Grover's Algorithm in Qiskit, Cirq, Q#, and Silq

Brian Goldsmith

Computer Science, Vanderbilt University
Nashville, United States of America
brian.d.goldsmith@vanderbilt.edu

***Abstract*— After writing Grover's algorithm for a 2 qubit system in Qiskit, Cirq, Q#, and Silq, I could see differences and similarities in these languages. It was especially helpful to compare the files in a stripped down version with comments and extra whitespaces removed. In this paper, I will review my observations of the languages as well my preferences.**

I. Introduction

This course has been a difficult and interesting journey and this final project is very fitting. I chose to write Grover's algorithm in Qiskit, Circ, Q#, and Silq. I chose four languages because for the first programming assignment, I did Grover's algorithm in Qiskit. Since I was reusing that code, I decided adding another language would be useful. When we first went over Grover's algorithm in class, I understood how the algorithm could be applied and how it was useful. However, I did not understand the programming or math behind the algorithm. While completing the first programming assignment, I spent a great deal of time going through the algorithm and code. At the end of the assignment, I was still confused about oracles, but I did have a better understanding of the math for Grover's algorithm thanks to the fantastic Qiskit learning materials. Now having completed Grover's algorithm in multiple languages, I would venture to say that I understand Grover's algorithm for a 2 qubit system. If I expand beyond a 2 qubit system to 3 qubits or n qubits, I would need to analyze the oracle and reflection to determine the appropriate gates. While this is not out of reach, I would need to work closely with the mathematics to determine which gates would be needed and how to arrange them for uncomputation.

II. The Languages

An easy way to comply with the conference paper formatting requirements is to use this document as a template and simply type your text into it.

A. Qiskit and Cirq

After writing Grover's algorithm in Qiskit and Cirq, I could see the similarities between them. Both languages operate at a low level and build the circuits by appending the different gates together. Because of these similarities, I have combined them into a single section. While programming Grover's algorithm is a limited test, the two languages seem to be very close. The design and implementation of the languages are similar and switching between the languages does not appear to be difficult.

B. Q#

Once I started working in Q#, there was a noticeable change with the use of within and apply. Q# attempts to advance to a mid level quantum computing language. Instead of appending gates to build out a circuit, gates are simply called with the understanding that a circuit is being generated. The within and apply functionality helps with uncomputation and with some practice, I would imagine that its usage would become second nature.

C. Silq

Finally, I used Silq which is a newer language which attempts to be a high level quantum programming language. I wrote my first paper in this class on Silq and was excited to implement Grover's algorithm in the language now that I have a better understanding of quantum computing and Grover's algorithm. There is definitely a higher level feel to Silq since qubits are not specifically defined or used. Gates are used, however for controlled gates an "if" statement is used. While uncomputation is handled automatically in Silq, one must still have the understanding of how and when it is used in order to produce error free code.

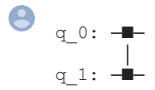
III. Conclusion

After this brief introduction to quantum programming in the different languages, I have found one consistent principle: to successfully program for quantum computers, one must understand the mathematics behind the code. While having a programming background is useful, a quantum programming language does not exist where the mathematics behind the code can be ignored. Personally, I prefer the low level languages of Qiskit and Circ because everything is very clear and straightforward. The language is not trying to provide shortcuts or do operations behind the scenes. In Q#, the use of within and applies is trying to be useful, but it ends up adding its own complexity. If a developer is experienced enough to understand the math and what the code needs to do, they do not need the assistance with uncomputation. This applies to Silq as well, since each shortcut the language tries to provide, at least to start, requires the developer to figure out what Silq is doing automatically. Since Silq aspires to be a high level language, not only does it attempt to do uncomputation automatically, but it introduces other concepts like qfree, mfree, and specifying quantum or classical types. Perhaps an experienced quantum programmer would find the shortcuts beneficial and perhaps in larger or more complex algorithms

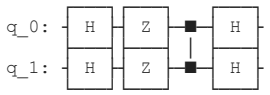
the benefits would truly shine. However, from a beginner's standpoint, I am not able to value the shortcuts since I still need to know the how/why the shortcuts are used. Thus, I appreciate the lower level languages where I have to explicitly go step by step through the algorithm to generate the circuit.

```
# Brian Goldsmith
# Grover's Algorithm
# used the following video: https://www.youtube.com/watch?v=0RPFWZj7Jm0

from qiskit import *
import matplotlib.pyplot as plt
import numpy as np
```

```
# define the oracle circuit to be used later
# first create a quantum circuit with 2 qubits named oracle
oracle = QuantumCircuit(2,name='oracle')

# add a controlled z gate to only change |11>
oracle.cz(0,1)

# add a controlled x gate to only change |10>
#oracle.cx(0,1)

# save the component to a gate for later use
oracle.to_gate()

oracle.draw()
```



