

**Dr. D. Y. Patil Institute of Technology**

**Pimpri Pune-411018**

**Department of Artificial Intelligence and Data Science**

**Laboratory Manual**

**Savitribai Phule Pune University**

**Second Year of Artificial Intelligence and Data Science (2020 Course)**

**Subject Code: 417526**

**Computer Laboratory II – QAI**

**Prepared by**

1. **Ritu S. Dudhmal**

Academic Year

2023-2024



**Dr. D. Y. Patil Institute of Technology**

**Pimpri Pune-411018**

**Department of Artificial Intelligence and Data Science**

**Vision of the Institute**

* Empowerment through knowledge

**Mission of the Institute**

* Developing human potential to serve the Nation
* Dedicated efforts for quality education.
* Yearning to promote research and development.
* Persistent endeavor to imbibe moral and professional ethics.
* Inculcating the concept of emotional intelligence.
* Emphasizing extension work to reach out to the society.
* Treading the path to meet the future challenges.

**Vision of the Department**

* To produce globally competent engineers in the field of Artificial Intelligence and Data Science with human values

**Mission of the Department**

* To develop students with a sound understanding in the area of Artificial Intelligence, Machine Learning and Data Science.
* To enable students to become innovators, researchers, entrepreneurs and leaders globally.
* Equip the department with new advancement in high performance equipments and software to carrying out research in emerging technologies in AI and DS.
* To meet the pressing demands of the nation in the areas of Artificial Intelligence and Data Science**.**

**Computer Laboratory II - QAI**

**417526**

|  |  |  |
| --- | --- | --- |
| **Teaching Scheme** | **Credit Scheme** | **Examination Scheme and Marks** |
| Practical: 04 Hours/Week | 02 | Term Work: 50 Marks  Practical: 25 Marks |

**Course Objectives:**

* To develop real-world problem-solving ability
* To enable the student to apply AI techniques in applications that involve perception,

reasoning, and planning

* To work in a team to build industry-compliant Quantum AI applications

**Course Outcomes:**

On completion of the course, learner will be able to–

**CO1 :** Evaluate and apply core knowledge of Quantum AI to various real-world problems.

**CO2 :** Illustrate and demonstrate Quantum AI tools for different dynamic applications.

**Guidelines for Instructor's Manual**

Lab Assignments: Following is a list of suggested laboratory assignments for reference. Laboratory Instructors may design a suitable set of assignments for their respective courses at their level. Beyond curriculum assignments, the mini-project is also included as a part of laboratory work. The Inclusion of a few optional assignments that are intricate and/or beyond the scope of the curriculum will surely be a valuable addition for the students and it will satisfy the intellectuals within the group of learners and will add to the perspective of the learners. For each laboratory assignment, it is essential for students to draw/write/generate flowcharts, algorithms, test cases, mathematical models, Test data sets, and comparative/complexity analysis (as applicable).

**Guidelines for Student's Laboratory Journal**

Program codes with sample output of all performed assignments are to be submitted as a softcopy.

The use of DVDs or similar media containing student programs maintained by the Laboratory Incharge is highly encouraged. For reference one or two journals may be maintained with program

prints in the Laboratory. As a conscious effort and little contribution towards Green IT and

environment awareness, attaching printed papers as part of write-ups and program listing to journals may be avoided. Submission of journal/ term work in the form of softcopy is desirable and

appreciated.

**Guidelines for Laboratory / Term Work Assessment**

Term work is a continuous assessment that evaluates a student's progress throughout the semester.

Term work assessment criteria specify the standards that must be met and the evidence that will be gathered to demonstrate the achievement of course outcomes. Categorical assessment criteria for the term work should establish unambiguous standards of achievement for each course outcome. They should describe what the learner is expected to perform in the laboratories or on the fields to show that the course outcomes have been achieved. It is recommended to conduct an internal monthly practical examination as part of continuous assessment.

**Guidelines for Practical Examination**

Problem statements must be decided jointly by the internal examiner and external examiner for

Elective III and Elective IV courses. **Student has to perform only one practical assignment during external evaluation either for Elective III and Elective IV courses**. During practical assessment, maximum weightage should be given to satisfactory implementation of the problem statement. Relevant questions may be asked at the time of evaluation to test the student’s understanding of the fundamentals, effective and efficient implementation. Adhere to these principles will consummate our team efforts to the promising start of student's academics.

**Guidelines for Laboratory Conduction**

Following is a list of suggested laboratory assignments for reference. Laboratory Instructors may

design a suitable set of assignments for respective courses at their level. Beyond curriculum

assignments and mini-project may be included as a part of laboratory work. The instructor may set

multiple sets of assignments and distribute them among batches of students. It is appreciated if the

assignments are based on real-world problems/applications. The Inclusion of a few optional

assignments that are intricate and/or beyond the scope of the curriculum will surely be a value

addition for the students and it will satisfy the intellectuals within the group of learners and will add to the perspective of the learners. For each laboratory assignment, it is essential for students to

draw/write/generate flowcharts, algorithms, test cases, mathematical models, Test data sets, and

comparative/complexity analysis (as applicable). Batch size for practical and tutorials may be as per guidelines of authority.

**Instructions:**

**1. Practical can be performed on suitable development platform.**

**2. Perform any 5 experiments.**

**Virtual Laboratory:**

1. https://learn.qiskit.org/course/quantum-hardware/introduction-to-quantum-errorcorrection-via-the-repetition-code

2. https://quantumcomputinguk.org/tutorials/16-qubit-random-number-generator

3. https://quantumcomputinguk.org/tutorials/quantum-fourier-transform-in-qiskit

4. https://www.sciencedaily.com/releases/2021/02/210212094105.htm

5. https://www.medrxiv.org/content/10.1101/2020.11.07.20227306v1.full

|  |  |
| --- | --- |
| **Practical No.** | **Assignment to be covered** |
| **Any 5 Assignments** | |
| 1 | Implementations of 16 Qubit Random Number Generator |
| 2 | Implement Quantum Teleportation algorithm in Python. |
| 3 | Implement Tarrataca’s quantum production system with the 3-puzzle problem |
| 4 | Tackle Noise with Error Correction |
| 5 | The Randomized Benchmarking Protocol |
| 6 | Implementing a 5 qubit Quantum Fourier Transform |

**Assignment 1**

**Problem Statement:**

Implementations of 16 Qubit Random Number Generator.

