Introduction to IBM Quantum & Qiskit

Mohd Harith Akmal Zulfaizal Fadillah, Bahari Idrus, Mohammad Khatim Hasan

Centre for Artificial Intelligence Technology (CAIT), Faculty of Information Science & Technology, Universiti Kebangsaan Malaysia, 43600 Bangi, Selangor

Abstract

IBM Quantum (IBM-Q) is an online platform that provides quantum computing services that allow the public and researchers for developments and experimentations on quantum circuits, algorithms and applications based on the framework of Qiskit. We will see the Qiskit code for the common quantum gates and go through the construction and execution of quantum circuits on simulators and IBMQ machines using Qiskit in Jupyter in implementing the Deutsch-Josza's, Grover's, Shor's and Harrow-Hassidim-Lloyd (HHL) algorithm on IBM-Q and Qiskit.

1 Introduction

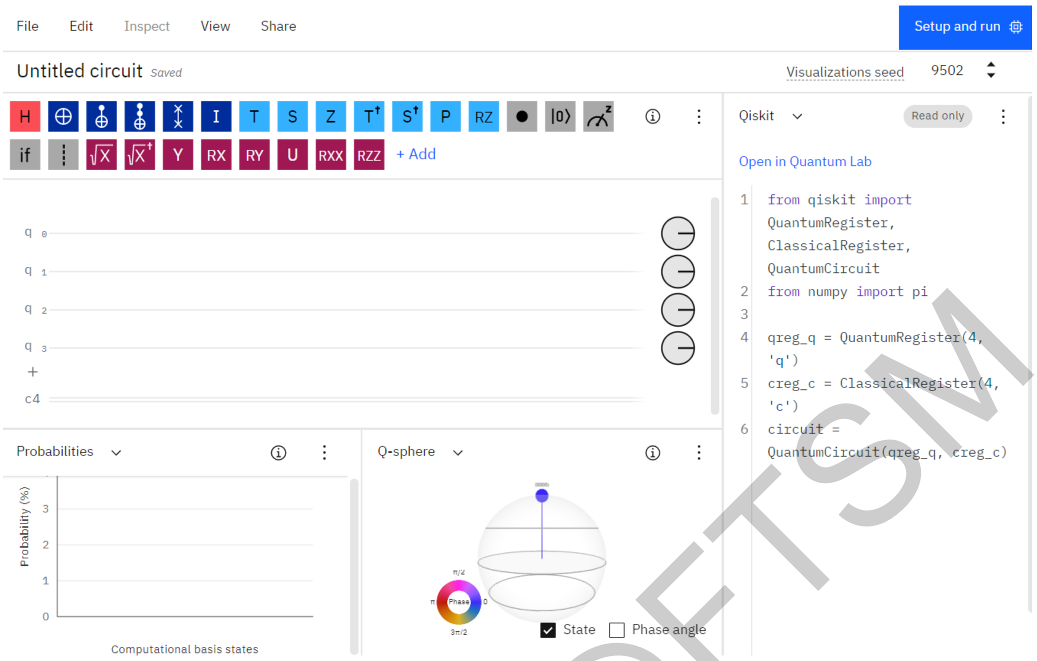
In the previous report, we have discussed on the fundamentals of quantum computing as well as have looked into several known quantum algorithms such as Deutsch-Josza's, Grover's, Shor's and Harrow-Hassidim-Lloyd (HHL) algorithm. We have studied the theoretical foundation for those algorithms and had some idea on what they do and the measurement that they give. In this report, we will implement those algorithms through IBM Quantum (IBM-Q), a platform for quantum computing founded by IBM and Qiskit which is the framework behind IBM-Q [1, 2].

From the theoretical understanding that we had on the algorithms, we will see how well it is translated into gate operations in the implementation and construction of quantum circuits which are much more complex than it seemed. These circuits of the algorithms that we will be constructing will be executed on both simulators and real quantum computers for us to observe the measurements that they yield and how well they compare to the theoretical results. Considerations for the various factors inherent to the quantum computers and from the circuits constructed will be explored with respect to the measurements that these computers output.

2 IBM-Q & Qiskit

IBM Quantum Composer and IBM Quantum Lab are the two online cloud-based quantum computing services that are offered by IBM [1]. They offer free public use of some of their own quantum machines to execute quantum circuits and algorithms alongside their private quantum computers reserved for researchers and partners. The quantum circuits can be constructed either graphically through IBM Quantum Composer or by writing Python codes using IBM Quantum Lab that is based on Jupyter notebooks.

2.1 IBM Quantum Composer



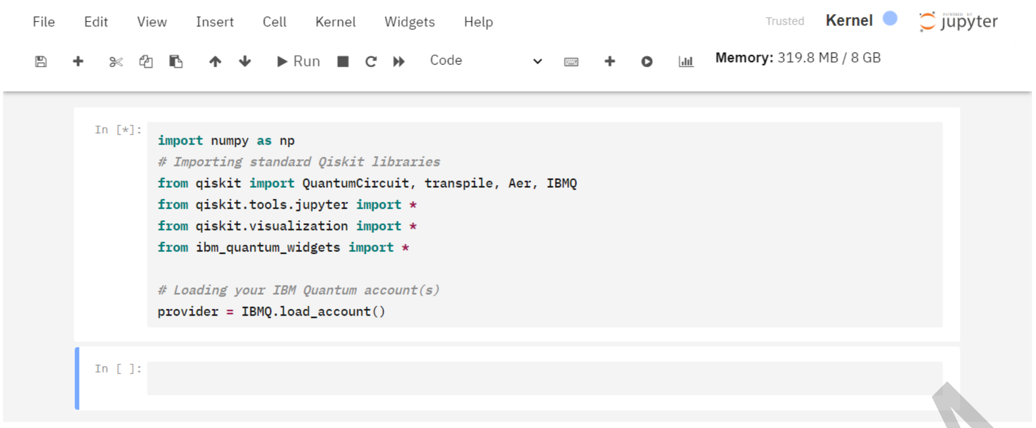
```
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
from numpy import pi

qreg_q = QuantumRegister(4, 'q')
creg_c = ClassicalRegister(4, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)
```

Figure 1: IBM Quantum Composer user interface

Figure 1 shows the IBM Quantum Composer which is a graphical user interface to compose, construct and execute quantum circuits. It lets the user to drag and drop quantum gates to the qubits to construct a circuit. It also shows real time probability distributions of the quantum states and the Bloch sphere or statevector of the qubits as the user constructs the circuit. It is useful to observe and understand graphically what happens to the qubits when certain gates are being applied to them. It also provides the Qiskit Python code of the circuit as the user constructs it which is useful to normalise the gates notations in Qiskit [1, 2]. Once the circuit is done, the user are able to execute the circuit using a real IBM-Q system from the available machines. The measurement can be observed later after the job is done at the Jobs webpage which can be accessed from the IBM Quantum frontpage.

2.2 IBM Quantum Lab & Qiskit

The IBM Quantum Lab uses Jupyter notebooks as its foundation for the online cloud-based programming of the quantum circuits. It uses Python as its language and the main module used to compose and construct quantum circuits is Qiskit which stands for Quantum Information Science Kit [2]. Qiskit is the framework for quantum computing founded by IBM Research that enables quantum circuits, algorithms, applications and experiments to be executed on their platform on your local simulators, IBM-Q cloud simulators or IBM-Q real systems [1]. Since IBM Quantum Lab uses Jupyter notebooks, it is completely fine to use your own local Jupyter which can be installed through Anaconda. The module Qiskit can then be installed through Anaconda Prompt by typing `pip install qiskit` in your chosen activated environment, `conda activate [insert-environment]`.



```
import numpy as np
# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, Aer, IBMQ
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *

# Loading your IBM Quantum account(s)
provider = IBMQ.load_account()
```

Figure 2: IBM Quantum Lab

As in Figure 2, we can see that the code starts with the import of the modules and classes that we will be using for the circuit. Those are some of the most common imports that we will be using throughout the report. The `IBMQ` class is used to execute the circuit on real IBM-Q systems. `Aer` is a component of Qiskit that provides local simulations such as statevector simulator which outputs a final statevector, unitary simulator which converts the circuit to a unitary matrix and QASM simulator mimics a real quantum machine. The `QuantumCircuit` class takes two arguments which are the number of qubits and classical bits and construct a quantum circuit from those arguments.

```
qreg = QuantumRegister(1, 'q')
creg = ClassicalRegister(1, 'c')
qc = QuantumCircuit(qreg, creg)

qc.h(qreg[0])
qc.measure(qreg[0], creg[0])
qc.draw('mpl')
```

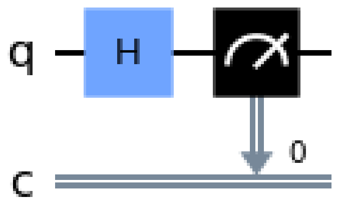


Figure 3: Single qubit circuit with Hadamard gate

The code above shows the construction of a single qubit circuit with Hadamard gate and a measurement at the end. We initialise a qubit and a classical bit before making the quantum circuit. The qubit always starts as the state $|0\rangle$. The Hadamard gate is then applied to the first qubit which is zero-indexed and measured with respect to the first classical bit. We can then use the method `.draw('mpl')` to show the constructed circuit as in Figure 3.

```
backend_sim = Aer.get_backend('qasm_simulator')
job_sim = execute(qc, backend_sim, shots=1000)
result_sim = job_sim.result()
counts = result_sim.get_counts(qc)
```

```
print(counts)

svsim = Aer.get_backend('statevector_simulator')
qobj = assemble(qc)
final_state = svsim.run(qobj).result().get_statevector()
print(final_state)

plot_histogram(counts)
```

