Beyond Classical

A crash course on Quantum Computing

using Qiskit and IBM Q

short line

Rishwi Thimmaraju

And

Harika Vajha

# Preface

This book is intended to be a course material for a university professional course on quantum computing. It gives students both theoretical and practical experience using Qiskit and IBMQ. This book is a 10-week course requiring at least 2 hours per week. The schema of the material is well structured and organized for the most efficient and encouraging learning experience. This book uses certain terminology intended only to make the learning process interesting. You can encounter a few terms which have letter Q embedded forcefully. This book includes basics, most important quantum algorithms, deep concepts like error correction and finally giving an introduction to the certain evolving applications of quantum computing in the field of Quantum Machine Learning as well as Quantum chemistry. This book is a good resource for novice, intermediate and also will act as a brush-up book for advanced level students.

# Schema of the book

The first four weeks include all the rudiments required to gain a grip on quantum computing. In the first four weeks learner will get an idea on IBM Q, Qiskit, Quantum Gates, and other basic algorithms with both theoretical explanation and practical implementation. Week -5, 6, 7 introduces various key algorithms. Week -8, 9 gives details of Quantum cryptography, errors and noise. It also introduces NISQ. Week 10 dives into research going on in Quantum Machine Learning and quantum chemistry.

# How to use this book

Spend at least 2 hours per week, Do weekly lessons and follow up exercises. While doing the lessons first get the theoretical essence and then try it out practically using Qiskit or IBM Q Experience. Make sure you spend extra time outside the learning time to do the exercise given in that week. If in case of any hurdle with practical experience all the code samples are available in the form of python notebooks in the below github link. Don’t look at exercise samples before even trying them out.

Github:: <https://rishwi.github.io/Beyond-Classical-a-quantum-computing-crash-course-using-qiskit/>

# Quontents(Contents)

Week - 1

1. How a Classical computer works
2. Difference between a Classical and a Quantum computer
3. Principles and Components of Quantum Computer
4. Hello World program ( Entanglement circuit using IBM Q Experience )

Week - 2

1. Quantum Gates ( Theory )
2. Installation of Qiskit and Quantum Gates ( Practical implementation using Qiskit )
3. Classical gates using Quantum gates
4. Exercise - 1

Week - 3

1. IBM Q / Qiskit Backends
2. Qiskit-Aer
3. Bernstein Vazirani Algorithm
4. Exercise - 2

Week - 4

1. Einstein’s Spooky action
2. Quantum Teleportation ( Theory + Practical using Qiskit )
3. No-teleportation theorem
4. Exercise - 3

Week - 5

1. Deutsch-Jozsa Algorithm
2. Simon’s Algorithm
3. Quantum Fourier Transform
4. Exercise - 4

Week - 6

1. RSA Encryption
2. Shor’s Algorithm ( Theory + Practical )
3. Exercise - 5

Week - 7

1. Amplitude amplification
2. Grover’s Algorithm
3. Exercise - 6

Week - 8

1. Quantum Noise
2. Quantum error correction
3. NISQ Computers
4. Exercise - 7

Week - 9

1. Quantum Cryptography
2. Quantum Key distribution
3. No-cloning Theorem
4. Exercise - 8

Week - 10

1. Quantum Machine Learning
2. Application of Quantum Computing in Chemistry.
3. Exercise - 9

# Welcome to WEEK 1

How Classical Computer works?

Let’s get to this in a bottom-up approach which starts with the stage of the program. First, the program is converted to assembly language which is then converted to binary bits. This whole process involves a lot of components like the compiler, loader, linker, syntax table, ...etc. So let’s go deeper and ask ourselves what are binary bits. The answer for that is the presence and absence of electrons which are represented as 1 and 0 respectively. As it is a classical computer it deals with a bunch of electrons. Even if an electron escapes from that bunch, there might not be an issue but when it comes to a quantum computer it is totally a different view. In a classical computer, a bit can take only two values either 1 or 0 at a time which makes classical computers less effective in a few cases.

Difference between Classical and Quantum computer

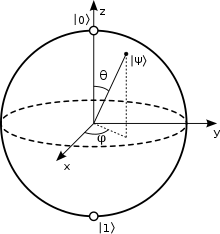
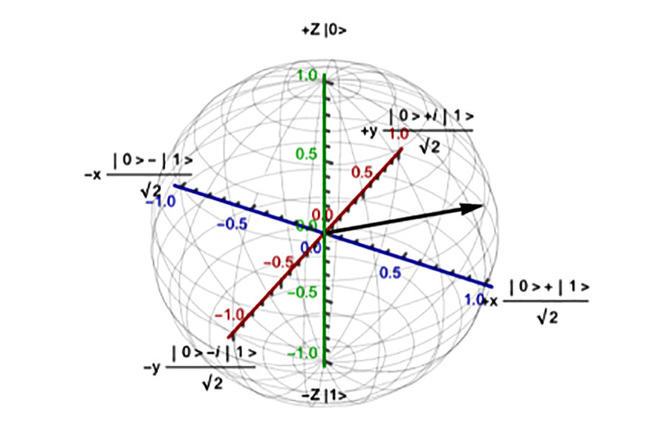
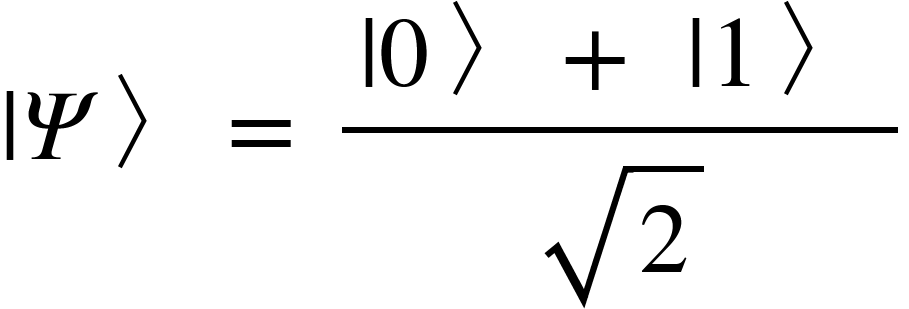
As told above bits in a classical computer are either 1 or 0 but when it comes to a quantum computer a bit is called “QUBIT” which has state |0〉 or |1〉 or both at a time. It is like a tossed coin that fell vertically and spinning. The state of the coin is not yet known because it is still spinning. In the same way, untill someone will measure its states, a qubit is in superposition which means both 1 and 0 at a time.

There is a famous thought experiment by Schrodinger which explains the state of a quantum particle by taking a cat that is placed inside a box along with poisoned food. There are many possibilities for the cat to be alive or dead. Here in this example cat being alive is |1〉 and cat being dead is |0〉 but we will know whether the cat is alive or dead only if someone opens the box which is tantamount to destroying the quantum state. Hence measuring a quantum bit turns it into a classical bit. There are many possibilities in this case. The cat may or may not eat the poisoned food. Instead, the cat can also die just because of suffocation inside the box. Let’s discuss superposition and other quantum phenomena later this week. When it comes to differences. If a classical computer does n operations in 1 sec, a quantum computer can perform  operations. Instead of dealing with a bunch of electrons, a quantum computer deals with a single electron or with particles so small, such as photons.

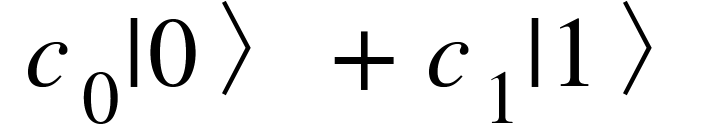
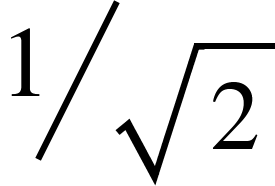
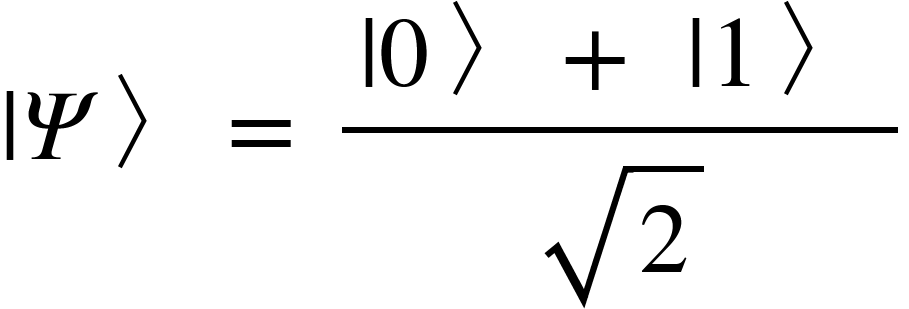
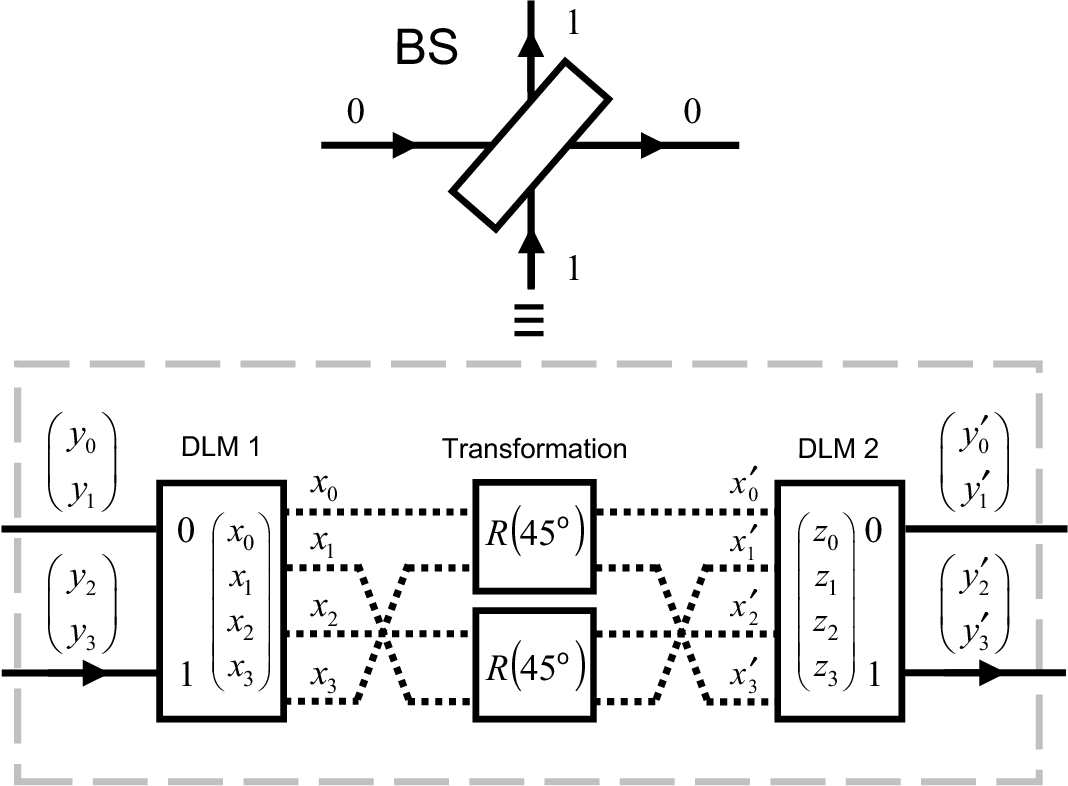
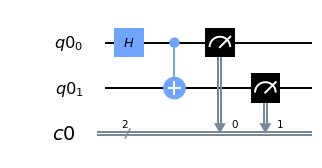
**Difference between conventional computing and quantum computing:**

|  |  |
| --- | --- |
| **CONVENTIONAL COMPUTING** | **QUANTUM COMPUTING** |
| Conventional computing is based on the classical phenomenon of electrical circuits being in a single state at a given time, either on or off. | Quantum computing is based on the phenomenon of Quantum Mechanics, such as superposition and entanglement, the phenomenon where it is possible to be in more than one state at a time. |
| Information storage and manipulation are based on “bit”, which is based on voltage or charge; low is 0 and high is 1. | Information storage and manipulation is based on Quantum Bit or “qubit”, which is based on the spin of electron or polarization of a single photon. |
| The circuit behavior is governed by classical physics. | The circuit behavior is governed by quantum physics or quantum mechanics. |
| Conventional computing uses binary codes i.e. bits 0 or 1 to represent information. | Quantum computing use Qubits i.e. 0, 1 and superposition state of both 0 and 1 to represent information. |
| CMOS transistors are the basic building blocks of conventional computers. | Superconducting Quantum Interference Device or SQUID or Quantum Transistors are the basic building blocks of quantum computers. |
| In conventional computers, data processing is done in the Central Processing Unit or CPU, which consists of Arithmetic and Logic Unit (ALU), processor registers and a control unit. | In quantum computers, data processing is done in the Quantum Processing Unit or QPU, which consists of a number of interconnected qubits. |

Principles and Components of Quantum Computer

1. **Qubit**: It is the basic unit of information in a quantum computer. A qubit can be in superposition state which means both |0〉and |1〉at a time. A qubit can be described as the Schrodinger's cat inside a box. The physical representation of a qubit can be denoted using a sphere called the Bloch sphere.  
     
      
     
   In the above sphere, there are two poles north and south which denotes state 0 and 1 respectively. Line passing from the origin represented with |Ψ〉 is the superposition state which is given as . Let us discuss further about this when we get into the topic of quantum gates. There are several qubit technologies theorized and developed.  
     
     
     
     
     
   **Qubit technologies / Physical Implementations:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Physical support** | **Name** | **Information support** | |0〉 | |1〉 |
| [Photon](https://en.wikipedia.org/wiki/Photon) | Polarization encoding | Polarization of light | Horizontal | Vertical |
| Number of photons | [Fock state](https://en.wikipedia.org/wiki/Fock_state) | Vacuum | Single-photon state |
| Time-bin encoding | Time of arrival | Early | Late |
| Coherent state of light | Squeezed light | Quadrature | Amplitude-squeezed state | Phase-squeezed state |
| Electrons | Electronic spin | Spin | Up | Down |
| Electron number | Charge | No electron | One electron |
| Nucleus | [Nuclear spin](https://en.wikipedia.org/wiki/Nuclear_spin) addressed through [NMR](https://en.wikipedia.org/wiki/Nuclear_magnetic_resonance) | Spin | Up | Down |
| Optical lattices | Atomic spin | Spin | Up | Down |
| Josephson junction | Superconducting charge qubit | Charge | Uncharged superconducting island (*Q*=0) | Charged superconducting island (*Q*=2*e*, one extra Cooper pair) |
| Superconducting flux qubit | Current | Clockwise current | Counterclockwise current |
| Superconducting phase qubit | Energy | Ground state | First excited state |
| Singly charged quantum dot pair | Electron localization | Charge | Electron on left dot | Electron on right dot |
| Quantum dot | Dot spin | Spin | Down | Up |
| Gapped topological system | Non-abelian [anyons](https://en.wikipedia.org/wiki/Anyon) | [Braiding of Excitations](https://en.wikipedia.org/wiki/Braid_group) | Depends on specific topological system | Depends on specific topological system |

1. **Superposition:** When two quantum states are superimposed on each other in a classical sense it can be described as two waves superimposed on each other which means adding them up. Mathematical representation is . If both the coefficients are equal to  then it is the superposition state , where the probability of occurrence of state 0 and 1 are equal which is ½. In quantum computing, superposition is achieved using a Hadamard gate about which we are going to discuss in further weeks.
2. **Interference:** This term immediately brings up a famous experiment by Thomas Young. Yes!! it is the “Young’s Double Slit” experiment. He demonstrated quantum interference properties of light particles contrasting Newton’s views of light as a stream of particles. As photons can be in different states at the same time, Interference is the property that defines the state occurrences. To state it in computation terms, Quantum Interference is a phenomenon which helps us bias the measurement of a qubit towards the desired state or set of states. The figure below illustrates how this phenomenon is processed using photon beam splitters.  
     
   A reflection will flip the phase of the photon where refraction will maintain the state of the photon.
3. **Quantum Circuits:** Quantum circuits are visually represented as lines where each line represents a qubit. Quantum gates are added to each of the lines. A combination of gates on qubit registers is called a quantum circuit. Fig below is an example of a basic quantum entanglement circuit which is, of course, a theoretical mystery.  
     
   Qiskit code:  
   ………………………………………...  
   from qiskit import \*  
   qr = QuantumRegister(2)  
   cr = ClassicalRegister(2)  
   circuit = QuantumCircuit(qr , cr)  
   circuit.h(qr[0])  
   circuit.cx(qr[0], qr[1])  
   circuit.measure(qr,cr)  
   circuit.draw(output='mpl')  
   ………………………………………….  
   Here ‘H’ is a Hadamard gate which is a single qubit gate unlike the ‘CNOT’ which is a multi-qubit gate. We are going to this further in the quantum gates topic.

Exercise 0:

As we discussed the entanglement circuit, go and try it out in IBM Q experience. It is a very pleasant interface to start experimenting with quantum circuits on the web. Just drag and drop the gates onto the qubit lines to form a circuit.

Link: <https://quantum-computing.ibm.com/>

Use the circuit composer to create quantum circuits. Once you are done submitting your circuit, you can check the results in the results pane.

Experiment with the teleportation on both a qasm\_simulator and an original quantum computer.

Simulators are not complete quantum computers but operate on optical characteristics based on the probabilistic computing approach.

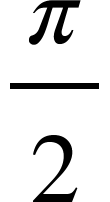
# Let’s dive into WEEK 2

Quantum Gates

**Single Qubit Gates**

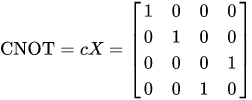
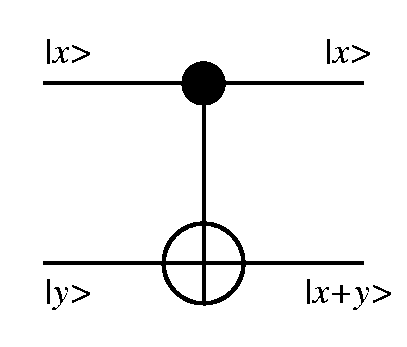
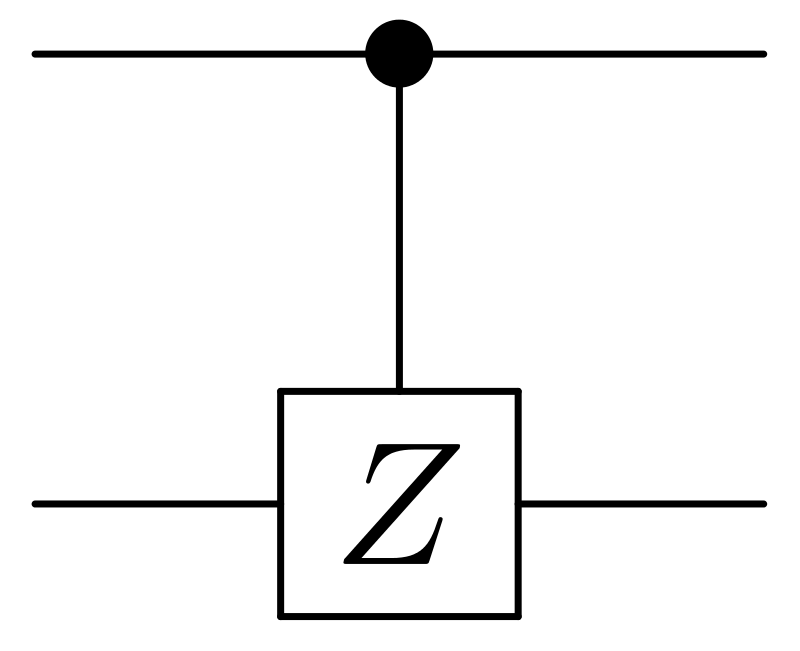
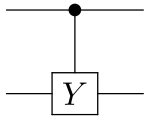
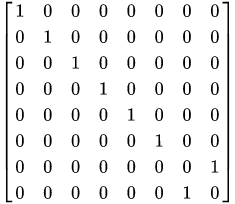
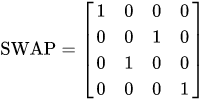
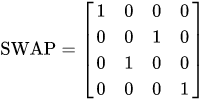
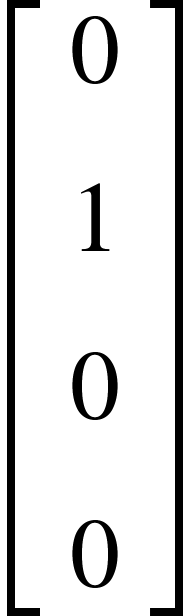
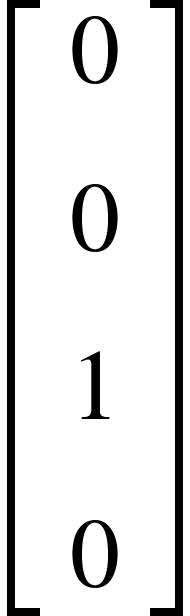
1. **Pauli X, Y, Z:** To put it in Bloch sphere representation these gates will perform a semi rotation of Bloch sphere around x, y and z axes respectively.   
     
   **X** gate will flip the qubit which is just like a not gate in conventional computing. So it is sometimes called bitflip. They can be formulated as follows.  
     
   **Y** gate will rotate the Bloch sphere around the y-axis about  radians.  
     
   **Z** gate will rotate the Bloch sphere around the z-axis about radians but it doesn’t change anything when the state is  and it does change  to .  
     
   Pauli matrices are Involuntary in nature. 

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Matrix** | **Circuit Symbol** | **Qiskit code** |
| Pauli *X*, *X*, NOT, bit flip, |  |  | qc.x(0) |
| Pauli *Y*, *Y,* |  |  | qc.y(0) |
| Pauli *Z*, *Z*, phase flip, |  |  | qc.z(0) |

1. **Hadamard gate:** It changes the base state of a qubit from  to  and  to  by making the probabilities of both basis states equal. It is made of two rotations, <math xmlns="http://www.w3.org/1998/Math/MathML"><mi>&#x3C0;</mi></math> around Z-axis and followed by  about Y-axis. Below is the representation of the Hadamard gate.   
   The resultant states of a Hadamard transform are |+⟩ and |−⟩ which are equal to and . So H|0⟩=|+⟩, H|1⟩=|−⟩, H|+⟩=|0⟩, H|−⟩=|1⟩.  
     
   H is certainly a unitary matrix .  
     
