Experiment No. 1

Aim: This practical manual aims to guide users in creating a 16-qubit random number generator using quantum computing techniques for scientific, cryptographic, and practical applications.

Objective: To Study:

1. Initialization of the quantum and classical registers
2. Creation of circuit & implementation of gates to all qubits
3. Measurement of qubits & creation of 16-qubit random number generator

Theory:

A random number generator (RNG) is a fundamental tool in various scientific, cryptographic, and practical applications. Quantum computing offers a unique approach to generating random numbers using the principles of quantum mechanics. In this section, we will explore how a 16qubit random number generator works and what its outputs mean.

Quantum Bits (Qubits):

Quantum bits, or qubits, are the fundamental building blocks of quantum computing. Unlike classical bits (0 or 1), qubits can exist in superpositions of states, representing both 0 and 1 simultaneously. A 16-qubit random number generator utilizes 16 qubits, allowing for an exponentially large state space and generating highly unpredictable results.

Quantum Circuit Design:

To create a random number generator, we design a quantum circuit using quantum gates. Quantum gates manipulate the state of qubits to perform operations. The circuit for random number generation involves applying a series of quantum gates to the initial qubit states to create a complex quantum superposition.

Quantum Superposition:

In quantum mechanics, qubits can exist in a superposition of states until measured. This means that before measurement, each qubit is in a combination of 0 and 1. The complex quantum superposition created by the circuit ensures that the outcome of a measurement is truly random and unpredictable.

Measurement:

To obtain a random number, the quantum circuit is measured, collapsing the qubits' superposition into a definite 0 or 1 for each qubit. Repeating this measurement process multiple times generates a sequence of 16 binary values, forming the random number.

Quantum Randomness:

The randomness of quantum measurements is a fundamental property of quantum mechanics. It arises from the probabilistic nature of measuring qubits in superposition. Unlike classical random number generators, which may have inherent biases or patterns, a quantum random number generator is theoretically unbiased and produces truly random and uncorrelated numbers.

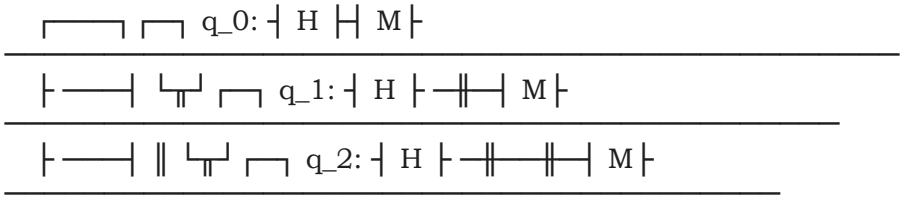
Applications:

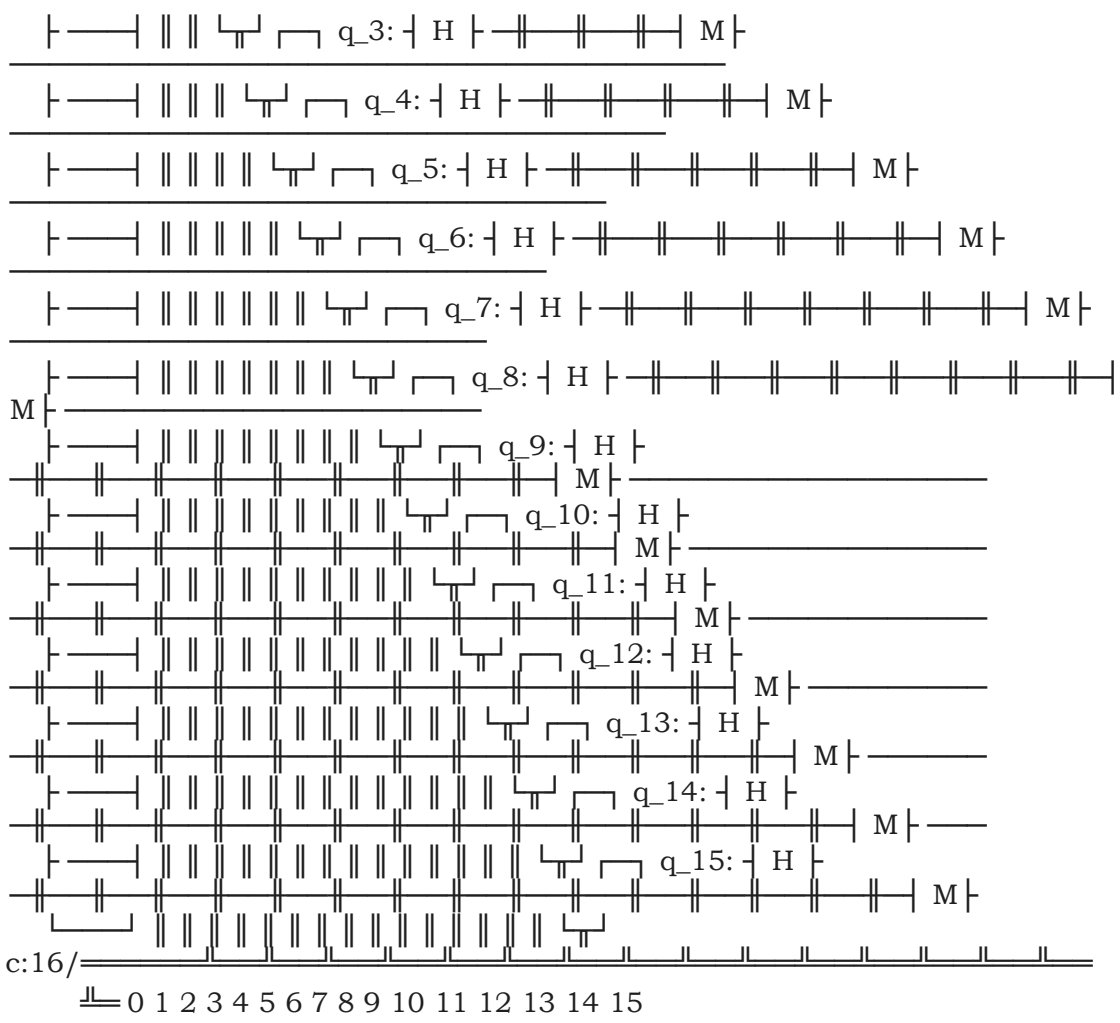
Quantum random numbers have applications in cryptography, where secure and unpredictable random numbers are crucial for encryption and key generation. They are also used in Monte Carlo simulations, scientific experiments, and games of chance where fairness and unpredictability are required.