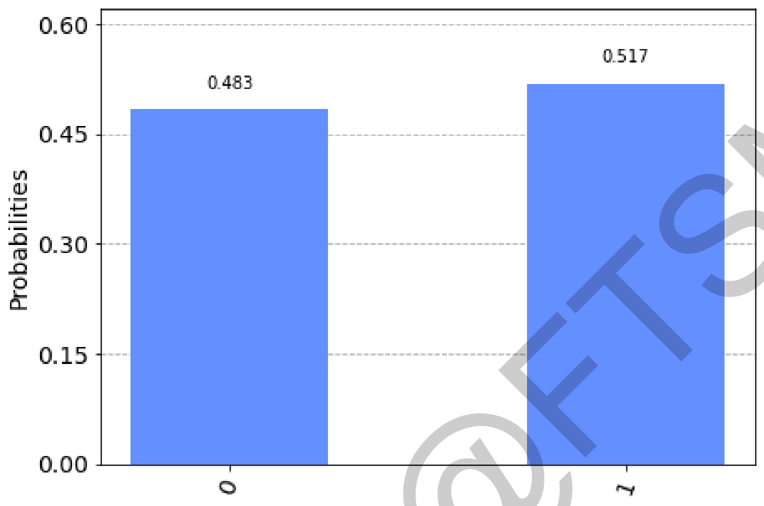


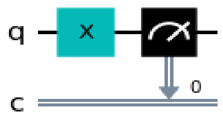
Figure 4: Probability distribution output of the single Hadamard circuit

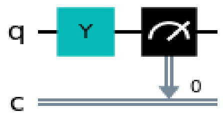
The `.get_backend` method lets you choose what backend or simulators to be used for the circuit. `.execute` executes the circuit on the chosen backend for a chosen number of shots which is the number of times the circuit is ran. The method `.assemble` simply assembles the circuit into an object to be ran, in this case through a statevector simulation, `svsim.run(qobj)`.

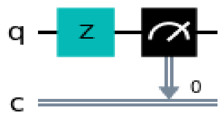
We can then measure the circuit by using both the local qasm_simulator and the statevector_simulator which will output the probability distribution of the quantum states or counts which is `['0': 494, '1': 506]` as in Figure 4 and the statevector of the final output which is `[1.+0.j 0.+0.j]`. We observe that the statevector_simulator outputs $|0\rangle$ instead of $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ because the quantum state collapses when it is measured and thus it can only either be $|0\rangle$ or $|1\rangle$ with equal probability.

Based on the quantum gates that we have studied on the previous report, we will now look into the Qiskit code for single qubit gates in Table 1 and for multi-qubit gates in Table 2.

Quantum gate	Quantum circuit representation	Qiskit code

Pauli-X gate		`qc.x(qreg[0])`
		`qc.measure(qreg[0], creg[0])`
		`qc.draw('mpl')`

Pauli-Y gate		`qc.y(qreg[0])`
		`qc.measure(qreg[0], creg[0])`
		`qc.draw('mpl')`

Pauli-Z gate		`qc.z(qreg[0])`
		`qc.measure(qreg[0], creg[0])`
		`qc.draw('mpl')`

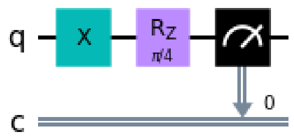
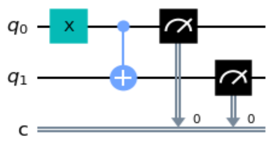
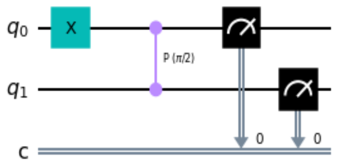
Phase shift (R_ϕ) gate		`qc.x(qreg[0])`
		`qc.rz(np.pi/4, qreg[0])`
		`qc.measure(qreg[0], creg[0])`
		`qc.draw('mpl')`

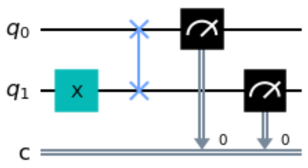
Table 1: Single qubit gates

For the case of the multi qubits gates, usually the method of the gates requires two to three arguments, where the first two are usually the control qubits while the last is the target qubit. For controlled gates that require phase, the first argument is the phase value, while the second and third arguments are the control and target qubits respectively.

Quantum gate	Quantum circuit representation	Qiskit code

CNOT gate		`qc.x(qreg[0])`
		`qc.cx(qreg[0], qreg[1])`
		`qc.measure(qreg[0], creg[0])`
		`qc.measure(qreg[1], creg[0])`
		`qc.draw('mpl')`

Controlled-U gate		`qc.x(qreg[0])`
		`qc.cp(np.pi/2, qreg[0], qreg [1])`
		`qc.measure(qreg[0], creg[0])`
		`qc.measure(qreg[1], creg[0])`
		`qc.draw('mpl')`

SWAP gate		`qc.x(qreg[1])`
		`qc.swap(qreg[0], qreg[1])`
		`qc.measure(qreg[0], creg[0])`
		`qc.measure(qreg[1], creg[0])`
		`qc.draw('mpl')`

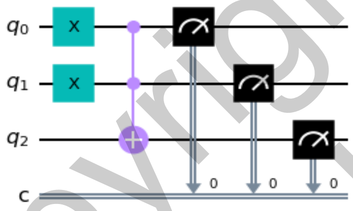
Toffoli gate		`qc.x(qreg[0])`
		`qc.x(qreg[1])`
		`qc.ccx(qreg[0], qreg[1], qreg [2])`
		`qc.measure(qreg[0], creg[0])`
		`qc.measure(qreg[1], creg[0])`
		`qc.measure(qreg[2], creg[0])`
		`qc.draw('mpl')`

Table 2: Multi qubits gates

The measurement of all these circuits can be done exactly as we did for the single Hadamard circuit by using the `Aer` backends for qasm_simulator and statevector_simulator. A Jupyter notebook that shows these measurements and simulations for the various quantum gates can be tried at this url: `https://colab.research.google.com/drive/1QxqDxQNHPRsd-Y8Lrn_8x7R-apYXyPwL?usp=sharing`.

Since we have already known much of the syntaxes used in Qiskit, we can try to develop a simple quantum circuit that we have seen in the previous report – the entangled Bell state. The quantum circuit for Bell state requires two qubits and two classical bits. A Hadamard gate is applied at the first qubit before a CNOT gate is applied to the second qubit with the first as its control as shown in Figure 5. Both of those qubits are then measured.

```
qreg_q = QuantumRegister(2, 'q')
creg_c = ClassicalRegister(2, 'c')
circuit = QuantumCircuit(qreg_q, creg_c)

circuit.h(qreg_q[0])
circuit.cx(qreg_q[0], qreg_q[1])

circuit.measure(qreg_q[0], creg_c[0])
circuit.measure(qreg_q[1], creg_c[1])

circuit.draw('mpl')
```

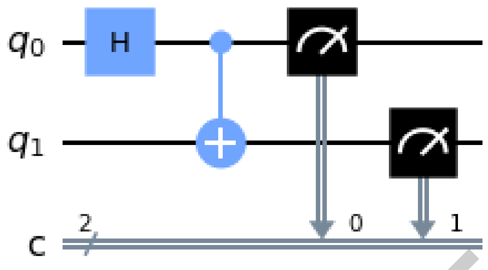


Figure 5: Bell state circuit

Theoretically, we should be able to measure either the states $|00\rangle$ or $|11\rangle$ which are both entangled states as the first measurement of the qubit will give the information for the second. From running the circuit on the simulators, both qasm and statevector, we observe the expected result in Figure 6.

```
>> Counts: {'11': 513, '00': 487}
>> Statevector: [0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 0.+0.j 1.+0.j]
```

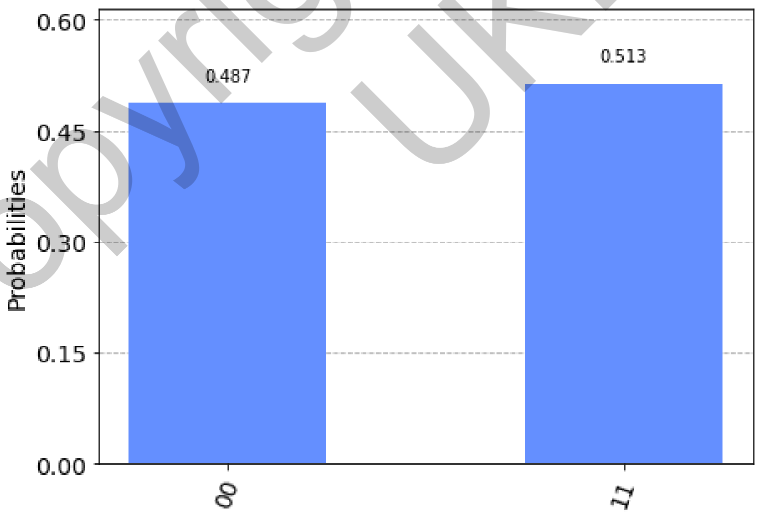


Figure 6: Probability distribution of the Bell state on qasm_simulator

Afterwards, we can run the circuit on a real IBM-Q machine. In this case, we are choosing the backend of IBM-Q Belem which is a 5-qubit system. We should expect a similar result to our simulation with a little bit of noise that will hopefully not disrupt the relative trend too much. The output from the real machine can be seen in Figure 7.

```
provider = IBMQ.load_account()
backend_ibmq = provider.get_backend('ibmq_belem')
job_ibmq = execute(circuit, backend_ibmq, shots=1000)
```

```
result_ibmq = job_ibmq.result()
counts = result_ibmq.get_counts(circuit)
plot_histogram(counts)
```