   Hadamard transform is used in the teleportation circuit, so the output that qubit decides the output of the second qubit, where the two qubits are joined using a CNOT gate about which we are going to discuss next.

**Multi-Qubit Gates**

They act on more than one qubit. Multi-qubit gates promote information exchange between qubits.

1. **CNOT** gate: This is a controlled not gate which is just like classical NOT gate but it includes two qubits unlike the conventional one. It has a control qubit and a target qubit. If the control qubit is one then the gate will flip the target qubit.   
      
   In Qiskit, the CNOT gate is referred to as cx which is an abbreviation for Controlled X-gate. The line with a black dot in the above figure is the control qubit whereas the bottom line is the target qubit.
2. **CZ gate(Controlled Z):** It is similar to the CNOT gate but does perform Z on target qubit.  
   
3. **CY gate(Controlled Y):** This is similar to the CNOT gate but does perform Y on target qubit.  
   
4. **Toffoli gate(CCNOT gate):** This is a three-qubit gate which is similar to a CNOT gate but has two control qubits and one target qubit. The matrix representation is given as follows.  
      
   This is also referred to as Deutsch  gate which flips the third qubit only if the first two qubits are .
5. **Swap Gate:** This gate swaps the two qubits in these basis states,, , ,  .  
   Matrix representation is .  
     
   Swapping : <math xmlns="http://www.w3.org/1998/Math/MathML"><mo>&#xD7;</mo></math> <math xmlns="http://www.w3.org/1998/Math/MathML"><mo>=</mo></math>

Installation of QISKIT

**Supporting Operating Systems:**

* Windows 7 or later versions

NOTE: For using Qiskit on Windows the system must have VC++ runtime components. The following are recommended:

1. Microsoft Visual C++ Redistributable for Visual Studio 2017
2. Microsoft Visual C++ Redistributable for Visual Studio 2015

* macOS 10.12.6 or later
* Ubuntu 16.04 or later

**Prerequisites**:

* Python 3.5 or later versions.
* Anaconda, a tool by Python for scientific computing.

**Installation:**

For better experience use Python virtual environments to cleanly separate Qiskit from other applications.

1. Open the terminal:

Open the terminal where you want to work and type the following commands:

Use conda command to specify the version of Python included with its libraries:

* conda create -n name\_of\_my\_env python=3

Activate the environment:

* source activate name\_of\_my\_env

**If it’s a Windows OS**:

1. Install Anaconda
2. Search for Anaconda Prompt
3. Open Anaconda Prompt

**Use:**

* conda create -n name\_of\_my\_env python=3
* activate name\_of\_my\_env

Now install Qiskit package that comes with Terra, Aer, Ignis, and Aqua.

* pip install qiskit

NOTE: If you already have Qiskit installed on your system and want to upgrade, first uninstall it using pip uninstall qiskit and install the latest version.

To verify the installation: For listing the packages use conda list command.

Optional dependencies: If you feel it essential, you can install by using:

* pip install qiskit-terra[visualization]

2. Importing Qiskit:

After completing all the above steps, import Qiskit to python using the command:

* import qiskit

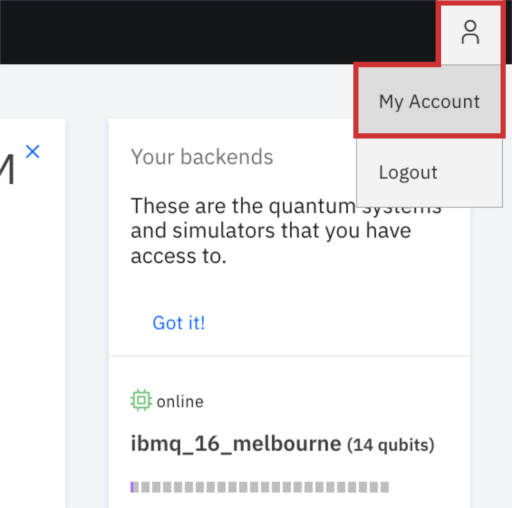
You can also get access to IBM Q systems:

IBM Q gives users get exposed to several real quantum computers and high-performance classical computing simulators through IBM Q Experience with Qiskit.

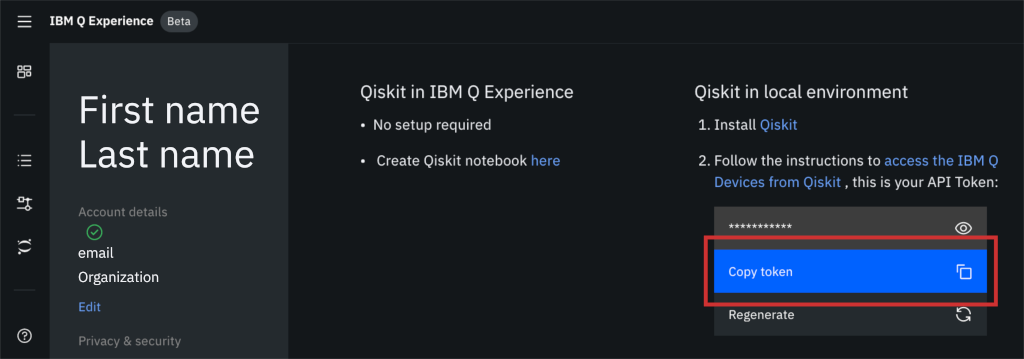
Follow the steps to set up Qiskit environment:

1. Create a free IBM Q experience account

2. Go to My Account to view your account settings.



3. Click on Copy token to copy the token. Paste the API token into a text editor for later use to create an account configuration file.



4. Run the below commands to store your API token for later use in a file called qiskitrc. Replace MY\_API\_TOKEN with the API token value that you previously stored in your text editor.

from qiskit import IBMQ

IBMQ.save\_account('MY\_API\_TOKEN')

**Checking the version:**

To check the version of Qiskit you’ve installed use command qiskit.\_\_version\_\_

Input: qiskit.\_\_version\_\_

Output: '0.9.1'

To get all the versions of the Qiskit elements in your environment you can use qiskit.\_\_qiskit\_version\_\_

Input:

Qiskit.\_\_qiskit\_version\_\_

Output:

{'qiskit-terra': '0.9.1',

'qiskit-ignis': '0.2.0',

'qiskit-aqua': '0.6.0',

'qiskit': 'dev-c0f998c6cc18408d019166d999eef9d2eaea39f6',

'qiskit-aer': '0.3.0',

'qiskit-ibmq-provider': '0.3.3'}

If you’ve followed all the above steps correctly you will get to use Qiskit. Happy Coding!

\*\*\* You can also use Qiskit notebooks in the IBMQ Experience website.

\*\*\*You can also use Google colab. Follow the below steps while using it.

1. pip install qiskit
2. Now load your IBM Q account.

Quantum gates using QISKIT

**Follow these initial steps:**

1. Make sure you have Qiskit setup (pip install qiskit).
2. **Importing packages:**from qiskit import \*
3. **Loading the account:**IBMQ.save\_account(‘API Token’)  
   IBMQ.load\_account()

**Implementation of the Quantum Gates using Qiskit**

**#xgate**

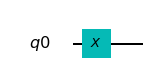
**q = QuantumRegister(1)**

**qc = QuantumCircuit(q)**

**qc.x(q[0])**

**qc.draw(output='mpl')**

**Output:**

****

**#zgate**

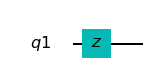
**q = QuantumRegister(1)**

**qc = QuantumCircuit(q)**

**qc.z(q[0])**

**qc.draw(output='mpl')**

**Output:**

****

**#hadamardgate**

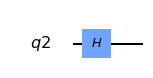
**q = QuantumRegister(1)**

**qc = QuantumCircuit(q)**

**qc.h(q[0])**

**qc.draw(output='mpl')**

**Output:**

****

**#cnotgate**

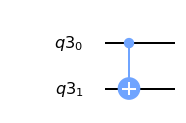
**q = QuantumRegister(2)**

**qc = QuantumCircuit(q)**

**qc.cx(q[0],q[1])**

**qc.draw(output='mpl')**

**Output:**

****

**#czgate**

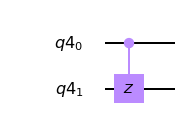
**q = QuantumRegister(2)**

**qc = QuantumCircuit(q)**

**qc.cz(q[0],q[1])**

**qc.draw(output='mpl')**

**Output:**

****

**#cz from h and cx**

**q = QuantumRegister(2)**

**qc = QuantumCircuit(q)**

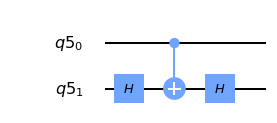
**qc.h(q[1])**

**qc.cx(q[0],q[1])**

**qc.h(q[1])**

**qc.draw(output='mpl')**

**Output:**

****

**#ccxgate or toffoli gate**

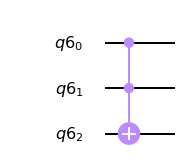
**q = QuantumRegister(3)**

**qc = QuantumCircuit(q)**

**qc.ccx(q[0],q[1],q[2])**

**qc.draw(output='mpl')**

**Output:**

****

**Creating classical Gates using Quantum Gates:**

Now let's start creating a classic logic gate using quantum gates. Each gates and their truth tables will be shown. Here we denote quantum registers as 'q', classical registers as 'c' where we encode the output of the measurement.

### **NOT Gate**

As it's mentioned before, a X gate can be considered as a NOT gate. Truth table for a NOT Gate looks like this:

|  |  |
| --- | --- |
| input | output |
| 0 | 1 |
| 1 | 0 |

**q = QuantumRegister(1)**

**c = ClassicalRegister(1)**

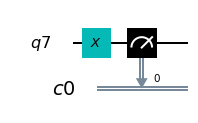
**qc = QuantumCircuit(q,c)**

**qc.x(q[0])**

**qc.measure(q[0], c[0])**

**qc.draw(output='mpl')**

**Output:**

****

### **AND Gate**

Truth table for an AND Gate looks like this:

It will only be true when both the conditions are true.

|  |  |  |
| --- | --- | --- |
| A(input) | B(input) | output |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

With a CCX gate, the result of an AND gate for 2 controlled bits will be output to its target bit.

**q = QuantumRegister(3)**

**c = ClassicalRegister(1)**

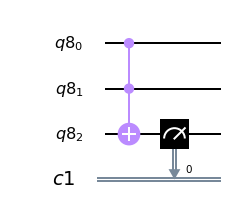
**qc = QuantumCircuit(q,c)**

**qc.ccx(q[0], q[1], q[2])**

**qc.measure(q[2], c[0])**

**qc.draw(output='mpl')**

**Output:**

****

### NAND Gate

A NAND gate can be made by applying a NOT gate after applying an AND gate.

|  |  |  |
| --- | --- | --- |
| A(input) | B(input) | output |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**q = QuantumRegister(3)**

**c = ClassicalRegister(1)**

**qc = QuantumCircuit(q,c)**

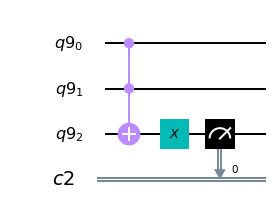
**qc.ccx(q[0], q[1], q[2])**

**qc.x(q[2])**

**qc.measure(q[2], c[0])**

**qc.draw(output='mpl')**

**Output:**

****

### OR Gate

|  |  |  |
| --- | --- | --- |
| A(input) | B(input) | output |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**q = QuantumRegister(3)**

**c = ClassicalRegister(1)**

**qc = QuantumCircuit(q,c)**

**qc.cx(q[1], q[2])**

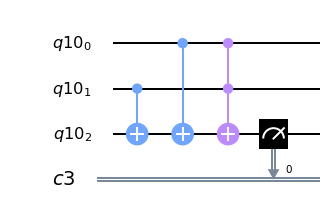
**qc.cx(q[0], q[2])**

**qc.ccx(q[0], q[1], q[2])**

**qc.measure(q[2], c[0])**

**qc.draw(output='mpl')**

**Output:**

****

### XOR Gate

|  |  |  |
| --- | --- | --- |
| A(input) | B(input) | output |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

In [12]:

**q = QuantumRegister(3)**

**c = ClassicalRegister(1)**

**qc = QuantumCircuit(q,c)**

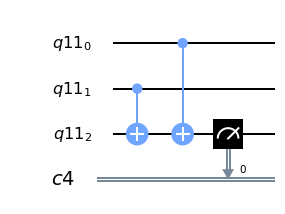
**qc.cx(q[1], q[2])**

**qc.cx(q[0], q[2])**

**qc.measure(q[2], c[0])**

**qc.draw(output='mpl')**

**Output:**

****

### NOR Gate

|  |  |  |
| --- | --- | --- |
| A(input) | B(input) | output |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

In [13]:

**q = QuantumRegister(3)**

**c = ClassicalRegister(1)**

**qc = QuantumCircuit(q,c)**

**qc.cx(q[1], q[2])**

**qc.cx(q[0], q[2])**

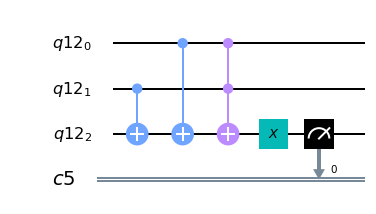
**qc.ccx(q[0], q[1], q[2])**

**qc.x(q[2])**

**qc.measure(q[2], c[0])**

**qc.draw(output='mpl')**

**Output:**

****

# **Adder Circuit**

An adder is a digital logic circuit that performs the addition of numbers. (i.e., )

In this example, we are going to take a look at the simplest adders, namely half adder and full adder.

## **Half Adder**

The half adder is used to add together the two least significant digits in a binary sum. It has two single binary inputs, called A and B, and two outputs C(carry out) and S(sum). The output C will be used as an input to the Full Adder, which will be explained later, for obtaining the value in the higher digit.

Half adders can be described with the truth table shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| A(input) | B(input) | S(sum) | C(carry out) |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

From the truth table, you should notice that the carry output C is a result of operating an AND gate against A and B、where the output S is a result of operating an XOR against A and B. As we have already created the AND and XOR gates, we can combine these gates and create a half adder as follows.

We denote our quantum register as 'q', classical registers as 'c', assign inputs A, B to q[0], q[1], the sum output S and carry output C to q[2] and q[3]."

***#Define registers and a quantum circuit***

**q = QuantumRegister(5)**

**c = ClassicalRegister(2)**

**qc = QuantumCircuit(q,c)**

***#XOR***

**qc.cx(q[1], q[2])**

**qc.cx(q[0], q[2])**

**qc.barrier(q)**

***#AND***

**qc.ccx(q[0], q[1], q[3])**

**qc.barrier(q)**

***#Sum***

**qc.measure(q[2], c[0])**

***#Carry out***

**qc.measure(q[3], c[1])**

**backend = Aer.get\_backend('qasm\_simulator')**

**job = execute(qc, backend, shots=1000)**

**result = job.result()**

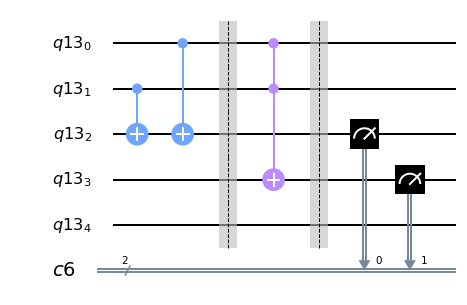
**count =result.get\_counts()**

**print(count)**

**qc.draw(output='mpl')**

**Output:**

**{'00': 1000}**

****

Exercise 1

The full adder takes two binary numbers plus an overflow bit as its input, which we will call X. Create a full adder from a quantum circuit. The truth table for the full adder is given below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A(input)** | **B(input)** | **X(carry input)** | **S(sum)** | **C(carry out)** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Denote your quantum register as 'q', classical register as 'c'. Assign inputs A, B and X to q[0], q[1] and q[2] respectively, the sum output S to c[0] and carry output C to c[1].

**Expected Outputs:**

Get the execution results (comma separated binary outputs) of the full adder you created for inputs 000 to 111.

**Additional Information:**

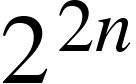
\*\*\*Code to check the gate count

qc.count\_ops()

Here qc is a quantum circuit.

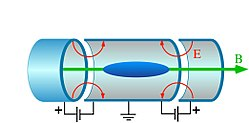
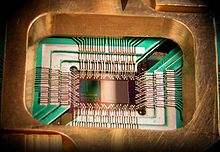
# Week 3 is up and running

Difference between Quantum simulators and Quantum computers

As we know Quantum computers process differently when compared to classical computers but according to church turing thesis even quantum computation algorithms can be simulated on a classical computer with a probabilistic approach. Quantum simulators will mimic the operations of a quantum computer. They can perform  operations in a given time.

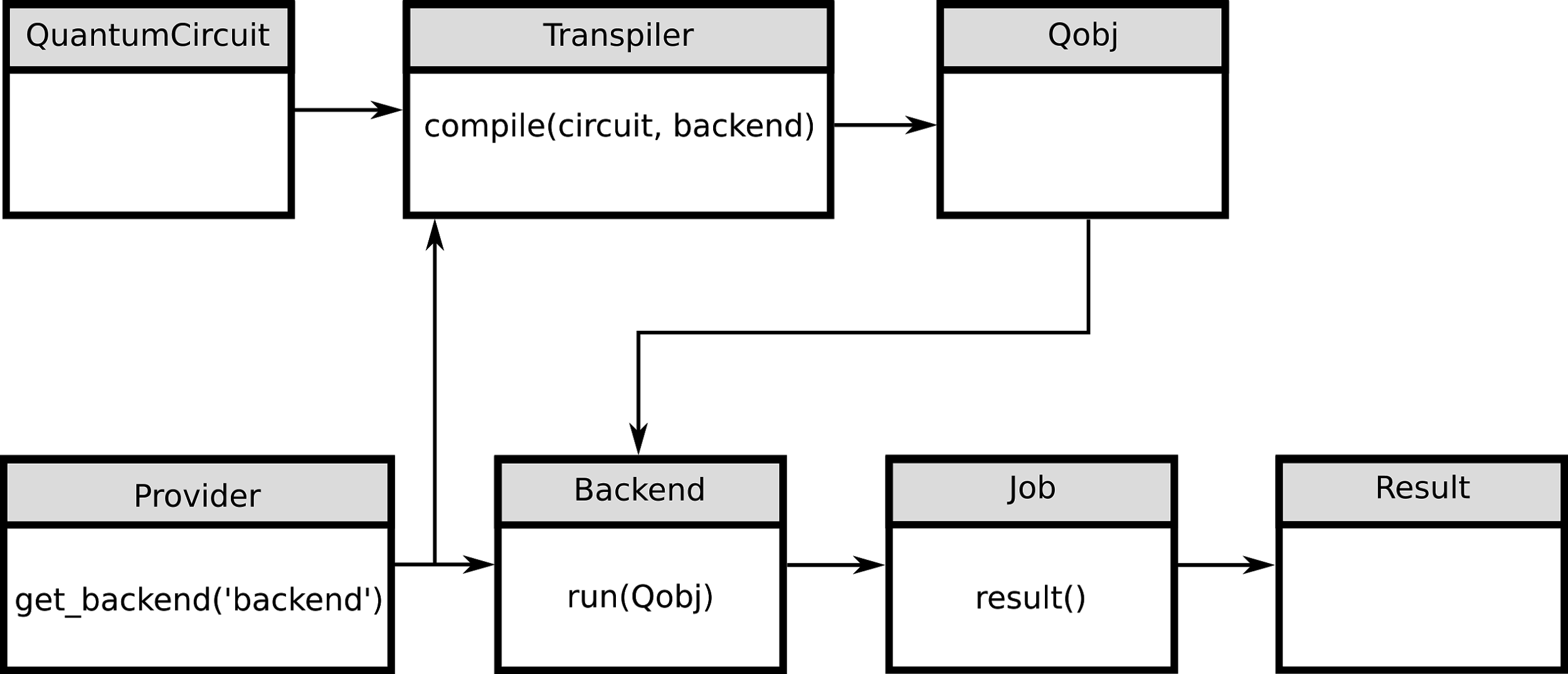
Every major company is trying to build these simulators to try and experiment with the quantum algorithms by making themselves well equipped with good software. Qasm\_simulator is IBM’s quantum assembly simulator which can be used to code and execute quantum algorithms in a probabilistic fashion. This is a good way for computer scientists to experience the quantum software. By the way a universal quantum simulator is proposed by Yuri Manin and Richard Feynman.

There are different types of quantum simulators available these days to solve complex physics problems, especially the problems that have quantum mechanics as their core.

1. **Trapped ION:**These are the most famous kind of simulators which are especially being used by many universities and research centers in many ways.  
     
   Trapped ion simulators consists of plane of beryllium ions which are of millimeter thick and arranged inside a penning trap machine. The outermost electron of the ions acts as a magnet and is controlled by cooling the ions using a laser. Microwaves and Laser pulses are used to make the qubits interact with each other and solve complex problems. Creating 1000’s of individual qubits is easy, but the toughest part is making them work together.  
   \*\*\* Penning trap is a device used to store charged particles using magnetic and electric fields.  
    
2. **Ultracold Atom simulators:**Ultracold gases are used to simulate quantum effects by controlling interparticle interactions using Feshbach resonances. Ultracold atoms offer remarkable opportunities for investigating quantum many-body problems that are relevant to fields as diverse as condensed matter physics, statistical physics, quantum chemistry, and high-energy physics.
3. **Superconducting qubits:**These include quantum annealing which is sometimes called adiabatic quantum computing. Quantum annealing initially starts with a superposition of all possible states with pre-defined equal weights. Then based on the Schrodinger time-dependent equation the system will evolve. The amplitudes of the states keep changing in accordance with the time-dependent transverse field and quantum parallelism.  
      
   DWave One, the first quantum annealer chip with 128 superconducting logic elements.

IBM Q / Qiskit Backends

It has three main components embedded into it

1. **Provider:** Gives you access to the backends and maintains the backend objects.
2. **Backend:** This will execute the quantum circuit.
3. **Job:**  This will keep track of the job once you hit the “run” button. In notebooks you can see the rate and a progress bar with the job queued.  
     
    

The figure above demonstrates how the transpiler, the provider and the backend will work together to compile and run the quantum circuit on the backend chosen. Finally “job” module will get the results and return that to the Results object.

**Providers:**

There are two types of providers available as the backends at IBMQ, they are Aer and IBMQ. Aer has quantum simulators whereas IBMQ has simulators and real quantum computers which can be accessed. BaseProvider is the class that is inherited by the provider to simulate the backends.