Input:

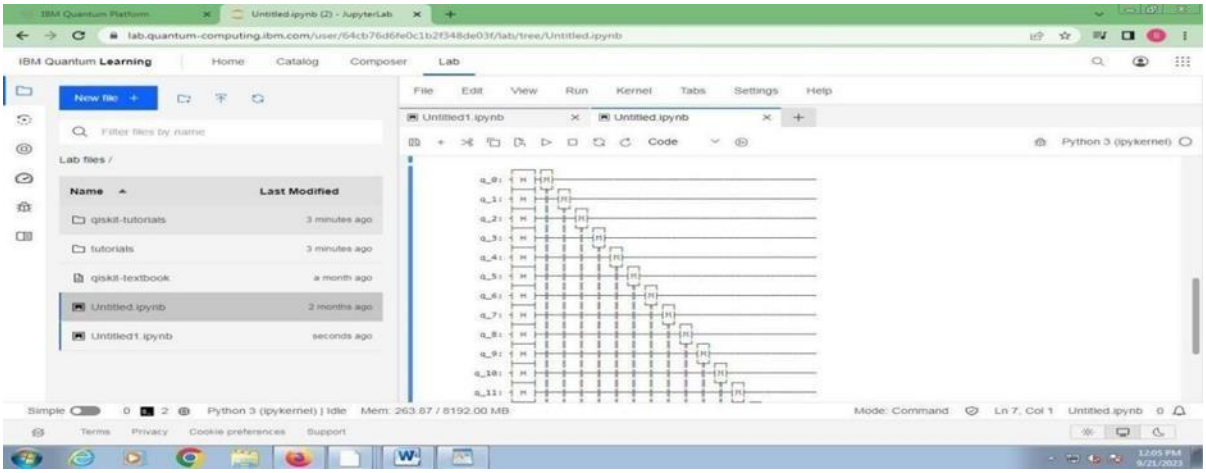
1. Initialize the quantum and classical registers
2. Create the circuit
3. Apply Hadamard gate to all qubits
4. Measure the qubits

Output:





Random number (decimal): 27678
Random number (binary): 0110110000011110



Conclusion:

Thus we have implemented 16-qubit random number generator using quantum computing techniques

Outcome:

Upon completion of this experiment, students will be able to:

Experiment level outcome (ELO1): Implement 16 Qubit Random Number Generator

Questions:

1. How do qubits in quantum computing differ from classical bits, and why is this difference essential for random number generation?

ANS:

Characteristic	Qubits (Quantum)	Bits (Classical)
Fundamental Unit	Quantum computing uses qubits as the fundamental unit.	Classical computing uses bits as the fundamental unit.
Information Storage	Can represent 0, 1, or any superposition of these states.	Can only represent 0 or 1.

Characteristic	Qubits (Quantum)	Bits (Classical)
Superposition	Qubits can exist in multiple states simultaneously, allowing parallel processing.	Classical bits can only exist in one state at a time, leading to sequential processing.
Entanglement	Qubits can become entangled, allowing for correlated behavior.	Classical bits do not exhibit entanglement.
Measurement	Measurement collapses a qubit's superposition into a definite state.	Measurement provides a deterministic value (0 or 1) without superposition.

The difference between qubits in quantum computing and classical bits is fundamental to random number generation because of the inherent properties of superposition and entanglement. These properties allow quantum computers to explore a vast number of possibilities simultaneously, making it exceptionally well-suited for generating truly random numbers.

Quantum random number generators exploit the superposition property to create a state where the qubit is in a superposition of 0 and 1. When measured, it collapses into either 0 or 1 with certain probabilities, creating random outcomes. Additionally, entanglement can be harnessed to create correlated random numbers.

2. What role does quantum superposition play in ensuring the randomness of quantum measurements in a 16-qubit random number generator?

ANS: Quantum superposition in a 16-qubit random number generator allows each qubit to exist in a combination of states simultaneously. When these qubits are measured, their collective superposition leads to a probabilistic outcome, generating truly random numbers. Superposition enhances the unpredictability of measurements, ensuring the randomness of the generated numbers.