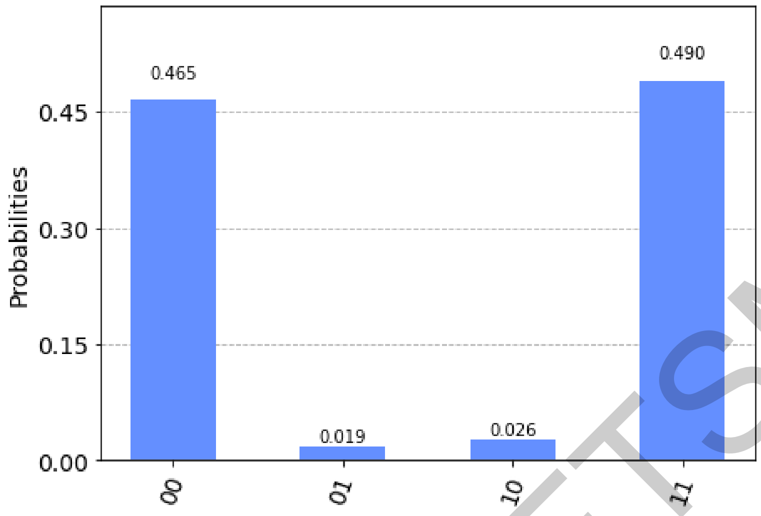


Figure 7: Probability distribution of the Bell state on IBM-Q Belem

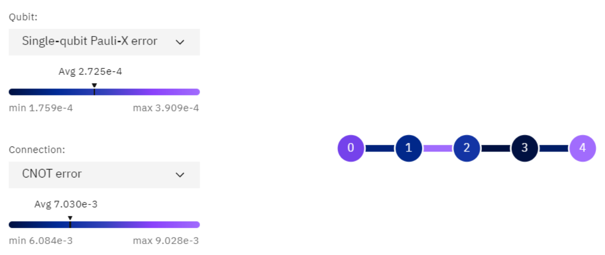
2.3 IBM Quantum System

The research and development for scalable real working quantum system are currently being pursued by many. Although we have already achieved up to more than 50 working qubits, so much more is needed to fully realize the potentials of quantum computers and algorithms [1]. However, our current quantum computers are not perfect yet – the qubits are noisy and prone to error due to environmental disturbances which affect the coherence time for these qubits. These quantum computers are usually called as noisy intermediate scale quantum computers or NISQ.

The basis of IBM's quantum computers are superconducting qubits that operate at extremely low temperature which require a cryogenic system enclosing the computer to keep the qubits in quantum states for some period of time. The qubits are controlled by microwave pulses that rotate their energy levels to apply gate operations. Currently, IBM has 28 quantum systems but only 9 of them are publicly available for use which varies from single qubit to 15-qubit quantum computers.

Due to these systems being NISQ, quantum computers that are on our hand right now might be good to implement and run smaller scale quantum circuits and algorithms but are not necessarily better than its classical counterparts on circuits and algorithms that are bigger and much more complex. Thus, it is important to know which IBM quantum system one should use that minimizes the gate errors that are already available in the system and the connections between the qubits [3].

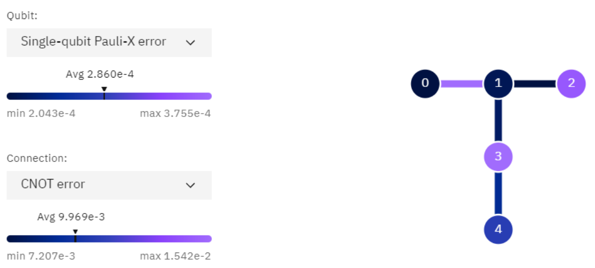
The mappings on the right of both Figures 8 and 9 show the connection between qubits of each system [1]. It is best practice to use the quantum system with qubit connections that closely resembles the connection of qubits in your designated quantum circuits. If the qubits are not connected in the real quantum machine, it is still possible to run the circuit but it will add SWAP gates to compensate which will in turn increase the number of gates used and the noise that comes with it. In general, the noise of the system increases as the number of qubits and gates increase.



Qubit:
Single-qubit Pauli-X error
Avg 2.725e-4
min 1.759e-4 max 3.909e-4

Connection:
CNOT error
Avg 7.030e-3
min 6.084e-3 max 9.028e-3

Figure 8: The qubit topology of IBM-Q Manila 5-qubit quantum system [1]



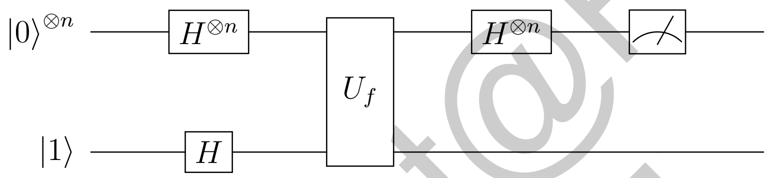
Qubit:
Single-qubit Pauli-X error
Avg 2.860e-4
min 2.043e-4 max 3.755e-4

Connection:
CNOT error
Avg 9.969e-3
min 7.207e-3 max 1.542e-2

Figure 9: The qubit topology of IBM-Q Belem 5-qubit quantum system [1]

3 Deutsch-Josza algorithm

The Deutsch-Josza algorithm is a black box algorithm that takes n binary inputs and outputs either 0 or 1. The function is balanced if the exactly half the outputs are 0s or 1s and the function is constant if all the outputs are either 0s or 1s [4].



As Deutsch and Deutsch-Josza is a black box solving algorithm, there is no one particular way to implement the black box, U_f. The simplest way to implement a constant function is by simply applying a Pauli-X gate on the $|1\rangle$ ancilla qubit on random to yield $|0\rangle^{\otimes n}|0\rangle$ output. The balanced function black box can be implemented by applying a CNOT gate in between Pauli-X gates on the qubits with the ancilla as the target qubit [1, 4].

```python
def dj_oracle(n, case):
    oracle_qc = QuantumCircuit(n+1)

    if case == "balanced":
        b = np.random.randint(1,2**n)
        b_str = format(b, '0'+str(n)+'b')

        for qubit in range(len(b_str)):
           if b_str[qubit] == '1':
               oracle_qc.x(qubit)

        for qubit in range(n):
            oracle_qc.cx(qubit, n)

        for qubit in range(len(b_str)):
           if b_str[qubit] == '1':
              oracle_qc.x(qubit)

    if case == "constant":
        output = np.random.randint(2)
        if output == 1:
            oracle_qc.x(n)

    oracle_gate = oracle_qc.to_gate()
    oracle_gate.name = "Oracle"
    return oracle_gate
```

We can then set the code to construct the circuit by initializing the Hadamard gates on the qubits before applying the oracle as in Figure 10. Another set of Hadamard transformations is applied to the qubits before measurement are made. We can use the NumPy random class to choose whether the function is balanced or constant randomly and we can also write an if statement to check whether it is balanced of constant based on the counts measurement.

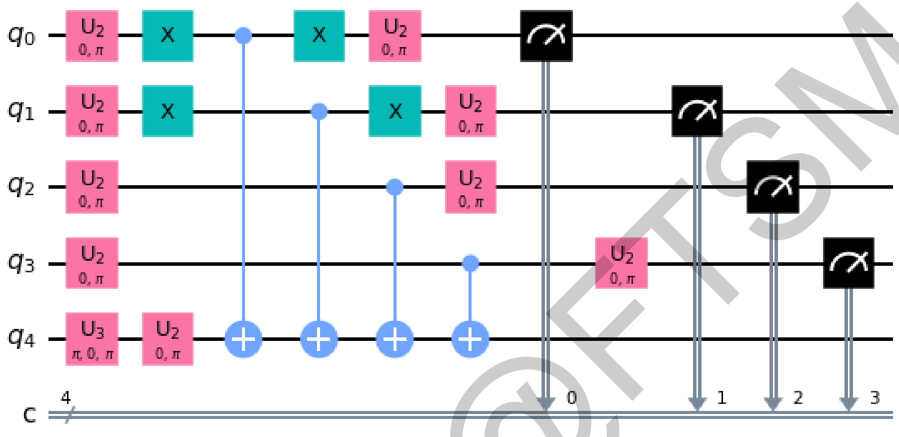


Figure 10: Decomposition of the circuit for balanced function

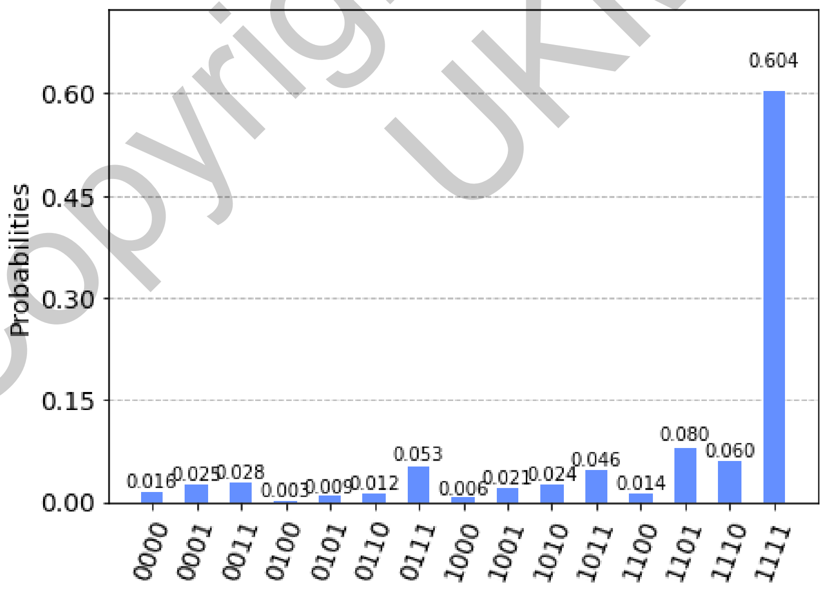


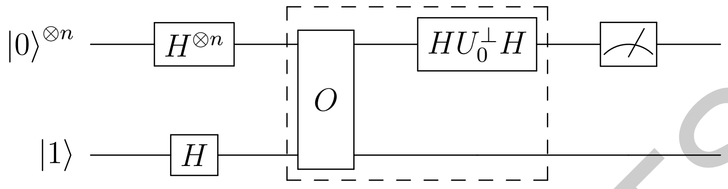
Figure 11: Measurement of the balanced function on IBM-Q Athens 5-qubit system

As seen in Figure 11, the probability for the measurement of $|1111\rangle$ is the most significant for balanced function which is true to what we have discussed on the previous report. The probability of the measurement is not exactly 1 as is theoretically because we have to consider the noise that is available from the IBM-Q system itself. If the function is constant, we should see the output measurement to be highly concentrated at $|0000\rangle$ for this case of $n = 4$.