Here is the code to look at the list of all quantum simulators.

from qiskit import Aer

for backend in Aer.backends():

print(backend.name())

**Output:**

qasm\_simulator

statevector\_simulator

Unitary\_simulator

Now to run your code on a simulator you will use the below code which you already used in week 2.

1. backend = Aer.get\_backend('qasm\_simulator')
2. job = execute(qc, backend, shots=1000)
3. result = job.result()
4. count =result.get\_counts()
5. print(count)
6. qc.draw(output='mpl')

In line[1] we are getting the qasm simulator as the backend.

In line[2] we are creating a job variable and using the execute() function which has the quantum circuit, backend, and shots as core parameters.

\*\*\*Shots means the number of times your algorithm will run on a simulator. So more shots give rise to more accurate results.

The job object has result as its inbuilt function which will get the result from the job and get counts method is used to know the number of times each possible state is achieved.

**IBMQ backends list:**

* ibmq\_qasm\_simulator in ibm-q/open/main
* ibmq\_16\_melbourne in ibm-q/open/main
* ibmq\_ourense in ibm-q/open/main
* ibmqx2 in ibm-q/open/main
* ibmq\_vigo in ibm-q/open/main
* ibmq\_london in ibm-q/open/main
* ibmq\_burlington in ibm-q/open/main
* ibmq\_essex in ibm-q/open/main

The backends will take the qobj which are quantum objects and compiles them to store the result in BaseJob object. The BaseJob object will allow asynchronous jobs and it retrieves the results as soon as the compilation is done.

Example:

from qiskit import IBMQ

from qiskit import \*

qc=QuantumCircuit(2,2)

qc.h(0)

qc.cx(0,1)

qc.measure(1,0)

# replace 'ibmq\_qasm\_simulator' with a name in ibmq.backends()

backend = IBMQ.get\_backend('ibmq\_qasm\_simulator', hub=None)

job=execute(qc,backend,shots=1000)

result = job.result()

print(result.get\_counts())

job=execute(qc,backend,shots=1000)

**Output:**

{'01': 518, '00': 482}

Berstein Vazirani Algorithm

Imagine a secret code locked inside a box, you are only allowed to perform operations on that number but cannot open and directly look for what that is. Let the enigma inside the box be 6 bits, how many steps do you think a classical computer takes to guess the number accurately? If your answer is 6, you are right. If there are n bits, it takes n steps for a classical computer to guess that string but a quantum computer can do it in only one shot using the Bernstein Vazirani algorithm.

First, let us understand how a classical computer does it. It will perform **AND** operation with each and every bit with one. If the result is 0 then the computer will know it is 0 in that place but if the result is 1 then it will understand that the bit is 1.

Now a quantum computer does this in one step but, Let me describe the process in 3 steps.

1. First, apply Hadamard transform to n qubit state   
    
2. Now apply CNOT gate on the qubits which when applied the oracle will turn the superposition into the following  
      
   Here oracle means a function f(x) which performs dot product x.s where s is the secret string. If the dot product is 1 this will flip the state x.
3. Finally, reapply Hadamard transform to the n qubit state which will convert  to  and  to .

Congrats you’ve unlocked the secret number. The trick here is Hadamard transform puts the qubits in superposition which means no uncle sam’s little steps anymore.

**Algorithm:**

s = '110101' #secret number

n = len(s)

circuit = QuantumCircuit(n+1,n)

circuit.x(n)

circuit.barrier()

circuit.h(range(n+1))

circuit.barrier() #To separate steps

for i, tf in enumerate(reversed(s)):

if tf == '1':

circuit.cx(i, n)

circuit.barrier()

circuit.h(range(n+1))

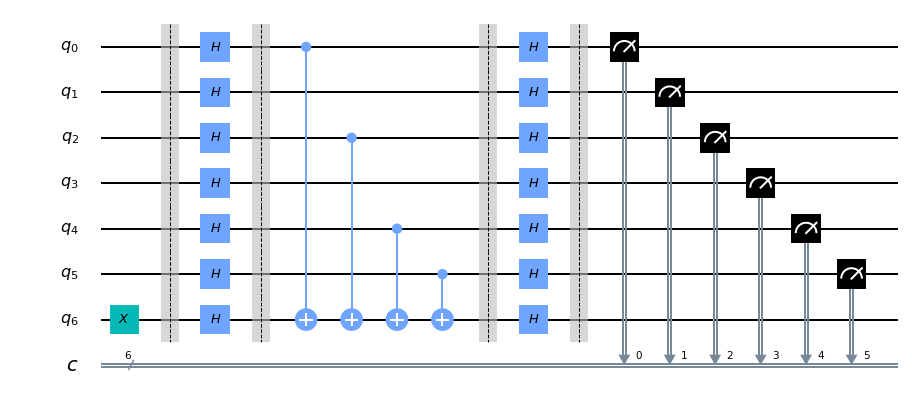
circuit.barrier()

circuit.measure(range(n), range(n))

%matplotlib inline

circuit.draw(output='mpl')

**Output**



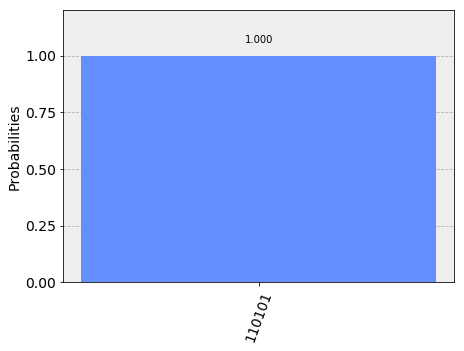
simulator = Aer.get\_backend('qasm\_simulator')

result = execute(circuit, backend=simulator, shots=1).result()

from qiskit.visualization import plot\_histogram

plot\_histogram(result.get\_counts(circuit))

**Output:**



Exercise - 2

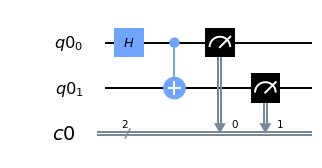
Write a program that will guess alphabetical secret strings using qiskit. Use qasm\_simulator to execute.

If you can successfully complete this exercise on your own upload an image of the string that you tried to guess on twitter and tag me with @RishwiB and #QCUIQbyRB.

# Week 4 - The Spooky Action

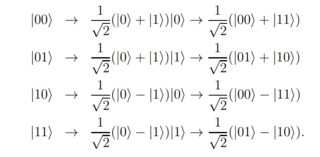
As you know it is called Quantum Entanglement. It means when two particles are entangled with each other measuring one particle will reveal the state of the other particle too, no matter the distance between those particles (or) When two particles are entangled together, their states cannot be explained individually. The states of the particles will be correlated when they are entangled together. This analogy is termed in the late 1990s by Sandu Popescu and Daniel Rollick. Einstein opposed the view of Neils Bohr by terming entanglement as “spooky action at a distance”. On the left is a real image of two entangled particles. Einstein said that nothing can travel faster than light but if entanglement is possible magically the measurement of the state of one particle is deciding the other by acknowledging it. This magic transmission was thought of as a paradox by Einstein, Podolsky, and Rosen in 1935. Einstein called these unmeasurable quantities as “Hidden Variables”.

When two particles are entangled they share the same spatial proximity in such a way that the state of a particle cannot be described independently from the other. Entanglement is a different phenomenon that no classical computer can demonstrate. Physicists sometimes call this phenomenon as Quantum nonlocality.

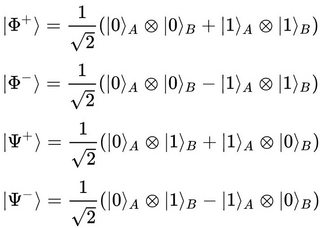


The figure demonstrates the entanglement circuit. Let’s go a little deep into it. Qubit q0 is pushed into superposition. It is now called the phase bit. In the meantime, the second qubit is attached to a CNOT. So now the second qubit is called parity bit.

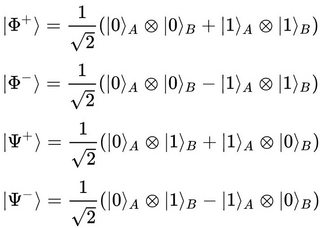
Phase bit will be in one of the four states. These states will result in an output.



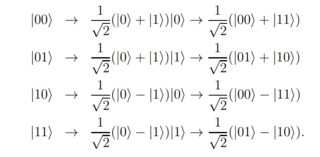
These are also called Bell states. They clearly represent the operation of the above circuit.

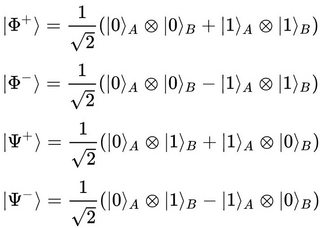


Now Let’s draw align circuits for the above equations on IBM Q Experience.

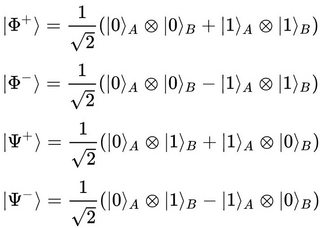
:

|  |  |
| --- | --- |
| Code | Circuit |
| qc = QuantumCircuit(2)  qc.h(0)  qc.cx(0,1)  %matplotlib inline  qc.draw(output='mpl') |  |

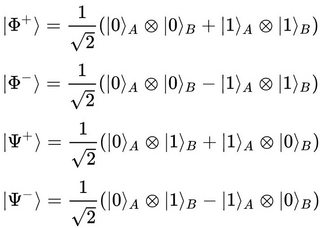
The control bit is 1 so it flips the multiplying qubit. Therefore after multiplying it will become

:

|  |  |
| --- | --- |
| Code | Circuit |
| qc = QuantumCircuit(2)  qc.h(0)  qc.cx(0,1)  qc.z(0)  %matplotlib inline  qc.draw(output='mpl') |  |

:

|  |  |
| --- | --- |
| Code | Circuit |
| qc = QuantumCircuit(2)  qc.h(0)  qc.cx(0,1)  qc.x(1)  %matplotlib inline  qc.draw(output='mpl') |  |



|  |  |
| --- | --- |
| Code | Circuit |
| qc = QuantumCircuit(2)  qc.h(0)  qc.cx(0,1)  qc.z(0)  qc.x(1)  %matplotlib inline  qc.draw(output='mpl') |  |

Try these codes using a simulator as well as an original quantum computer. Find the probabilities of the states.

Quantum Teleportation

This is an application based on Entanglement phenomenon where

We may not be able to transport people or macroscopic scaled objects at the moment but can surely transport qubits over a particular distance by entangling them and transmitting the state of the original qubit’s state using the entanglement phenomenon.

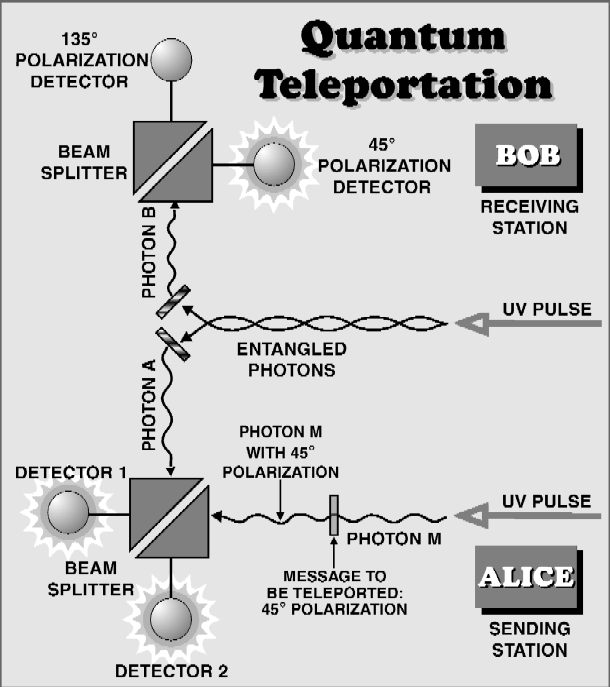
Quantum entanglement occurs when the qubits are forced to hold mutually exclusive states, which brings the observer to a situation where he will be able to determine the state of a qubit if the state of its entangled qubit is known.

Think of it in this way, you bought two books from a shop, a black-covered and a blue-covered one, packed differently. So on opening a cover, you’ll be able to decide what’s in the other one. If the cover you opened has a blue-covered book then you know that black-covered book is in the other cover or vice-versa.

Steps for quantum teleportation:

* First, the desired qubit that has to be teleported is identified.
* Then a classical channel that is capable of transmitting two bits(classical) is created, and an EPR entangled pair is generated.
* Each of these qubits that are entangled is transported to two different locations, say A and B.
* Bell’s measurement is performed on one of the qubits and information is obtained, then the state of another qubit is manipulated accordingly.

Let us try teleportation with information Alice wants to send to Bob using quantum entanglement.



Let us assume that the information Alice wants to send Bob is:



Alice and Bob share maximum entanglement, so the state of the information can be in one of the four possible states:









Let us assume that Alice and Bob share  Now the complete system has three particles, Alice has two qubits( C, the one that is to be teleported and A, qubit that is one of the EPR pair) and Bob has one qubit that is one of the EPR pair.

So, the system at A can be given by:



Alice then makes a bell measurement which will lead her to the superposition of the Bell’s basis:

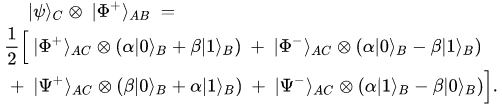








Thus, the total system together will give:



No measurement has been done so far, hence there will be no change in the states of the qubits.

The change will occur when a measurement is done for teleportation on:



The measurement can be done by directing laser pulses on the qubits. So after measurement qubits would collapse into one of the following states:

* 
* 
* 
* 

You can now see that the entanglement between Alice and Bob is broken and is formed in between the qubits that Alice possesses. And note one of the 4 superposition states of qubit Bob has exactly matched to the qubit that has to be teleported.

The result of Bell’s measurement will let Alice know which of the 4 possible states is she in so that she can send the information using the classical channel to Bob.

After receiving the information from Alice, if it seems to be:

* , then Bob does nothing as it is already in the desired state.
* , then Bob applies a unitary quantum gate to recover the state that is given by Pauli matrix:

,

* If it is , then Bob applies the following gate to his qubit.



* If it is the last case then he will apply the following gate:



And thereby teleportation is achieved.

Results:

* Bob qubit’s state will be , and Alice qubit’s state will be undefined, thus not violating in the no-cloning theorem.
* The information is not physically moved, so there is no transfer of energy.
* The classical bits do not know about the teleportation completely, so if an eavesdropper tries to interfere, Alice might interact with Bob using the entanglement.

**No-teleportation theorem**

It states that a random quantum state cannot be converted into a sequence of traditional classical bits and neither can they be used to reconstruct the original state.

Put it in other words, it is an extension for no-cloning theorem. If you can convert a qubit into a series of classical bits, then it would be easy to copy the qubit which is against the quantum laws, this will lead to the destruction of the original state. So a qubit cannot be copied as the series of classical bits.

We’ve so far discussed the topics related to teleportation. Now let us take a look at the advancements in Quantum mechanics regarding the subject.

**Implementation of Quantum teleportation using Qiskit:**

**Algorithm:**

**from qiskit import \***

**%matplotlib inline**

**circuit = QuantumCircuit(3,3)**

**circuit.x(0)**

**circuit.barrier()**

**circuit.h(1)**

**circuit.cx(1,2)**

**circuit.cx(0,1)**

**circuit.h(0)**

**circuit.barrier()**

**circuit.measure([0,1],[0,1])**

**circuit.barrier()**

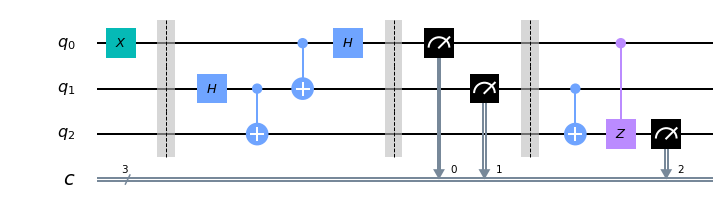
**circuit.cx(1,2)**

**circuit.cz(0,2)**

**circuit.measure(2,2)**

**circuit.draw(output='mpl')**

**Output:**

****

**Implementation using qasm simulator**

**simulator = Aer.get\_backend('qasm\_simulator')**

**result = execute(circuit, backend=simulator, shots=1024).result()**

**Plotting the result:**

**from qiskit.visualization import plot\_histogram**

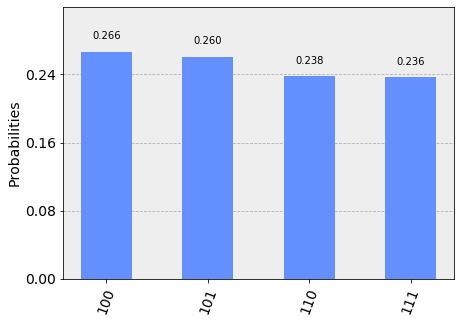
**counts=result.get\_counts()**

**print(counts)**

**plot\_histogram(counts)**

**Output:**

**{'101': 266, '110': 244, '111': 242, '100': 272}**

****

**Advancements in Quantum Teleportation:**

This concept of physics where it is possible that a particle can travel faster than light bothered Einstein so much that he started thinking of it in a different way and also wrote a paper on spooky action at a distance.

* In 2014, the scientists proved that it is possible and entangled two qubits that are separated by 25km.
* In 2015, the separation was increased to 100km, and it is the physical limit because the material they used it a fiber-optic cable and if the limit is exceeded it would cause too many losses.
* In 2017, scientists from China tested this phenomenon in space. They’ve successfully sent a photon from Earth to the quantum satellite. The scientists created an exact replica of the state of the photon(through which we will be able to describe the behavior and movement of the photons) on the Earth to the outer orbit.
* This was not an ordinary experiment that is done using entanglement between the qubits for over short distances. The teleportation was performed between two sources that are separated by 1400km. They were successful in performing the same experiment for over 900 times.
* They had an entangled pair of photons, one of them on the Earth and the other on the satellite. They beamed them down to two different stations on the Earth that are 1000km apart.
* In the experiment, they tried to create a replica of the state of polarisation of a single photon using quantum entanglement. The satellite is named after a Chinese philosopher, Micius. The two entangled bits are called as *Ngari* and *Micius.*
* Ngari performs a joint measurement on the photon 1 (that is to be teleported) and the photon 2 from the entangled pair, projecting the photons into one of the four Bell states. To maximize the count rate in the experiment they used four ultra-bright photon sources and they achieved.
* And in 2019, the sharp-witted scientists moved a step ahead and instead of teleporting a qubit they tried to teleport a qutrit, is a part of quantum information that is in a superposition of three mutually orthogonal states. It is surely a big step towards the future.
* The state that is to be teleported is represented in the possible ways the photon can travel. You can just imagine this as paths as three different optical fibers. The most fun part is you can see the same photon in three different paths at the same time. They used bell’s measurement which is based on a multiport beam splitter that will connect all the optical fibers that direct’s photons through many inputs and outputs. They also used auxiliary photons that may interfere with other photons.
* So, now they transferred information from an input photon to another one that is physically separated through a wise selection of interference patterns. They say that this phenomenon is not bounded by any dimensions. Hence, this process might take place even in higher dimensions.

**Real-time applications of teleportation:**

The idea of trying to transfer information without any physical movement sounds fascinating, but what can it be used for? Scientists see this as a solution to building an ideal quantum internet.

Though it is still under development it may replace the traditional network in the near future.

* Quantum internet is a reliable network that connects devices to a remote quantum computer that can perform operations way faster than a classical computer without even actually finding out what the computations are for.
* In quantum cryptography that involves more than two parties, for example, consider a tripartite quantum network for three parties (Alice, Bob, and Claire) who share tripartite entanglement In such case quantum teleportation will be successful between any two members with the help of the third one. This can serve as a reliable tripartite quantum protocol based on the physics of teleportation that will be a failure without the help of the third member. In other words, there can be no bipartite entanglement in the tripartite entangled state.

Exercise 3

1. Study about No-cloning theorem
2. Teleport state 0 using the above algorithm
3. Try the teleportation with superposition state.( hint: apply H gate instead of X for q0).

# Week - 5, Get Ready for some DJ

Deutsch-Josza Algorithm

It is the simplest and first algorithm invented by David Deutsch and Richard Josza where it takes a conventional computer 2n-1 evaluations when compared to a quantum computer that can do it in 1 shot. It basically takes n inputs and gives one-bit value as the output. But the point here is not whether the output is 0 or 1. It is about the function . It is about deciding whether the function is constant or balanced.

People termed something called and Oracle which performs an XOR operation. This problem is called a black box. Here are the steps to demonstrate how the algorithm works.

1. We need to take n qubits where n-1 are in the state  which, let us term as the upper states and the last one is in the state  which will be the lower state.
2. Apply Hadamard transform on all the n qubits.
3. Now apply the oracle on the upper and lower state where it performs XOR.
4. After that reapply the Hadamard transform to all the upper states.
5. Finally, measure all the qubits.

So, after measuring the answer qubit if it turns out to be 0, then the function is constant else it is balanced. Here constant means the function will output the same value independent of the input, whereas balanced means the function generates half a bunch of 0’s and another half of 1’s.

Here goes the algorithm using Qiskit…

import numpy as np

import matplotlib.pyplot as plt

from qiskit import BasicAer, IBMQ

from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, execute

from qiskit.compiler import transpile

from qiskit.tools.monitor import job\_monitor

from qiskit.tools.visualization import plot\_histogram

%matplotlib inline

n = 13 # No of qubits in state 0

# Choose a type of oracle at random. With probability half it is constant,

# and with the same probability it is balanced

oracleType, oracleValue = np.random.randint(2), np.random.randint(2)

if oracleType == 0:

print("The oracle returns a constant value ", oracleValue)

else:

print("The oracle returns a balanced function")

a = np.random.randint(1,2\*\*n) # this is a hidden parameter for balanced oracle.

