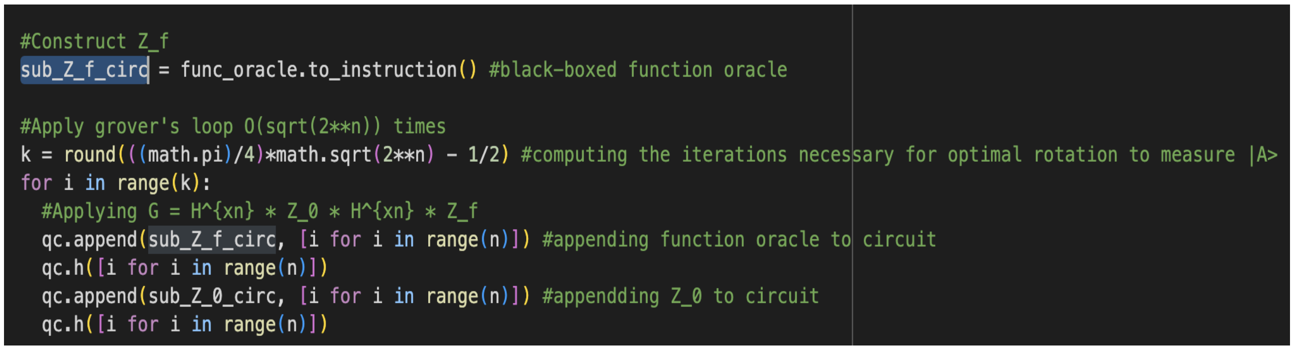
Four Algorithms in Qiskit

Rishab Khurana

November 2023

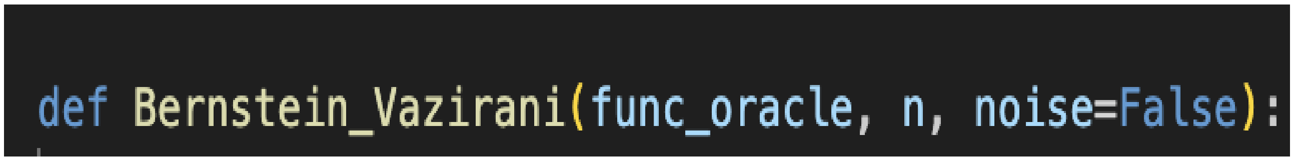
1 Design

The black-box function U_f/Z_f was implemented by it being taken as input as a QuantumCircuit object to each algorithm. Then, using the Qiskit library, I converted it to a "sub-circuit" and appended it to the algorithm's circuit when needed. For example, for Grover's Algorithm, I had the following code snippet to implement the function oracle Z_f (where "func_oracle" is the QuantumCircuit representing Z_f):



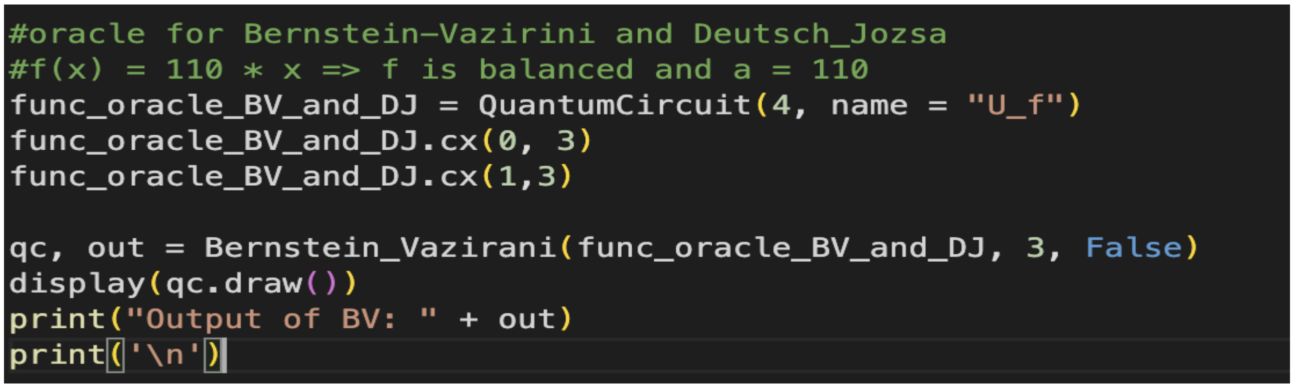
```
#Construct Z_f
sub_Z_f_circ = func_oracle.to_instruction() #black-boxed function oracle

#Apply grover's loop O(sqrt(2**n)) times
k = round(((math.pi)/4)*math.sqrt(2**n) - 1/2) #computing the iterations necessary for optimal rotation to measure |A>
for i in range(k):
  #Applying G = H^{xn} * Z_0 * H^{xn} * Z_f
  qc.append(sub_Z_f_circ, [i for i in range(n)]) #appending function oracle to circuit
  qc.h([i for i in range(n)])
  qc.append(sub_Z_0_circ, [i for i in range(n)]) #appendding Z_0 to circuit
  qc.h([i for i in range(n)])
```