**Objective:**

1. Understand creation of Qubit circuit

2. Create 16 Qubit Random Number Generator

**Outcome:**

Displays 16 Qubit Random Number

**Theory:**

To create a Random Number Generator in qiskit for IBMs quantum computers using 16 qubits.

Requirements:

Python 3.x or above (available here: https://www.python.org/)

Pip: A package management system for Python (included with Python 3.x)

IBM Q Account: This is so you can run your programs on IBM quantum devices. You can sign up for one here: https://quantum-computing.ibm.com

Installation :

Install Python 3.x (Make sure Python is added to Path and Pip is checked)

Open Command Prompt and type in: pip install qiskit

**Steps to perform:**

**STEP 1** : INITIALISE THE QUANTUM AND CLASSICAL REGISTERS

The first step is to initialise a 16 qubit register . This is done by the following code:

q = QuantumRegister(16,’q’)

Next we initialise the 16 bit classical register with the following code:

c = ClassicalRegister(16,’c’)

**STEP 2** : CREATE THE CIRCUIT

Next we create a quantum circuit using the following code:

circuit = QuantumCircuit(q,c)

**STEP 3 :** APPLY A HADAMARD GATE TO ALL QUBITS

Then we need to apply a Hadamard gate. This gate is used to put a qubit in to a superposition of 1 and 0 such that when we measure the qubit it will be 1 or a 0 with equal probability.

This is done with the following code:

circuit.h(q)

**STEP 4** : MEASURE THE QUBITS

After this we measure the qubits. This measurement will collapse the qubits superposition in to either a 1 or a 0.

This is done with the following code:

circuit.measure(q,c)

**ALGORITHM :**

1. Start
2. pip install qiskit
3. Initialise the quantum and classical registers
4. Create the circuit
5. Apply a hadamard gate to all qubits
6. Measure the qubits
7. Stop

**How to run the program :**

Write the code in to a python file.

Enter your API token in the IBMQ.enable\_account(‘Insert API token here’) part

Save and run.

**Conclusion:**

By this way, we can generate 16 Qubit Random Number.

**Oral Questions:**

1. What is Qubit?

2. What are different steps performed to generate random number?

3. What are different types of registers?

4. What is quantum circuit?

5. How you measure the qubits?

6. What is Hadamard gate?

**Code :**

from qiskit import QuantumRegister, ClassicalRegister,

QuantumCircuit, execute,IBMQ

IBMQ.enable\_account('ENTER API TOKEN HERE')

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

q = QuantumRegister(16,'q')

c = ClassicalRegister(16,'c')

circuit = QuantumCircuit(q,c)

circuit.h(q) # Applies hadamard gate to all qubits

circuit.measure(q,c) # Measures all qubits

backend = provider.get\_backend('ibmq\_qasm\_simulator')

job = execute(circuit, backend, shots=1)

print('Executing Job...\n')

result = job.result()

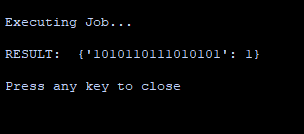
counts = result.get\_counts(circuit)

print('RESULT: ',counts,'\n')

print('Press any key to close')

input()

**Output :**



**Implementation :**

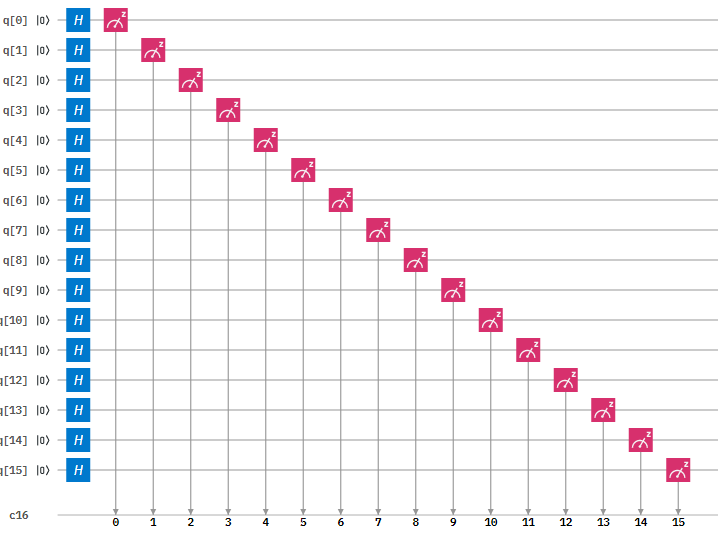


Figure 1: Circuit Diagram of the 16-qubit Random Number Generator

**Assignment 2**

**Problem Statement:**

Implement Quantum Teleportation algorithm in Python.

**Objective:**

1. Understand Quantum Teleportation algorithm.

2. Create quantum circuit.

3. Understand to perform Entanglement.

**Outcome**

Displays circuit for Quantum Teleportation

**Theory:**

**What is Quantum Teleportation?**

Quantum Computers are not really similar to Classical computers, the difference between the two is copying data because here Quantum bits is that they remain in the Quantum state till they are unobserved. As soon as we observe (or click) on them, they collapse to one of the known states. This is also called the no-cloning theorem. Hence, to copy data on a Quantum Computer, we need the process of Quantum Teleportation.

The theory behind the algorithm Let’s assume there are two friends, Kartik and Sharanya. Sharanya wants to send some form of Quantum data, possibly a qubit to Kartik. Since she can’t observe what the state of the qubit is due to the no-cloning theorem, she takes the help of the so-called ‘portal’, to transfer the data. So what the portal basically does is, that it creates entanglement between one qubit from Sharanya and one qubit of its own and sends the entangled pair towards Kartik. Then, Kartik would have to perform some actions to remove the entanglement and receive the output.