4 Grover's algorithm

In the previous report, we have seen the theoretical foundation for Grover's algorithm. The algorithm implementation should follow the quantum circuit based on the algorithm. Based on the quantum circuit for Grover's algorithm, we observe that there are two main parts of the algorithm – the Grover Oracle, O for the phase inversion and the Grover Diffuser, $HU_0^\perp H$ for the inversion about the mean [4].

Grover's algorithm quantum circuit



The Grover Oracle is where we will mark the key value that we are trying to find. There are multiple ways this can be implemented that can even let you mark several key values to be found. In this implementation, we will mark the key value by using a controlled Pauli-Z gate which will phase flip the intended key value. The only thing we will have to do is transform the initial qubit to the key value by using Pauli-X gates. In Figure 12 where our key value for a 5-qubit system is 10011, the circuit for the Grover Oracle is,

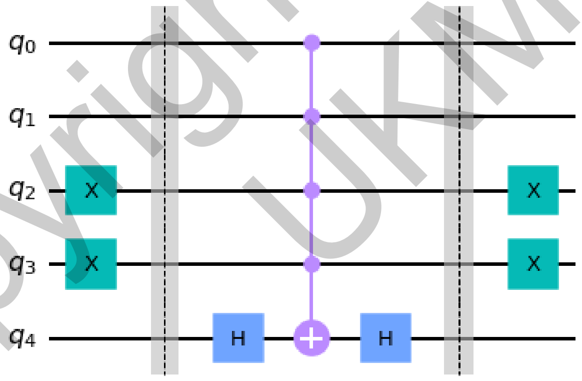


Figure 12: Grover Oracle for key value 10011

The Pauli-X gates is applied on the 0s of the key values, where it is the second and third number but since the qubit representation on IBM-Q and Qiskit is reversed, the key value should be read from the bottom to up and hence q3 and q2 is marked for the key value 10011. There is also no multi-controlled Pauli-Z gate in Qiskit, however it can be constructed with a multi-controlled Pauli-X gate in between two Hadamard gates since $Z = HXH$.

```
n = 5
key = '10011'

qc = QuantumCircuit(n)
lst = []
for i, num in enumerate(key[::-1]):
    print(i, num)
    if num == '0':
        qc.x(i)
        lst.append(i)

qc.barrier()
qc.h(n-1)
qc.mct(list(range(n-1)), n-1)
qc.h(n-1)
qc.barrier()

for j in lst:
    qc.x(j)
```

The Grover Diffuser's, $HU_0^{\perp}H$ main purpose is to invert the amplitude of the quantum states about the mean. First, the Hadamard gate and Pauli-X gate are applied on all the qubits in order for the states to be in superposition. The superposition states will then be applied to a multi-controlled Pauli-Z gate before the Pauli-X and Hadamard gates are applied again as in Figure 13. Essentially, this combination of gates inverts the amplitude of the marked key value above the mean while all the other states below the mean.

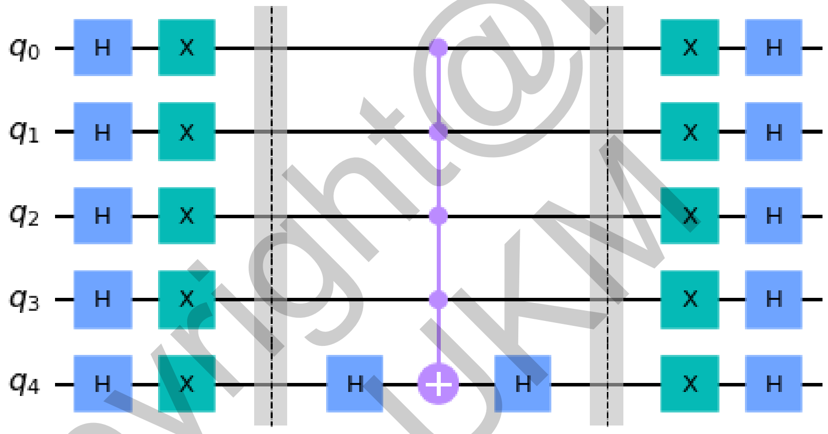


Figure 13: Grover Diffuser for 5-qubit system

```
qc = QuantumCircuit(n)

def ccz(n, qc):
    qc.h(n-1)
    qc.mct(list(range(n-1)), n-1)
    qc.h(n-1)

for qubit in range(n):
    qc.h(qubit)
    qc.x(qubit)

qc.barrier()
ccz(n, qc)
qc.barrier()

for qubit in range(n):
    qc.x(qubit)
    qc.h(qubit)
```

Now that we have all the components for the Grover's algorithm, we are able to run the actual algorithm. Both the Oracle and the Diffuser need to be run iteratively for $\frac{\pi\sqrt{N}}{4}$ times to get an accurate result of the key value. This can simply be done by encompassing both of these circuits in a for loop using Qiskit on Python. We have tested the algorithm for 2, 3, 4 and 5 qubits on both simulators and real quantum machine. Below are the results from real quantum machine runs using the backend of IBM-Q Santiago for 6000 shots,

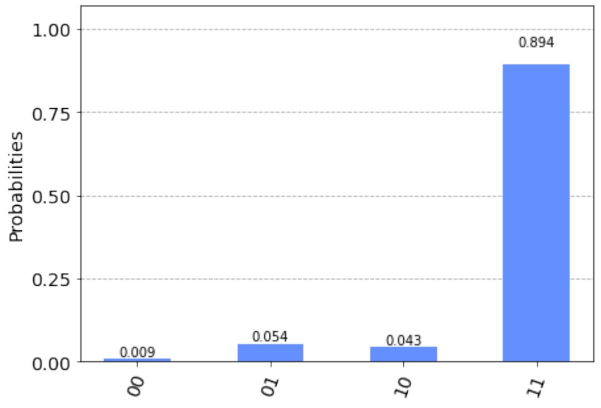


Figure 14: Grover's search for 11

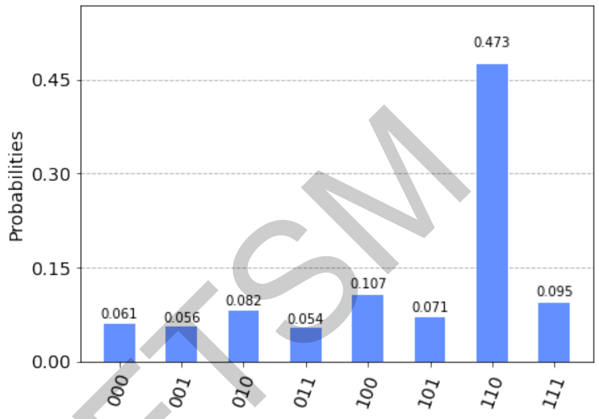


Figure 15: Grover's search for 110

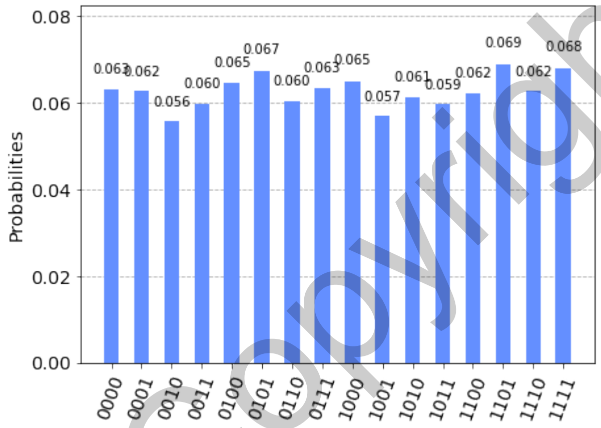


Figure 16: Grover's search for 1101

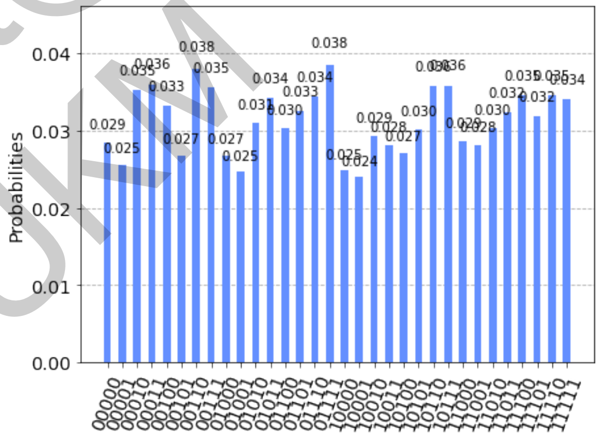


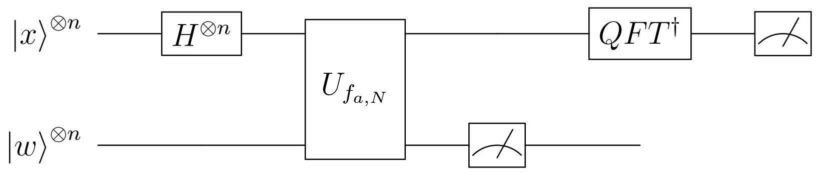
Figure 17: Grover's search for 11011

From Figures 14, 15, 16 and 17, we can observe that the result gets more inaccurate as the number of qubits increases. This can be due to multiple reasons mainly that as the number of qubits increases, the number of gates used in the circuit increases which will be prone to more error on top of the already existing error of the quantum machine and the potential addition of SWAP gates due to the connectivity of the qubits from the machine.