Observe that in Grover's Loop (applying G $O(\sqrt{2^n})$ times), the black boz Z_f is turned into a sub-circuit using the to_instruction() method and then it is appended to the circuit using the append() method. This is how U_f/Z_f is implemented for all algorithms – using the to_instruction() and append() method from Qiskit. Below is an example of a header (Bernstein-Vazirini's header, but every algorithm had the same header) in which I take func_oracle as an input:



```
def Bernstein_Vazirani(func_oracle, n, noise=False):
```

Then, after creating each algorithm, I tested them by constructing such function oracles (as QuantumCircuits) and sending them as input. Below is an example of the construction of a function oracle for BV and is passed as an input to the Bernstein-Vazirini algorithm:



```
#oracle for Bernstein-Vazirini and Deutsch_Jozsa
#f(x) = 110 * x => f is balanced and a = 110
func_oracle_BV_and_DJ = QuantumCircuit(4, name = "U_f")
func_oracle_BV_and_DJ.cx(0, 3)
func_oracle_BV_and_DJ.cx(1,3)

qc, out = Bernstein_Vazirani(func_oracle_BV_and_DJ, 3, False)
display(qc.draw())
print("Output of BV: " + out)
print('\n')
```

For parameterizing the number of qubits, I took n as an input to each algorithm and constructed the generalized quantum circuit for each algorithm. For example, for Simon's circuit, I constructed a $2n$-qubit quantum circuit with n classical bits. The first n qubits were then sent through the gates required for Simon's algorithm and measured into the n classical bits, and the second n qubits were helper bits. The only thing making the parameterization tricky would have been U_f/Z_f for each circuit, but that issue was solved by taking the oracles as inputs and assuming they took in the assumed amount of qubits.

2 Evaluation

Bernstein-Vazirani and Deutsch-Jozsa have nearly identical code with one minor difference – in accordance with the algorithms, Bernstein-Vazirani has n classical bits and it measures the first n qubits and outputs those, whereas Deutsch-Jozsa just measures the first bit; this created minor differences in the initialization of the algorithm and the measurement, but other than that, the two are nearly identical. Simon is quite similar to Bernstein-Vazirani, with the exception that there are n helper bits rather than just 1 (so there are $2n$ total qubits in Simon's) – this is to support the different oracle U_f for Simon's; Moreover, Simon's has less applications of the Hadamard gate; this means that the only difference from Simon's to Bernstein-Vazirani is the initialization and the body is ever so slightly different – just changes in the bounds for the for loop that applies hadamards. The most different algorithm from the rest is Grover's. This is because Grover's is a fundamentally different algorithm (it is iterative), and has the implementation of Z_0 as well. All the algorithms other than Grover had around 45 lines of about 90% the same implementation, whereas Grover had around 60 lines of maybe 40% the same implementation as the rest of them – the only similarity of

Grover to the rest is the initial application of the Hadamards, the measurement of the first n qubits, and the black-box use of a function oracle. All in all, Bernstein-Vazirani, Deutsch-Jozsa, and Simon's have a lot of code reusability, but for Grover's I essentially needed to write the whole algorithm from scratch.