3. In what practical applications can the output of a 16 qubit random number generator be utilized, and why is quantum randomness significant in these contexts?

ANS: The output of a 16-qubit random number generator can be utilized in applications such as cryptography, secure communications, Monte Carlo simulations, and randomized algorithms. Quantum randomness is significant in these contexts because it ensures unpredictability and enhanced security, making it extremely challenging for adversaries to predict or manipulate the generated random numbers, thereby safeguarding sensitive information and enhancing the reliability of simulations and algorithms.

Experiment No. 2

Aim: Tackle Noise with Error Correction

Objective:

To Study:

1. Identification of the noisy channel & selection of an error correction technique.
2. Implementation of an error detection and correction.
3. Integration of it into your system, Testing and optimization
4. Monitoring and maintenance

Theory:

Quantum error correction is theorized 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.

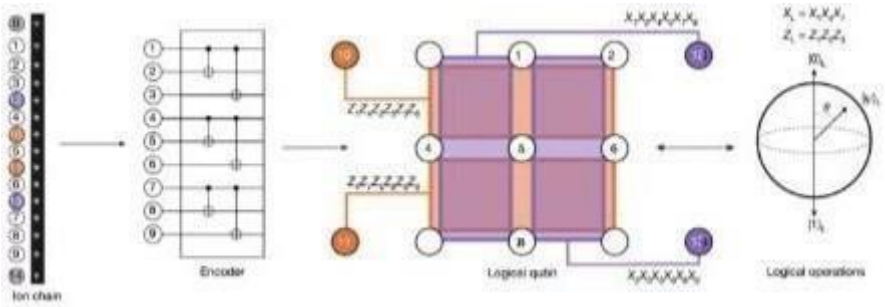
Here is an example of how QEC can be used to tackle noise in a quantum communication system:

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.

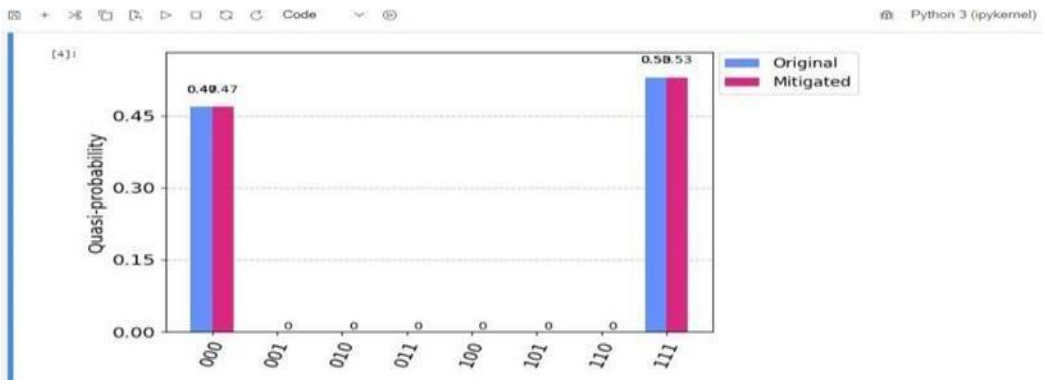
QEC is a powerful tool for tackling noise in quantum computing and communication. As QEC codes improve and become more efficient, they will play an essential role in building large-scale quantum computers and enabling reliable quantum communication.



Input:

1. Identify the noisy channel.
2. Choose an error correction technique.
3. Implement error detection and correction.
4. Integrate into your system.
5. Test and optimize.
6. Monitor and maintain.

Output:



Conclusion:

Thus we have Tackled Noise with Error Correction

Outcome:

Upon completion of this experiment, students will be able to:

Experiment level outcome (ELO2): Tackle Noise with Error Correction

Questions:

1. 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?

ANS: Adapting error correction codes to different types of noise involves customizing the code to the specific error characteristics. For example, using surface codes for qubits vulnerable to correlated errors or designing codes that suit depolarizing noise for independent errors. Scalability to large qubit numbers involves finding efficient codes and optimizing error correction procedures to handle the increased complexity, often through techniques like hierarchical or concatenated codes.

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

ANS:

To minimize error correction overhead in quantum computing:

1. Choose efficient error correction codes.
2. Optimize qubit allocation and layout.
3. Employ local error correction.
4. Use fault-tolerant techniques like stitching.
5. Develop accurate error models.
6. Implement error correction when needed, considering the error correction threshold.

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

ANS:

Combining error correction with quantum feedback control can enhance quantum device performance by dynamically adapting error correction based on real-time error monitoring, optimizing resource allocation, and actively mitigating errors as they occur, resulting in improved reliability and efficiency.

Experiment No. 3

Aim: Implement Tarrataca's quantum production system with the 3-puzzle problem

Objective:

To Study:

1. Creation of a quantum circuit with the desired number of qubits and classical bits.
2. Application of gate operations & measuring the qubits and storing the measurement results in classical bits
3. Selection of a backend simulator & execution of the quantum circuit on the chosen backend with a specified number of shots.
4. Retrieval and analysis of the measurement results.

Theory:

Tarrataca's quantum production system is a quantum algorithm for solving combinatorial optimization problems. It works by constructing a quantum circuit that represents the problem space and then applying a sequence of quantum operations to the circuit to search for the optimal solution.