# Creating registers

# n qubits for querying the oracle and one qubit for storing the answer

qr = QuantumRegister(n+1) #all qubits are initialized to zero

# for recording the measurement on the first register

cr = ClassicalRegister(n)

circuitName = "DeutschJozsa"

djCircuit = QuantumCircuit(qr, cr)

# Create the superposition of all input queries in the first register by applying the Hadamard gate to each qubit.

for i in range(n):

djCircuit.h(qr[i])

# Flip the second register and apply the Hadamard gate.

djCircuit.x(qr[n])

djCircuit.h(qr[n])

# Apply barrier to mark the beginning of the oracle

djCircuit.barrier()

if oracleType == 0:#If the oracleType is "0", the oracle returns oracleValue for all input.

if oracleValue == 1:

djCircuit.x(qr[n])

else:

djCircuit.iden(qr[n])

else: # Otherwise, it returns the inner product of the input with a (non-zero bitstring)

for i in range(n):

if (a & (1 << i)):

djCircuit.cx(qr[i], qr[n])

# Apply barrier to mark the end of the oracle

djCircuit.barrier()

# Apply Hadamard gates after querying the oracle

for i in range(n):

djCircuit.h(qr[i])

# Measurement

djCircuit.barrier()

for i in range(n):

djCircuit.measure(qr[i], cr[i])

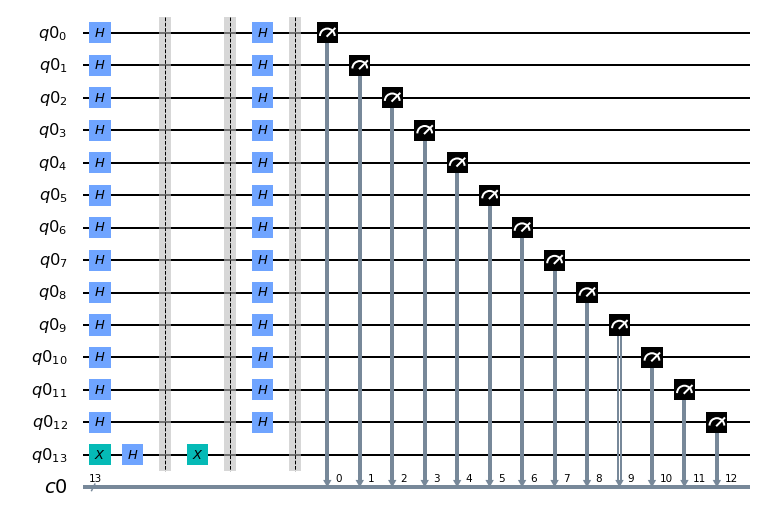
**Output:**

The oracle returns a constant value 1

#draw the circuit

djCircuit.draw(output='mpl',scale=0.5)

**Output:**



#executing on simulator

backend = BasicAer.get\_backend('qasm\_simulator')

shots = 1000

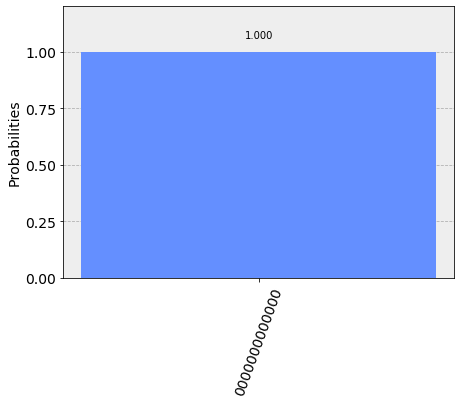
job = execute(djCircuit, backend=backend, shots=shots)

results = job.result()

answer = results.get\_counts()

plot\_histogram(answer)

**Output:**

****

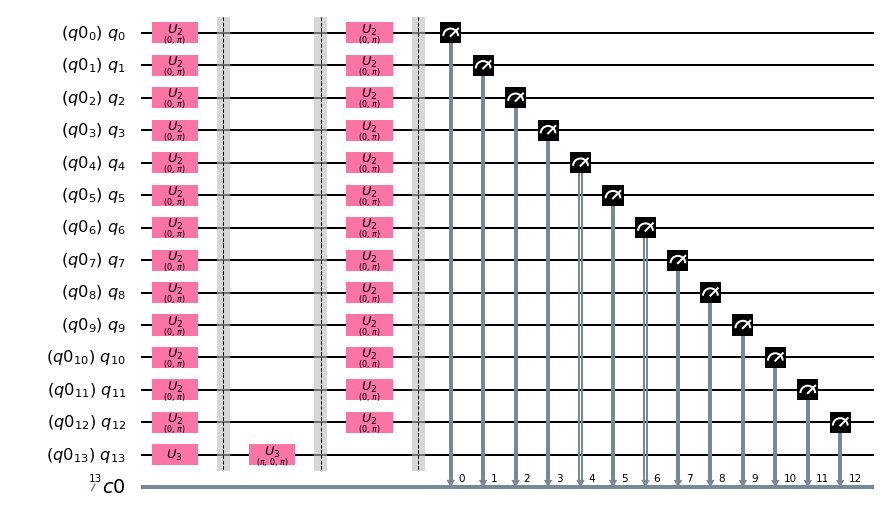
**#Now let's implement it on Real device**

**backend = IBMQ.get\_backend('ibmq\_16\_melbourne')**

**djCompiled = transpile(djCircuit, backend=backend, optimization\_level=1)**

**djCompiled.draw(output='mpl',scale=0.5)**

**Output:**

****

**#This step might take some time**

**job = execute(djCompiled, backend=backend, shots=1024)**

**job\_monitor(job)**

**results = job.result()**

**answer = results.get\_counts()**

**threshold = int(0.01 \* shots) # the threshold of plotting significant measurements**

**filteredAnswer = {k: v for k,v in answer.items() if v >= threshold} # filter the answer for better view of plots**

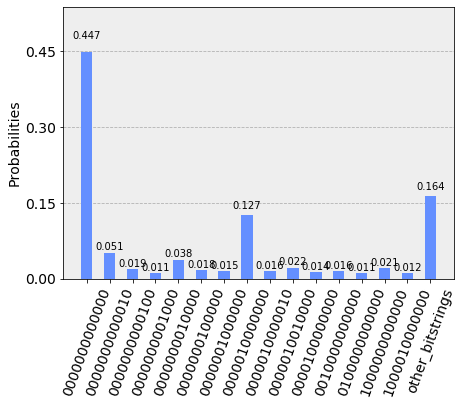
**removedCounts = np.sum([ v for k,v in answer.items() if v < threshold ]) # number of counts removed**

**filteredAnswer['other\_bitstrings'] = removedCounts # the removed counts are assigned to a new index**

**plot\_histogram(filteredAnswer)**

**#Quantum computers have noise so they wouldn't produce discrete outputs like simulators - that only demonstrates the superioity of the quantum over classcial**

**Output:**

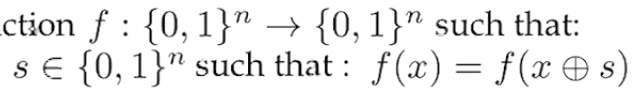
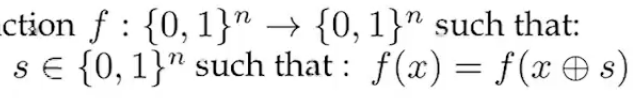
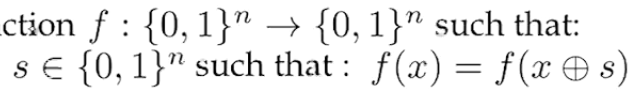
****

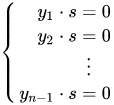
**print(filteredAnswer)**

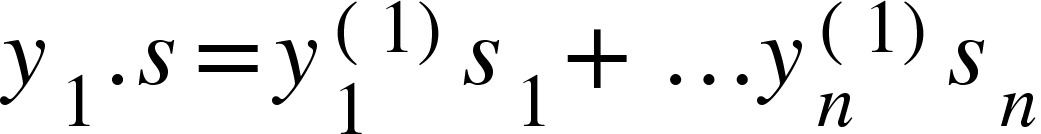
**Output:**

**{'0000000000010': 52, '0000000010000': 39, '0000010000010': 16, '0000001000000': 15, '1000010000000': 12, '0000010010000': 23, '0000000000000': 458, '0010000000000': 16, '0000000001000': 11, '0100000000000': 11, '1000000000000': 22, '0000000000100': 19, '0000010000000': 130, '0000000100000': 18, '0000100000000': 14, 'other\_bitstrings': 168}**

Simon’s Algorithm

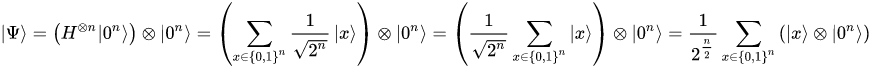
It takes  as input, two sets of n qubits. Assume there is a secret number  so that . We need to build a quantum circuit and iterate it n-1 times to find n-bit linearly independent strings, i.e, 

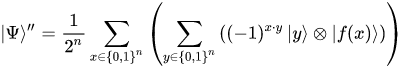


These are the n-1, n-bit equations, where .

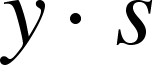
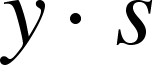
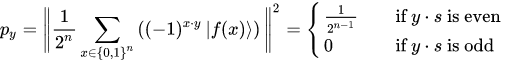
Below are the steps involved in it.

1. First, it will put the first set of n qubits in superposition with Hadamard transform.



1. Apply the oracle that we learned in DJ algorithm above, without phase shifts. This will result in  
   
2. Now, finally, apply the Hadamard transform to the n qubits again, which will result in  
   where 
3. Don’t forget to measure at the end.

There are two types of outputs for this algorithm, they are

1.  where  is a one-to-one function. This is the case where probabilities () of n-bit string are uniformly distributed.
2.  and  , which will give a resultant as follows  
     
   This again has two follow up cases  
   I. If  is an odd number then it turns out to be zero which is a no result outcome.  
   II. If  is an even number then   
   

**Implementation using Qiskit:**

from qiskit import \*

import matplotlib.pyplot as plt

%matplotlib inline

import numpy as np

from qiskit.providers.ibmq import least\_busy

from qiskit.tools.visualization import plot\_histogram

s = '10'

#create a quantum register which is double the size of the secret string.

qr = QuantumRegister(2\*len(str(s)))

cr = ClassicalRegister(2\*len(str(s)))

circuit = QuantumCircuit(qr, cr)

# First hadamard phase

for i in range(len(str(s))):

circuit.h(qr[i])

# Apply barrier just to separate

circuit.barrier()

# 2 qubit oracle for s = 10

circuit.cx(qr[0], qr[len(str(s)) + 0])

circuit.cx(qr[0], qr[len(str(s)) + 1])

circuit.cx(qr[1], qr[len(str(s)) + 0])

circuit.cx(qr[1], qr[len(str(s)) + 1])

circuit.barrier()

# Measure ancilla qubits

for i in range(len(str(s)), 2\*len(str(s))):

circuit.measure(qr[i], cr[i])

circuit.barrier()

# Another hadamard phase

for i in range(len(str(s))):

circuit.h(qr[i])

circuit.barrier()

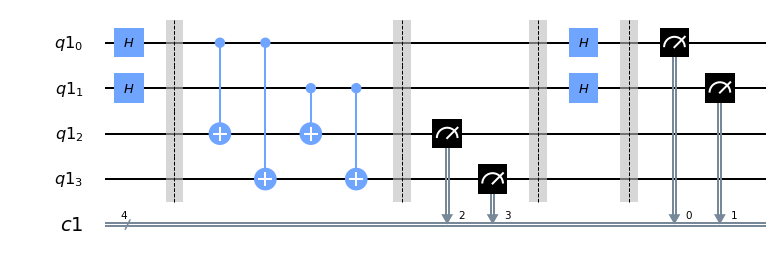
# Measure the input register

for i in range(len(str(s))):

circuit.measure(qr[i], cr[i])

circuit.draw(output='mpl')

**Output:**

****

**# First execute with simulator**

**backend = BasicAer.get\_backend('qasm\_simulator')**

**shots = 1024**

**results = execute(circuit, backend=backend, shots=shots).result()**

**answer = results.get\_counts()**

**# Categorize measurements by input register values**

**answer\_plot = {}**

**for measresult in answer.keys():**

**measresult\_input = measresult[len(str(s)):]**

**if measresult\_input in answer\_plot:**

**answer\_plot[measresult\_input] += answer[measresult]**

**else:**

**answer\_plot[measresult\_input] = answer[measresult]**

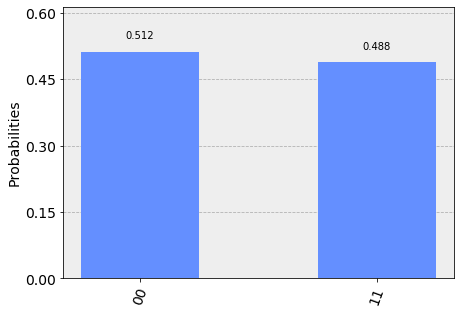
**# Plot the categorized results**

**print( answer\_plot )**

**plot\_histogram(answer\_plot)**

**Output:**

**{'11': 500, '00': 524}**

****

**# Calculate the dot product of the results**

**def sdotz(a, b):**

**accum = 0**

**for i in range(len(a)):**

**accum += int(a[i]) \* int(b[i])**

**return (accum % 2)**

**print('s, z, s.z (mod 2)')**

**for z\_rev in answer\_plot:**

**z = z\_rev[::-1]**

**print( '{}, {}, {}.{}={}'.format(s, z, s,z,sdotz(s,z)) )**

**Output:**

s, z, s.z (mod 2)

11, 11, 11.11=0

11, 00, 11.00=0

# Load our saved IBMQ accounts and get the least busy backend device with less than or equal to 5 qubits

provider = IBMQ.get\_provider(hub='ibm-q')

provider.backends()

backend = least\_busy(provider.backends(filters=lambda x: x.configuration().n\_qubits <= 5 and not x.configuration().simulator and x.status().operational==True))

print("least busy backend: ", backend)

# Run our circuit on the least busy backend. Monitor the execution of the job in the queue

from qiskit.tools.monitor import job\_monitor

shots = 1024

job = execute(circuit, backend=backend, shots=shots)

job\_monitor(job, interval = 2)

# Categorize measurements by input register values

answer\_plot = {}

for measresult in answer.keys():

measresult\_input = measresult[len(str(s)):]

if measresult\_input in answer\_plot:

answer\_plot[measresult\_input] += answer[measresult]

else:

answer\_plot[measresult\_input] = answer[measresult]

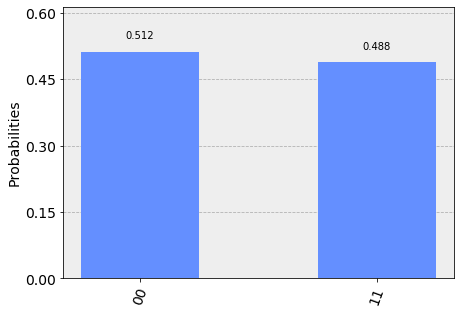
# Plot the categorized results

print( answer\_plot )

plot\_histogram(answer\_plot)

**Output:**

**{'11': 500, '00': 524}**

****

**# Calculate the dot product of the most significant results**

**print('s, z, s.z (mod 2)')**

**for z\_rev in answer\_plot:**

**if answer\_plot[z\_rev] >= 0.1\*shots:**

**z = z\_rev[::-1]**

**print( '{}, {}, {}.{}={}'.format(s, z, s,z,sdotz(s,z)) )**

**Output:**

**s, z, s.z (mod 2)**

**11, 11, 11.11=0**

**11, 00, 11.00=0**

As we understood initial and important algorithms, now lets discuss about Quantum Fourier Transform which has its own importance in various algorithm like the Shor’s.

Quantum Fourier Transform

It is a basic building block for famous algorithms such as the Shor’s algorithm It is a process of linear mapping of fundamental algebraic structures. It was invented by Don CopperSmith. A quantum computer can perform discrete quantum fourier transform in using Hadamard gates and some phase shift gates when compared to classical one does it in , where quantum computer shows exponential growth in time complexity.

First, to understand Quantum Fourier Transform also called as QFT you need to be clear with two points.

1. It maps one vector to another.
2. In classical sense a Fourier Transform maps Vector X =  to vector Y =  using a well known formula.  
     
    where 

Now, coming to the quantum sense there is only a minute change that a QFT acts on quantum states. Consider a basis state , applying QFT on it will result following operation.

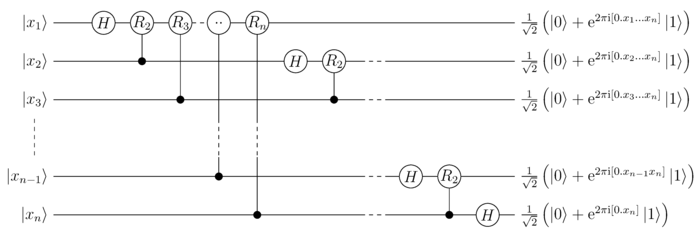


Let’s implement this using IBM Q Experience or Qiskit

Before diving into it you need to know what are phase shift gates. They are single qubit gates which impact state  and will map it to . It shifts the phase of a vector in bloch sphere. Even swap gate and Z-gate are sometimes represented using this phase gates where  and  respectively. It’s matrix representation is as follows



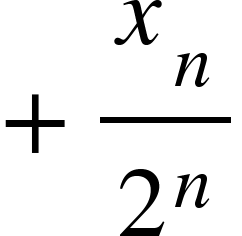
First column is the case of basis state  and the second one is for state  where it multiplies with .



The above is the image of the quantum Fourier transform where the phase shift part is clearly demonstrated at the right end of the image. The tensor product is applied to all possible states of a qubit. The following is the equation to demonstrate the tensor products.

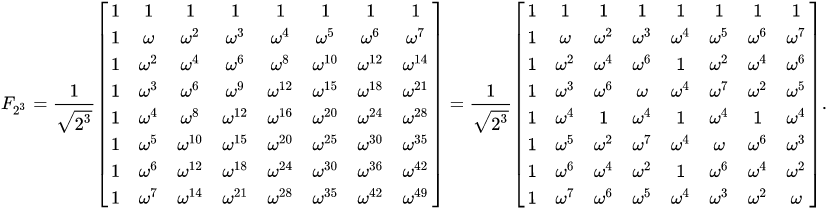


In the above equation 

where , ......., =.......

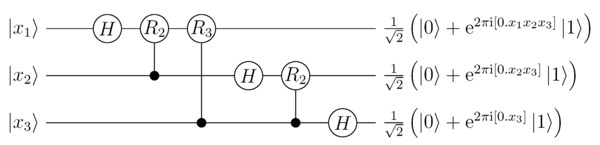
If you take a 3-qubit system the equation for QFT will be as follows



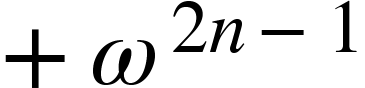
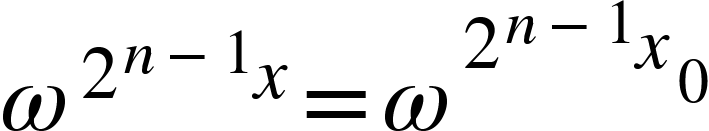


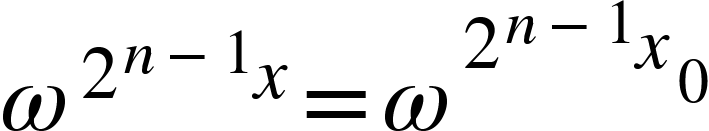
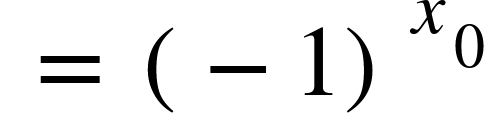
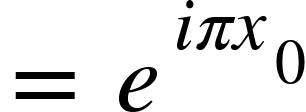
The base of F is  which denotes dimension of Hilbert space for a n-qubit system.

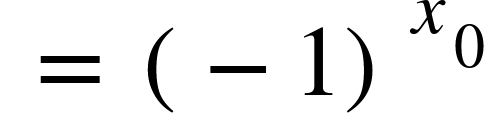
The circuit diagram for the 3-qubit system is as follows



Assume q[n] is an array of qubits where q[n-1] is called the most significant bit.

 - ( 1 ), this is the representation of the most significant bit. Now the value of  can be calculated and that shows us the following.

  - ( 2 )

Taking equation ( 2 ) into consideration ( 1 ) can be written as <math xmlns="http://www.w3.org/1998/Math/MathML"><mo>+</mo></math>. This can be obtained by applying Hadamard transform to the least significant state . For the states q[n-2] …. Q[0] we will be applying a Hadamard transform and followed by controlled phase shift gates.

