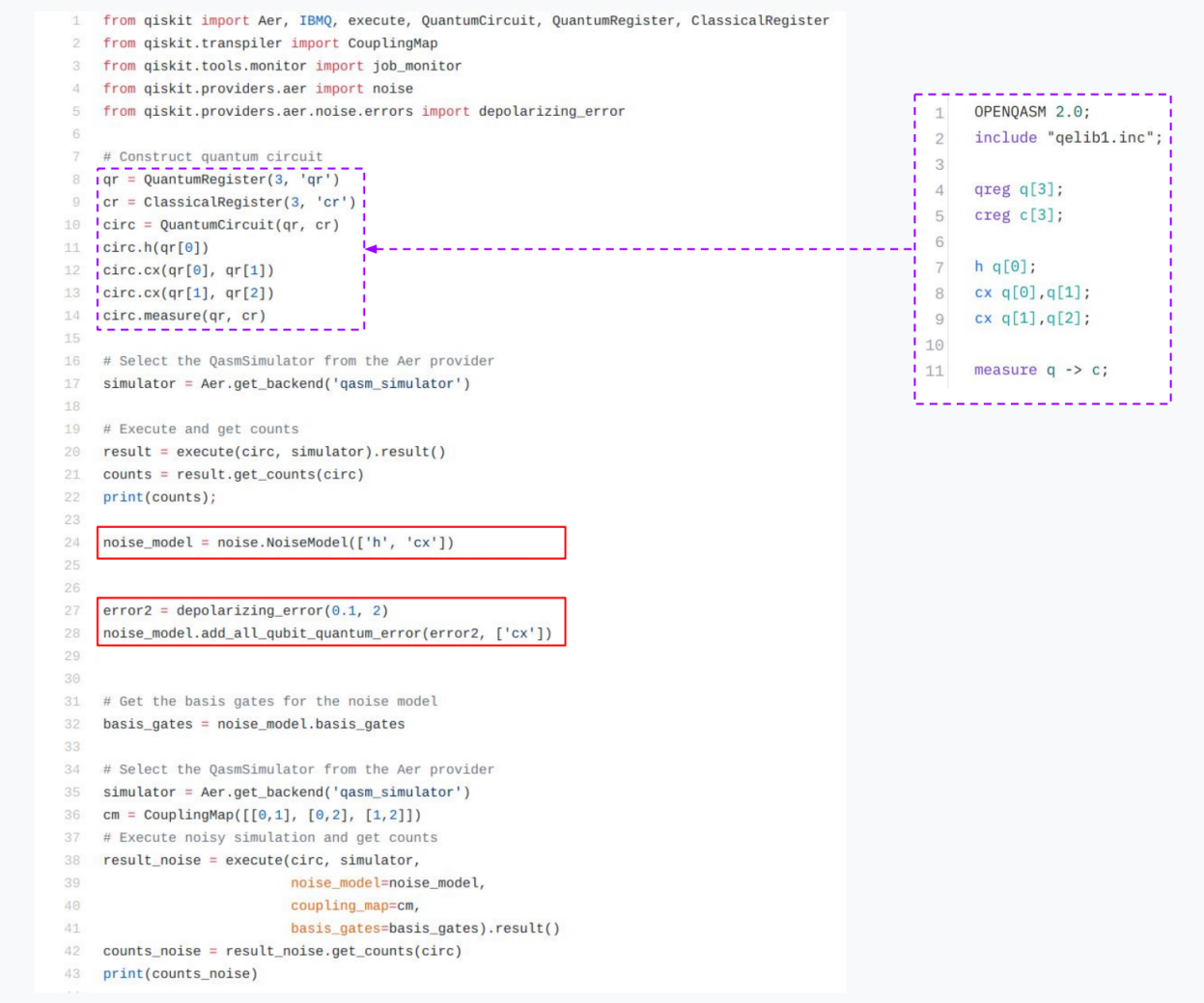


Figure 4.4 IBM Q Experience - Quantum Computer

It is not difficult to see that the histogram of the simulator and of the quantum computer are very different. This is because the simulator generates a theoretically perfect result whereas the quantum computer *suffers* from error. The first question to be answered is to figure out the ways to simulate the error of a quantum computer with the simulator on a classical computer. Clearly, we can simulate quantum computation on a classical computer (up to some threshold of qubits) however, the distinction here is to be able to mimic the error displayed by the quantum computer.