The 3-puzzle problem is a combinatorial optimization problem where the goal is to arrange three tiles in order, given an initial state. Each tile can be moved either left or right, and the cost of a move is equal to the distance that the tile is moved.

To implement Tarrataca's quantum production system for the 3-puzzle problem, we can use the following steps:

Encode the problem state as a quantum state. We can use a qubit to represent each tile, and the state of the qubit will represent the position of the tile. For example, if a tile is in the leftmost position, we can represent it with the qubit state |0>, and if it is in the rightmost position, we can represent it with the qubit state |1>.

Construct a quantum circuit that represents the problem space. The quantum circuit will have a qubit for each tile, and the gates will represent the possible moves that can be made. For example, we can use a CNOT gate to represent a move that exchanges the positions of two tiles.

Apply a sequence of quantum operations to the circuit to search for the optimal solution. We can use a variety of quantum algorithms to search for the optimal solution, such as Grover's algorithm or amplitude amplification.

Measure the qubits to obtain the optimal solution. Once we have found the optimal solution, we can measure the qubits to obtain the positions of the tiles.

Here is an example of a quantum circuit that can be used to solve the 3-puzzle problem:

```
q0: ---
q1: ---
q2: ---


CNOT q0 q1
CNOT q1 q2

H q0
H q1
H q2


Grover's algorithm


M q0
M q1
M q2
```

This circuit starts with the three qubits in the state |000>. The first two CNOT gates exchange the positions of the first two qubits and the second two qubits, respectively. The three Hadamard gates put the qubits into a superposition of states.

Grover's algorithm is then used to search for the optimal solution. Grover's algorithm is a quantum algorithm that can amplify the probability of finding a solution to a search problem.

Finally, the three qubits are measured to obtain the optimal solution.

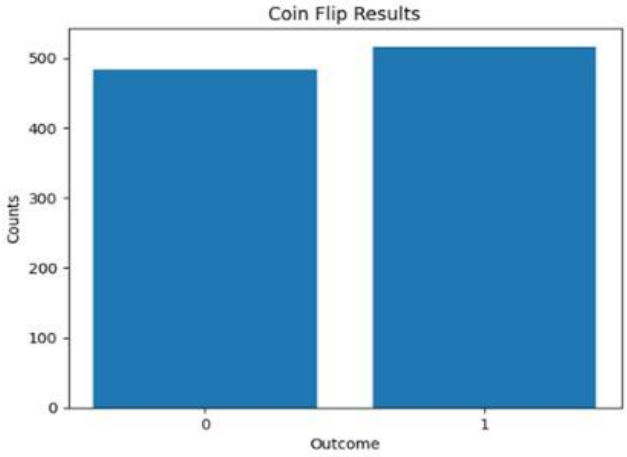
Tarrataca's quantum production system is a powerful algorithm for solving combinatorial optimization problems. It has the potential to solve problems that are intractable for classical computers.

Input:

1. Install Qiskit and the required dependencies.
2. Import the necessary Qiskit libraries

3. Define the initial state and the target state.
4. Create a quantum circuit with the desired number of qubits and classical bits.
5. Apply gate operations to transform the initial state to the target state.
6. Measure the qubits and store the measurement results in classical bits.
7. Choose a backend simulator Execute the quantum circuit on the chosen backend with a specified number of shots.
8. Retrieve and analyze the measurement results. 9. Display the results by printing the counts and visualizing them using a histogram.

Output:



Conclusion:

Thus we have implemented Tarrataca's quantum production system with the 3-puzzle problem

Outcome:

Upon completion of this experiment, students will be able to:

Experiment level outcome (ELO3): Write a Python Program using Tarrataca's quantum production system with the 3-puzzle problem.

Questions:

1. How does Tarrataca's quantum production system encode the problem state as a quantum state?

ANS: Tarrataca's quantum production system encodes the problem state as a quantum state by mapping the problem's information onto a quantum superposition, where each possible solution is represented as a probability amplitude. This encoding allows quantum algorithms to simultaneously explore multiple solutions, potentially providing a speedup in problem-solving.

2. How does Tarrataca's quantum production system construct a quantum circuit that represents the problem space?

ANS: Tarrataca's quantum production system constructs a quantum circuit that represents the problem space by creating a sequence of quantum gates that manipulate qubits to encode the problem's constraints and variables. These gates perform operations that correspond to the problem's logic, allowing quantum algorithms to search for solutions within the quantum circuit's superposition.

3. What are some of the quantum algorithms that can be used to search for the optimal solution to the 3-puzzle problem using Tarrataca's quantum production system?

ANS: Some quantum algorithms that can be used to search for the optimal solution to the 3-puzzle problem using Tarrataca's quantum production system include Grover's algorithm and quantum annealing techniques, which aim to find the most efficient configuration of puzzle pieces within the quantum state, potentially leading to the optimal solution.

4. How can we measure the qubits at the end of the quantum circuit to obtain the optimal solution to the 3-puzzle problem?

ANS: To obtain the optimal solution to the 3-puzzle problem, you can measure the qubits at the end of the quantum circuit and decode the resulting bitstring. This bitstring represents the state of the puzzle pieces, providing a solution that optimally arranges them to solve the puzzle.