Following is the Qiskit code to execute QFT:

from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit

from qiskit.visualization import circuit\_drawer as drawer

from qiskit import execute

from qiskit import Aer

import numpy as np

%matplotlib inline

def QFT(q,c,n=3):

circuit = QuantumCircuit(q,c)

# First get the most significant bit

for k in range(n):

j = n - k

# Now add the hadamard transform to qubit j-1

circuit.h(q[j-1])

# Now each qubit from the lowest significance takes one conditional phase shift

for i in reversed(range(j-1)):

circuit.cu1(2\*np.pi/2\*\*(j-i),q[i], q[j-1])

# Finally swap the qubits

#This gate swaps the order of the qubits

for i in range(n//2):

circuit.swap(q[i], q[n-i-1])

return circuit

#QFT is represented in a matrix form with 2^n rows and columns

#n represents number of qubits

def QFTmatrix(n):

qft = np.zeros([2\*\*n,2\*\*n], dtype=complex)

for i in range(2\*\*n):

for j in range(2\*\*n):

qft[i,j] = np.exp(i\*j\*2\*1j\*np.pi/(2\*\*n))

return 1/np.sqrt(2\*\*n)\*qft

def QFTCircuit(n):

q = QuantumRegister(n,"x")

c = ClassicalRegister(n,"c")

circuit = QFT(q,c,n)

backend = Aer.get\_backend('unitary\_simulator')

job = execute(circuit, backend)

actual = job.result().get\_unitary()

np.around(actual,2)

expected = QFTmatrix(n)

delta = actual - expected

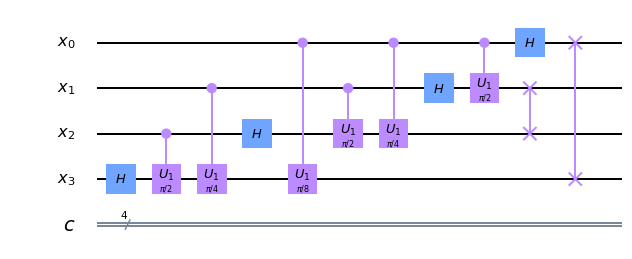
print("Deviation: ", round(np.linalg.norm(delta),10))

return circuit

circuit = QFTCircuit(n=4)

drawer(circuit, output="mpl")

**Output:**



n=4

q = QuantumRegister(n,"x")

c = ClassicalRegister(n,"c")

qftCircuit = QFT(q,c,n)

initCircuit = QuantumCircuit(q,c)

for i in range(n):

initCircuit.h(q[i])

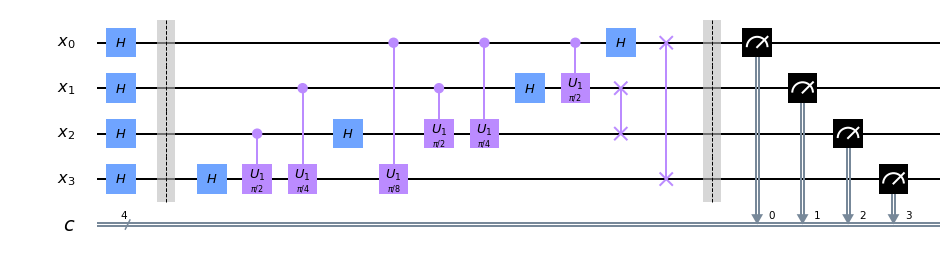
initCircuit.barrier(q)

circuit = initCircuit + qftCircuit

circuit.barrier(q)

circuit.measure(q,c)

drawer(circuit, output="mpl")



# On simulator

backend = Aer.get\_backend('qasm\_simulator')

job = execute(circuit, backend)

k=job.result().get\_counts()

# On Quantum Computer

from qiskit import IBMQ

backend = IBMQ.get\_backend('ibmq\_16\_melbourne')

print("Status of backend: ", backend.status())

job = execute(circuit, backend=backend, shots=1024)

import time

lapse = 0

# This step might take sometime.

time.sleep(3)

interval = 60

while (job.status().name != 'DONE') and (job.status().name != 'CANCELLED') and (job.status().name != 'ERROR'):

print('Status @ {} seconds'.format(interval \* lapse))

print(job.status())

print(job.queue\_position())

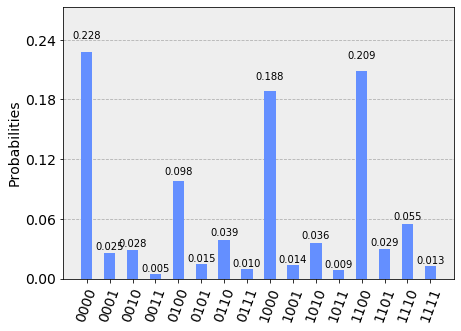
time.sleep(interval)

lapse += 1

print(job.status())

from qiskit.tools.visualization import plot\_histogram

plot\_histogram(job.result().get\_counts())



job.result().get\_counts()

Output:

{'0000': 233,

'0001': 26,

'0010': 29,

'0011': 5,

'0100': 100,

'0101': 15,

'0110': 40,

'0111': 10,

'1000': 193,

'1001': 14,

'1010': 37,

'1011': 9,

'1100': 214,

'1101': 30,

'1110': 56,

'1111': 13}

Exercise 4:

Write, simulate and execute QFT for 3 qubit system and 5 qubit system.

Hint: Change the value of n

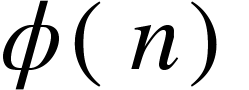
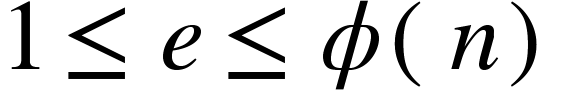
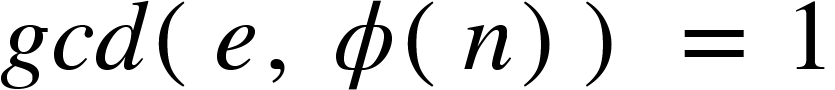
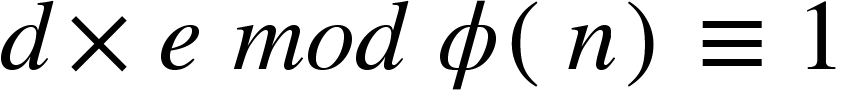
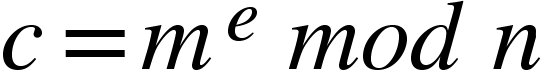
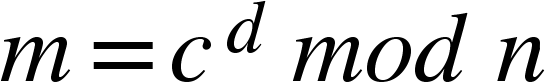
# Woohoo it’s a Sixer - (Week -6)

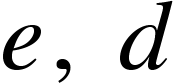
RSA Encryption

Rivest - Shamir - Adleman algorithm is the most famous encryption algorithm which is still being used by many major companies and banks. But it may not be the future of encryption if the quantum era takes a leap. RSA uses prime numbers to encrypt the data so as to decrypt it without the key one should factorize the prime numbers which can be done for small prime number like 5, 13 with the quantum technology that exist today which would take a classical computer several years. But no AI can yet accurately predict the future, so by the time you complete reading this book there may be a major leap.

RSA is sometimes addressed as public crypto system which is asymmetric and uses two keys, a public and a private. Public is accessible to anyone which is used to encrypt the data and which produces a private key which is sent to the receiver through a secure channel, so that he can use that private key to decrypt.

Here are the steps for RSA Algorithm:

1. Take two random large prime numbers p, q.
2. Now assume n = p\*q and = ( p - 1 )\*( q - 1 )
3. Let <math xmlns="http://www.w3.org/1998/Math/MathML"><mi>e</mi></math> be the encryption key where  and 
4. If d is the decryption key then it should satisfy 
5. Assume that <math xmlns="http://www.w3.org/1998/Math/MathML"><mi>m</mi></math> is plain text and <math xmlns="http://www.w3.org/1998/Math/MathML"><mi>c</mi></math> is ciphertext that we acquire after encrypting the plain text as follows 
6. Decryption will go as follows 

( public key, private key ) = (  )

( plain text, ciphertext ) = ( <math xmlns="http://www.w3.org/1998/Math/MathML"><mi>m</mi><mo>,</mo><mo>&#xA0;</mo><mi>c</mi></math> )

If you take a small number like 3, the text can be easily decrypted without factorizing, so choose a large prime number.

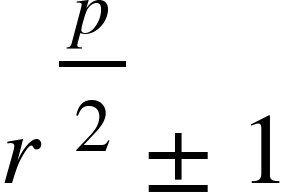
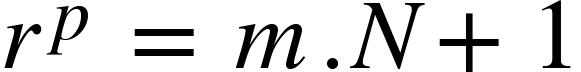
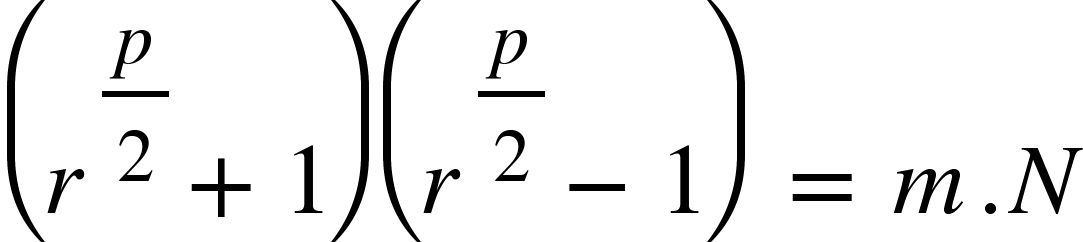
These days industries are using 2048 bit RSA which they recently updated from 1024 bit version because of advancements in the cryptographic field.

Shor’s Algorithm

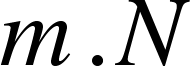
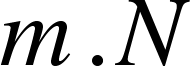
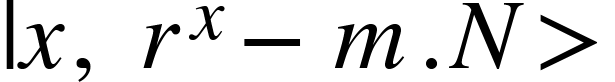
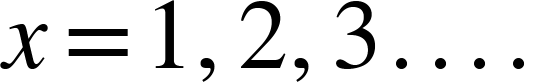
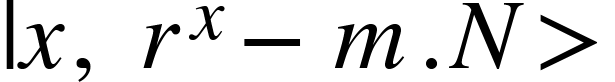
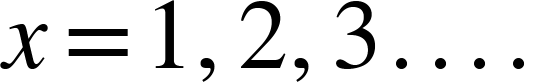
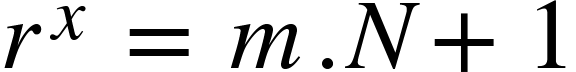
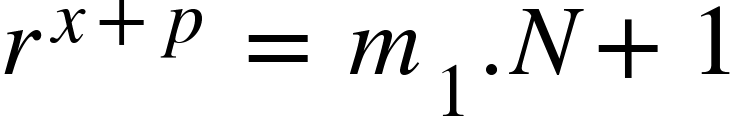
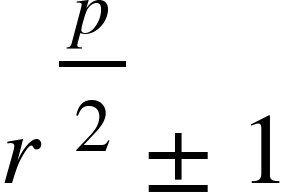
To simply define Shor’s algorithm in one line, it finds prime factors for an integer N. It is a quantum algorithm developed by peter shor in 1994. With the help of a few classical lines of code, it is used to hack the RSA encryption. Although it needs enormously more qubits than those exist currently, as quantum technology takes a leap, a quantum computer can even hack a 2048 bit RSA encryption.

Shor’s algorithm gives exponential speed up over well known classical algorithm for factorizing. This is because of the usage of Quantum Fourier transform and a method called Modular Exponentiation which is used to find the remainder when a variable is raised to the exponent power and divided by a positive integer. This clears up the hardest task that stops a classical computer to factor prime numbers.

Although the decoherence and errors cause issues, IBM was successful to factor 15 to 3 x 5 in the year 2001. They used NMR technology which is an abbreviation for Nuclear Magnetic Resonance, where it uses Nuclear spins as qubits. They control the states and measure them using Radio Frequency waves. In 2012, number 21 was factored which is the largest number factored till now.

This algorithm can be classified into two parts, a classical part which is executed on a classical computer and a quantum part which takes the help of a quantum computer. First, if there is a number  used for RSA encryption, one will need factors of that number to decrypt it. In the classical case, we should go one guessing one number at a time and check if it works. This could take an eternity. Even Shor’s algorithm starts with a random guess of a number. Let that random number be r. But it then will try to find two numbers  because it is obvious that  is true, which can also be written as .

Steps:

1. Take a value x and raise it to the power of r i.e <math xmlns="http://www.w3.org/1998/Math/MathML"><msup><mi>r</mi><mi>x</mi></msup></math> and find how much bigger it is than  which should give a result of the power raised and the excess amount to , i.e. A quantum computer will not do all this by taking one number at a time. Instead it puts <math xmlns="http://www.w3.org/1998/Math/MathML"><msup><mi>r</mi><mi>x</mi></msup></math> in a superposition state in which  at the same time.
2. In the second stage it doesn’t just calculate the gap for one <math xmlns="http://www.w3.org/1998/Math/MathML"><msup><mi>r</mi><mi>x</mi></msup></math> at a time. Instead it uses superposition to calculate  where . So this is the benefit with a quantum computer.
3. If  then there should be a condition where . Now these values maintain a frequency. Its frequency can be calculated by doing inverse of the period, where Quantum Fourier transform that we learned last week comes into the play.
4. QFT can now perform destructive interfernce and can give us that one value of p. If p is even, then substitute it in  to get the factors.
5. If p is odd the then start again from guessing a random number.

**Algorithm:**

**MIT License**

**Copyright (c) 2018 christianb93**

**from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit**

**import numpy as np**

**%matplotlib inline**

**from qiskit import Aer**

**from qiskit import execute**

**from qiskit.visualization import circuit\_drawer as drawer**

**from qiskit import IBMQ**

**from qiskit import \***

**from qiskit.tools.visualization import plot\_histogram**

**M = 15**

**a = 11**

**print("a\*\*2: ", a\*a % M)**

**Output:**

**a\*\*2: 1**

**def oneQbitCircuit(p, w, c):**

**circuit = QuantumCircuit(w,p, c)**

**# Prepare initial state 1 in primary register**

**circuit.x(p[0])**

**circuit.barrier(p)**

**circuit.barrier(w)**

**# Add Hadamard gate to working register**

**circuit.h(w[0])**

**# Add conditional multiplication by a to primary register**

**circuit.cx(w[0], p[1])**

**circuit.cx(w[0], p[3])**

**return circuit**

**p = QuantumRegister(4,"p")**

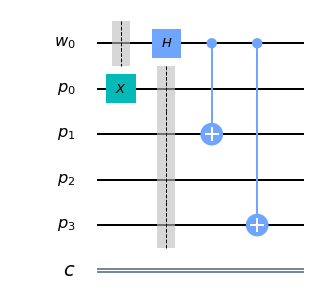
**w = QuantumRegister(1,"w")**

**c = ClassicalRegister(1, "c")**

**circuit = oneQbitCircuit(p,w,c)**

**drawer(circuit, output='mpl')**

**Output:**

****

**backend = Aer.get\_backend('statevector\_simulator')**

**job = execute(circuit, backend)**

**job.result().get\_statevector()**

**Output:**

**array([0. +0.j, 0. +0.j, 0.70710678+0.j, 0. +0.j,**

**0. +0.j, 0. +0.j, 0. +0.j, 0. +0.j,**

**0. +0.j, 0. +0.j, 0. +0.j, 0. +0.j,**

**0. +0.j, 0. +0.j, 0. +0.j, 0. +0.j,**

**0. +0.j, 0. +0.j, 0. +0.j, 0. +0.j,**

**0. +0.j, 0. +0.j, 0. +0.j, 0.70710678+0.j,**

**0. +0.j, 0. +0.j, 0. +0.j, 0. +0.j,**

**0. +0.j, 0. +0.j, 0. +0.j, 0. +0.j])**

**def twoQbitCircuit(p, w, c):**

**circuit = QuantumCircuit(w,p, c)**

**# Prepare initial state 1 in primary register**

**circuit.x(p[0])**

**circuit.barrier(p)**

**circuit.barrier(w)**

**# Add Hadamard gate to working register**

**circuit.h(w[0])**

**circuit.h(w[1])**

**# Add conditional multiplication by a to primary register**

**circuit.cx(w[0], p[1])**

**circuit.cx(w[0], p[3])**

**return circuit**

**p = QuantumRegister(4,"p")**

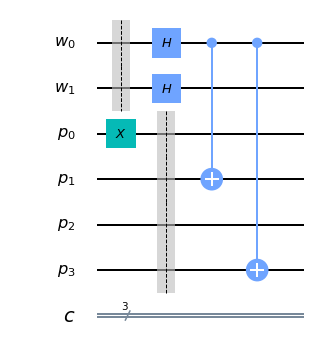
**w = QuantumRegister(2,"w")**

**c = ClassicalRegister(3, "c")**

**circuit = twoQbitCircuit(p,w,c)**

**drawer(circuit, output='mpl')**

**Output:**

****

**backend = Aer.get\_backend('statevector\_simulator')**

**job = execute(circuit, backend)**

**state = np.around(job.result().get\_statevector(), 2)**

**for i in range(2\*\*6):**

**if (state[i] != 0):**

**print("|",i,"> ---> ", state[i])**

**Output:**

**| 4 > ---> (0.5+0j)**

**| 6 > ---> (0.5+0j)**

**| 45 > ---> (0.5+0j)**

**| 47 > ---> (0.5+0j)**

**def threeQbitCircuit(p, w, c):**

**circuit = QuantumCircuit(w,p, c)**

**# Prepare initial state 1 in primary register**

**circuit.x(p[0])**

**circuit.barrier(p)**

**circuit.barrier(w)**

**# Add Hadamard gates to working register**

**circuit.h(w[0])**

**circuit.h(w[1])**

**circuit.h(w[2])**

**# Add conditional multiplication by a to primary register**

**circuit.cx(w[0], p[1])**

**circuit.cx(w[0], p[3])**

**return circuit**

**p = QuantumRegister(4,"p")**

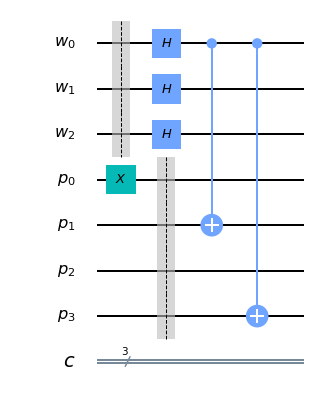
**w = QuantumRegister(3,"w")**

**c = ClassicalRegister(3, "c")**

**circuit = threeQbitCircuit(p,w,c)**

**drawer(circuit, output='mpl')**

**Output:**

****

**#**

**# Print out expected amplitudes up to normalisation**

**#**

**for s in range(2\*\*3):**

**x = a\*\*s % 15**

**print("|", x\*8 + s, "> = |",x,">|", s,">")**

**Output:**

**| 8 > = | 1 >| 0 >**

**| 89 > = | 11 >| 1 >**

**| 10 > = | 1 >| 2 >**

**| 91 > = | 11 >| 3 >**

**| 12 > = | 1 >| 4 >**

**| 93 > = | 11 >| 5 >**

**| 14 > = | 1 >| 6 >**

**| 95 > = | 11 >| 7 >**

**backend = Aer.get\_backend('statevector\_simulator')**

**job = execute(circuit, backend)**

**state = np.around(job.result().get\_statevector(), 2)**

**for i in range(2\*\*7):**

**if (state[i] != 0):**

**print("|",i,"> = |", i // 8, ">|", i % 8,"> ---> ", state[i])**

**Output:**

**| 8 > = | 1 >| 0 > ---> (0.35+0j)**

**| 10 > = | 1 >| 2 > ---> (0.35+0j)**

**| 12 > = | 1 >| 4 > ---> (0.35+0j)**

**| 14 > = | 1 >| 6 > ---> (0.35+0j)**

**| 89 > = | 11 >| 1 > ---> (0.35+0j)**

**| 91 > = | 11 >| 3 > ---> (0.35+0j)**

**| 93 > = | 11 >| 5 > ---> (0.35+0j)**

**| 95 > = | 11 >| 7 > ---> (0.35+0j)**

**def nBitQFT(q,c,n=3):**

**circuit = QuantumCircuit(q,c)**

**#**

**# We start with the most significant bit**

**#**

**for k in range(n):**

**j = n - k**

**# Add the Hadamard to qubit j-1**

**circuit.h(q[j-1])**

**#**

**# there is one conditional rotation for**

**# each qubit with lower significance**

**for i in reversed(range(j-1)):**

**circuit.cu1(2\*np.pi/2\*\*(j-i),q[i], q[j-1])**

**#**

**# Finally we need to swap qubits**

**#**

**for i in range(n//2):**

**circuit.swap(q[i], q[n-i-1])**

**return circuit**

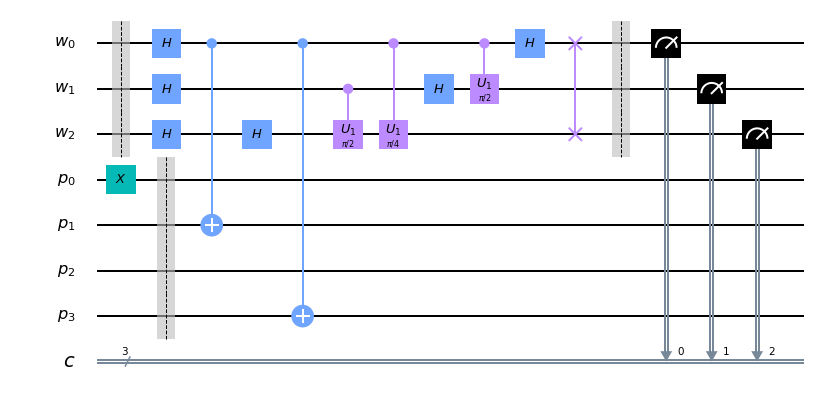
**circuit = threeQbitCircuit(p,w,c) + nBitQFT(w,c,n=3)**

**circuit.barrier(w)**

**circuit.measure(w,c)**

**drawer(circuit, output='mpl')**

**Output:**

****

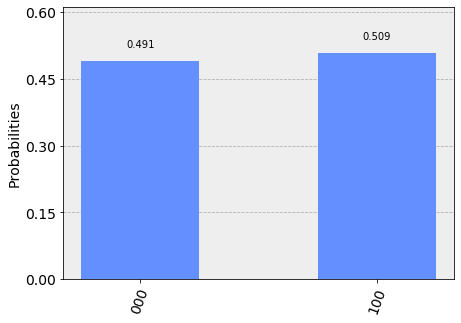
**backend = Aer.get\_backend('qasm\_simulator')**

**job = execute(circuit, backend)**

**counts = job.result().get\_counts()**

**plot\_histogram(counts)**

**Output:**

****

**def shorAlgorithm(n=3):**

**# Create registers and circuit**

**p = QuantumRegister(4,"p")**

**w = QuantumRegister(n,"w")**

**c = ClassicalRegister(n, "c")**

**circuit = QuantumCircuit(w,p,c)**

**# Add Hadamard gates to working register**

**circuit.h(w[0])**

**circuit.h(w[1])**

**# Add conditional multiplication by a to primary register**

**circuit.cx(w[0], p[1])**

**circuit.cx(w[0], p[3])**

**#**

**# Now build the QFT part. We start with the most significant bit**

**#**

**for k in range(n):**

**j = n - k**

**# Add the Hadamard to qubit j-1**

**if (j - 1) != 2:**

**circuit.h(w[j-1])**

**#**

**# there is one conditional rotation for**

**# each qubit with lower significance**

**for i in reversed(range(j-1)):**

**circuit.cu1(2\*np.pi/2\*\*(j-i),w[i], w[j-1])**

**#**

**# and add the measurements**

**#**

**circuit.barrier(w)**

**circuit.measure(w[0],c[2])**

**circuit.measure(w[2], c[0])**

**circuit.measure(w[1], c[1])**

**return circuit**

**circuit = shorAlgorithm()**

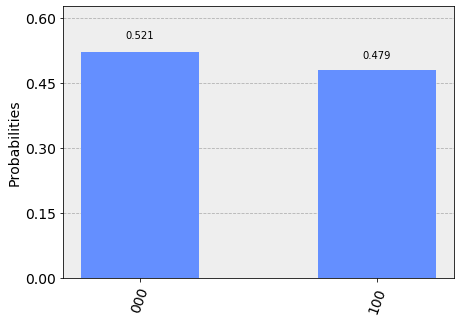
**backend = Aer.get\_backend('qasm\_simulator')**

**job = execute(circuit, backend)**

**counts = job.result().get\_counts()**

**plot\_histogram(counts)**

**Output:**

****

**IBMQ.load\_accounts()**

**backend = IBMQ.get\_backend('ibmq\_16\_melbourne')**

**job = execute(circuit, backend)**

**counts = job.result().get\_counts()**

**plot\_histogram(counts)**

**Output:**

**/usr/local/lib/python3.6/dist-packages/qiskit/providers/ibmq/utils/deprecation.py:53: DeprecationWarning: IBMQ.load\_accounts() is being deprecated. Please use IBM Q Experience v2 credentials and IBMQ.load\_account() (note the singular form) instead. You can find the instructions to make the updates here:**

[**https://github.com/Qiskit/qiskit-ibmq-provider#updating-to-the-new-ibm-q-experience**](https://github.com/Qiskit/qiskit-ibmq-provider#updating-to-the-new-ibm-q-experience)

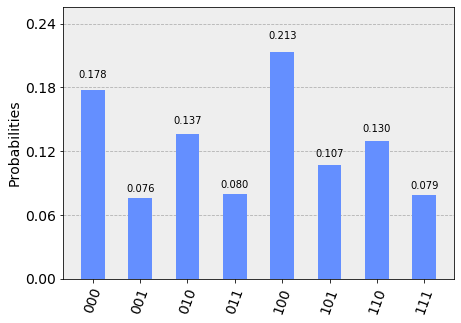
**DeprecationWarning)**

**/usr/local/lib/python3.6/dist-packages/qiskit/providers/ibmq/ibmqfactory.py:483: DeprecationWarning: Calling IBMQ.load\_accounts() with v2 credentials. This is provided for backwards compatibility and may lead to unexpected behaviour when mixing v1 and v2 account credentials.**

**'v1 and v2 account credentials.', DeprecationWarning)**

**/usr/local/lib/python3.6/dist-packages/qiskit/providers/ibmq/ibmqfactory.py:595: DeprecationWarning: IBMQ.get\_backend() is being deprecated. Please use IBMQ.get\_provider() to retrieve a provider and AccountProvider.get\_backend("name") to retrieve a backend.**

**DeprecationWarning)**

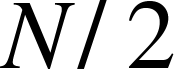
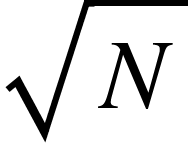
****

Exercise 5

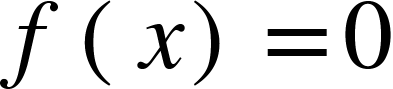
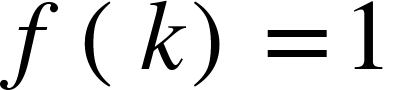
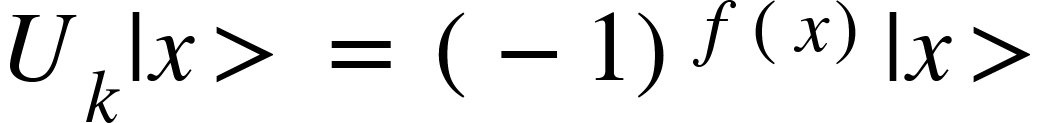
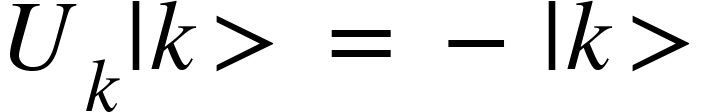
Take the random number ‘a’ as 7 and apply the algorithm.