**Algorithm:**

Step 1: The portal creates an entangled pair of Qubits, which is a special pair known as Bell’s pair. In order to create a Bell’s pair using Quantum Circuits, we need to take one qubit and turn it into the (|+> or |->) state using a Hadamard gate and then using a CNOT gate on the other qubit, which will be controlled by the first qubit. One of the qubits is given to Sharanya (say Q1), the other to Kartik (say Q2).

Step 2: Let’s say that the qubit Sharanya wants to send is |Ψ> = |∝> + |β>. She needs to applied a CNOT gate to Q1, controlled by |Ψ>. A CNOT gate is basically the ‘if this, then that’ condition of the Quantum world.

Step 3: Sharanya takes a measurement of the two qubits that she has, and stores them in two classical bits. She then sends this information to Kartik (A transfer can be made since classical bits are being sent). Since qubits can handle 2n classical bits, we can say that the outputs Sharanya will get with her calculation will always be a probabilistic answer containing 00, 01, 10, and 11 (all permutations of 0 and 1).

Step 4: Now, all that Kartik needs to do is perform certain transformations on the qubit he has, Q2, which is a part of an entangled pair. This part comes from Quantum Mechanics, so you can just know it as a fact, or the complexity of the article will increase manifolds. So, if Kartik gets a 00, he needs to apply for an I gate. For 01, a X gate needs to be applied, for 10, a Z gate needs to be applied and for 11, a ZX gate needs to be applied.

And there, we have it. Kartik now has a qubit in the same state as the state Sharanya initially had her qubit in.

Module needed Qiskit: Qiskit is an open-source framework for quantum computing. It provides tools for creating and manipulating quantum programs and running them on prototype quantum devices on IBM Q Experience or on simulators on a local computer. Let’s see how we can create a simple Quantum circuit and test it on a real Quantum computer or simulate it in our computer locally.

**Installation:**

pip install qiskit

**Stepwise implementation :**

**Step 1**: Creating the Quantum Circuit on which we will be doing operations.

QuantumCircuit takes in 2 arguments, the number of qubits that we want to take and the number of classical bits that we want to take.

from qiskit import \*

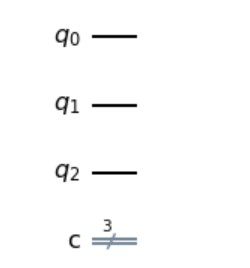
circuit = QuantumCircuit(3, 3)

%matplotlib inline # enables the drawing of matplotlib figures in the IPython environment.

# Whenever during any point of the program we want to see how our circuit looks like, this is what we will be doing.

circuit.draw(output='mpl')

Output:



This is how our circuit looks right now. We have three quantum bits and 3 classical bits, which will be used to measure the values of these Qubits, whenever we want. Right now they don’t have any value in them.

**Step 2**: Applying an X gate on the qubit which we have to teleport. We will also be adding a barrier, just to make the circuit more clear.

Now here, what we have done till now is that we have 3 qubits. What we’ll be doing is that we will be using q1 to transport data from q0 to q3. For this, we will use an X gate to init q0 to 1 state

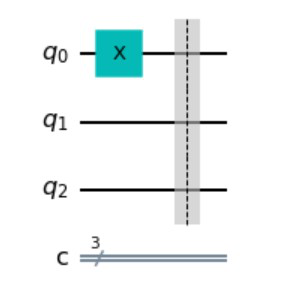
circuit.x(0) # used to apply an X gate.

# This is done to make the circuit look more organized and clear.

circuit.barrier()

circuit.draw(output='mpl')

Output:



This is how our circuit looks right now. On one of the Qubits, we have added an X gate, shown by the X. A barrier is added to make the circuit more organized as we keep on adding gates and other things. The classical bits are still untouched.

**Step 3**: Creating entanglement between Q1 and Q2 by applying a Hadamard gate on Q1, and a CX gate on Q1 and Q2 in such a way that the behavior of Q1 affects the behavior of Q2.

Here, what we’ll do is that we will create entanglement so that the behavior of the first qubit affects the behavior of the second qubit.

# This is how we apply a Hadamard gate on Q1.

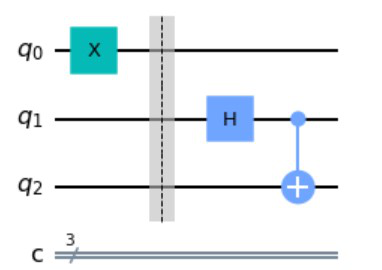
circuit.h(1)

# This is the CX gate, which takes two parameters, one being the control qubit and the other being the target qubit.

circuit.cx(1, 2)

circuit.draw(output='mpl')

Output:



Here we can see that after this step, this is how our circuit looks. We have a Hadamard gate applied to Q1, shown by the ‘H’ Symbol, and a Controlled NOT(CX) gate on Q1 and Q2. The classical bits are still untouched.

Creating entanglement between Q0 and Q1 by applying a Hadamard gate on Q0, and a CX gate on Q0 and Q1 in such a way that the behavior of Q1 affects the behavior of Q0. So essentially, we have a system where the behavior of either of the Qubits will affect the behavior of all the Qubits.

Q1 can be considered as the portal we talked of, above. We will also now project the Qubits on the classical bits and measure the values of Q0 and Q1.