5. What are some of the advantages and disadvantages of using Tarrataca's quantum production system to solve the 3-puzzle problem?

ANS: Advantages of Tarrataca's quantum production system for the 3-puzzle problem:

-Potential for quantum speedup in problem-solving.- Efficient representation of problem space using quantum circuits.

Disadvantages:

- Susceptible to noise and decoherence.- Limited scalability for complex problems.- Quantum hardware constraints.

Experiment No. 4

Aim: Write a Python program to Implement Quantum Teleportation algorithm.

Objective:

To Study:

1. Creation of a quantum circuit with three qubits and three classical bits.
2. Preparation of the initial state by applying gates to create entanglement
3. Implementation of teleportation protocol by applying gates and measurements

Theory:

Introduction:

Quantum teleportation is a fundamental concept in quantum information theory that allows for the transfer of quantum states from one location to another using two entangled particles and classical communication. This algorithm plays a crucial role in quantum computing and quantum communication protocols. In this theoretical document, we will outline the steps to implement the quantum teleportation algorithm in Python

Quantum Teleportation Overview:

Quantum teleportation consists of three main participants: Alice, Bob, and Charlie. Alice wants to send an arbitrary quantum state (qubit) to Bob. To achieve this, they need to share an entangled pair of qubits and perform a series of quantum operations and classical communication. The high-level steps are as follows:

1. Create an entangled pair:

Alice and Bob need to create an entangled pair of qubits, typically using a Bell state, such as the Bell state $|\Phi+\rangle = (|00\rangle + |11\rangle) / \sqrt{2}$. This pair is shared between Alice and Bob.

2. Prepare the quantum state to be teleported:

Alice has the quantum state she wants to teleport, denoted as $|\psi>$. She combines her state with one of the qubits from the entangled pair using a controlled-Not (CNOT) gate and a Hadamard gate.

3. Perform measurements:

Alice performs measurements on her two qubits (the one from the entangled pair and the one representing |ψ>) in the Bell basis, which consists of the states |Φ+> ,Φ-> , |Ψ+> , and |Ψ-> .

4. Transmit measurement results classically:

Alice communicates the results of her measurements to Bob using classical communication. These results will help Bob transform his qubit to recreate the original quantum state |ψ⟩ .

5.Bob's operations:

Based on the classical information received from Alice, Bob applies a series of quantum gates to his qubit to transform it into the desired quantum state |ψ⟩ .

6.Python Implementation:

Now, let's discuss how to implement this algorithm in Python. We'll use a quantum computing library like Qiskit, which provides the necessary tools for quantum programming.

Import the required libraries: from qiskit import QuantumCircuit, Aer,

transpile, assemble, execute from qiskit.visualization import

plot_bloch_multivector

7.Create a quantum circuit:

Initialize a quantum circuit with three qubits, one for Alice, one for Bob, and one for the entangled pair.

8.Apply gates and measurements:

Implement the steps outlined above using quantum gates and measurements. This involves applying CNOT and Hadamard gates, measuring Alice's qubits in the Bell basis, and performing Bob's operations based on the measurement results.

9.Simulate and visualize results:

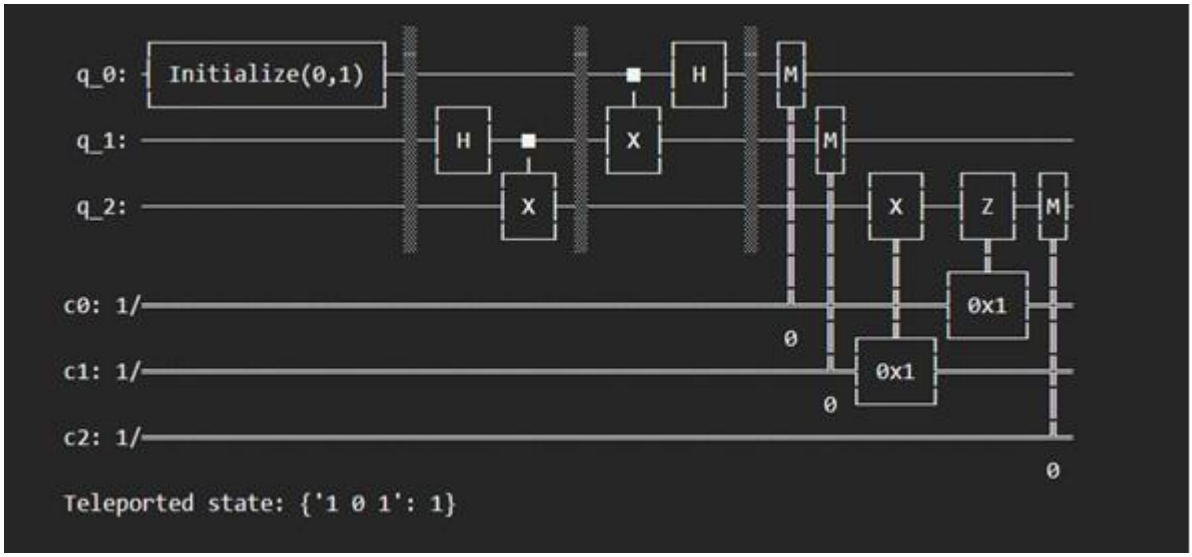
Use Qiskit's simulator to simulate the quantum teleportation process and visualize the final state of Bob's qubit.