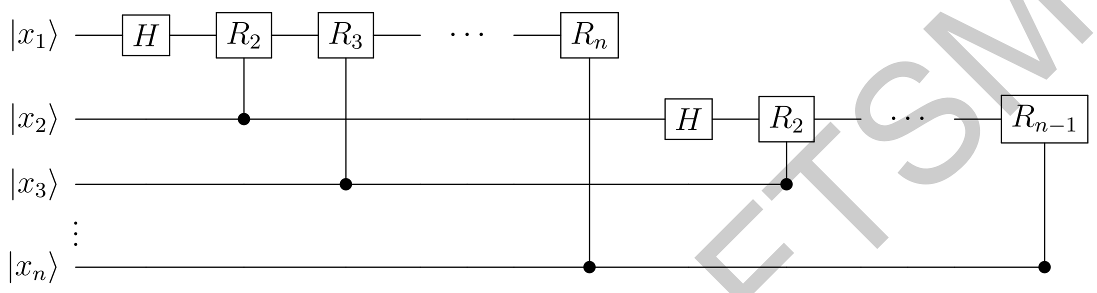
5 Shor's algorithm

The Shor's algorithm is moderately complex as compared to Grover's algorithm circuit. The two main parts comprised in the algorithm is the modular exponentiation function, $U_{f_{a,N}}$ and the inverse Quantum Fourier Transformation (IQFT), $QFT^{\dagger}$.

Shor's algorithm quantum circuit



The explanation of QFT can be found in the appendix of the previous report but essentially the quantum circuit for QFT is constructed by using the unitary rotation gate, R_k, where the quantum circuit for QFT is,



The implementation of quantum gates for IQFT is rather straightforward as in Figure 18. We can use the controlled rotation/phase gate available in Qiskit that takes the (negative) phase, $-\frac{\pi}{2^{j-m}}$ where j and m are the controlled and target qubits respectively. The qubits should also be swapped as the output state are reversed.

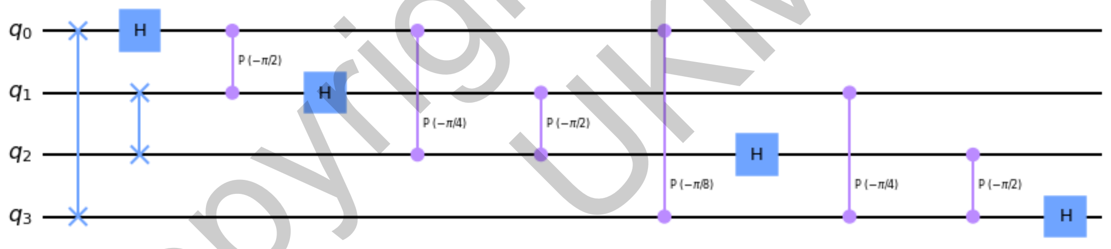


Figure 18: Inverse QFT

```
n = 4
qc = QuantumCircuit(n)

for qubit in range(n//2):
    qc.swap(qubit, n-qubit-1)
for j in range(n):
    for m in range(j):
        qc.cp(-np.pi/float(2**(j-m)), m, j)
    qc.h(j)
```

Conversely, the modular exponentiation implementation is rather elaborate which would require strong understanding on adder, modular adder and controlled multiplier gates. The modular exponentiation computes the function $f(x) = a^x \mod N$ on the second register, w. From the Qiskit textbook, it can be implemented by using SWAP and Pauli-X gates based on the chosen a value for the period finding of $a^x \mod 15$ [2, 4]. Thus, this particular circuit can only be used for this respective case.

```python
def c_amod15(a, power):
    if a not in [2,7,8,11,13]:
        raise ValueError("'a' must be 2,7,8,11 or 13")
    U = QuantumCircuit(4)
    for iteration in range(power):
        if a in [2,13]:
            U.swap(0,1)
            U.swap(1,2)
            U.swap(2,3)
        if a in [7,8]:
            U.swap(2,3)
            U.swap(1,2)
            U.swap(0,1)
        if a == 11:
            U.swap(1,3)
            U.swap(0,2)
        if a in [7,11,13]:
            for q in range(4):
                U.x(q)
    U = U.to_gate()
    U.name = "%i^%i mod 15" % (a, power)
    c_U = U.control()
    return c_U
```

Taking the example from the Qiskit textbook of finding the period of 15, we start with an 8 qubit system where the first 4 qubits are the counting qubits and the rest are the auxiliary qubits where the modular exponentiation is going to happen shown in Figure 19 [2]. Hadamard gates are applied to all the counting qubits and the last auxiliary qubit is applied with a Pauli-X gate. The modular exponentiation function is then applied to the qubits before the inverse QFT is applied to the counting qubits which are then measured.

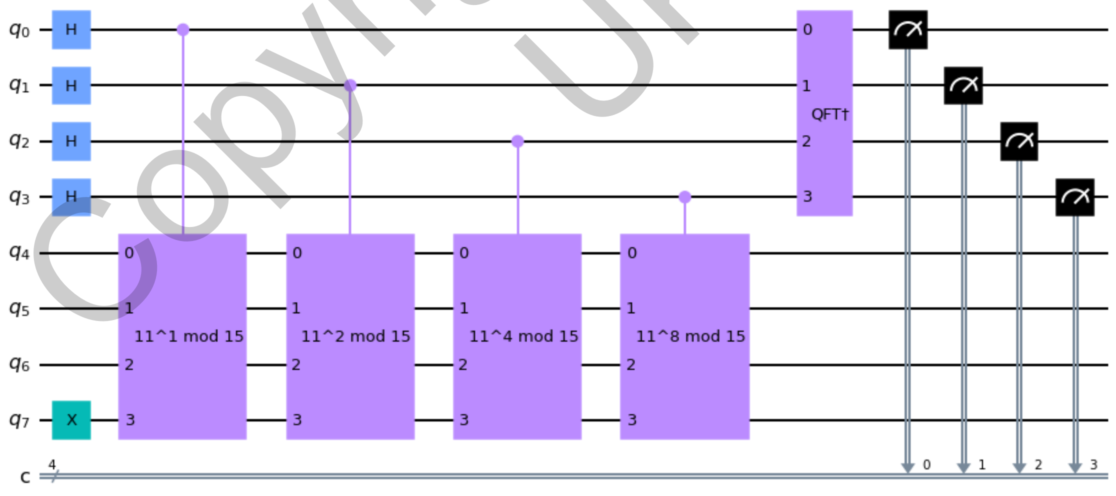


Figure 19: Shor's period finding algorithm for $a^x \mod 15$

The measured quantum state can be converted to the phase reading by dividing the decimal value of the state and 2^n where n is the number of counting qubits. By transforming the phase into a fraction, it is possible to get the potential value of the period, r. The factors can be found by calculating the greatest common divisor, $\gcd(a^{r/2} - 1, N)$ and $\gcd(a^{r/2} + 1, N)$.

We tried running this circuit on the simulator and it was able to run and yielded the correct factors. However, this circuit is way too big in qubits and gates number to be ran via a real quantum machine. Since, it uses 8 qubits, the only machine available is the IBM-Q Melbourne's 16 qubit system which has a relatively high CX error rate. Not to mention, the machine architecture is not necessarily optimized to the circuit and the circuit alone has a depth of 125 gates. However, it is possible to run an optimized circuit for Shor's algorithm that was developed for experimental realization back in 2001 via nuclear magnetic resonance in Figure 20 [5]. This circuit that solves the period finding of $a^x \mod 15$ is significantly more compact and uses less gates. This optimized Shor's algorithm circuit is ran on IBM-Q Manila (5-qubit system) for 6000 shots.

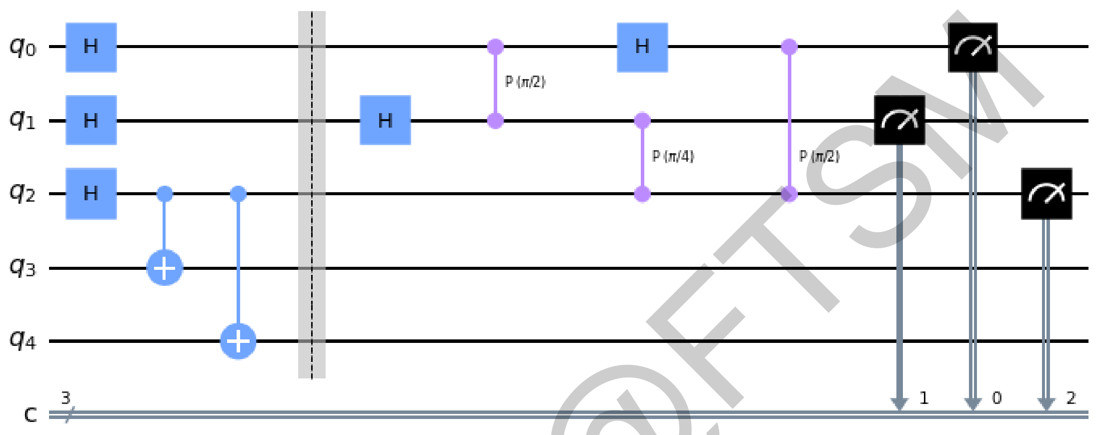


Figure 20: Optimized Shor's period finding algorithm for $a^x \mod 15$

Running this optimized Shor's algorithm, we are able to attain two quantum states with significant probabilities, 000 and 100. We can disregard 000 as it gives us no real information, but 100 can be converted to 4 in decimal where the phase is $\frac{4}{2^3} = 0.5$. Converting the phase to fraction, we can find the period, $r = 2$ which is the denominator of the fraction. We can finally find the factors of 15 by calculating the greatest common divisor. This will result in the values of 3 and 5 which is the factors of 15 in Figure 21 [5, 2].