5 Simulating Noise and Error

Lucky for us, Qiskit enables us to do just that. Using a so called "Noise Model" and "depolarizing error" we can select which gates to add noise to and how much. This is useful for our purpose. The purpose is to illustrate that for quantum circuits with few qubits, we can produce results that would be difficult to distinguish coming from a true quantum computer. Below (Figure 5.1) illustrates the equivalent python program for the quantum circuit of the GHZ state. In this figure one can see that noise is added to the Haddamard and CNOT gates. There is also additional error added to the CNOT gates. In the next section we will discuss and compare the results constructed on Circuit Composer run on one of IBM's quantum computers and results of the previously displayed python program being run on a simulator on a local computer.



```
from qiskit import Aer, IBMQ, execute, QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit.transpiler import CouplingMap
from qiskit.tools.monitor import job_monitor
from qiskit.providers.aer import noise
from qiskit.providers.aer.noise.errors import depolarizing_error

# Construct quantum circuit
qr = QuantumRegister(3, 'qr')
cr = ClassicalRegister(3, 'cr')
circ = QuantumCircuit(qr, cr)
circ.h(qr[0])
circ.cx(qr[0], qr[1])
circ.cx(qr[1], qr[2])
circ.measure(qr, cr)

# Select the QasmSimulator from the Aer provider
simulator = Aer.get_backend('qasm_simulator')

# Execute and get counts
result = execute(circ, simulator).result()
counts = result.get_counts(circ)
print(counts);

noise_model = noise.NoiseModel(['h', 'cx'])


error2 = depolarizing_error(0.1, 2)
noise_model.add_all_qubit_quantum_error(error2, ['cx'])


# Get the basis gates for the noise model
basis_gates = noise_model.basis_gates

# Select the QasmSimulator from the Aer provider
simulator = Aer.get_backend('qasm_simulator')
cm = CouplingMap([[0,1], [0,2], [1,2]])
# Execute noisy simulation and get counts
result_noise = execute(circ, simulator,
                       noise_model=noise_model,
                       coupling_map=cm,
                       basis_gates=basis_gates).result()
counts_noise = result_noise.get_counts(circ)
print(counts_noise)
```

```
OPENQASM 2.0;
include "qelib1.inc";

qreg q[3];
creg c[3];

h q[0];
cx q[0],q[1];
cx q[1],q[2];

measure q -> c;
```

Figure 5.1 GHZ State With Simulated Error Added

6 Results - Simulating Noise and Error

Remember that in the probability distribution as seen in the histogram of Figure 4.3 there were only two states. Now, while still using a simulator on a classical computer with custom added noise, we were able to mimic the probability distribution of results generated by a quantum computer of the GHZ state. To an untrained eye, the results could, however, seem equivalent. The basis of this paper is to illustrate that with programming tools of today and a limit to the number of qubits used in a quantum circuit not only can we simulate and generate results that are theoretically perfect but we can ALSO simulate error in the results of the computation. To further these findings, we need to develop heuristics or conventions to simply generate weighted histograms randomly without the need to simulate an actual quantum circuit.