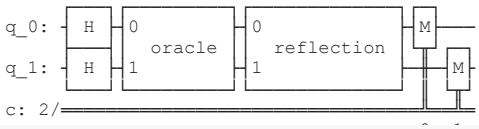

```
# create reflection quantum circuit with 2 qubits and named reflection
reflection = QuantumCircuit(2,name='reflection')

# apply a negative phase on only 00 state

# add a Hadamard gate
reflection.h([0,1])

# add a z gate
reflection.z([0,1])

# add a controlled z gate
reflection.cz(0,1)

# add another Hadamard gate
reflection.h([0,1])

# save the component to a single gate to use later
reflection.to_gate()

reflection.draw()
```




```
# create backend for simulation
backend = Aer.get_backend('qasm_simulator')

# create another quantum circuit (could have also used previous grover_circ)
grover_circ = QuantumCircuit(2,2)

# initialize the qubits with Hadamard gate to ensure superposition
grover_circ.h([0,1])

# add the oracle from above to switch the sign for |11>
grover_circ.append(oracle,[0,1])

# add the reflection from above to reflect/diffuse
# this means get the |11> probability higher
grover_circ.append(reflection,[0,1])

# add measurements for the qubits and save the info to the classical bits
grover_circ.measure([0,1],[0,1])
```

```
<qiskit.circuit.instructionset.InstructionSet at 0x15f04d0d0>
```

```
grover_circ.draw()
```




```
job = execute(grover_circ, backend, shots=1)
# save the results from the simulation
result = job.result()
result.get_counts()
```

```
{'11': 1}
```

```
try:
    import cirq
except ImportError:
    print("installing cirq...")
    !pip install --quiet cirq
    import cirq

    print("installed cirq.")
```

```
"""Get qubits to use in the circuit for Grover's algorithm."""
# Number of qubits n.
nqubits = 2

# Get qubit registers.
qubits = cirq.LineQubit.range(nqubits)
```

```
def make_oracle(qubits):
    # call a CZ gate
    yield cirq.CZ(qubits[0], qubits[1])
```

```
def grover_iteration(qubits, oracle):
    """Performs one round of the Grover iteration."""
    circuit = cirq.Circuit()

    # Create an equal superposition over input qubits.
    circuit.append(cirq.H.on_each(*qubits))


    # Query the oracle.
    circuit.append(oracle)

    # Construct Grover operator / Grover Diffuser
    circuit.append(cirq.H.on_each(*qubits))
    circuit.append(cirq.Z.on_each(*qubits))
    circuit.append(cirq.CZ(qubits[0], qubits[1]))
    circuit.append(cirq.H.on_each(*qubits))

    # Measure the input register.
    circuit.append(cirq.measure(*qubits, key="result"))

    return circuit
```

```
# Make oracle
oracle = make_oracle(qubits)

# Embed the oracle into quantum circuit with Grover's algorithm
circuit = grover_iteration(qubits, oracle)
print("Circuit for Grover's algorithm:")
print(circuit)
```

```
Circuit for Grover's algorithm:
0: ———H———@———H———Z———@———H———M('result')———
          │         │         │
1: ———H———@———H———Z———@———H———M———————————————
```

```
"""Simulate the circuit for Grover's algorithm and check the output."""
# Helper function.
def bitstring(bits):
    return "".join(str(int(b)) for b in bits)

# Sample from the circuit a couple times.
simulator = cirq.Simulator()
result = simulator.run(circuit, repetitions=1)

# Look at the sampled bitstrings.
frequencies = result.histogram(key="result", fold_func=bitstring)
print('Sampled results:\n{}'.format(frequencies))

# Check if we actually found the secret value.
most_common_bitstring = frequencies.most_common(1)[0][0]
print("\nMost common bitstring: {}".format(most_common_bitstring))
print("Found a match? {}".format(most_common_bitstring == bitstring([1,1])))
```

```
Sampled results:
Counter({'11': 1})

Most common bitstring: 11
Found a match? True
```

```
// Grover's alogrithm in Silq. This is a 2 qubit version of Grover's algorithm
// in Q#. The following references were used:
// https://www.youtube.com/watch?v=Q9O93Gavj1A
// https://learn.microsoft.com/en-us/azure/quantum/tutorial-qdk-grovers-search?tabs=tabid-visualstudio

open Microsoft.Quantum.Arrays;
open Microsoft.Quantum.Measurement;

operation Grovers() : Result[]{
    // using 2 qubits
    use qubits = Qubit[2];

    // apply Hadamard to assure superposition
    ApplyToEachCA(H, qubits);

    // number of iterations
    // since we are only using 2 qubits we do 1 iteration
    for _ in 0..0 {
        // apply Oracle
        Oracle(qubits);

        // apply reflection
        Reflection(qubits);
    }

    // since on a simulator there is no need to measure
    return ForEach(MResetZ, qubits);
}

operation Oracle(inputQubits : Qubit[]) : Unit {
    // Do a control Z gate to return |11> as the answer
    CZ(inputQubits[0], inputQubits[1]);
}

operation Reflection(inputQubits : Qubit[]) : Unit {
    within {
        // do a hadamard gate on each qubit which will be uncomputed
        Adjoint ApplyToEachCA(H, inputQubits);


    } apply {

        // do a Z gate on each quibit
        ApplyToEachCA(Z, inputQubits);
        // do a control Z gate to get the reflection
        CZ(inputQubits[0], inputQubits[1]);
    }
}
```