To test the four programs, I created oracles for different functions for input to each algorithm and tested under the Aer simulator for accuracy (make sure my algorithms were correct). For Deutsch-Jozsa and Bernstein-Vazirani, I created oracles for several functions of the form $f(x) = a \cdot x$ and made sure that Deutsch-Jozsa outputted whether or not it was balanced or constant with 100% probability, and Bernstein-Vazirani outputted a with 100% probability. For Simon's, I made oracles for several functions that are 2-to-1 (mainly two and three-qubit functions) and made sure that outputs were always orthogonal to the s (0 dot product with s) for each function. For Grover's I implemented Z_0 as a unitary matrix in the actual algorithm and fed Z_f as input to the function – I constructed Z_f for functions of f of the form $f(x_0) = 1$ for some x_0 and $f(x) = 0$ for every other $x \neq x_0$, and I tested whether or not x_0 was output with high probability. Note that most of these algorithms are deterministic in the sense that every time I ran them, the output needed to satisfy some criteria – for DJ and BV, the output needed to be the answer; for Simon's the output needed to be orthogonal to s. The only exception was Grover's, which outputs the correct output with high probability for $n > 2$; for $n = 2$, it outputs the correct output with 100% probability. Thus, to test this I made sure for $n = 2$, the correct output was always measured, and for $n > 2$, I made sure that the correct output was measured *most* of the time – i.e. 9/10 times. For execution times, I tested on a noise simulator, and the execution times varied greatly from run to run, but I noticed about a 0.35-second execution time for BV and DJ, a 0.25-second execution time for Simons, and a 0.3-second execution time for Grover's; I would say all 4 algorithms had fairly similar execution times (none of the four stood out too much).

Noise made a significant impact. Using the FakeManila noise simulator, even the deterministic algorithms (DJ, BV) outputted incorrect results quite frequently – about 4 out of 10 times, the incorrect result was given for DJ and BV. For Simon's Algorithm, the error rate was the least of all other algorithms, but I think this is because the amount of acceptable outputs is much higher with Simon's (DJ, BV, and Grover's all expect one output, whereas Simon's expects all outputs orthogonal to s). For Grover's with $n = 2$, there was about a 3 for 10 error rate in my testing with noise – but I wouldn't consider this a statistically significant difference from the DJ and BV error rates.

Scalability for qubits can be seen in the following diagram for DJ/BV and $a = 10^{n-1}$ (i.e.

$f(x) = 10^{n-1} \cdot x$, which is a balanced function for DJ and $a = 10^{n-1}$ for BV):

$$1\ qubit \rightarrow 0.2\ secs\ execution$$

$$2\ qubit \rightarrow 0.3\ secs\ execution$$

$$3\ qubit \rightarrow 0.5\ secs\ execution$$

$$3\ qubit \rightarrow 0.6\ secs\ execution$$

It seems that under my testing conditions (note that I am using a fake noise simulator due to the time restrictions of using an actual quantum computer), the execition time is growing linearly with the number of qubits.

3 Instructions/README

To run the algorithms, you need to specify the number of qubits you want and construct an oracle U_f/Z_f for each algorithm – you can construct this oracle using a computer or manually. For DJ and BV, your oracle must satisfy

$$U_f|x\rangle|b\rangle = |x\rangle|b \oplus f(x)\rangle, \forall x \in \{0,1\}^n$$

For Simon's your oracle is the same as for DJ and BV except $b \oplus f(x) \in \{0,1\}^n$ instead of $\{0,1\}$ because $b, f(x) \in \{0,1\}^n$ for Simon's. For Grover's, your oracle must me

$$Z_f|x\rangle = (-1)^{f(x)}|x\rangle, \forall x \in \{0,1\}^n$$

Once you compute these oracles, create their implementation as a QuantumCircuit object in Qiskit (you can see an example of this implementation for the BV algorithm in the Design section), and then send the oracle as input to the algorithm you want. Each algorithm has the first input as the number of qubits n, the second input as function oracle, and the third input as weather or not you want noise. If you choose to use noise, then a noise simulator is used due to the large queue time of running on an actual quantum computer.