# Welcome to the Heaven - It’s Week 7

This week we are going to learn about Grover’s algorithm which is used to search unstructured data. This gives quadratic speedup over classical algorithms. It uses a trick called amplitude amplification, to elevate the output from the unstructured mess.

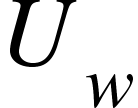
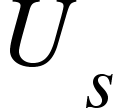
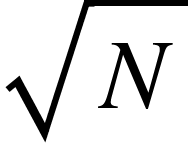
Using a linear classical search the worst-case scenario is  and the average case is  whereas in the case of a quantum computer it’s  which give quadratic speed up.

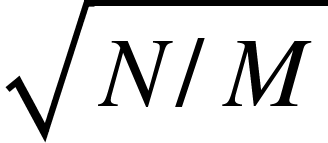
Oracle creation

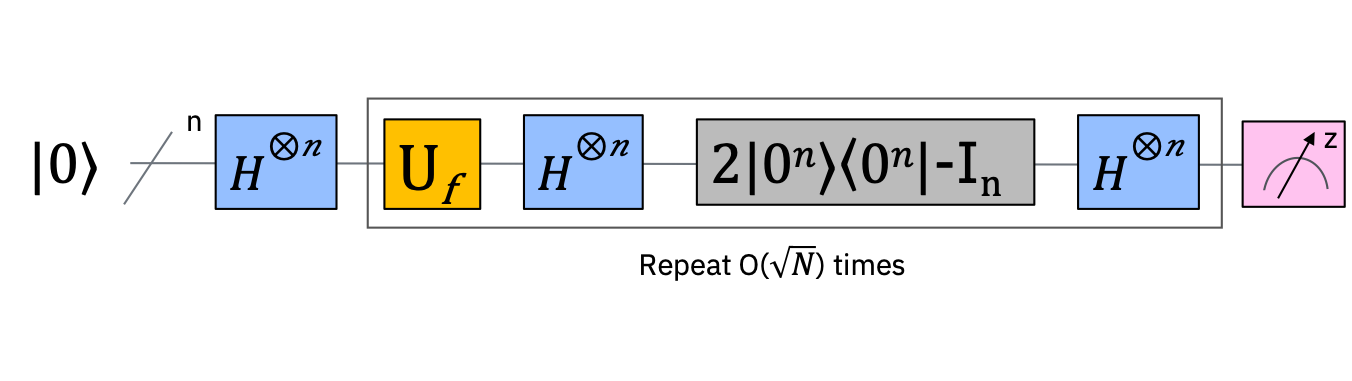
It has an important part in Grover’s search algorithm. Initially, we feed this oracle function with a superposition of states. The main aim of this oracle is to pick the king from the multifarious types of people. At last, the function returns  for all unmarked items and  for the king. The oracle is simply a unitary matrix. Primarily all the basis states are in the form . As we know if the output of f(x) is 0, which means if it is an unmarked item, which in turn means it is an ignore case. But when f(x)=1, i.e when x=k then . This state will correspond to a reflection along with the origin.

If it was any classical algorithms we should have measured  times, but thanks to superposition, it made our job easy. Now there is a phenomenon called amplitude amplification which stretches the amplitude of the king, therefore, reducing the amplitude of other items. Now the final measurement will result in the right answer, i.e the king.

Steps for Amplitude amplification:

1. For this to happen first there should be a uniform superposition of states.  
   
2. Apply the oracle reflection to the king. This turns the amplitude of the king to negative.
3. After the unitary transformation  in the above step, without any relaxation, apply another unitary transform  which will do not negate it more but instead it will amplify it three times and decreases the amplitudes of other items. This happens because the secondary unitary transformation that we applied will flip the entire average amplitude which was decreased by negating the amplitude of the king due to the unitary transformation in step 2. Now repeat steps 2 and 3 nearly  times.

\*\*\* In case of multiple search solutions Repeat the above steps 2 & 3  times.



**Implementation of Grover’s Algorithm using Qiskit**

#importing stuff

import matplotlib.pyplot as plt

%matplotlib inline

import numpy as np

from qiskit import IBMQ, BasicAer

from qiskit.providers.ibmq import least\_busy

from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, execute

from qiskit.tools.visualization import plot\_histogram

#This is the code to create the oracle

def phase\_oracle(circuit, register):

circuit.cz(register[0], register[1])

circuit.barrier()

#This barrier separates step 1 and 2 from the step 3

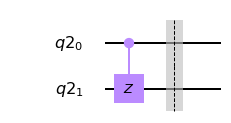
qr = QuantumRegister(2)

oracleCircuit = QuantumCircuit(qr)

phase\_oracle(oracleCircuit, qr)

oracleCircuit.draw(output="mpl")

**Output:**

****

**#This is the average inversion function which i mentioned in step 3**

**def inversion\_about\_average(circuit, register):**

**"""Apply inversion about the average step of Grover's algorithm."""**

**circuit.h(register)**

**circuit.x(register)**

**circuit.h(register[1])**

**circuit.cx(register[0], register[1])**

**circuit.h(register[1])**

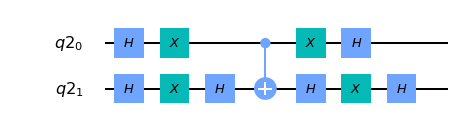
**circuit.x(register)**

**circuit.h(register)**

**qAverage = QuantumCircuit(qr)**

**inversion\_about\_average(qAverage, qr)**

**qAverage.draw(output='mpl')**

****

**qr = QuantumRegister(2)**

**cr = ClassicalRegister(2)**

**groverCircuit = QuantumCircuit(qr,cr)**

**groverCircuit.h(qr)**

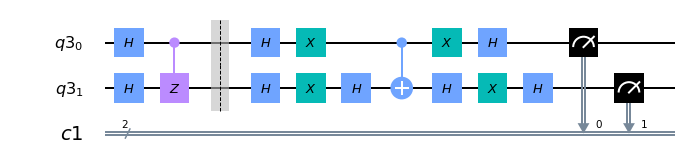
**phase\_oracle(groverCircuit, qr)**

**inversion\_about\_average(groverCircuit, qr)**

**groverCircuit.measure(qr,cr)**

**groverCircuit.draw(output="mpl")**

**Output:**

****

**backend = BasicAer.get\_backend('qasm\_simulator')**

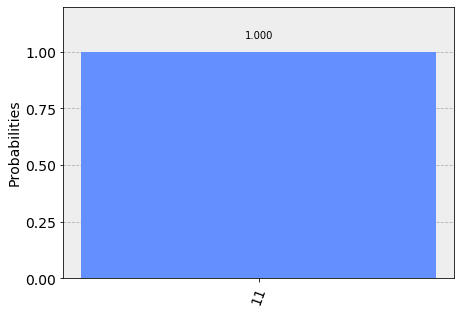
**shots = 1024**

**results = execute(groverCircuit, backend=backend, shots=shots).result()**

**answer = results.get\_counts()**

**plot\_histogram(answer)**

**Output:**

****

**#now lets run it on a real quantum computer**

**IBMQ.load\_account()**

**provider = IBMQ.get\_provider(group='open')**

**backend\_lb = least\_busy(provider.backends(simulator=False, operational=True))**

**print("Least busy backend: ", backend\_lb)**

**from qiskit.tools.monitor import job\_monitor**

**backend = backend\_lb**

**shots = 1024**

**job\_exp = execute(groverCircuit, backend=backend, shots=shots)**

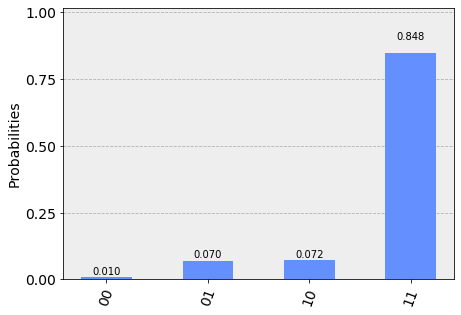
**job\_monitor(job\_exp, interval = 2)**

**results = job\_exp.result()**

**answer = results.get\_counts(groverCircuit)**

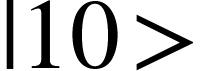
**plot\_histogram(answer)**

**Output:**

****

As you can see 11 got the highest probability. The reason for getting a slight deviation in the probabilities of the other three states is due to the errors in the quantum computer. Don’t worry quantum computers are still evolving. By the time you read this book the error rate may go down a bit, so try out the algorithm yourself.

Exercise 6

Try to find the state  using Grover’s algorithm above.

Hint: Slight modification in oracle will suffice; Two crosses on either side.

# Week 8, A Fight with Errors and Noise

Quantum Noise

Till now we have seen many principles and techniques to attain maximum efficiency in the world of computing. But to what extent do they provide efficient services? All the principles that are revised for quantum computers are generally for the ideal state, which means they execute the tasks without any interruption, but we know that practically, this idealistic nature doesn’t exist. Hence, we have to design quantum computers in such a way that even if any interruption occurs it should be solved in minimal time.

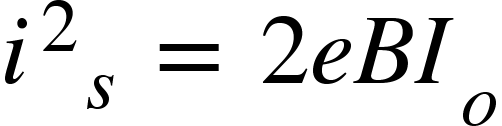
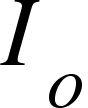
Uncertainty in the physical state of a system that occurs due to the change in the environment or due to the introduction of new particles to the system may create a disturbance in the functionality. This disturbance which disrupts the task execution can be termed as noise and as it is associated with quantum computing it is known as quantum noise.

Let us consider an example, where we are trying to achieve photonic entanglement that will further provide quantum internet within the necessary distance. In doing so, if any condition such as a change in polarisation due to temperature rise/drop will lead to decoherence and the complete setup will go in vain.

In other situations, the quantum noise can occur in the form of Poisson noise or otherwise known as shot noise, which occurs due to the discrete nature of the electric charge. This can be seen during amplifying the amplitude in any algorithm because of the use of optical communication devices. In such cases, there might be uncertainty in phase as well as amplitude modulation. And thus will disrupt the whole machine thereby decreasing the efficiency of the system.

In classical computing, we call term this as error and when it comes to quantum computing, we call it “quantum noise”. The noise can be observed in any system and can occur in any of the forms like vibrations, industrial noise, thermal noise(due to Brownian motion) and fluctuations in power supply, etc.

Shot noise is arisen due to the static nature of photoelectrons, it can be mathematically given as:

, where  is the photo-electric current.

The need for studying quantum noise is to know the quantum limits of digital transmission. Quantum limit can be described as the minimum energy required to maintain error bits in quantum computing, i.e minimum level of noise during the execution of the process.

**Effects of quantum noise:**

And after knowing what noise is and the forms in which it can occur, we now have to analyze the threats it possesses to a quantum system and find a way to prevent it from occurring so that one might get a quantum computer that is equally ideal.

Though this appears to be a minute issue, it disturbs the whole equipment without leaving any traces. Some of the effects might include:

* Fidelity in the state.
* Reduced feedback strength, i.e efficiency of the output is decreased.
* Complete shut-down of the system.

In order to avoid these threatening disruptions to the system, we have to maintain accuracy in the system and avoid the system in leading to a state where quantum noise occurs. In classical terms, we can say that we should correct the errors.

Quantum Error Correction

In classical computers, it is evident that error correction can be done in two ways:

* Guessing the original data
* Retransmission of data, which includes copying the original data.

When it comes to the quantum point of view, we know that copying the original data is against the laws, or can be stated as practically impossible due to no-cloning theorem which states that it is impossible to create an identical copy of the original quantum state. This no-cloning theorem seems to be an obstacle in error-correction techniques.

Problems that occur while correcting the errors in quantum computing:

* No-cloning theorem (as mentioned above).
* Continuous error on a single qubit.
* Measurement leads to the destruction of the state, i.e the original state will be collapsed

And now how do we solve the problem? It is practically not possible to create an identical state but it is possible to pass the information of the qubit to the highly entangled qubits, i.e state of the several entangled qubits. This was first discovered by Peter Shor, where he demonstrated by spreading the information of one qubit to the highly entangled state of nine qubits.

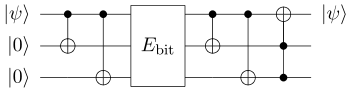
In order to know which qubit got affected and the possible ways of it affecting the system, we use syndrome measurement. Syndrome measurement is the measurement performed on a computing system to get information about the physical state of the bit and the ways in which it got affected due to the noise that occurred in the system.

After performing the syndrome measurement, it can be concluded that the arbitrary noise was arisen due to one or the superposition of the following reasons:

* Bit flip
* Phase or sign flip
* Both bit and phase flip

Let us carefully take a look at each of the error correction techniques mentioned above

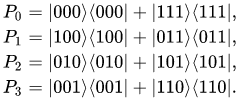
1. **The bit flip code:**



This technique uses entanglement as the base phenomenon and for better understanding let us consider a symmetric quantum system with a bit-flip error X that has the probability of occurrence p.

On encoding the qubit , using repetition it will lead to  or  or .

Now we perform syndrome measurement on the system. Based on the projections, there can be 4 error syndromes. They are:



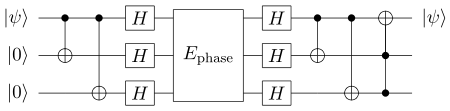
Where P0 corresponds to no error, P1 to an error on the first qubit, P2 to error on the second qubit, P3 to error on the third qubit.

Let us assume that the error has occurred on the first qubit, so it means= 0 |100> + 1 |011 > and the bit flip should be performed on the bit. Note: The state will not be destroyed as we are just trying to know the information on where the error has occurred and not the state of the qubit.

Similarly, 

Now, we have encoded the qubits and performed syndrome measurement and so we have all the information needed to know about the error qubit. Now we just have to apply the bit flip on the qubit that is associated with the noise.

1. **The phase flip code:**



In this error correction technique, the error is associated with the phase of the qubit and that should be corrected.

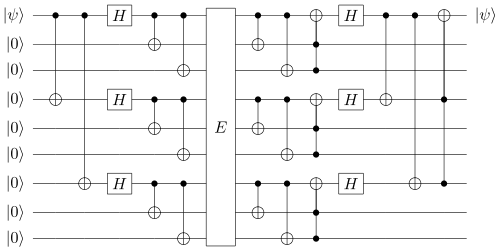
For instance, we can say that  will be flipped to  and a qubit in the state  might be flipped to 

Let us consider that there is an error associated with the qubit , using the phase flip technique we obtain the new state of the qubit with no error as 

Usually, in the Hadamard basis, phase flips become bit flips and vice versa. So let  be the phase flip that has at most one phase flip, now, the bit flip code is applied to  which transforms before and after the  transmission.

This has remarkable features just as a bit flip technique which involves minimum fidelity.

1. **The Shor code:**



This code serves as the protection for a single qubit on which any arbitrary error occurs. It is the combination of bit flip code and the phase flip code.





Firstly, the phase flip code is applied on the qubit for encoding and then bit-flip code is applied.

If  is a quantum channel with a single corrupted qubit, then 1st, 4th and 7th qubits are designed for the phase flip code and (1,2,3), (4,5,6), and (7,8,9) are for the bit-flip code.

On applying Shor code to the , it will transform into  with 9 qubits.

Syndrome measurement will be performed on each of (1,2,3), (4,5,6), and (7,8,9) if there is a bit flip error it is corrected.

Consider (1,2,3), (4,5,6), and (7,8,9) as the inputs, then the Shor code is used to reduce the circuit as a phase flip code. This proves that it can be used both as a bit-flip code and a phase flip code.

For better clarification, let us consider that a unitary transformation *U*  is applied on a qubit ,

U can be given as 

Where, , ,  and  are complex constants and I am the identity matrix. Then the Pauli matrices are given as:







And after measurement, if the results show:

* If *U* = I, then there is no error.
* If *U* = , then there is a bit-flip error.
* If *U* = , then there is a phase flip error.
* And if *U* =  then there is both bit-flip and sign flip error.

And they can be further corrected using the Shor code.

1. **The Classical code:**

In a classical code, let us consider that k bits of information has to be sent.

And the error detection techniques used are:

* Cyclic redundancy check
* Hamming distance, etc.

The errors are corrected either by re-transmitting the whole data or by guessing the part of the information which consists of error.

**Error correction using ancilla qubits:**

Why do we use ancilla qubits for error correction? Utmost importance is given in the quantum error correction to ensure that the original state isn’t collapsed.

For example, If we measure the qubits we might get the qubits to

|0⟩ or |1⟩ and lose the original information about the coefficients α|0⟩+β|1⟩.

On measuring with ancilla qubits we will know what has happened to the qubits without actually measuring the qubits, this will enable us to correct errors in a non-destructive way, and further, we can carry on with our quantum operation.

Even after correcting the errors researchers were given a task to create a practical real-time scenario to quantum computers that will surpass the existing computers and work efficiently. And therefore they brought up the idea of NISQ (Noisy Intermediate Scale Quantum Computing).

NISQ Computers

NISQ is short for “Noisy Intermediate Scale Quantum Computing”. This name is given keeping in mind that an intermediate scale, i.e a quantum computer can be made using qubits ranging from 50-100, but the problem is if we use so many qubits there is a high probability that it will be exposed to noise.

When we think of quantum computers, we generally mean fault-tolerant devices. The computers will be able to run Shor's algorithm for factoring large prime numbers and on the other hand the algorithms that have been developed so far. But the power comes at a cost: to solve a factoring problem that is not feasible when it is computed on a classical computer, why? Because we need millions of qubits. This process requires error correction since most algorithms are extremely sensitive to noise.

Even then, the programs that run on devices beyond 50 qubits in size are extremely difficult to implement and simulate on classical computers. The possibility that devices of this kind may be used to perform the demonstration of a quantum computer doing something that is infeasible for a classical one. Just like Google’s sycamore processor that claims to have achieved Quantum Supremacy by running an algorithm on its 53 qubits.

We now know that devices can do things that classical computers can't, but they aren’t big enough to provide fault-tolerant implementations of the algorithms we know so far. The term NISQ can be used to describe this era.

It is believed that noise in the quantum gates will limit the power of creating efficient quantum circuits. These computers can be of much use in the field of business, medicine and solving space-related problems.

At the moment, we can’t say that NISQ will change the world, but can definitely be a huge milestone in creating more powerful quantum technologies that will solve classical problems faster than ever.

For example, an algorithm can be written that has chemistry applications where the ground state of a molecule is approximated.

To start off, first, the user should give an approximation of the ground state as input to the qubits. Then it is the work of quantum computing to improve the initial guess by applying logic gates sequentially that depend on a set of parameters, analogous to weights in a neural network.

And finally, the state which is retrieved is fed to the classical computer, which instructs the user how to pull those weights in the quantum computer. The entire process iterates are then automated.

When it comes to software implementation, researchers say that this can be a potential step when integrated with Machine Learning. Scientists have performed experiments on a machine for classifying the data, one might get a doubt that we currently have classical computers doing the task, but with the quantum computer the results are obtained faster.

On the other hand, hardware specialists are still trying to figure out the type of qubit (ion, quantum dot, neutral atom or superconducting loop) works the best.

Exercise 7

Write Grover’s algorithm that you have learned last week using ancilla qubits,

# Qrypto Week 9

Cryptography

When we hear the word cryptography we think of two types in it.