10. Execute the circuit:

Compile and run the quantum circuit on a quantum computer or simulator.

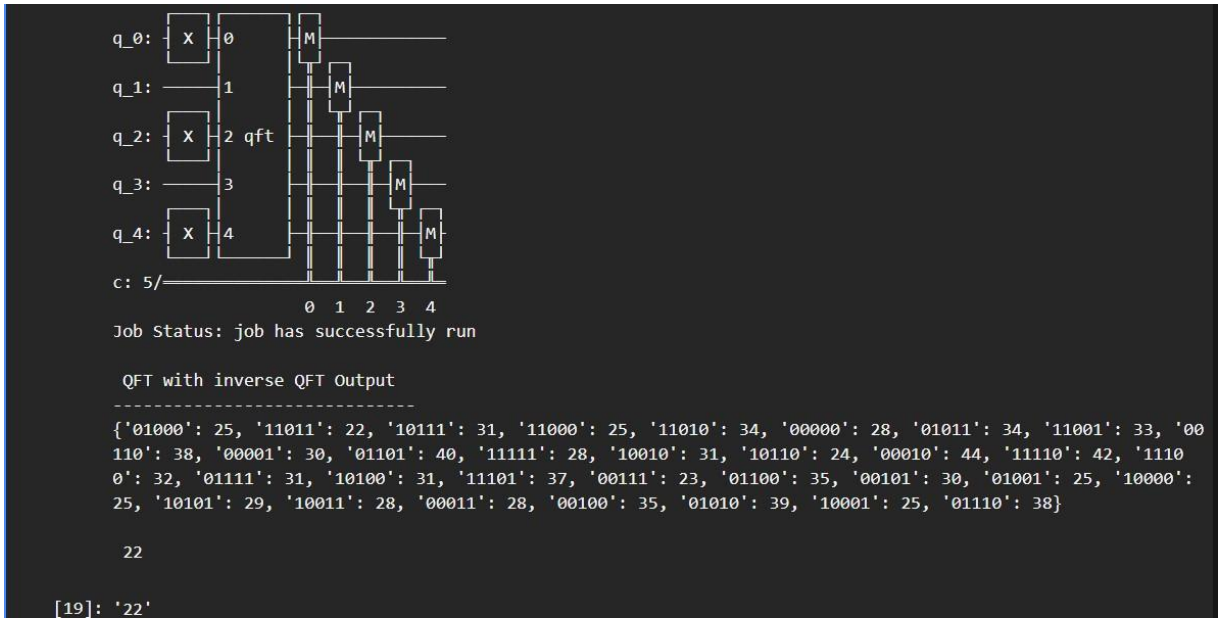
Input:

1. Import the necessary libraries (Quantum Circuit, Aer, execute, plot_bloch_multivector, and plot_histogram).
2. Create a quantum circuit with three qubits and three classical bits.
3. Prepare the initial state by applying gates to create entanglement.
4. Perform the teleportation protocol by applying gates and measurements **Output:**




```
Job Status: job has successfully run

 QFT Output
-------------
{'00001': 36, '01101': 20, '00110': 28, '11011': 24, '11000': 34, '10111': 36, '11010': 26, '00100': 29, '00
011': 37, '11001': 39, '10011': 25, '01110': 28, '11100': 31, '11110': 26, '10101': 35, '10000': 34, '0001
0': 26, '11101': 36, '10100': 31, '01000': 41, '10110': 25, '10010': 32, '11111': 25, '01111': 36, '10001':
25, '01010': 31, '01011': 27, '00000': 36, '00111': 37, '01100': 36, '00101': 37, '01001': 31}
```


```
Job Status: job has successfully run

 QFT Output
-------------
{'00010': 23, '01100': 29, '01000': 26, '11010': 32, '00100': 25, '00011': 24, '11011': 26, '11100': 32, '11
110': 38, '00110': 31, '00001': 31, '01101': 19, '00101': 38, '01001': 30, '10101': 38, '10000': 36, '1011
0': 29, '11111': 32, '10010': 32, '01111': 29, '01010': 29, '10001': 34, '01110': 32, '11001': 35, '10011':
34, '10100': 39, '11101': 38, '00111': 32, '01011': 22, '00000': 45, '11000': 26, '10111': 34}

 23
```



Conclusion:

Quantum teleportation is a fascinating concept that demonstrates the principles of quantum entanglement and the transfer of quantum information. Implementing the algorithm in Python using a quantum computing library like Qiskit allows for practical experimentation and exploration of quantum teleportation. The key is to follow the steps outlined in the algorithm while taking advantage of the powerful tools provided by the library to work with quantum circuits and simulate quantum processes.

Outcome:

Upon completion of this experiment, students will be able to:

Experiment level outcome (ELO4): Write a Python program to implement Quantum Teleportation algorithm in Python

Questions:

1. What are the fundamental principles behind quantum teleportation, and why is it a crucial concept in quantum information theory?

ANS: Quantum teleportation is based on the principles of entanglement and quantum measurement. It's crucial in quantum information theory because it enables the instantaneous transfer of quantum information between distant qubits, without the physical transfer of particles. This is vital for secure quantum communication and quantum computing**.**

2. In the context of quantum teleportation, what is the significance of the Bell state, and how does it play a pivotal role in the algorithm's implementation?