- Grovers
- Oracle
- Reflection

```
%simulate Grovers
```

- One
- One

final/Grovers_Silq.slq

```
/*
 * Grover's alogrithm in Silq. This is a 2 qubit version of Grover's algorithm
 * in Silq. The following references were used:
 * https://silq.ethz.ch/overview#/overview/2_grover
 * https://github.com/eth-sri/silq/blob/master/test/grover2.slq
 * https://www.youtube.com/watch?v=AMKYgINLUKw
 */

def grover[n:!N](f:const uint[n]!-> qfree B): !N{
    // hardcode to do 1 iteration since we know with 2 qubits 1 iteration is enough
    nIterations := 1;

    // this sets up an array of qubits (though it appears to be an array of ints
    // the default values are set to 0
    cand:=0:uint[n];

    for k in [0..n) {
        // for each qubit perform a Hadamard transform to put it in superposition
        cand[k] := H(cand[k]);
    }



    // for the number of specified iterations
    for k in [0..nIterations){
        // if the result of calling function f is true call a phase flip.
        //
        // if the result of calling "the oracle" is true call a phase flip
        //
        // if the correct search result is found, return true then call a
        // phase flip on the combination of qubits so that "candidate" (cand)
        // or answer has a different probability. In this case, the probability
        // drops to -0.5 and it is distinguished from the other options.
        //
        // function f does NOT consume cand. We know this because of "const" above
        // in (f:const int[n]!-> B)
        if (f(cand)) { phase(π); }


        // debug to dump the current state
        // this is helpful to see the quantum states BEFORE diffusion
        dump();


        // call helper method groverDiff which performs the diffusion reflection
        //
        // the call returns the new probabilities for all candidates where the
        // answer's probability is increased and non-answers' porbabilities are decreased
        //
        // groverDiff consumes cand and the result of the groverDiff call is then written
        // into the variable cand
        cand := groverDiff[n](cand);

    }

    // return the measurement as a natural number
    return measure(cand) as !N;
```

```
}

// function takes an array of integers and performs Hadamard and a controlled phase shift
// in order to relfect over s / do amplitude ampificatoin
def groverDiff[n:!N](cand:uint[n]) mfree: uint[n] {

    // call a Hadamard gate on each qubit to change from z to x basis
    for k in [0..n) { cand[k] := H(cand[k]); }

    // this does a Controlled Z gate with the "if" as the condition
    // then phase(π) as the Z gate
    if cand !=0 {
        phase(π);
    }

    // call a Hadamard again to change back from the x basis to the z basis
    for k in [0..n) { cand[k] := H(cand[k]); }

    return cand;
}

// this is the Oracle that returns a boolean TRUE for |11> or |3> because that is the
// "answer"
def f[n:!N](const cand:uint[n])qfree:B{
    return (cand = 3);
}

def main(){
    n := 2:!N;

    return grover[n](f[n]);
}
```