Yehuda Naveh Follow
Feb 25 · 6 min read

Classical Simulators for Quantum Computers

Gadi Aleksandrowicz, Yael Ben-Haim, Yehuda Naveh, Jay M. Gambetta

Why use classical simulators?

Nowadays we have actual quantum computers that can be accessed via a cloud service, so why do we need classical simulators to simulate quantum computers? After all, if quantum computers could be efficiently simulated on classical computers we wouldn't need quantum computers at all.

The thing is, quantum computer time is a scarce resource. When building and testing a new quantum circuit or algorithm, we want to run it many times, to verify things work. We want to test it over various noise scenarios. We don't want to wait until the quantum computer is available after every change we make in the circuit we're building. And we certainly can't control the noise in a real quantum machine. And of course, current quantum machines are limited in the number of qubits and the noise levels they provide.

Which simulators to use?

Writing a simple quantum computing simulator is easy; when first learning quantum computing, we can think "Oh, the mathematical formulation is actually very easy! We can write a short script in Python which performs quantum computing right now!" and indeed, if we want to adopt the simplest viewpoint possible, we can think of quantum computing as nothing but "a vector is multiplied by a sequence of matrices"—where the (2^n)-sized vector represents our n-qubit quantum state at any point in time, and the matrices represent the quantum gates composing the circuit. However, our needs are rarely this simple.

We may want to capture "snapshots" of the quantum states in the middle of computations; we may want to perform measurements and determine the rest of the circuit according to the result; we may want to simulate quantum noises, to see how sensitive our algorithm is to noise (or how well it manages to fix it). Maybe we want the density operator of the system and not only the result of a specified number of measurements. And obviously, we want it to be fast and consume as

small amounts of classical memory as possible. We can achieve all of this, but usually one comes at the expense of the other.

Let us consider two examples. If a circuit is composed exclusively from so-called Clifford gates (everything that can be built from the Hadamard, Phase, CNOT, and Measurement gates) then the Gottesman–Knill theorem tells us how to simulate this circuit on a classical computer in polynomial time. The simulator ("stabilizer simulator") uses a vastly different representation of the quantum state —instead of storing a large vector, it stores the generators of a group stabilized by that state vector. This representation is very efficient, both memory- and time-wise; but it can only represent a very restricted set of quantum states—ones obtained from |0>^n by the operation of Clifford gates only.

For the second example, let's look at the next section

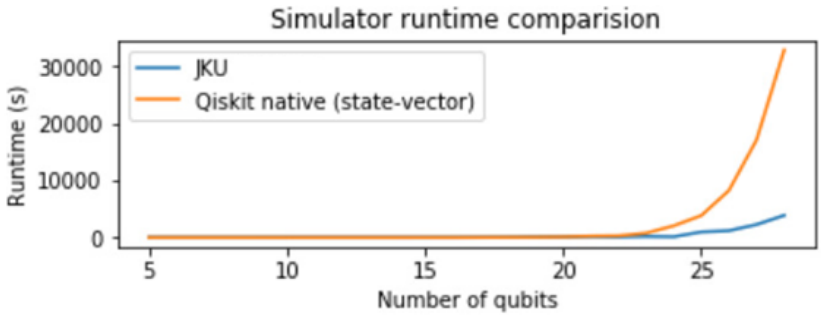
The JKU simulator

Qiskit is designed to include third-party simulators contributed by the quantum community at large. A prime example of such a simulator is the one created by Alwin Zulehner and Robert Wille from the quantum computation team at Johannes Kepler University in Linz, Austria. Let's call this the JKU simulator. This simulator stores quantum states using a data structure which is based on classical decision diagrams. This representation is more complex than simply storing a state vector, but if the vector has regular multiplicities, then it results in much more efficient storage space and manipulation time, while remaining able to deal with any quantum circuit, not limited, e.g., to Clifford gates.

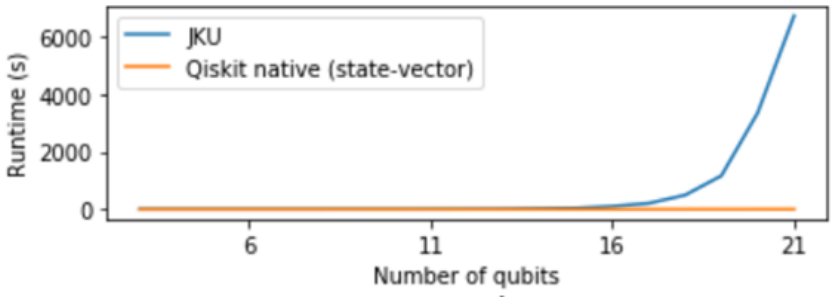
The JKU simulator gives better run-times on many circuits, even random ones; but on circuits which create extremely diverse quantum states with few repeating values its effectiveness diminishes compared to a standard, state vector simulator. Both simulators are outperformed by the stabilizer simulator, in the cases it can be used.

