Quantum Simulator Report

Rishab Khurana

December 2023

1 Design

To parameterize the solution in the number of qubits, I created a vector of size 2^n with initial state $[1+0i, 0+0i, 0+0i, ..., 0+0i]$, representing $|00...00\rangle$. The index (zero-indexed) of each element represents a potential state of the system, and the value at the index represents the coefficient of that state (so the norm squared of the coefficient is the probability of getting that state). Because this vector is the representation of qubits for which we can apply unitaries, I just directly apply a unitary to this vector to get the next state of the system when parsing a QASM file. However, before applying a unitary U to register i, I needed to tensor it with $I^{\otimes(i-1)}$ from the left and $I^{\otimes(n-i)}$ from the right (n is the number of qubits in the system). This is because we need a $2^n \times 2^n$ matrix to step to the next state for the 2^n vector – Moreover because apply U to register i, we apply I to all other registers other than i. Below is the code that initializes the state:

```
self.num_qubits = int(qubits)

#initial state of the system is all zeros ==> |0> x |0> x |0> x |0> x .... x |0>  = (1, 0, 0, 0, ..., 0):
self.qubits_arr.append(1 + 0j)
for i in range(1, 2**(self.num_qubits)):
    self.qubits_arr.append(0 + 0j)
```

And below is the code the applies a single-qubit gate to the state of the system:

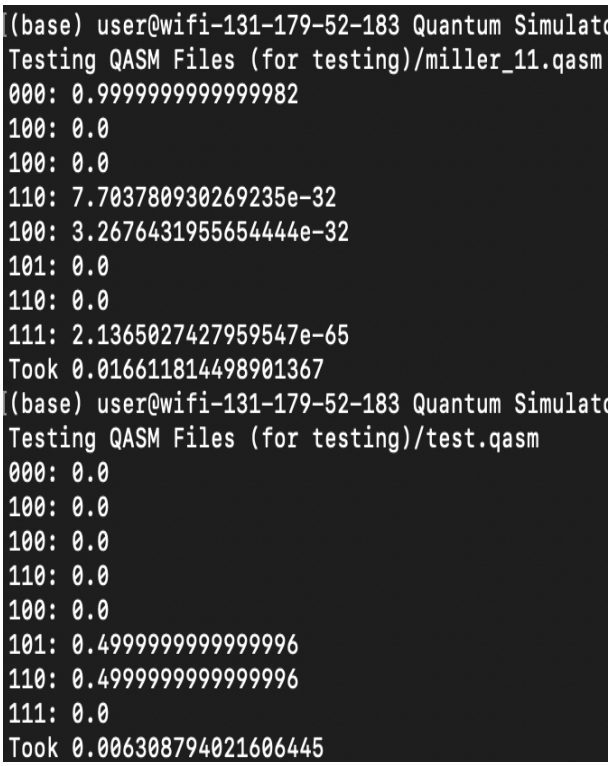
```
def apply_single_qubit_gate(self, gate, reg):
    #Input: A single qubit gate and a qubit register
    #Output: The state of the system with the gate applied to the qubit
    GATE = [[1 + 0j]]
    for i in range(reg):
        GATE = self.tensor_product(GATE, self.I)
    GATE = self.tensor_product(GATE, gate)
    for i in range(reg + 1, self.num_qubits):
        GATE = self.tensor_product(GATE, self.I)
    self.qubits_arr = np.dot(GATE, self.qubits_arr)
```

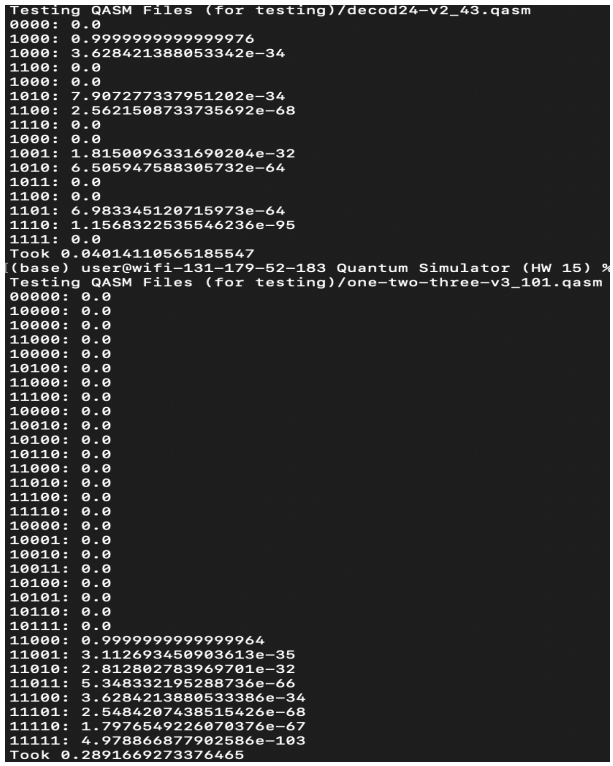
Using this representation of the system, the QASM file just needs to specify the number of qubits and everything is taken care of in the code by just applying these unitaries to the

state vector.