1. Private cryptosystem
2. Public cryptosystem

As we know cryptography is a process of hiding the data so that it can only be accessed by the one who is intended to access it. In the private key cryptography, there will be only one key to encrypt as well as decrypt. This creates privacy issues where one has a threat of key being stolen or data being leaked. This is sometimes referred to as Symmetric key encryption.

Whereas public key cryptosystem is an asymmetric type of encryption that has two keys, one being public and another being private key. Best known classical encryption algorithms use public cryptosystems, for example, the RSA algorithm that we have learned and that which is used by many banks and major industries. Secure Socket Layer algorithm which is uses public cryptosystem to encrypt your web session, especially while you are making a purchase. They are also called as asymmetric encryptions. These cryptosystems work on a principle that “some math problems are easy to do but hard to undo”. For example, it is easy to multiply two large prime numbers but it is hard to factorize the result back. Already the key sizes of RSA have been increased due to the technical leap fever which is expected at any time. NIST has confirmed that 1024 bit RSA is no longer safer and recommended 2048 bit RSA encryption what is slower than the previous one and also costlier.

Quantum Cryptography

It is similar to the cryptographic process described above but uses physical laws to encrypt the data. The application of quantum mechanical principles makes quantum cryptography so strong and quite unbreakable. Theories such as no-cloning theorem will always guard its privacy. It is hard to measure any state without causing disturbances in the channel because of quantum properties. Fortunately, photons have such properties that are comparatively superior and easily understood.

Quantum Key Distribution

It is a part of quantum cryptography. QKD is a method used to send encrypted keys using the peculiar and natural features of subatomic particles, which makes it unhackable and unbreakable. Basically, photons are sent one after the other through optical fibers. If there is an eavesdropper in the channel, the polarization shifts which alerts the recipient Other than optical fibers there is another spooky way to communicate, i.e Entanglement. Using the entanglement phenomenon, the information can be transferred safely and securely.

BB84 Algorithm ( Bennett and Brassard in 1984 )

Bennet and Brassard developed a model of quantum cryptography in 1984. Assume that there are two people namely Alice and Bob, who are willing to share information. Assume Alice initiates the message by sending Bob a key. Now imagine the key to be a stream of photons. If we introduce a polarizer in front of the traveling stream it alters the oscillation of the photons. The very basic nature of polarizer is, it permits certain photons to pass through it with the same oscillation as before and changes the state of oscillation for the other photons.it may not allow some photons to pass through it (exceptional case). The oscillations can be divided into four types up/down, left/right, up-left/right-down and up-right/left-down. All these four states have specific angles which are known as polarization of the photon, they are 0°, 90°, 45°, and 135° respectively.

There are two polarization filters

1. Rectilinear (UP/DOWN, LEFT/RIGHT)
2. Diagonal (UP-LEFT/RIGHT-DOWN, UP-RIGHT/LEFT-DOWN)

Alice maps a random selection of either of the polarization filters for the transfer of each bit of information. She swaps between rectilinear and diagonal filters for different bits. Now on the Bob side while he receives every bit she should apply one of the two filters available. There is a 50-50 chance for Bob to be using the right filter or the wrong one. Due to the No-Cloning theorem he is unable to recognize the base Alice used to send the photons. If Bob uses a rectilinear filter to identify a diagonally polarized photon then the photon passes by, changing its oscillation. The four states are given as following.









You may also encounter this kind of representation in some material.

**Rectilinear **

**diagonal** .

Bob measures some photons correctly and some photons incorrectly. So to correct this Alice and Bob will establish a secure channel of communication that can be eavesdropped. Now through the channel, Alice will tell Bob the polarizer she used to send a particular bit.

\*\*\* She does not say how she sent the photons which means she will not say whether she sent a UP-DOWN or LEFT-RIGHT but tells which filter she used, i.e, either rectilinear or diagonal. Based on this information Bob will arrange his version of polarizers.

If a guy named Eve will eavesdrop in between the communication, possessing the similar kind of polarizers that Bob have he may be guessing half the photons right just as Bob did before establishing direct communication with Alice. So Bob is on the advantage side, as he can confirm with Alice, the polarizer used for each photon/qubit.

The eavesdropping can even be known by Bob. For example, assume Alice sends a photon in UP-LEFT/RIGHT-DOWN filter but Eve will use a rectilinear type of polarizer then he will convert the photon to UP-DOWN or LEFT-RIGHT, which reaches Bob. Here there are two cases.

1. If Bob uses a rectilinear filter the photon will pass through it without any changes but when the bob verifies with Allice about the polarizer used for that bit, they will realize that there has been some eavesdropping. Then Alice will discard that bit from the key.
2. Bob will use a diagonal filter that changes the oscillation of the rectilinearly aligned photons to UP-LEFT/RIGHT-DOWN state which is what Alice has sent to Bob. But as I said Alice will communicate about the polarizer used but not how she sent the photons.

**Implementation of BB84 using Qiskit**

Import the required packages first.

import numpy as np

from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, execute, BasicAer

from qiskit.tools.visualization import plot\_histogram

Now declare the quantum and classical register. Also set available qubit number to 16.

n = 16 # for a local backend n can go as up as 23, after that it raises a Memory Error

qr = QuantumRegister(n, name='qr')

cr = ClassicalRegister(n, name='cr')

Now let’s create Alice’s circuit with n qubits and n classical bits. Let’s then take a random number using numpy which will act as our key, which we will then convert into binary format and store in a variable.

# Quantum circuit for alice state

alice = QuantumCircuit(qr, cr, name='Alice')

# Generate a random number in the range of available qubits [0,65536))

alice\_key = np.random.randint(0, high=2\*\*n)

# Cast key to binary for encoding

# range: key[0]-key[15] with key[15] least significant figure

alice\_key = np.binary\_repr(alice\_key, n) # n is the width

In the next step, we are going to apply rotation to half of these qubits so that 50% of them will be in one of the eigenstates off-diagonal basis. we will also try to record the basis choice in a list that will be used for key verification.

# Encode key as alice qubits

# IBM's qubits are all set to |0> initially

for index, digit in enumerate(alice\_key):

if digit == '1':

alice.x(qr[index]) # if key has a '1', change state to |1>

# Switch randomly about half qubits to diagonal basis

alice\_table = [] # Create empty basis table

for index in range(len(qr)): # BUG: enumerate(q) raises an out of range error

if 0.5 < np.random.random(): # With 50% chance...

alice.h(qr[index]) # ...change to diagonal basis

alice\_table.append('X') # character for diagonal basis

else:

alice\_table.append('Z') # character for computational basis

Now let's look into Bob's circuit. we don't have another Quantum computer, but we can create another Quantum circuit that will act as Bob's circuit. Helper function named SendState that is declared in the code below retrieves the qasm code of a given Quantum circuit (that is Alice's circuit) and extract the gates applied which eventually produces new instructions used to initialise another circuit. Qiskit has a feature of maintaining python dictionaries of quantum circuits with their relative qasm instructions..

\*\*\*Qasm stands for Quantum assembler.

def SendState(qc1, qc2, qc1\_name):

''' This function takes the output of a circuit qc1 (made up only of x and

h gates and initializes another circuit qc2 with the same state

'''

# Quantum state is retrieved from qasm code of qc1

qs = qc1.qasm().split(sep=';')[4:-1]

# Process the code to get the instructions

for index, instruction in enumerate(qs):

qs[index] = instruction.lstrip()

# Parse the instructions and apply to new circuit

for instruction in qs:

if instruction[0] == 'x':

old\_qr = int(instruction[5:-1])

qc2.x(qr[old\_qr])

elif instruction[0] == 'h':

old\_qr = int(instruction[5:-1])

qc2.h(qr[old\_qr])

elif instruction[0] == 'm': # exclude measuring:

pass

else:

raise Exception('Unable to parse instruction')

Now let's create Bob's and send Alice's qubits to Bob pretending that the state is unknown to Bob.

bob = QuantumCircuit(qr, cr, name='Bob')

SendState(alice, bob, 'Alice')

# Bob doesn't know which basis to use

bob\_table = []

for index in range(len(qr)):

if 0.5 < np.random.random(): # With 50% chance...

bob.h(qr[index]) # ...change to diagonal basis

bob\_table.append('X')

else:

bob\_table.append('Z')

Now Bob can measure all the qubits and store them in the classical register. While executing the circuit remember to assign shots = 1 because Bob has only one measurement chance.

# Measure all qubits

for index in range(len(qr)):

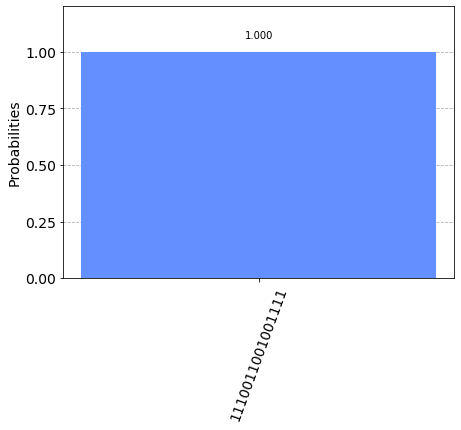
bob.measure(qr[index], cr[index])

backend = BasicAer.get\_backend('qasm\_simulator')

result = execute(bob, backend=backend, shots=1).result()

plot\_histogram(result.get\_counts(bob))

**Output:**



# Result of the measure is Bob's key candidate

bob\_key = list(result.get\_counts(bob))[0]

bob\_key = bob\_key[::-1] # key is reversed so that first qubit is the first element of the list

Now, let’s compare their basis table lists and discard qubits measured using different basis over the classical channel.

keep = []

discard = []

for qubit, basis in enumerate(zip(alice\_table, bob\_table)):

if basis[0] == basis[1]:

print("Same choice for qubit: {}, basis: {}" .format(qubit, basis[0]))

keep.append(qubit)

else:

print("Different choice for qubit: {}, Alice has {}, Bob has {}" .format(qubit, basis[0], basis[1]))

discard.append(qubit)

**Output:**

**Different choice for qubit: 0, Alice has Z, Bob has X**

**Different choice for qubit: 1, Alice has X, Bob has Z**

**Same choice for qubit: 2, basis: Z**

**Same choice for qubit: 3, basis: X**

**Same choice for qubit: 4, basis: X**

**Same choice for qubit: 5, basis: X**

**Different choice for qubit: 6, Alice has Z, Bob has X**

**Different choice for qubit: 7, Alice has X, Bob has Z**

**Same choice for qubit: 8, basis: X**

**Different choice for qubit: 9, Alice has Z, Bob has X**

**Same choice for qubit: 10, basis: Z**

**Different choice for qubit: 11, Alice has Z, Bob has X**

**Different choice for qubit: 12, Alice has X, Bob has Z**

**Different choice for qubit: 13, Alice has Z, Bob has X**

**Same choice for qubit: 14, basis: Z**

**Different choice for qubit: 15, Alice has X, Bob has Z**

We now know that Bob picks 50% of the choices wrong. So let's measure the percentage of qubits to be discarded.

acc = 0

for bit in zip(alice\_key, bob\_key):

if bit[0] == bit[1]:

acc += 1

print('Percentage of qubits to be discarded according to table comparison: ', len(keep)/n)

print('Measurement convergence by additional chance: ', acc/n)

**Output:**

**Percentage of qubits to be discarded according to table comparison: 0.4375**

**Measurement convergence by additional chance: 0.75**

We compare the value of qubits and then shift keys. Since we have only 16 qubits we will limit the procedure to exchange 16 qubits at a time and which is repeated as many times as needed

new\_alice\_key = [alice\_key[qubit] for qubit in keep]

new\_bob\_key = [bob\_key[qubit] for qubit in keep]

acc = 0

for bit in zip(new\_alice\_key, new\_bob\_key):

if bit[0] == bit[1]:

acc += 1

print('Percentage of similarity between the keys: ', acc/len(new\_alice\_key))

**Output:**

Percentage of similarity between the keys: 1.0

if (acc//len(new\_alice\_key) == 1):

print("Key exchange has been successfull")

print("New Alice's key: ", new\_alice\_key)

print("New Bob's key: ", new\_bob\_key)

else:

print("Key exchange has been tampered! Check for eavesdropper or try again")

print("New Alice's key is invalid: ", new\_alice\_key)

print("New Bob's key is invalid: ", new\_bob\_key)

**Output:**

Key exchange has been successfull

New Alice's key: ['1', '1', '0', '0', '0', '1', '1']

New Bob's key: ['1', '1', '0', '0', '0', '1', '1']

It's time to introduce an eavesdropper and see what happens. First, let’s create a circuit called Eve and let's initialize it to Alice’s state.

eve = QuantumCircuit(qr, cr, name='Eve')

SendState(alice, eve, 'Alice')

eve\_table = []

for index in range(len(qr)):

if 0.5 < np.random.random():

eve.h(qr[index])

eve\_table.append('X')

else:

eve\_table.append('Z')

for index in range(len(qr)):

eve.measure(qr[index], cr[index])

# Execute (build and run) the quantum circuit

backend = BasicAer.get\_backend('qasm\_simulator')

result = execute(eve, backend=backend, shots=1).result()

# Result of the measure is Eve's key

eve\_key = list(result.get\_counts(eve))[0]

eve\_key = eve\_key[::-1]

# Update states to new eigenstates (of wrongly chosen basis)

for qubit, basis in enumerate(zip(alice\_table, eve\_table)):

if basis[0] == basis[1]:

print("Same choice for qubit: {}, basis: {}" .format(qubit, basis[0]))

else:

print("Different choice for qubit: {}, Alice has {}, Eve has {}" .format(qubit, basis[0], basis[1]))

if eve\_key[qubit] == alice\_key[qubit]:

eve.h(qr[qubit])

else:

if basis[0] == 'X' and basis[1] == 'Z':

eve.h(qr[qubit])

eve.x(qr[qubit])

else:

eve.x(qr[qubit])

eve.h(qr[qubit])

**Output:**

Same choice for qubit: 0, basis: Z

Different choice for qubit: 1, Alice has X, Eve has Z

Same choice for qubit: 2, basis: Z

Same choice for qubit: 3, basis: X

Different choice for qubit: 4, Alice has X, Eve has Z

Different choice for qubit: 5, Alice has X, Eve has Z

Different choice for qubit: 6, Alice has Z, Eve has X

Same choice for qubit: 7, basis: X

Different choice for qubit: 8, Alice has X, Eve has Z

Same choice for qubit: 9, basis: Z

Same choice for qubit: 10, basis: Z

Same choice for qubit: 11, basis: Z

Different choice for qubit: 12, Alice has X, Eve has Z

Same choice for qubit: 13, basis: Z

Different choice for qubit: 14, Alice has Z, Eve has X

Same choice for qubit: 15, basis: X

Eve’s altered state is now sent to Bob.

SendState(eve, bob, 'Eve')

bob\_table = []

for index in range(len(qr)):

if 0.5 < np.random.random():

bob.h(qr[index])

bob\_table.append('X')

else:

bob\_table.append('Z')

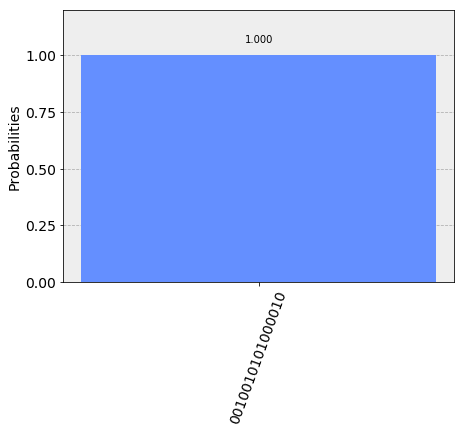
for index in range(len(qr)):

bob.measure(qr[index], cr[index])

result = execute(bob, backend=backend, shots=1).result()

plot\_histogram(result.get\_counts(bob))

**Output:**



bob\_key = list(result.get\_counts(bob))[0]

bob\_key = bob\_key[::-1]

Now Alice and Bob will share their table data and perform checking operations.

keep = []

discard = []

for qubit, basis in enumerate(zip(alice\_table, bob\_table)):

if basis[0] == basis[1]:

print("Same choice for qubit: {}, basis: {}" .format(qubit, basis[0]))

keep.append(qubit)

else:

print("Different choice for qubit: {}, Alice has {}, Bob has {}" .format(qubit, basis[0], basis[1]))

discard.append(qubit)

acc = 0

for bit in zip(alice\_key, bob\_key):

if bit[0] == bit[1]:

acc += 1

print('\nPercentage of qubits to be discarded according to table comparison: ', len(keep)/n)

print('Measurement convergence by additional chance: ', acc/n)

new\_alice\_key = [alice\_key[qubit] for qubit in keep]

new\_bob\_key = [bob\_key[qubit] for qubit in keep]

acc = 0

for bit in zip(new\_alice\_key, new\_bob\_key):

if bit[0] == bit[1]:

acc += 1

print('\nPercentage of similarity between the keys: ', acc/len(new\_alice\_key))

if (acc//len(new\_alice\_key) == 1):

print("\nKey exchange has been successfull")

print("New Alice's key: ", new\_alice\_key)

print("New Bob's key: ", new\_bob\_key)

else:

print("\nKey exchange has been tampered! Check for eavesdropper or try again")

print("New Alice's key is invalid: ", new\_alice\_key)

print("New Bob's key is invalid: ", new\_bob\_key)

**Output:**

Same choice for qubit: 0, basis: Z

Same choice for qubit: 1, basis: X

Different choice for qubit: 2, Alice has Z, Bob has X

Same choice for qubit: 3, basis: X

Same choice for qubit: 4, basis: X

Different choice for qubit: 5, Alice has X, Bob has Z

Different choice for qubit: 6, Alice has Z, Bob has X

Different choice for qubit: 7, Alice has X, Bob has Z

Same choice for qubit: 8, basis: X

Same choice for qubit: 9, basis: Z

Same choice for qubit: 10, basis: Z

Same choice for qubit: 11, basis: Z

Different choice for qubit: 12, Alice has X, Bob has Z

Different choice for qubit: 13, Alice has Z, Bob has X

Different choice for qubit: 14, Alice has Z, Bob has X

Different choice for qubit: 15, Alice has X, Bob has Z

Percentage of qubits to be discarded according to table comparison: 0.5

Measurement convergence by additional chance: 0.4375

Percentage of similarity between the keys: 0.375

Key exchange has been tampered! Check for eavesdropper or try again

New Alice's key is invalid: ['0', '0', '1', '0', '0', '1', '1', '1']

New Bob's key is invalid: ['1', '1', '1', '0', '0', '0', '0', '0']

Exercise 8

Learn and write the B92 algorithm which is almost similar to BB84 that you learned this week.

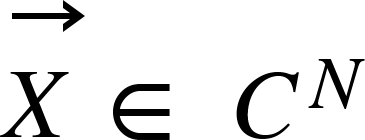
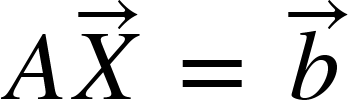
# Final Week - 10

Quantum Machine Learning

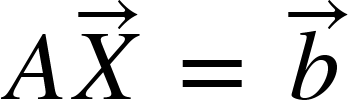
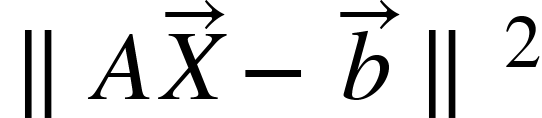
Quantum machine learning is an emerging research area that deals with both Quantum computing as well as machine learning, the two booming technologies in the market as of 2019. Quantum machine learning is nothing but classical machine learning algorithms executed using a Quantum computer. It's giving a machine power to learn itself and also enhancing its processing power with the support of quantum mechanical laws. The classical deep learning algorithms, neural networks, can be combined with the techniques in quantum physics. Quantum machine learning is mainly about finding difficult models in machine learning, for example, probabilistic models of over graph structures that resist calculations of classical computers.

Some of the quantum machine learning algorithms are implemented based on a concept called amplitude encoding. The main goal is to make Quantum algorithmic resources grow polynomial with the number of qubits. Basically, amplitude encoding is a process of associating the amplitudes of quantum States with the inputs and outputs. If there are n qubits then there will be  complex amplitudes. There is a lot of play that goes based on the Quantum algorithm for linear systems of equations that are designed by Aram Harrow, Avintan Hassidim, and Seth Lloyd.

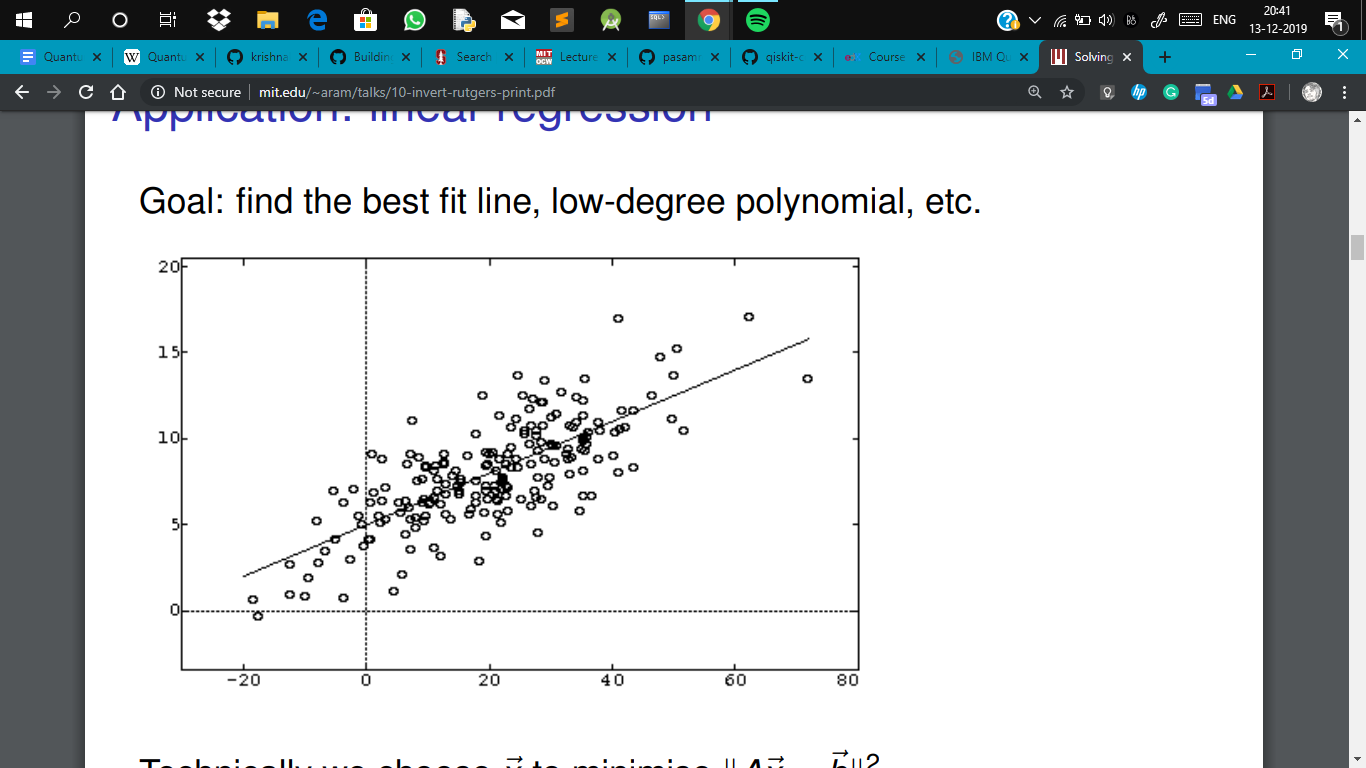
Quantum Matrix inversion can also be applied to machine learning algorithms which can reduce the training time to solve linear systems of equations for example,

We need to find  in .