# The next step is to create a controlled gate between qubit 0 and qubit 1.

# Also we will be applying a Hadamard gate to q0.

circuit.cx(0, 1)

circuit.h(0)

# Done for clarification of the circuit again.

circuit.barrier()

# the next step is to do the two measurements on q0 and q1.

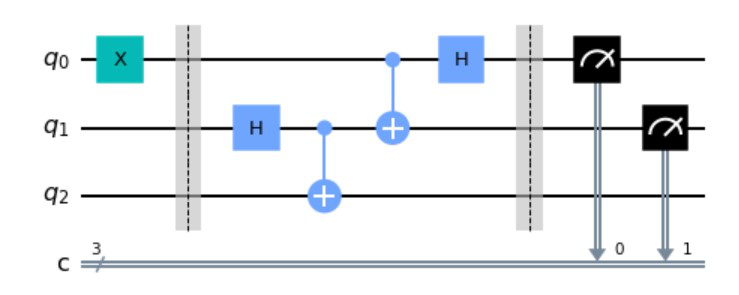
circuit.measure([0, 1], [0, 1])

# circuit.measure can take any number of arguments, and has the following parameters:

# [qubit whose value is to be measured, classical bit where the value is stored]

circuit.draw(output='mpl')

Output:



Here we can see how our circuit looks. With the help of barriers, you can easily distinguish what was done in this step. We have a Hadamard gate applied to Q0, shown by the ‘H’ Symbol, and a Controlled NOT(CX) gate on Q0 and Q1. The classical bits are now put to use, with the black symbols showing that we have taken the value of Q0 and Q1 and stored it in classical bit 1 and classical bit 2.

The actual explanation will require an understanding of Quantum mechanics, so one can just understand that for a 00 measurement of the classical bits, we need to apply an I gate. For 01, an X gate needs to be applied, for 10, a Z gate needs to be applied and for 11, a ZX gate needs to be applied. Since we don’t know what the value will be stored in classical bits, we are applying the more generalized Control X gates and Control Z gates.

The last step is to add two more gates, a controlled x gate and a controlled z gate (We have talked about this step in the text above, which tells what gates to apply for different measurements).

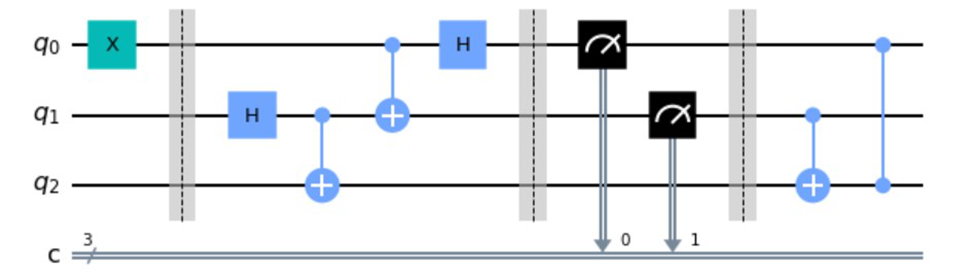
circuit.barrier()

circuit.cx(1, 2)

circuit.cz(0, 2)

circuit.draw(output='mpl')

Output:



This is the final required circuit for Quantum Teleportation. After the barrier, we can see that a Controlled X gate has been applied on Q1 and Q2, such that the behavior of Q2 affects the behavior of Q1. Also, a Control Z gate has been applied as shown between Q0 and Q2.

Now that our circuit has been made, all we need to do is to pass that circuit into a simulator so that we can get the results from the circuit back. Once we get the results back, we are plotting a histogram from the values of the classical bits that we have received. You can think of whatever we have done now as an Abstract Data Type created by us, and the code below is the main function where we will put it to use. Each step has been explained in complete detail for easy understanding of the reader.

# The first step is to call a simulator

# which we will use to perform simulations.

from qiskit.tools.visualization import plot\_histogram

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

# here, like before, we have given the

# classical bit 2 the value of the Quantum bit 2.

circuit.measure(2, 2)

# Now, we run the execute function,

# which takes our quantum circuit,

# the backend which we are using and

# the number of shots we want

# (shots are to increase accuracy and

# mitigate errors in Quantum Computing).

# All of this is stored in a variable called result

result = execute(circuit, backend=sim, shots=1000).result()

counts = result.get\_counts()

# This counts variable shows that for each possible combination,

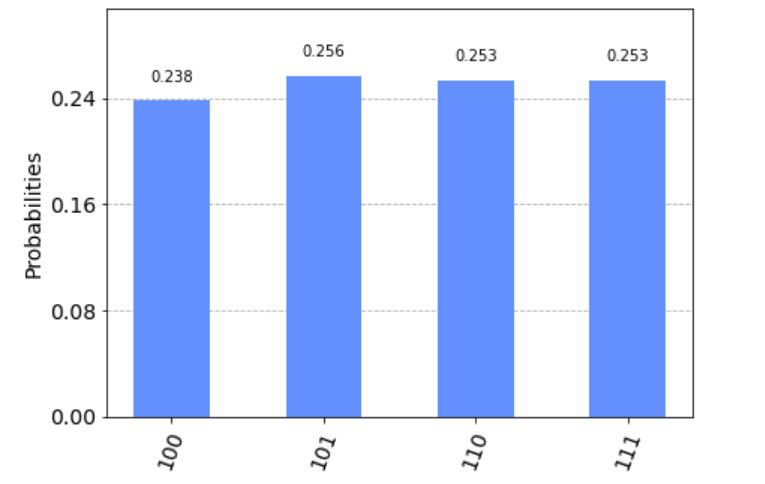
# how many times the circuit gave a similar output

# (for example, 111 came x times, 101 came y times etc.)

# importing plot\_histogram which will help us visualize the results.

plot\_histogram(counts)

Output:



**Algorithm:**

1. Start
2. Creating the Quantum Circuit.
3. Applying an X gate on the qubit which we have to teleport.
4. Creating entanglement between Q1 and Q2 to create a controlled gate between qubit 0 and qubit 1.
5. Apply a Hadamard gate to q0.
6. Do the two measurements on q0 and q1.
7. Add two more gates, a controlled x gate and a controlled z gate
8. Stop.

**Conclusion:**

By this way, we have Implemented quantum teleportation algorithm in python.

**Oral Questions:**

1. What is entanglement?
2. What are different gates used in teleportation?
3. What is teleportation?
4. How Quantum circuit is designed?

**Assignment 3**

**Problem Statement:**

Implementing a 5 qubit Quantum Fourier Transform

**Objective:**

1. Understand Quantum Fourier Transform

2. Create 5 qubit Quantum Fourier Transform

**Outcome**

Displays circuit for 5 qubit Quantum Fourier Transform.