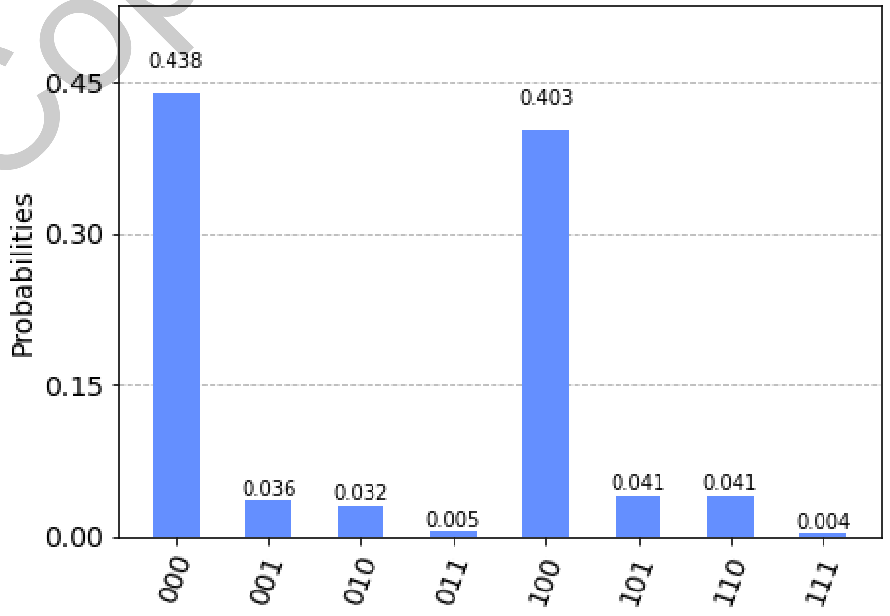


Figure 21: Optimized Shor's period finding probability output

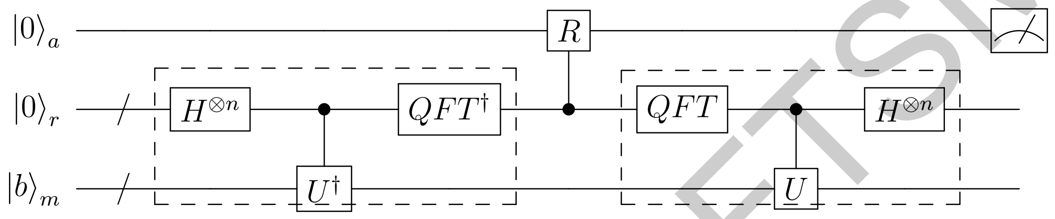
6 Harrow-Hassidim-Lloyd (HHL) algorithm

Based on the previous report, the HHL algorithm essentially rearranges the general equation of $A\,|x\rangle = |b\rangle$ to obtain,

$$|x\rangle = A^{-1}\,|b\rangle = \sum_{j=0}^{N-1} \lambda_j^{-1} b_j\,|u_j\rangle$$

The circuit for HHL uses the subroutine of Quantum Phase Estimation (QPE) which is similar to the Shor's algorithm where it consists of controlled-U rotations and QFT. Much like Shor's, the gate implementation for the controlled-U rotations and the R phase rotations between the QPEs are intricate [1].

HHL algorithm quantum circuit



In the case of the implementation of HHL, we will be using Qiskit Aqua which is a library of pre-built quantum algorithms for ease of use [2]. The HHL function takes the input of an eigenvalue circuit, the size of the circuit, matrix A, vector b, and the size of the matrix, n [1, 3]. However, another special function which utilizes numerical approach, `EigsQPE` is needed for a circuit to extract the eigenvalues from the matrix which is the QPE part of the HHL algorithm. The controlled phase rotation can be done using the `LookupRotation` method [3]. The matrix A must also be Hermitian, otherwise it should be expanded such that,

$$A_{\text{new}} = \begin{bmatrix} 0 & A \\ A^H & 0 \end{bmatrix}$$

```python
def create_eigs(matrix, num_auxiliary, num_time_slices, negative_evals):
    ne_qfts = [None, None]
    if negative_evals:
        num_auxiliary += 1
        ne_qfts = [QFT(num_auxiliary - 1), QFT(num_auxiliary - 1).inverse()]

    return EigsQPE(MatrixOperator(matrix=matrix),
                   QFT(num_auxiliary).inverse(),
                   num_time_slices=num_time_slices,
                   num_ancillae=num_auxiliary,
                   expansion_mode='suzuki',
                   expansion_order=1,
                   evo_time=None,
                   negative_evals=negative_evals,
                   ne_qfts=ne_qfts)

orig_size = len(vector)
matrix, vector, truncate_powerdim, truncate_hermitian = HHL.matrix_resize(
    matrix, vector) #Expansion of matrix A if needed

eigs = create_eigs(matrix, 3, 50, False) #EigsQPE
num_q, num_a = eigs.get_register_sizes()

init_state = Custom(num_q, state_vector=vector)
```

```
reci = LookupRotation(negative_evals=eigs._negative_evals, evo_time=eigs.
    _evo_time) #controlled phase rotations

algo = HHL(matrix, vector, truncate_powerdim, truncate_hermitian, eigs,
            init_state, reci, num_q, num_a, orig_size)
```

The HHL circuit is constructed by the `HHL` function which can be observed by using `algo.construct_circuit().draw(output='mpl')`. Since it uses a general algorithm for the circuit, it is a complex circuit with a circuit depth of 101 and 54 CNOT gates which makes running the circuit on a real quantum machine impossible. However, it is still possible to run the circuit on a statevector simulator. The result from the simulation is the final statevector which is the solution of x. A classical linear solver can be used to find the solution classically and the fidelity which is the ratio between the normalized HHL solution to the normalized classical solution can be calculated.

```
result = algo.run(QuantumInstance(Aer.get_backend('statevector_simulator')))
print("Solution:\t\t", np.round(result['solution'], 5))

result_ref = NumPyLSsolver(matrix, vector).run()
print("Classical Solution:\t", np.round(result_ref['solution'], 5))

print("Probability:\t\t %f" % result['probability_result'])
fidelity(result['solution'], result_ref['solution'])
```

We can try the algorithm with a matrix A and vector b,

$$A = \begin{bmatrix} 1.5 & 0.5 \\ 0.5 & 1.5 \end{bmatrix} \qquad b = \begin{bmatrix} \sqrt{2}/2 \\ \sqrt{2}/2 \end{bmatrix}$$

which would output the result,

```
>> Solution:      [0.35355-0.j 0.35355-0.j]
>> Classical Solution:    [0.35355 0.35355]
>> Time to process (HHL): 3.485805599999992
>> Time to process (Classical): 0.00020649999999902284
>> Probability:     0.395009
>> Fidelity:     1.000000
```

with a fidelity of 1 which shows that the solution from the HHL is exactly the same as the classical solution. The probability is the probability of the quantum state of the final statevector. If we were to use qasm simulator or a real quantum machine, it will only output the probability distribution of the quantum states making it impossible to obtain the statevector of the solution. However the quantum state of the solution, $|x\rangle$ can be used for further applications and algorithms. As we can see, the time taken for the algorithm to process and execute is around 3.5 seconds which is much longer than the time the classical linear solver took. This can be attributed to the complexity of the general unoptimized algorithm.

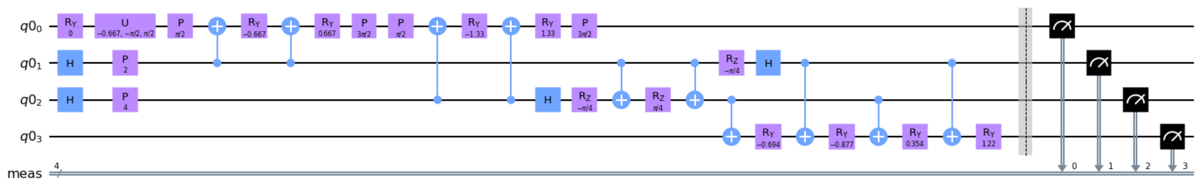


Figure 22: Optimized HHL algorithm circuit for 2x2 matrix with $a = 1$ and $b = -1/3$

Figure 22 shows an optimized HHL circuit for a 2x2 matrix in the form of $\begin{bmatrix} a & b \\ b & a \end{bmatrix}$ with $|b\rangle = (\cos\theta, \sin\theta)^\intercal$ from [2, 6]. Qubit 1, q00 is where the state preparation happens and is also the solution qubit, while q02 is the conditional rotation qubit. The results from Figures 23 and 24 that can be obtained from this circuit is the norm of the solution vector which is the probability of the conditional rotation qubit measuring $|1\rangle$ and the absolute average of the solution by applying a Hadamard gate on the solution qubit and measuring $|0\rangle$ on the conditional qubit [6]. This circuit is executed for the norm and the absolute average of the solution for the initial phase, θ of 0, 0.5, 1, 1.5, 2, 2.5 and 3 on IBM-Q Belem for 8192 shots.