**Classical solution**

Linear regression is an application, which is a process of fitting a line that has the least sum of squares. It models the relationship between a dependent variable and multiple independent variables. As there is only one independent variable in  it comes under simple linear regression. We calculate the least squares using .

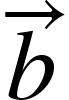
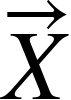
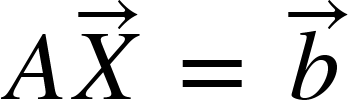
The line is rearranged till the lowest degree polynomial is found. The learning rate is used to change the parametric values in each turn of the line. Finally, the output values are acquired from the line that best fits.

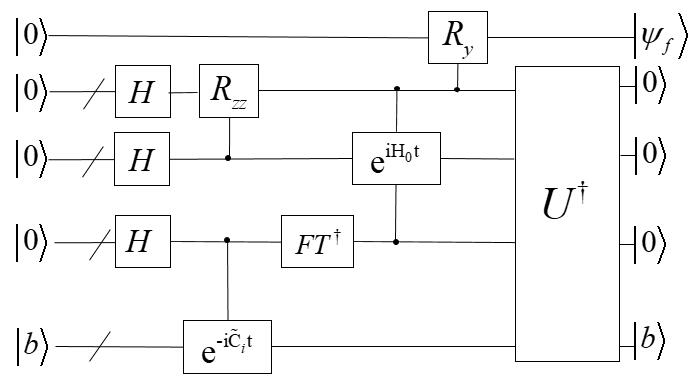


**Implementation of Quantum Algorithm for Linear System of Equations ( HHL Algorithm )**

Solving linear equations is a common task in the field of machine learning, but as the dataset grows in size, a classical computer’s time complexity increases. So a quantum computer will out in such situations by reducing the computational as well as time complexity.

**Steps**

1. First, we have a  Hermitian matrix A and unit vector . We need to find  in .
2. Represent  as a quantum state .
3. (2) Apply the conditional Hamiltonian evolution  to  for a superposition of different times . With the phase estimation algorithm, we can decompose  in the eigenbasis of and to find the corresponding eigenvalues . After this stage, the state of the system is close to , where  is the eigenvector basis of  and .
4. (3)Uncompute the  register and we get a state which is proportional to .
5. The schematic diagram of quantum K-Means is the following picture. And to make our algorithm can be run using Qiskit, we design a more detailed circuit to achieve our algorithm in the next section.



**Program**

#import the tools required

from math import pi

from qiskit import Aer, execute

from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister

from qiskit.tools.visualization import plot\_histogram

backend = Aer.get\_backend('qasm\_simulator')

# create Quantum Register called "qr" with 4 qubits

qr = QuantumRegister(4, name="qr")

# create Quantum Register called "cr" with 4 qubits

cr = ClassicalRegister(4, name="cr")

# Creating Quantum Circuit called "qc" involving your Quantum Register "qr"

# and your Classical Register "cr"

qc = QuantumCircuit(qr, cr, name="solve\_linear\_sys")

# Initialize times that we get the result vector

n0 = 0

n1 = 0

for i in range(10):

#Set the input|b> state"

qc.x(qr[2])

#Set the phase estimation circuit

qc.h(qr[0])

qc.h(qr[1])

qc.u1(pi, qr[0])

qc.u1(pi/2, qr[1])

qc.cx(qr[1], qr[2])

#The quantum inverse Fourier transform

qc.h(qr[0])

qc.cu1(-pi/2, qr[0], qr[1])

qc.h(qr[1])

#R（lamda^-1） Rotation

qc.x(qr[1])

qc.cu3(pi/16, 0, 0, qr[0], qr[3])

qc.cu3(pi/8, 0, 0, qr[1], qr[3])

#Uncomputation

qc.x(qr[1])

qc.h(qr[1])

qc.cu1(pi/2, qr[0], qr[1])

qc.h(qr[0])

qc.cx(qr[1], qr[2])

qc.u1(-pi/2, qr[1])

qc.u1(-pi, qr[0])

qc.h(qr[1])

qc.h(qr[0])

# To measure the whole quantum register

qc.measure(qr[0], cr[0])

qc.measure(qr[1], cr[1])

qc.measure(qr[2], cr[2])

qc.measure(qr[3], cr[3])

job = execute(qc, backend=backend, shots=8192,)

result = job.result()

# Get the sum og all results

n0 = n0 + result.get\_data("solve\_linear\_sys")['counts']['1000']

n1 = n1 + result.get\_data("solve\_linear\_sys")['counts']['1100']

# print the result

print(result)

# print(result.get\_data(qc))

plot\_histogram(result.get\_counts())

# Reset the circuit

qc.reset(qr)

# calculate the scale of the elements in result vector and print it.

p = n0/n1

print(n0)

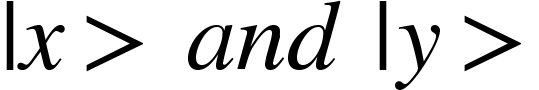
print(n1)

print(p)

**K Means Clustering:**

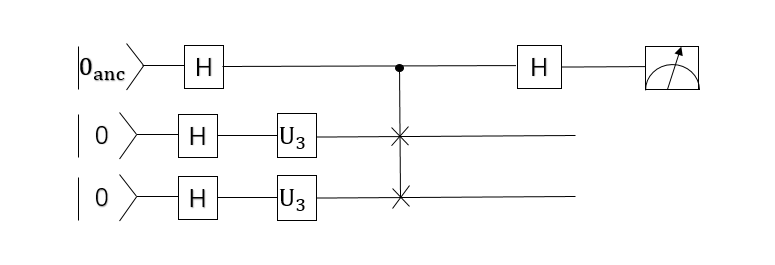
It is an unsupervised machine learning algorithm used for clustering. All the items are thrown into clusters with which they have less mean distance. It will partition N items into k clusters. K means clustering algorithm divides the n items  into multiple sets  so as to reduce the sum of squares error. The most commonly used formula to form samples of similar items is the Euclidean distance. This is a classical point of view. In the quantum sense, the core part of the algorithm remains the same where few quantum mechanical laws act upon.

The quantum K-Means algorithm mainly uses the swap test to compare the distances among the input data points. Select K points randomly from N data points as centroids, measure the distance from each point to each centroid, and assign it to the nearest centroid- class, repeat the step for the classes that have been obtained, and iterate 2 to 3 steps until the new centroid is equal to or less than the specified threshold, which will give you the final output. In the example, we are going to select 6 data points, 2 centroids, and use the swap test circuit to calculate the distance.

Here  is an Auxilary Qubit which is first put into superposition using an H-gate. Now  as control bit apply swap gate to swat the vectors . Finally, apply the Hadamard transform once more and measure the Auxilary Qubit. The result will look like this.



**Quantum K-means Circuit diagram:**

oouQ

**Quantum K-means Program using Qiskit**

**from math import pi**

**from qiskit import Aer, IBMQ, execute**

**from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister**

**from qiskit.tools.visualization import plot\_histogram**

**backend = Aer.get\_backend('qasm\_simulator')**

**theta\_list = [0.01, 0.02, 0.03, 0.04, 0.05, 1.31, 1.32, 1.33, 1.34, 1.35]**

**# create Quantum Register called "qr" with 5 qubits**

**qr = QuantumRegister(5, name="qr")**

**# create Classical Register called "cr" with 5 bits**

**cr = ClassicalRegister(5, name="cr")**

**# Creating Quantum Circuit called "qc" involving your Quantum Register "qr"**

**# and your Classical Register "cr"**

**qc = QuantumCircuit(qr, cr, name="k\_means")**

**#Define a loop to compute the distance between each pair of points**

**for i in range(9):**

**for j in range(1,10-i):**

**# Set the parament theta about different point**

**theta\_1 = theta\_list[i]**

**theta\_2 = theta\_list[i+j]**

**#Achieve the quantum circuit via qiskit**

**qc.h(qr[2])**

**qc.h(qr[1])**

**qc.h(qr[4])**

**qc.u3(theta\_1, pi, pi, qr[1])**

**qc.u3(theta\_2, pi, pi, qr[4])**

**qc.cswap(qr[2], qr[1], qr[4])**

**qc.h(qr[2])**

**qc.measure(qr[2], cr[2])**

**qc.reset(qr)**

**job = execute(qc, backend=backend, shots=1024)**

**result = job.result()**

**print(result)**

**print('theta\_1:' + str(theta\_1))**

**print('theta\_2:' + str(theta\_2))**

**# print( result.get\_data(qc))**

**plot\_histogram(result.get\_counts())**

Its output is so big. So, I will paste the first iteration of the output below. theta\_1 and theta\_2 are the first two centroids in the output shown below.

**Output:**

Result(backend\_name='qasm\_simulator', backend\_version='0.3.4', date=datetime.datetime(2019, 12, 13, 20, 52, 30, 13434), header=Obj(backend\_name='qasm\_simulator', backend\_version='0.3.4'), job\_id='7ecb518a-cf1a-4df7-8882-e66aa7c45bf0', metadata={'max\_memory\_mb': 6511, 'omp\_enabled': True, 'parallel\_experiments': 1, 'time\_taken': 0.023093791000000002}, qobj\_id='ae270e3c-7ee8-4632-b14d-f15eef5743be', results=[ExperimentResult(data=ExperimentResultData(counts=Obj(0x0=1024)), header=Obj(clbit\_labels=[['cr', 0], ['cr', 1], ['cr', 2], ['cr', 3], ['cr', 4]], creg\_sizes=[['cr', 5]], memory\_slots=5, n\_qubits=5, name='k\_means', qreg\_sizes=[['qr', 5]], qubit\_labels=[['qr', 0], ['qr', 1], ['qr', 2], ['qr', 3], ['qr', 4]]), meas\_level=<MeasLevel.CLASSIFIED: 2>, metadata={'measure\_sampling': False, 'method': 'statevector', 'parallel\_shots': 2, 'parallel\_state\_update': 1}, seed\_simulator=1049265097, shots=1024, status='DONE', success=True, time\_taken=0.022903236)], status='COMPLETED', success=True, time\_taken=0.028276443481445312)

theta\_1:0.01

Theta\_2:0.02

Quantum Chemistry

This is an interesting branch of science which deals with the principles of quantum physics to experiment with the chemical processes, molecules, and synthesis. Popularly known as molecular quantum mechanics.

For example, let us consider a hypothetical situation where a scientist is trying to obtain a chemical that will reverse the process of HIV, in short, he’s trying to get a reversible reaction with the molecules that cause HIV. The problem here is the unexpected destructive results. So he might want to first know the possibility theoretically which will take millions of years to compute in a classical computer, but whereas in the quantum computer all the constraints are considered and will take the millions of equations in order that will lead to the desired results with the information it has. That’s how powerful quantum chemistry is!

**What does it consist of?**

This deals with the quantum particles(say bosons, fermions, etc) their spins, and their momentum. It serves as a powerful tool in studying the properties of molecules and their reactions. But one might ask what is the need of studying all these? The answer is pretty simple, let me break it down for you, macroparticles are the combination of many microparticles and together they form a shape. So in order to get a clear understanding of the macroscopic world, we first need to comprehend the mysteries of microscopic particles.

**History:**

In earlier days, scientists used hydrogen as the default atom to test any theory on and they’ve conducted a series of experiments that lead to revolutionary results in the department of chemistry. But the problem was the theories were restricted to the empirical formulas of the molecules.

The biggest task of all time for the scientists is to make the systems capable of computing the large molecules. They could do all the wonders with the single-electron system, but when the size increases complexity increases too and they had to add many variables to the existing equations.

**Advancements:**

Researchers found that it is important to know the structure and characteristics of the polymer to reverse the polymeric reactions. They described it as follows:

* Synthetic polymer’s properties strongly depend on the polymerization techniques and a wide range of chemicals required to make the polymer. And it is known that the rate of polymerization is faster than any other reaction in different mediums, and it is highly difficult to measure the concentrations of all the chemicals involved. So, they’ve performed a million experiments and discovered that if a polymer is formed, it might be difficult to change the native properties. Hence, scientists deduced that this polymeric point which will let us know about the irreversibility of reaction plays a major role in understanding the favorable directions. This problem can be successfully solved by combining quantum chemistry, macroscopic-scale mechanics, and spectroscopic methods.
* Another theory suggests that electronic models can be used for self-initiated reactions in chemistry by using quantum computing. which is based on the theoretical study of thermal self-initiation of acrylates. It gives information about how an electronic model has been used to screen reaction mechanism models, meaning to create the self-initiation reaction mechanisms and to compute the kinetic parameters in the process. They compared the theoretical predictions made using the electronic model with the estimations obtained from macroscopic mechanics and the measurements made on polymers during the experiments, which proved the high reliability of electronic models. This reliability can create a great difference in the history of quantum chemistry by making use of electronic-level reaction modeling tools for silicon investigation.

Google claimed that it took another by achieving the first-ever biggest quantum chemistry simulation on 12 December 2019. Here’s the journey to the path-breaking innovation.

The quantum computer of Google, which is Sycamore uses superconducting devices as its qubits used its computational power to know the electronic structure of the hydrogen atom, which broadened the scope of investigating macroscopic compounds. It wasn’t the first time such experiments were conducted, but the amazing part here was, it did not limit the degree of complexity by using the classical computer for the initial process. The quantum prototype follows two methods namely variational quantum eigensolver (VQE) and phase-estimation algorithm (PEA). In reality, it was evident that errors are absolutely present, and that’s when the VQE method comes into the picture. It involves a series of algorithms that incrementally improve on the initial guess of the molecule’s wavefunction. It is truly possible to compensate for the errors just by adjusting the parameters in the wavefunction and we would still get an answer for the dissociation energy, The researchers believe that it is already possible to simulate more complicated molecules than hydrogen with their device.

**Algorithm:**

**import numpy as np**

**import pylab**

**from qiskit.chemistry import QiskitChemistry**

**# Input dictionary to configure Qiskit Chemistry for the chemistry problem.**

**qiskit\_chemistry\_dict = {**

**'problem': {'random\_seed': 750},**

**'driver': {'name': 'PYSCF'},**

**'PYSCF': {'atom': '', 'basis': 'sto3g'},**

**'operator': {'name': 'hamiltonian', 'transformation': 'full',**

**'qubit\_mapping': 'parity', 'two\_qubit\_reduction': True},**

**'algorithm': {},**

**}**

**molecule = 'H .0 .0 -{0}; H .0 .0 {0}'**

**algorithms = [{'name': 'VQE', 'operator\_mode': 'paulis'},**

**{'name': 'ExactEigensolver'}**

**]**

**optimizer = {'name': 'SPSA', 'max\_trials': 200}**

**variational\_form = {'name': 'RYRZ', 'depth': 3, 'entanglement': 'full'}**

**backend = {'provider': 'qiskit.BasicAer', 'name': 'qasm\_simulator', 'shots': 1024}**

**start = 0.5 # Start distance**

**by = 0.5 # How much to increase distance by**

**steps = 20 # Number of steps to increase by**

**energies = np.empty([len(algorithms), steps+1])**

**hf\_energies = np.empty(steps+1)**

**distances = np.empty(steps+1)**

**print('Processing step \_\_', end='')**

**for i in range(steps+1):**

**print('\b\b{:2d}'.format(i), end='', flush=True)**

**d = start + i\*by/steps**

**qiskit\_chemistry\_dict['PYSCF']['atom'] = molecule.format(d/2)**

**for j in range(len(algorithms)):**

**qiskit\_chemistry\_dict['algorithm'] = algorithms[j]**

**if algorithms[j]['name'] == 'VQE':**

**qiskit\_chemistry\_dict['optimizer'] = optimizer**

**qiskit\_chemistry\_dict['variational\_form'] = variational\_form**

**qiskit\_chemistry\_dict['backend'] = backend**

**else:**

**qiskit\_chemistry\_dict.pop('optimizer')**

**qiskit\_chemistry\_dict.pop('variational\_form')**

**qiskit\_chemistry\_dict.pop('backend')**

**solver = QiskitChemistry()**

**result = solver.run(qiskit\_chemistry\_dict)**

**energies[j][i] = result['energy']**

**hf\_energies[i] = result['hf\_energy']**

**distances[i] = d**

**print(' --- complete')**

**print('Distances: ', distances)**

**print('Energies:', energies)**

**print('Hartree-Fock energies:', hf\_energies)**

**Output:**

**Processing step 20 --- complete**

**Distances: [0.5 0.525 0.55 0.575 0.6 0.625 0.65 0.675 0.7 0.725 0.75 0.775**

**0.8 0.825 0.85 0.875 0.9 0.925 0.95 0.975 1. ]**

**Energies: [[-1.06086904 -1.07138175 -1.09113875 -1.10744489 -1.11953674 -1.13116184**

**-1.13320145 -1.13667867 -1.13892688 -1.13662612 -1.13536438 -1.13603326**

**-1.13339153 -1.1308772 -1.12739979 -1.12469779 -1.12047399 -1.11336415**

**-1.11008319 -1.10846154 -1.10181643]**

**[-1.05515979 -1.07591366 -1.09262991 -1.10591805 -1.11628601 -1.12416092**

**-1.12990478 -1.13382622 -1.13618945 -1.13722138 -1.13711707 -1.13604436**

**-1.13414767 -1.13155121 -1.12836188 -1.12467175 -1.12056028 -1.11609624**

**-1.11133942 -1.10634211 -1.10115033]]**

**Hartree-Fock energies: [-1.04299627 -1.06306214 -1.07905074 -1.0915705 -1.10112824 -1.10814999**

**-1.11299655 -1.11597526 -1.11734903 -1.11734327 -1.11615145 -1.11393966**

**-1.1108504 -1.10700581 -1.10251055 -1.09745432 -1.09191404 -1.08595587**

**-1.07963693 -1.07300676 -1.06610865]**

**pylab.plot(distances, hf\_energies, label='Hartree-Fock')**

**for j in range(len(algorithms)):**

**pylab.plot(distances, energies[j], label=algorithms[j])**

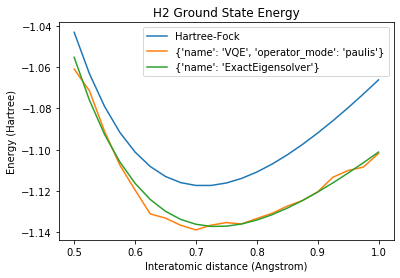
**pylab.xlabel('Interatomic distance (Angstrom)')**

**pylab.ylabel('Energy (Hartree)')**

**pylab.title('H2 Ground State Energy')**

**pylab.legend(loc='upper right');**

**Output:**

****

**pylab.plot(distances, np.subtract(hf\_energies, energies[1]), label='Hartree-Fock')**

**pylab.plot(distances, np.subtract(energies[0], energies[1]), label=algorithms[0])**

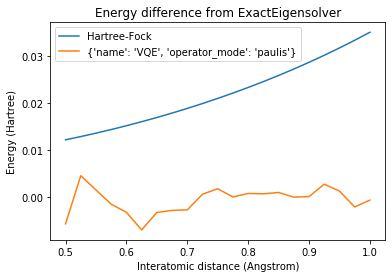
**pylab.xlabel('Interatomic distance (Angstrom)')**

**pylab.ylabel('Energy (Hartree)')**

**pylab.title('Energy difference from ExactEigensolver')**

**pylab.legend(loc='upper left');**

**Output:**

****

Exercise 9

Learn and implement the algorithm for Support Vector Machines using Quantum computing.

**\*\_ \_ The End \_ \_\***

## **References**

1. M. Nielsen, I. Chuang, Quantum Computation and Quantum Information, Cambridge University Press 2010

2. R. Cleve, A. Ekert, C. Macchiavello, M. Mosca, Quantum Algorithms revisited, arXiv:9708016

<https://github.com/Qiskit/qiskit-community-tutorials/blob/master/algorithms/deutsch_jozsa.ipynb>

<https://community.qiskit.org/textbook/ch-algorithms/simon.html#implementation>

<https://qiskit.org/documentation/_modules/qiskit/aqua/algorithms/single_sample/shor/shor.html>

C. H. Bennett and G. Brassard. "Quantum cryptography: Public key distribution and coin tossing". In *Proceedings of IEEE International Conference on Computers, Systems and Signal Processing*, volume 175, page 8. New York, 1984. <http://researcher.watson.ibm.com/researcher/files/us-bennetc/BB84highest.pdf>

Bennett, Charles H.; Brassard, Gilles (2014-12-04). "Quantum cryptography: Public key distribution and coin tossing". *Theoretical Computer Science*. Theoretical Aspects of Quantum Cryptography – celebrating 30 years of BB84. 560, Part 1: 7–11. [doi](https://en.wikipedia.org/wiki/Digital_object_identifier):[10.1016/j.tcs.2014.05.025](https://doi.org/10.1016%2Fj.tcs.2014.05.025)

<https://github.com/Qiskit/qiskit-community-tutorials/blob/master/awards/teach_me_qiskit_2018/cryptography/Cryptography.ipynb>

Zhao, Zhikuan et al. “Bayesian Deep Learning on a Quantum Computer.” Quantum Machine Intelligence 1.1-2 (2019): 41–51. Crossref. Web.

<https://github.com/Qiskit/qiskit-community-tutorials/tree/1cf11ad977aec8e0680e32c38110aa8cf3677e60>