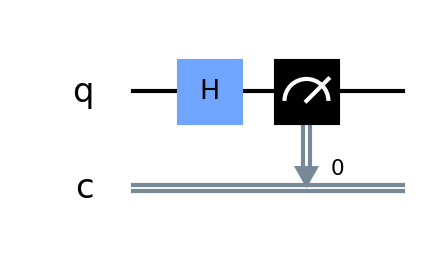
**Theory:**

What is the Quantum Fourier Transform?

The Quantum Fourier Transform (QFT) is a circuit that transforms the state of the qubit from the computational basis to the Fourier basis. *Note that the Fourier basis is just another term for the Hadamard basis.* As such the easiest way to implement a QFT is with Hadamard gates and Controlled U1 gates.

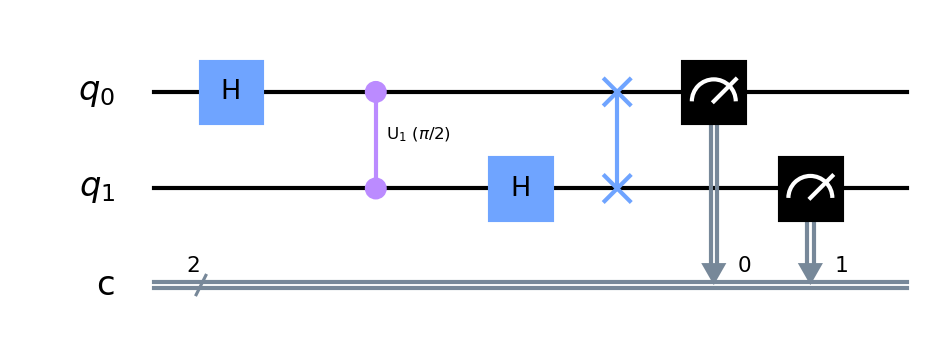
Note: A Controlled U1 gate is just a gate that implements a single rotation around the Z-axis (phase) of the target qubit if the control qubit is 1.

Circuit diagram of a 1 qubit QFT



The simplest QFT is a 1 qubit QFT which just implements a Hadamard gate.

However if we implement a 2 qubit QFT then you can see how the controlled U1 are used:

2 qubit QFT

First we implement a Hadamard gate which puts q0 in to superposition. Next we apply a controlled U1 gate with a rotation of pi/2 to q1. After this a Hadamard gate is applied to q1. Next we apply a swap gate to q0 and q1.

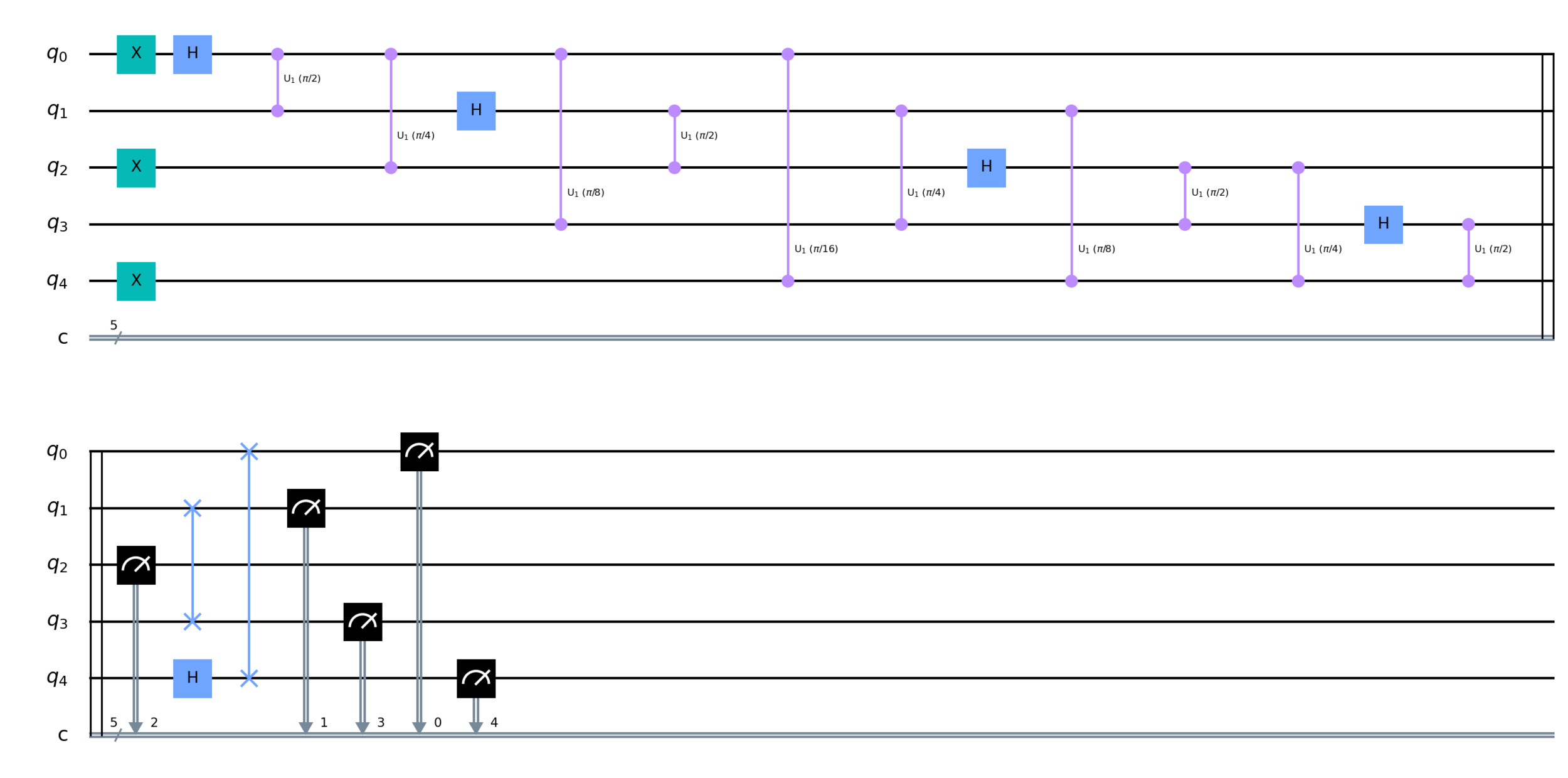
**Note that** these swap gates are not needed if the QFT is implemented at the end of your circuit.