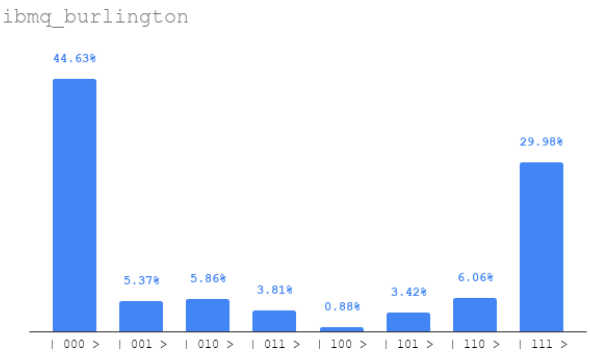


Figure 6.1 Results from IBM quantum computer

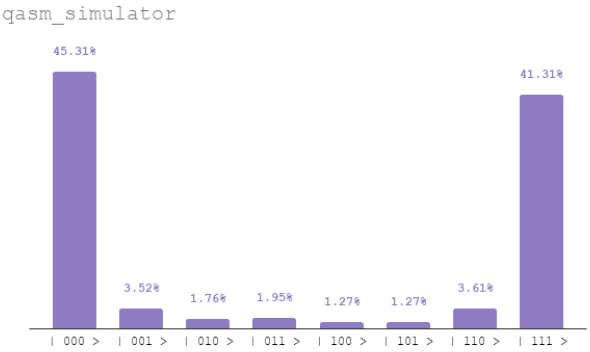


Figure 6.2 Results from Local Computer

7 Future Work

First, find how can we distinguish the results of the simulator with fake error and the results of the quantum computer. Next, if we can readily see that the simulator does not exhibit true randomness then, how can we alter the error generated to make the simulator and QC results nearly identical. Lastly and most importantly, provided that we know the number of states we should find in a given measurement, can we just randomly generate a probability distribution based on convention or heuristics.

8 Figures

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[4];
creg c[4];
x q[0];
x q[2];
barrier q;
h q[0];
cu1(pi/2) q[1],q[0];
h q[1];
cu1(pi/4) q[2],q[0];
cu1(pi/2) q[2],q[1];
h q[2];
cu1(pi/8) q[3],q[0];
cu1(pi/4) q[3],q[1];
cu1(pi/2) q[3],q[2];
h q[3];
measure q -> c;
cx q[1],q[2];
u1(-pi/4) q[2];
cx q[1],q[2];
u1(pi/4) q[2]; // end cu1
h q[2];

```

Figure 3.3 QASM for the Quantum Fourier Transform

References

[1] Qiskit *Introducing Qiskit Aer: A high performance simulator framework for quantum circuits* 2018, Retrieved from https://medium.com/qiskit/qiskit-aer-d09d0fac7759

[2] Gil Kalai: *The Argument against Quantum Computers*, 2019; arXiv:1908.02499.

[3] Zhao-Yun Chen, Qi Zhou, Cheng Xue, Xia Yang, Guang-Can Guo: *64-Qubit Quantum Circuit Simulation*, 2018, Science Bulletin, 2018, 63(15):964-971; arXiv:1802.06952. DOI: 10.1016/j.scib.2018.06.007

[4] IBM Q Experience, https://quantum-computing.ibm.com/support/guides/introduction-to-quantum-circuits?section=5cae613866c1694be21df8cc

[5] Andrew W. Cross, Lev S. Bishop, John A. Smolin: *Open Quantum Assembly Language*, 2017; arXiv:170703429

[6] CouplingMap "0.13.0" documentation, Retrieved from https://qiskit.org/documentation/api/qiskit.transpiler.CouplingMap.html

[7] Atkin, S. (2018, December 21). Demystifying Superdense Coding. Retrieved from https://medium.com/qiskit/demystifying-superdense-coding-41d46401910e

[8] Qiskit qiskit terra, Retrieved from https://github.com/Qiskit/qiskit-terra/blob/master/examples/python/qft.py

[9] NoiseModel — Qiskit "0.13.0" documentation. Retrieved from https://qiskit.org/documentation/api/qiskit.providers.aer.noise.NoiseModel.html