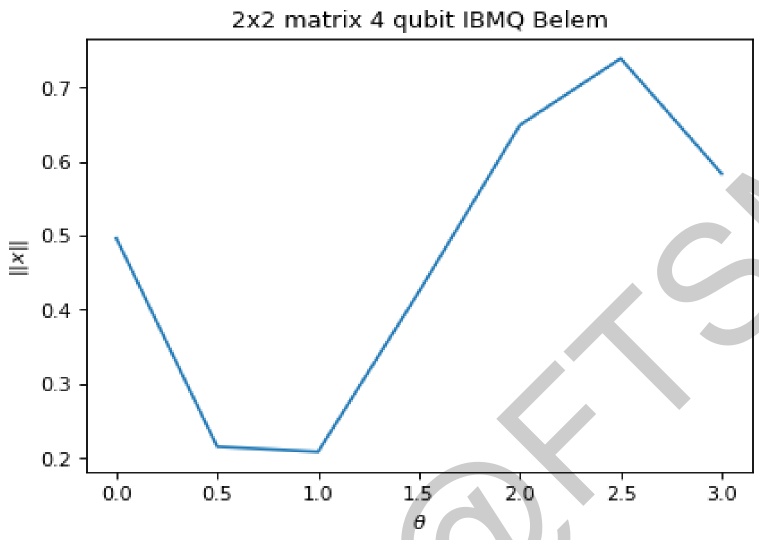


Figure 23: Norm of the solution for the 2x2 matrix

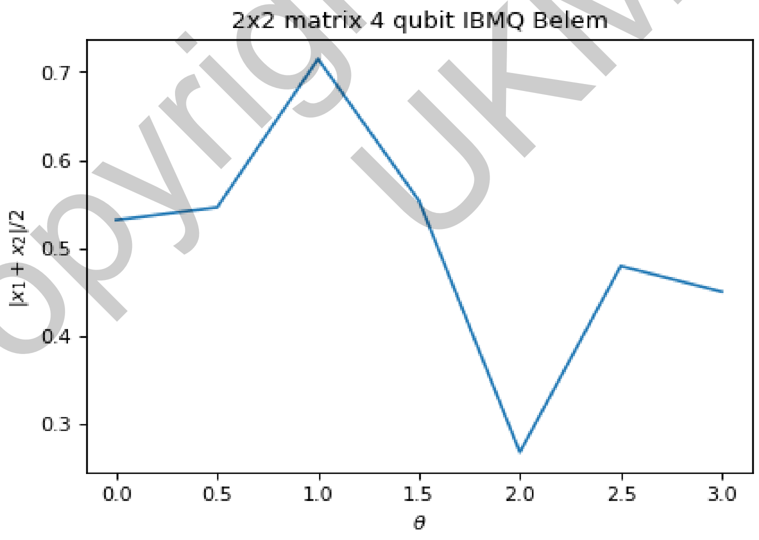


Figure 24: Absolute average of the solution for the 2x2 matrix

7 Conclusion

IBM-Q as a platform is a reliable entry to most people on understanding and implementing quantum computing and algorithms. We have seen that the quantum machines publicly available is capable of running smaller scale quantum algorithms with good accuracy. However, the hardware limitations still remain for all quantum computers. Constructing and implementing an algorithm into a quantum circuit on a real quantum machine are not as straightforward as it seemed to be. Researchers have to compensate with the hardware limitations by constructing optimized smaller scale circuits that still give the expected outputs by considering the amount of qubits used, the amount of quantum gates used in the circuit, the architecture and topology of the quantum machine used among other various factors.

In this report, we observed that without any sort of error mitigation and correction code used, the Grover's algorithm was able to locate the key value up until 3 qubits while the optimized Shor's algorithm for period finding of $a^x \mod 15$ was able to yield the period and the optimized HHL algorithm was able to give the norm and the absolute average of the solution.

Acknowledgements

This work is related to the ongoing research by Ts. Dr. Bahari Idrus and Assoc. Professor. Dr. Mohammad Khatim Hasan funded by GUP-2020-061.

We acknowledge the use of IBM Quantum services for this work. The views expressed are those of the authors, and do not reflect the official policy or position of IBM or the IBM Quantum team. In this paper we used ibmq_manila, ibmq_belem, ibmq_santiago and ibmq_athens which are part of the IBM Quantum Falcon Processors. The codes for the algorithms are largely based on the textbook by Qiskit.

References

[1] *IBM Quantum*. 2021. URL: https://quantum-computing.ibm.com/.

[2] Abraham Asfaw et al. *Learn Quantum Computation Using Qiskit*. 2020. URL: http://community.qiskit.org/textbook.

[3] Sigurdur Ag. Sigurdsson. "Implementations of Quantum Algorithms for Solving Linear Systems". Jan. 2021. URL: http://resolver.tudelft.nl/uuid:820157f7-5350-4908-acba-7f419f0a7ec9.

[4] Abhijith J. et al. *Quantum Algorithm Implementations for Beginners*. 2020. arXiv: 1804.03719 [cs.ET].

[5] Lieven M. K. Vandersypen et al. "Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance". In: *Nature* 414.6866 (Dec. 2001), pp. 883–887. ISSN: 1476-4687. DOI: 10.1038/414883a. URL: http://dx.doi.org/10.1038/414883a.

[6] Almudena Carrera Vazquez, Ralf Hiptmair, and Stefan Woerner. *Enhancing the Quantum Linear Systems Algorithm using Richardson Extrapolation*. 2020. arXiv: 2009.04484 [quant-ph].

A Deutsch-Josza's algorithm

```python
import numpy as np

from qiskit import IBMQ, Aer, QuantumCircuit, assemble, transpile
from qiskit.visualization import plot_histogram

def dj_oracle(n, case):
    oracle_qc = QuantumCircuit(n+1)

    if case == "balanced":
        b = np.random.randint(1,2**n)
        b_str = format(b, '0'+str(n)+'b')

        for qubit in range(len(b_str)):
            if b_str[qubit] == '1':
                 oracle_qc.x(qubit)

        for qubit in range(n):
            oracle_qc.cx(qubit, n)

        for qubit in range(len(b_str)):
            if b_str[qubit] == '1':
                oracle_qc.x(qubit)

    if case == "constant":
        output = np.random.randint(2)
        if output == 1:
            oracle_qc.x(n)

    oracle_gate = oracle_qc.to_gate()
    oracle_gate.name = "Oracle"
    return oracle_gate

def dj_algorithm(n, case='random'):
    dj_circuit = QuantumCircuit(n+1, n)
    dj_circuit.x(n)
    dj_circuit.h(n)

    for qubit in range(n):
        dj_circuit.h(qubit)

    oracle = dj_oracle(n, case)
    dj_circuit.append(oracle, range(n+1))

    for qubit in range(n):
        dj_circuit.h(qubit)

    for i in range(n):
        dj_circuit.measure(i, i)

    return dj_circuit

n = 4
dj_circuit = dj_algorithm(n, case='balanced')
dj_circuit.draw(output='mpl')

qasm_sim = Aer.get_backend('qasm_simulator')
shots = 1024
transpiled_dj_circuit = transpile(dj_circuit, qasm_sim)
```

```
qobj = assemble(transpiled_dj_circuit)
results = qasm_sim.run(qobj).result()
counts = results.get_counts()
plot_histogram(counts)

if '0'*n in counts:
    zeroMeasurements = counts['0'*n]
else:
    zeroMeasurements = 0
if (zeroMeasurements < 100):
    print("Function is balanced")
else:
    print("Function is constant")

provider = IBMQ.load_account()
backend = provider.get_backend('ibmq_athens')
transpiled_dj_circuit = transpile(dj_circuit, backend)
qobj = assemble(transpiled_dj_circuit)
results = backend.run(qobj).result()
counts = results.get_counts()
plot_histogram(counts)
```

B Grover's algorithm

```
%matplotlib inline

from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit,
    execute, Aer, IBMQ
from qiskit.compiler import transpile, assemble
from qiskit.tools.monitor import job_monitor
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from qiskit.providers.aer import noise

import numpy as np

def init(qc, qubits):
    for q in qubits:
        qc.h(q)
    return qc

def ccz(n, qc):
    qc.h(n-1)
    qc.mct(list(range(n-1)), n-1)
    qc.h(n-1)

def groveracle(n, key):
    if n==2:
        qc = QuantumCircuit(n)
        qc.h(1)
        qc.cx(0,1)
        qc.h(1)

        goracle = qc.to_gate()
        goracle.name = "Oracle"
        return goracle

    qc = QuantumCircuit(n)
    lst = []
    for i, num in enumerate(key[::-1]):
        #print(i, num)
        if num == '0':
            qc.x(i)
            lst.append(i)
    ccz(n, qc)
    for j in lst:
        qc.x(j)

    goracle = qc.to_gate()
    goracle.name = "Oracle"
    return goracle

def diffuser(n):
    qc = QuantumCircuit(n)

    for qubit in range(n):
        qc.h(qubit)
        qc.x(qubit)

    ccz(n, qc)

    for qubit in range(n):
        qc.x(qubit)
        qc.h(qubit)

    HUH = qc.to_gate()
    HUH.name = "Diffuser"
    return HUH

n = 5
key = '11011'

iterations = int(round(np.pi/4 * np.sqrt(2**n)))
if n==2:
    iterations = 1

grover_circuit = QuantumCircuit(n)
grover_circuit = init(grover_circuit, list(range(n)))

for i in range(iterations):
    grover_circuit.append(groveracle(n, key), list(range(n)))
    grover_circuit.barrier()
    grover_circuit.append(diffuser(n), list(range(n)))
    grover_circuit.barrier()

grover_circuit.measure_all()
grover_circuit.draw(output='mpl')

backend_sim = Aer.get_backend('qasm_simulator')
job_sim = execute(grover_circuit, backend_sim, shots=6000)
result_sim = job_sim.result()
counts = result_sim.get_counts(grover_circuit)
print(counts)
plot_histogram(counts)

provider = IBMQ.load_account()
backend_ibmq = provider.get_backend('ibmq_belem')

tp_grover = transpile(grover_circuit, backend_ibmq)
job_ibmq = execute(grover_circuit, backend_ibmq, shots=6000,
   optimization_level=3)
job_monitor(job_ibmq)
result_ibmq = job_ibmq.result()

counts = result_ibmq.get_counts(grover_circuit)
plot_histogram(counts)
```