ANS: The Bell state is significant in quantum teleportation as it serves as the entangled resource between two qubits, allowing for the transfer of quantum information. It plays a pivotal role by enabling the measurement of one qubit in the Bell state to extract and reconstruct the state of the other qubit, leading to the successful teleportation of quantum information.

3. Can you explain the role of classical communication in quantum teleportation and why it is essential for the algorithm's success?

ANS: Classical communication is essential in quantum teleportation as it conveys the measurement outcomes to the receiving end. It enables the recipient to perform operations based on these outcomes to recreate the teleported quantum state faithfully. Without classical communication, the teleportation process would be incomplete, making it a crucial element for the algorithm's success.

Experiment No. 5

Aim:

The Randomized Benchmarking Protocol

Objective:

To Study:

1. Defining a benchmarking sequence of random Clifford gates & Creation of benchmarking circuits by applying the sequence to qubits
2. Execution of the circuits on a quantum simulator or device.
3. Analysis of the measured data to calculate fidelity and error rates

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 errorcorrecting 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.

Input:

1. Import the necessary libraries (qiskit, numpy, matplotlib, etc.).
2. Define a benchmarking sequence of random Clifford gates.
3. Create benchmarking circuits by applying the sequence to qubits.
4. Execute the circuits on a quantum simulator or device.
5. Analyze the measurement data to calculate fidelity and error rates.
6. Visualize the results by plotting the fidelity decay curve.

Output:

Result: [0.6116681128758453, 0.7011809859732847,0.7120854233100897,0.6813325854678913]

Conclusion:

Thus we have implemented Randomized Benchmarking Protocol **Outcome:**

Upon completion of this experiment, students will be able to:
Experiment level outcome (ELO5): Implement Randomized Benchmarking Protocol

Questions:

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

ANS:

Characteristic	Randomized Benchmarking	Other Techniques in Quantum Error Characterization
Scope	Provides an aggregate error metric for a set of gates.	Characterizes individual errors and noise sources.
Error Metric	Yields an average error rate of quantum gates.	Identifies specific error channels, probabilities, and error matrices.
Simplicity	Offers a relatively simple and efficient method.	Often involves more complex, resource-intensive procedures.
Robustness	Resilient to systematic errors and fluctuations.	May be sensitive to systematic errors and noise variations.
Scalability	Scales well for assessing the overall error performance.	May require extensive resources for comprehensive characterization.

2. What Mathematical Models are used to Analyze RB Data and Extract Error Rates?

ANS: The mathematical models commonly used to analyze RB data and extract error rates include:

1. Exponential Decay Model: Describes the fidelity decay as an exponential function, with the error rate obtained from the decay constant.

2. Clifford Model: Models RB data based on the properties of Clifford gates, offering insights into the error rates of individual gates.

3. Polynomial Fitting: Employs polynomial functions to approximate the relationship between RB sequence length and gate fidelity.

4. Bayesian Inference: Utilizes Bayesian analysis to estimate error rates and confidence intervals from RB data.

5. Chi-Square Fitting: Fits experimental data to a chi-square distribution to estimate error rates.

These models are essential for quantifying and understanding the error rates of quantum gates in RB experiments.

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

ANS: Certainly, here's a concise list of steps to apply Randomized Benchmarking (RB) to specific quantum hardware or softwar

1. Design Sequences: Create RB sequences tailored to the quantum system.

2. Collect Data: Implement sequences, run experiments, and collect data.

3. Analyze Errors: Use mathematical models to analyze error rates.

4. Optimize: Improve the system based on error rate insights.

5. Validate: Verify improvements with additional RB runs.

6. Benchmark & Certify: Use RB for system benchmarking and certification.

Experiment No. 6

Aim:

Write a Python program to implementing a 5 qubit Quantum Fourier Transform

Objective:

To Study:

1. Creation of a quantum circuit with 5 qubits.
2. Application of gates to each qubit to implement the QFT.
3. Measurement of the qubits to obtain the result

Theory:

Introduction:

Quantum computing is a rapidly advancing field that promises to revolutionize computation by leveraging the principles of quantum mechanics to perform complex calculations efficiently. The Quantum Fourier Transform (QFT) is a pivotal quantum algorithm that finds applications in various quantum algorithms, such as Shor's algorithm for factoring large numbers and quantum phase estimation. This paper delves into the theory and techniques for implementing a 5-qubit Quantum Fourier Transform, an essential step toward building more advanced quantum algorithms.

Quantum Fourier Transform (QFT) Overview

The Quantum Fourier Transform is the quantum analogue of the classical discrete Fourier transform (DFT). It transforms an input quantum state |x> into an output state |y>, where each basis state |y> corresponds to a frequency component of the input state |x> . The mathematical representation of the QFT is as follows:

Here, N represents the number of basis states (in our case, 2^5 = 32), and |k> denotes the basis states in the computational basis.

Implementing the 5-Qubit QFT

To implement the 5-qubit Quantum Fourier Transform, we will discuss the fundamental steps involved:

1. State Preparation: Begin by initializing an initial quantum state |x>, which is typically a superposition of basis states.
2. Hadamard Transform: Apply a Hadamard gate (H) to each qubit in the superposition, creating entanglement and preparing the input state for the Fourier transform
3. Controlled Phase Shifts: Implement controlled-phase gates (CROT gates) to introduce phase shifts, representing the frequency components of the QFT
4. Bit Reversal: Perform a bit reversal operation to reorder the qubits in the correct sequence, aligning them with the output state.