2 Evaluation

I tested my simulator on several of the QASM files provided. However, I needed to change the qubits specified in those files to the minimum number of qubits needed because 16 qubits timed out – this is likely because tensor products take $O(i^2 j^2)$ time for tensoring two square matrices of size $i \times i$ and $j \times j$. Moreover, we are tensoring a matrix of size 2^k for $k < n$ (where n is number of qubits), and a 2×2 for each operation, meaning the upper bound is $O(2^{2n} \cdot 2^{2n}) = O(2^{4n})$ for just tensoring matrices. Then, for each operation, we get $O(l \cdot 2^{4n})$, where l is the amount of operations (lines) in the QASM file. Thus, we have on the order of 2^{64} code operations for a 16 qubit circuit. On the files I did test, though, I seemed to get promising results:





All the probabilities line up and look correct in accordance with the Cirq simulator. As you can see execution times were quite fast for 1-5 qubits; the above tests along with more of my own tests yield that:

$$n = 3 \rightarrow \approx 0.01 \; secs$$

$$n = 4 \rightarrow \approx 0.04 \; secs$$

$$n = 5 \rightarrow \approx 0.28 \; secs$$

$$n = 6 \rightarrow \approx 1.2 \; secs$$

$$n = 7 \rightarrow \approx 3.4 \; secs$$

$$n = 8 \rightarrow \approx 12.88 \; secs$$

$$n = 9 \rightarrow \approx 50 \; secs$$

After $n = 9$, the simulation took too long. Below is the test output for $n = 9$ (I tested with the same file used for $n = 4$, but I forced the number of qubits to be 9, so I am assuming it would take even longer if all 9 qubits were used):

```
111001010: 0.0
111001011: 0.0
111001100: 0.0
111001101: 0.0
111001110: 0.0
111001111: 0.0
111010000: 0.0
111010001: 0.0
111010010: 0.0
111010011: 0.0
111010100: 0.0
111010101: 0.0
111010110: 0.0
111010111: 0.0
111011000: 0.0
111011001: 0.0
111011010: 0.0
111011011: 0.0
111011100: 0.0
111011101: 0.0
111011110: 0.0
111011111: 0.0
111100000: 0.0
111100001: 0.0
111100010: 0.0
111100011: 0.0
111100100: 0.0
111100101: 0.0
111100110: 0.0
111100111: 0.0
111101000: 0.0
111101001: 0.0
111101010: 0.0
111101011: 0.0
111101100: 0.0
111101101: 0.0
111101110: 0.0
111101111: 0.0
111110000: 0.0
111110001: 0.0
111110010: 0.0
111110011: 0.0
111110100: 0.0
111110101: 0.0
111110110: 0.0
111110111: 0.0
111111000: 0.0
111111001: 0.0
111111010: 0.0
111111011: 0.0
111111100: 0.0
111111101: 0.0
111111110: 0.0
111111111: 0.0
Took 48.613478899002075
```

Thus, there is a very clear trend that as qubits grow, the execution time increases exponentially, making it very hard to simulate anything over 10 qubits, which agrees with my analysis given in the beginning.

3 README

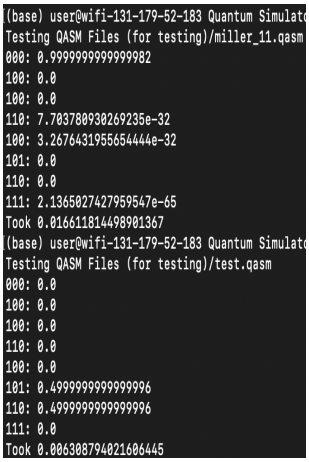
To execute the simulator, a few steps need to be taken. First, the user needs to install the python numpy library by execution

pip install numpy

in their terminal. After this is installed, one needs to create a folder with the Simulator.py script (my simulator) and the QASM files they want to test on. Then, in terminal and under the directory with Simulator.py and the QASM files, the user must execute the following command:

python3 Simulator.py "QASM_filename".qasm

Where "QASM_filename" is the name of the QASM file the user is trying to simulate. The desired output should be in the terminal. Note that the output is formatted with the least significant bit on the very right. So, for the below example,



the state reg 0 = 1, reg 1 = 0 and reg 2 = 1 has a 50 percent chance of being outputted for test.qasm. And the state reg 0 = 1, reg 1 = 1, and reg 2 = 0 has a 50 percent chance of being outputted for test.qasm