After this both qubits will be in superposition but whatever computational value (1 or 0) will be encoded in to the Hadamard basis of the qubit.

To encode values on N qubits we have to double the rotation value of each qubit. For example the diagram below shows a 5 qubit QFT.

Notice how for q0 it applies a rotation of pi/2 for q1 then pi/4 for q2 then pi/8 for q3 and so on. This pattern repeats for each qubit. When all rotations have been applied to a qubit it is put in to superposition using a Hadamard gate. Then it can be used as a control qubit to apply rotations to target qubits below it.

Implementation



Circuit diagram of a 5 qubit QFT

Implementing a 5 qubit Quantum Fourier Transform in qiskit

In qiskit we could implement the 5 qubit QFT by implementing all the gates in the diagram above.

**In qiskit you can use the QFT() function as follows:**

QFT(num\_qubits=None, approximation\_degree=0, do\_swaps=True, inverse=False, insert\_barriers=False, name='qft')

Where:

num\_qubits: The number of qubits we want to add to the QFT (in our case it is 5)

approximation\_degree: This allows us to reduce circuit depth by ignoring phase rotations under a certain value

do\_swaps: If set to true then we use swap gates in the QFT

inverse: If set to true we implement the inverse QFT

insert barrier: If set to true then we insert barriers

For example in our 5 qubit QFT we implement the following:

QFT(num\_qubits=5, approximation\_degree=0, do\_swaps=True, inverse=False, insert\_barriers=True, name='qft')

If we encode 1010 on to a QFT and then measure it we will get random values since the qubits have been put in to superposition and the values we encoded in to the computational basis are now encoded in the Hadamard basis of each qubit via the controlled U1 gates.

Inverse Quantum Fourier Transform

To get our values back we can use the inverse QFT. This reverses all the rotations done in the QFT above.

For example is there was a rotation of Pi in the QFT then the inverse QFT will do a rotation of -Pi.

In qiskit we can get the values back by implementing an inverse QFT by setting inverse to true.

**For example:**

QFT(num\_qubits=5, approximation\_degree=0, do\_swaps=True, inverse=True, insert\_barriers=True, name='qft')

**How to run the program**

Copy and paste the code below in to a python file

Enter your API token in the IBMQ.enable\_account('Insert API token here') part

Save and run

**Code :**

from qiskit import QuantumRegister, ClassicalRegister

from qiskit import QuantumCircuit, execute,IBMQ

from qiskit.tools.monitor import job\_monitor

from qiskit.circuit.library import QFT

import numpy as np

pi = np.pi

IBMQ.enable\_account(‘ENTER API KEY HERE’)

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

backend = provider.get\_backend('ibmq\_qasm\_simulator')

q = QuantumRegister(5,'q')

c = ClassicalRegister(5,'c')

circuit = QuantumCircuit(q,c)

circuit.x(q[4])

circuit.x(q[2])

circuit.x(q[0])

circuit += QFT(num\_qubits=5, approximation\_degree=0, do\_swaps=True, inverse=False, insert\_barriers=False, name='qft')

circuit.measure(q,c)

circuit.draw(output='mpl', filename='qft1.png')

print(circuit)

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

job\_monitor(job)

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

print("\n QFT Output")

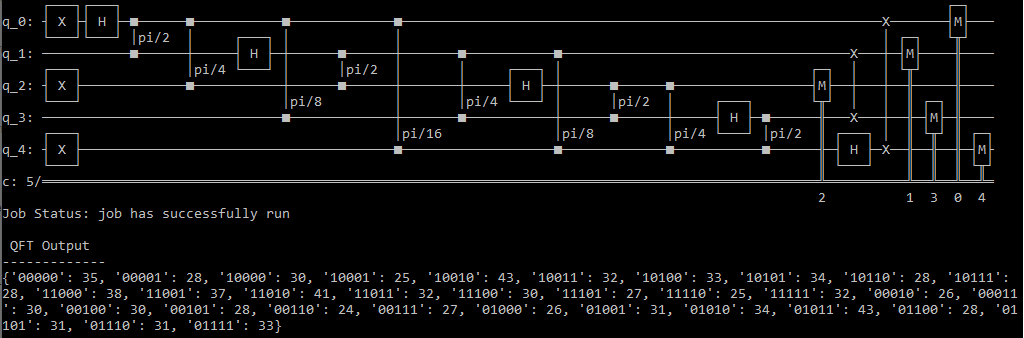
print("-------------")

print(counts)

input()

**Output**

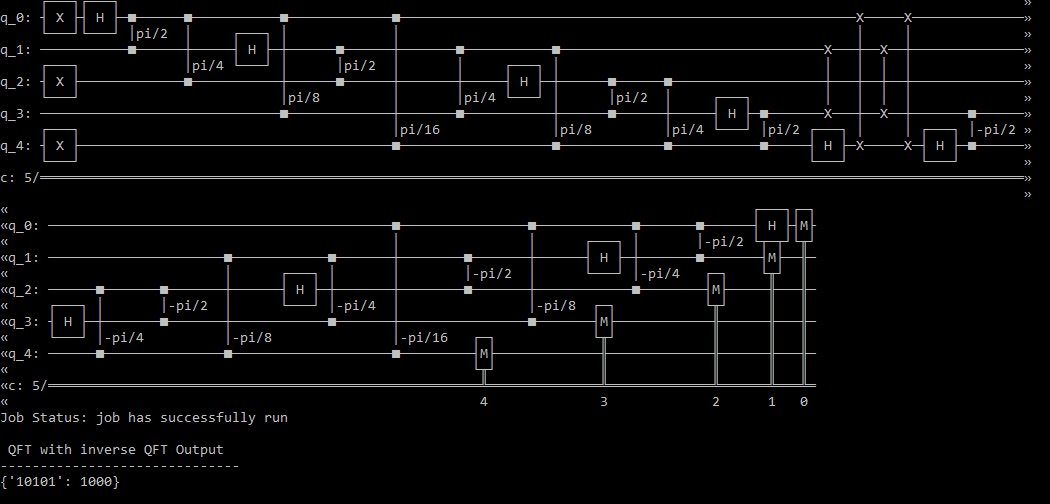
Here is the output when running the QFT:



Output when running the 5 qubit QFT. Notice how we get multiple values back since the qubits are in superposition and the value is encoded in the Hadamard basis.

Here is the output when running the QFT with the inverse QFT.

**Oral Questions:**

Notice how we get the 1010 back!

**Assignment 4**

**Problem Statement:**

Tackle Noise with Error Correction

**Objective:**

1. Understand Tackle Noise with Error Correction

**Outcome**

Displays circuit for Tackle Noise with Error Correction.

**Software Requirement**: Ubuntu OS, Quskit (Cloud based)

**Theory:**

Quantum error correction is theorised as essential to achieve fault tolerant quantum computing that can reduce the effects of noise on stored quantum information, faulty quantum gates, faulty quantum preparation, and faulty measurements. This would allow algorithms of greater circuit depth.

Noise is a major challenge in quantum computing, as it can cause errors in quantum computations. Quantum error correction (QEC) is a technique that can be used to tackle noise and improve the reliability of quantum computers.

QEC works by encoding a single logical quantum bit (qubit) into multiple physical qubits. This redundancy allows QEC to detect and correct errors that occur on the physical qubits.

There are a number of different QEC codes, each with its own strengths and weaknesses. Some QEC codes are more efficient, while others are more robust to noise.

QEC is still under development, but it has already been demonstrated in small-scale experiments. As QEC codes improve and become more efficient, they will be essential for building large-scale quantum computers.

Here is an example of how QEC can be used to tackle noise in a quantum communication system:

The sender encodes a single logical qubit into multiple physical qubits using a QEC code.

The sender transmits the physical qubits to the receiver.

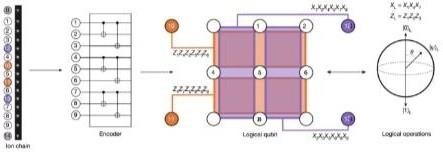
The receiver uses the QEC code to detect and correct any errors that occurred during transmission.

The receiver decodes the physical qubits to recover the logical qubit.

If a small number of errors occur during transmission, the QEC code will be able to correct them and the receiver will be able to recover the logical qubit accurately. However, if too many errors occur, the QEC code will not be able to correct them and the receiver will not be able to recover the logical qubit accurately.

QEC can also be used to tackle noise in quantum computers. For example, QEC can be used to protect the qubits in a quantum computer from noise caused by faulty quantum gates or environmental interactions.

QEC is a powerful tool for tackling noise in quantum computing and communication. As QEC codes improve and become more efficient, they will play an essential role in building large-scale quantum computers and enabling reliable quantum communication.



**How to run the program**

Copy and paste the code below in to a python file

Enter your API token in the IBMQ.enable\_account('Insert API token here') part

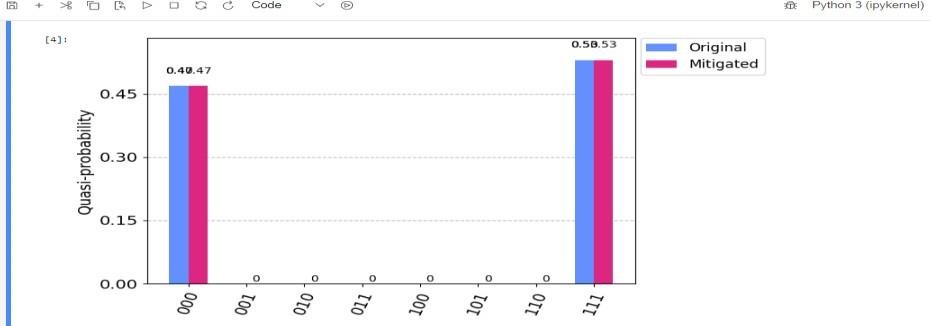
Save and run

**Code :**



plot\_histogram([counts, mitigated\_counts], legend=['Original', 'Mitigated'])

Output:



**Oral Questions:**

Q1. How can we adapt error correction codes to different types of noise? How can we design error correction codes that are scalable to large numbers of qubits?

Q2. How can we implement error correction in a way that minimizes overhead?

Q3. How can we combine error correction with other techniques, such as quantum feedback control, to improve the performance of quantum devices?

**Experiment No – 5**

**Aim:** The Randomized Benchmarking Protocol.

**Outcome:** At end of this experiment, student will be able The Randomized Benchmarking Protocol.

**Software Requirement:** Ubuntu OS,Python Editor(Pyhton Interpreter).

## Theory: The Randomized Benchmarking:

## The term "Randomized Benchmarking" (RB) refers to a specific experimental technique in quantum computing used to assess the performance and error rates of quantum gates or operations. It is a protocol designed to measure the average fidelity or error rate of a set of quantum gates or operations, often referred to as "Clifford gates." Randomized Benchmarking is widely used to quantify the quality of quantum hardware and provides a benchmark for evaluating the noise and errors in a quantum processor.

## The Randomized Benchmarking Protocol:

## The Randomized Benchmarking Protocol (RB) is a widely used technique in quantum computing for assessing and quantifying the error rates in a quantum processor's gate operations. It provides a robust and efficient method to estimate the average error rate of a set of quantum gates. RB is designed to be relatively insensitive to the specifics of the quantum hardware and is often used to benchmark the performance of quantum processors. Here's an overview of the theory behind the Randomized Benchmarking Protocol:

## 1. Error Rates in Quantum Gates:

## In quantum computing, gates are used to perform operations on qubits. These gates are subject to errors due to various sources, such as decoherence, control imperfections, and environmental noise.