As we see, different simulators are suitable for different purposes. In fact, this is exactly the behavior we expect from a problem that is inherently exponential. So rather than sticking with any one simulation approach, we would like to have at our disposal a full portfolio of simulators.

Picture 1: Comparison of state vector and JKU simulators



(a) Random circuit



(b) Aqua VQE problem

How to use simulators?

But with a portfolio of different simulators, we don't want to download each simulator separately, and then learn how to use it and interface to it manually; we want things to just work, and we want them to work with the same look-and-feel, the same interface. Enter Qiskit. Using Qiskit, writing a quantum circuit is extremely simple—simply create a Python code doing what you want, or write a QASM file and load it.

Tinkering with the configuration is easy. Adding noises is easy. Sending the circuit to a real quantum computer is easy—and running simulators is easy: create the circuit you want, decide on your backend simulator, call "execute", and you are done. Under the covers, a very different simulation scheme can run. But from your point of view, all you needed to change is the name of the simulator.

Picture 2: Full code example of running Qiskit simulator backends

```
from qiskit import ClassicalRegister, QuantumRegister,
QuantumCircuit, execute
from qiskit_jku_provider import JKUProvider
JKU = JKUProvider()

qubits_num = 5
```

```
q = QuantumRegister(qubits_num, "q")
c = ClassicalRegister(qubits_num, "c")
qc = QuantumCircuit(q, c, name='ghz')

# Create a 5-qubit GHZ state
qc.h(q[0])
for i in range(qubits_num-1):
qc.cx(q[i], q[i+1])
# Insert a barrier before measurement
qc.barrier()
# Measure all qubits in the standard basis
for i in range(qubits_num):
qc.measure(q[i], c[i])

jku_backend = JKU.get_backend('qasm_simulator')
job = execute(qc, backend=jku_backend, shots=1024, seed=42)
result = job.result()
print(result.get_counts())
```

Being able to run every simulator you want is only part of the story; most of the time we **don't** want to run a specific simulator—we provide the circuit, and we'd like someone to choose the best simulator for us. Or maybe switch to another simulator if the run on one simulator takes too long or explodes in memory. At this point in time, you may run a simple script that does just that. As additional simulation schemes become part of Qiskit, Qiskit will also provide the smart way of doing such portfolio choices on any relevant subsets of the simulators.

In many cases we don't create circuits ourselves; for example, the Qiskit Aqua package provides a set of advanced algorithms using a quantum simulator (or a real quantum computer) as one major building block. When we run such algorithms, the simulators are called on the complex quantum circuits created automatically by the algorithms. Here again, having multitude simulation schemes can help find the best approach to solve any Aqua problem we may wish to experiment.

How to make your simulator a Qiskit simulator?

Suppose you have written a wonderful new classical simulator of a quantum computer and want people to use it. Your next step is to contact the Qiskit team. Then we may together follow the JKU simulator path to success: Us at the Qiskit team collaborated with the JKU creators in adding the JKU simulator as a Qiskit provider simulator. Qiskit users can now seamlessly enjoy the smartness of the simulator's

algorithm and data structures with the ease of use and flexibility of Qiskit. The JKU team in return increases their opportunity to create impact, while retaining all credit for the cases their simulator was able to solve problems which were hard for other simulation schemes.

As is usual for exponential problems, the JKU simulation approach can only be good on a limited subset of circuits. So if your simulator works in sufficiently different ways than existing simulators in Qiskit, we have all reasons to hope that it will be the winner on a different subset of circuits. We would love to have it then as part of the Qiskit simulation portfolio. We invite you to repeat the success of the JKU/Qiskit framework. Submit your simulator as a Qiskit provider by following the scheme outlined in this tutorial, and help cover as much of the quantum circuit simulation space as possibly can.

Ready to get started?

To get started with Qiskit, click here to find the installation and setup guides, as well as many tutorials, including the simulator example described above. We look forward to welcoming you to the Qiskit contributors team!