C Optimized Shor's algorithm

```
%matplotlib inline

from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit,
    execute, Aer, IBMQ
from qiskit.compiler import transpile, assemble
from qiskit.tools.monitor import job_monitor
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from qiskit.providers.aer import noise
from math import gcd
from fractions import Fraction
import numpy as np

qc = QuantumCircuit(5,3)

for i in range(3):
    qc.h(i)

qc.cx(2, 3)
qc.cx(2, 4)
qc.barrier()

qc.h(1)
qc.cp(np.pi/2, 1, 0)
qc.h(0)
qc.cp(np.pi/4, 1, 2)
qc.cp(np.pi/2, 0, 2)

for _ in range(3):
    qc.measure(_, _)

qc.draw('mpl')

qasm_sim = Aer.get_backend('qasm_simulator')
t_qc = transpile(qc, qasm_sim)
qobj = assemble(t_qc)
results = qasm_sim.run(qobj).result()
counts = results.get_counts()
plot_histogram(counts)

provider = IBMQ.load_account()
backend_ibmq = provider.get_backend('ibmq_manila')
tp = transpile(qc, backend_ibmq)
job_ibmq = execute(qc, backend_ibmq, shots=6000, optimization_level=3)
job_monitor(job_ibmq)
result_ibmq = job_ibmq.result()
counts = result_ibmq.get_counts(qc)
plot_histogram(counts)

x = int('100', 2)
N = 15
a = 4
phase = x/2**3
frac = Fraction(phase).limit_denominator(2**3)
print(frac)r = frac.denominator
guesses = [gcd(a**(r//2)-1,15), gcd(a**(r//2)+1,15)]

for guess in guesses:
    if guess not in [1,N] and (N % guess) == 0:
        print("--- Non-trivial factor found: %i ---" % guess)
        factor_found = True
```

D HHL algorithm

```python
from qiskit import Aer, transpile, assemble, IBMQ
from qiskit.circuit.library import QFT
from qiskit.aqua import QuantumInstance, aqua_globals
from qiskit.quantum_info import state_fidelity
from qiskit.aqua.algorithms import HHL, NumPyLSsolver
from qiskit.aqua.components.eigs import EigsQPE
from qiskit.aqua.components.reciprocals import LookupRotation
from qiskit.aqua.operators import MatrixOperator
from qiskit.aqua.components.initial_states import Custom
import numpy as np
import time

def create_eigs(matrix, num_auxiliary, num_time_slices, negative_evals):
    ne_qfts = [None, None]
    if negative_evals:
        num_auxiliary += 1
        ne_qfts = [QFT(num_auxiliary - 1), QFT(num_auxiliary - 1).inverse()]

    return EigsQPE(MatrixOperator(matrix=matrix),
                   QFT(num_auxiliary).inverse(),
                   num_time_slices=num_time_slices,
                   num_ancillae=num_auxiliary,
                   expansion_mode='suzuki',
                   expansion_order=1,
                   evo_time=None,  # This is t, can set to: np.pi*3/4
                   negative_evals=negative_evals,
                   ne_qfts=ne_qfts)

def fidelity(hhl, ref):
    solution_hhl_normed = hhl / np.linalg.norm(hhl)
    solution_ref_normed = ref / np.linalg.norm(ref)
    fidelity = state_fidelity(solution_hhl_normed, solution_ref_normed)
    print("Fidelity:", fidelity)

matrix = [[1.5,0.5],[0.5,1.5]]
vector = [np.sqrt(2)/2, np.sqrt(2)/2]

orig_size = len(vector)
matrix, vector, truncate_powerdim, truncate_hermitian = HHL.matrix_resize(
    matrix, vector)
print(matrix, truncate_powerdim)

# Initialize eigenvalue finding module
eigs = create_eigs(matrix, 3, 50, False)
num_q, num_a = eigs.get_register_sizes()
print(num_q, num_a)

# Initialize initial state module
init_state = Custom(num_q, state_vector=vector)

# Initialize reciprocal rotation module
reci = LookupRotation(negative_evals=eigs._negative_evals, evo_time=eigs.
    _evo_time)
print(reci)

algo = HHL(matrix, vector, truncate_powerdim, truncate_hermitian, eigs,
           init_state, reci, num_q, num_a, orig_size)

```

```
t0 = time.perf_counter()
result = algo.run(QuantumInstance(Aer.get_backend('statevector_simulator')))
print(result)
t1 = time.perf_counter()
print("Solution:", np.round(result['solution'], 5), "Time taken:", t1-t0)

t2 = time.perf_counter()
result_ref = NumPyLSsolver(matrix, vector).run()
t3 = time.perf_counter()
print("Classical Solution:", np.round(result_ref['solution'], 5), "Time
    taken:", t3-t2)

print("Probability:", result['probability_result'])
fidelity(result['solution'], result_ref['solution'])
```

E Optimized HHL algorithm

```python
from qiskit import QuantumRegister, QuantumCircuit
from qiskit.visualization import plot_histogram
import numpy as np

t = 2.344915690192344

nqubits = 4
nb = 1  #Solution qubit
nl = 2  #Eigenvalues qubits

thetas = [0, 0.5, 1.0, 1.5, 2, 2.5, 3]
norm = []

a = 1
b = -1/3

for theta in thetas:
    qr = QuantumRegister(nqubits)
    qc = QuantumCircuit(qr)

    qrb = qr[0:nb]
    qrl = qr[nb:nb+nl]
    qra = qr[nb+nl:nb+nl+1]

    qc.ry(2*theta, qrb[0])

    # QPE with e^{iAt}
    for qu in qrl:
        qc.h(qu)

    qc.p(a*t, qrl[0])
    qc.p(a*t*2, qrl[1])

    qc.u(b*t, -np.pi/2, np.pi/2, qrb[0])

    # Controlled e^{iAt} on \lambda_{1}:
    params=b*t

    qc.p(np.pi/2,qrb[0])
    qc.cx(qrl[0],qrb[0])
    qc.ry(params,qrb[0])
    qc.cx(qrl[0],qrb[0])
    qc.ry(-params,qrb[0])
    qc.p(3*np.pi/2,qrb[0])

    # Controlled e^{2iAt} on \lambda_{2}:
    params = b*t*2

    qc.p(np.pi/2,qrb[0])
    qc.cx(qrl[1],qrb[0])
    qc.ry(params,qrb[0])
    qc.cx(qrl[1],qrb[0])
    qc.ry(-params,qrb[0])
    qc.p(3*np.pi/2,qrb[0])

    # Inverse QFT
    qc.h(qrl[1])
    qc.rz(-np.pi/4,qrl[1])
    qc.cx(qrl[0],qrl[1])
    qc.rz(np.pi/4,qrl[1])
    qc.cx(qrl[0],qrl[1])
    qc.rz(-np.pi/4,qrl[0])
    qc.h(qrl[0])

    # Eigenvalue rotation
    t1=(-np.pi +np.pi/3 - 2*np.arcsin(1/3))/4
    t2=(-np.pi -np.pi/3 + 2*np.arcsin(1/3))/4
    t3=(np.pi -np.pi/3 - 2*np.arcsin(1/3))/4
    t4=(np.pi +np.pi/3 + 2*np.arcsin(1/3))/4

    qc.cx(qrl[1],qra[0])
    qc.ry(t1,qra[0])
    qc.cx(qrl[0],qra[0])
    qc.ry(t2,qra[0])
    qc.cx(qrl[1],qra[0])
    qc.ry(t3,qra[0])
    qc.cx(qrl[0],qra[0])
    qc.ry(t4,qra[0])

    qc.measure_all()

    print("Depth:", qc.depth())
    print("CNOTS:", qc.count_ops()['cx'])
    qc.draw(fold=-1, output='mpl')

    from qiskit import execute, BasicAer, ClassicalRegister, IBMQ
    from qiskit.compiler import transpile

    provider = IBMQ.load_account()
    backend = provider.get_backend('ibmq_athens')
    qc_qa_cx = transpile(qc, backend=backend)
    job = execute(qc_qa_cx, backend=backend, shots=8192, optimization_level=3)
    result = job.result()
    counts = result.get_counts(qc_qa_cx)

    sum = 0
    for i in range(2**4):
        bit_str = format(i,'04b')
        if bit_str[1]=='1':
            print(bit_str, counts[bit_str])
            sum += counts[bit_str]

    norm_x = sum/8192
    print(theta, norm_x)
    norm.append(norm_x)

import matplotlib.pyplot as plt
plt.figure(figsize=(6, 4), dpi=80)

plt.plot(thetas, norm)
plt.xlabel(r'$\theta$')
plt.title('2x2 matrix 4 qubit IBMQ Belem')
plt.ylabel(r'$||x||$')
plt.show()
```