## 2. The Goal of RB:

## The primary goal of RB is to provide a reliable way to estimate the average error rate of a set of quantum gates, often referred to as the "Clifford gates." These gates form a basis set that can be used to create any quantum state.

## 3. Clifford Group:

## The Clifford group is a specific set of quantum gates known for their stability and error-correcting properties. They include gates like the Hadamard gate, CNOT gate,

## 

## and phase gate.

## 4. Randomized Benchmarking Sequence:

## RB sequences are constructed by applying random sequences of Clifford gates followed by an inverse sequence to undo the gate operations. These sequences are designed to be "random" to ensure that errors accumulate over time.

## 5. Expected Outcome:

## In the absence of errors, applying a random sequence of Clifford gates and then its inverse should return the qubits to their initial state. However, errors cause deviations from this ideal behavior.

## 6. Decay of Quantum Coherence:

## As RB sequences get longer, the quantum coherence of the qubits decays due to errors. RB quantifies this decay by measuring how the fidelity (similarity to the ideal state) of the output state decreases as a function of sequence length.

## 7. Error Rate Estimation:

## By performing RB experiments with sequences of different lengths, you can estimate the error rate by fitting the observed fidelity decay to a specific mathematical model. This model accounts for the accumulation of errors over sequence length.

## 8. Robustness:

## One of the strengths of RB is its robustness. RB error estimates are relatively insensitive to the specific details of the quantum hardware and can provide a reliable benchmark even when the hardware is noisy.

## 9. Scalability:

## RB can be scaled to assess the error rates of large sets of gates and can be used to compare the performance of different quantum processors.

## 10. Benchmarking Progress:

## RB can be used to track the progress of quantum hardware and software improvements over time. By regularly performing RB experiments, researchers can monitor changes in error rates and identify areas for improvement.

## In summary, the Randomized Benchmarking Protocol is a valuable tool in the field of quantum computing for characterizing the performance of quantum gates. It provides a practical way to estimate

## and track error rates, which is crucial for assessing the reliability of quantum processors and for advancing the field of quantum error correction.

## 

## Program :

## import numpy as np

## from qiskit import QuantumCircuit, transpile, Aer, execute

## # Generate a random quantum circuit

## def generate\_random\_circuit(num\_qubits, depth): circuit = QuantumCircuit(num\_qubits, num\_qubits) for \_ in range(depth):

## for qubit in range(num\_qubits): circuit.rx(np.random.uniform(0, 2 \* np.pi), qubit) circuit.ry(np.random.uniform(0, 2 \* np.pi), qubit) circuit.rz(np.random.uniform(0, 2 \* np.pi), qubit)

## for qubit in range(num\_qubits - 1): circuit.cz(qubit, qubit + 1)

## return circuit

## # Perform randomized benchmarking

## def randomized\_benchmarking(num\_qubits, depths, num\_sequences, shots): backend = Aer.get\_backend('statevector\_simulator')

## results = []

## for depth in depths:

## success\_counts = 0

## for \_ in range(num\_sequences):

## # Generate a random circuit and the corresponding inverse circuit circuit = generate\_random\_circuit(num\_qubits, depth) inverse\_circuit = circuit.inverse()

## # Apply the circuit and obtain the final statevector circuit\_result = execute(circuit, backend=backend).result() final\_statevector = circuit\_result.get\_statevector()

## # Apply the inverse circuit and obtain the final statevector inverse\_result = execute(inverse\_circuit, backend=backend).result() inverse\_statevector = inverse\_result.get\_statevector()

## # Calculate the success rate based on state fidelity

## fidelity = np.abs(np.dot(final\_statevector, inverse\_statevector.conj()))

\*\*2

success\_counts += shots \* (1 - fidelity)

success\_rate = success\_counts / (num\_sequences \* shots)

results.append (success\_rate)

return results

# Example usage num\_qubits = 2

depths = [1, 2, 3, 4]

## num\_sequences = 100

## shots = 1024

## 

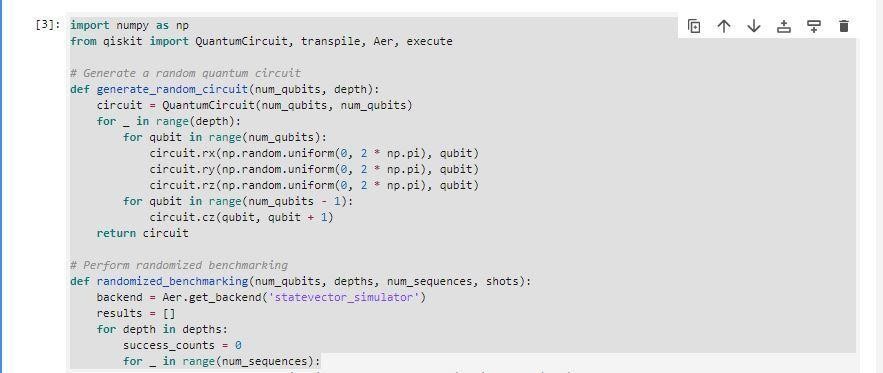
## results = randomized\_benchmarking(num\_qubits, depths, num\_sequences, shots)

## print(results)

## Output :

## Result : [0.6116681128758453, 0.7011809859732847, 0.7120854233100897,

## 0.6813325854678913]

**Code**:

## 

## 02.JPG

Conclusion: - The Randomized Benchmarking Protocol is performed successfully.

## Questions:

## Q1. How Does Randomized Benchmarking Differ from Other Quantum Error Characterization Techniques?

## Q2 What Mathematical Models are Used to Analyze RB Data and Extract Error Rates?

## Q3 How Can RB Be Applied to Specific Quantum Hardware or Software.