Step-by-Step Explanation

State Preparation:

The process begins with initializing the input state |x>, which is often a superposition of basis states. For example, to prepare the state $|01101\rangle$, apply X (bit-flip) gates to the corresponding qubits.

Hadamard Transform:

Apply a Hadamard gate (H) to each qubit in the prepared state. The Hadamard gate creates a superposition of basis states, which is essential for the subsequent steps of the algorithm.

Controlled Phase Shifts:

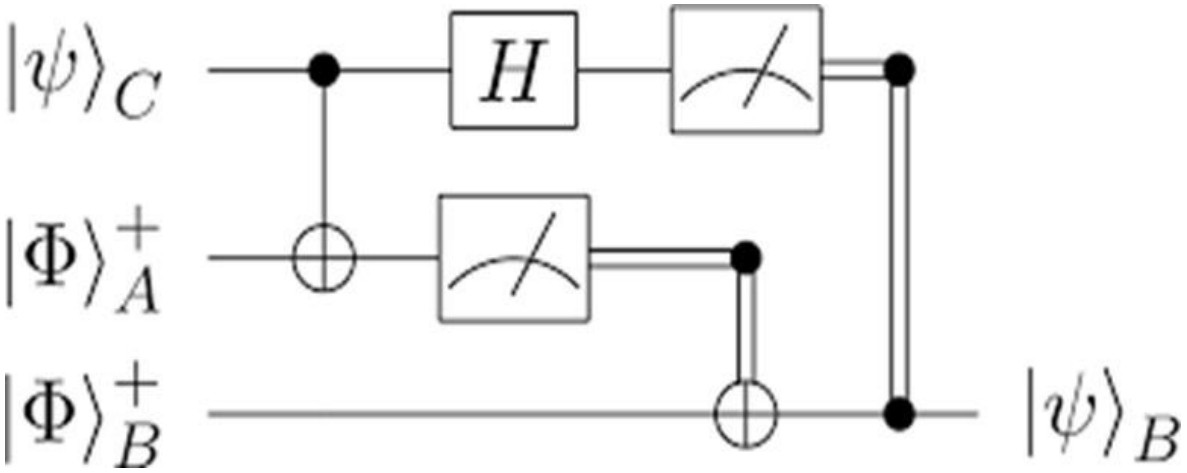
The core of the QFT involves applying controlled-phase gates (CROT gates). Each CROT gate corresponds to a specific frequency component in the QFT. The controlled-phase gate CROT_k multiplies the phase of $|k\rangle$ by $2\pi/N$ when the control qubit is in state $|1\rangle$. Apply these gates for k = 1 to 5 (as we are implementing a 5-qubit QFT).

Bit Reversal:

To ensure that the qubits are ordered correctly in the final state, perform a bit reversal operation on the qubits. This step is necessary to align the qubits with the output state.

Quantum Circuit Representation

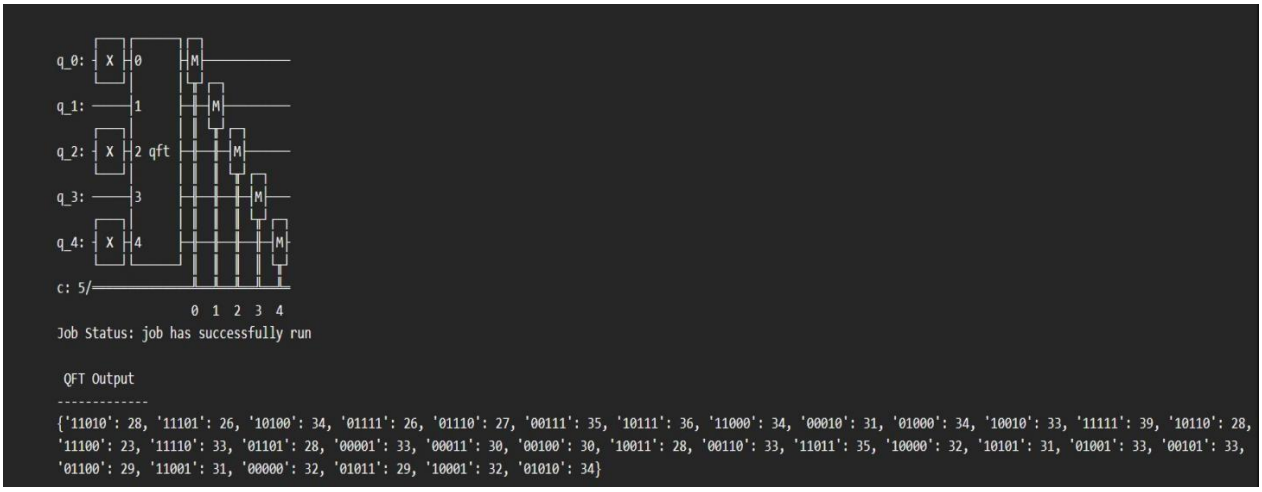
A quantum circuit diagram representing the 5-qubit QFT is provided below for clarity:

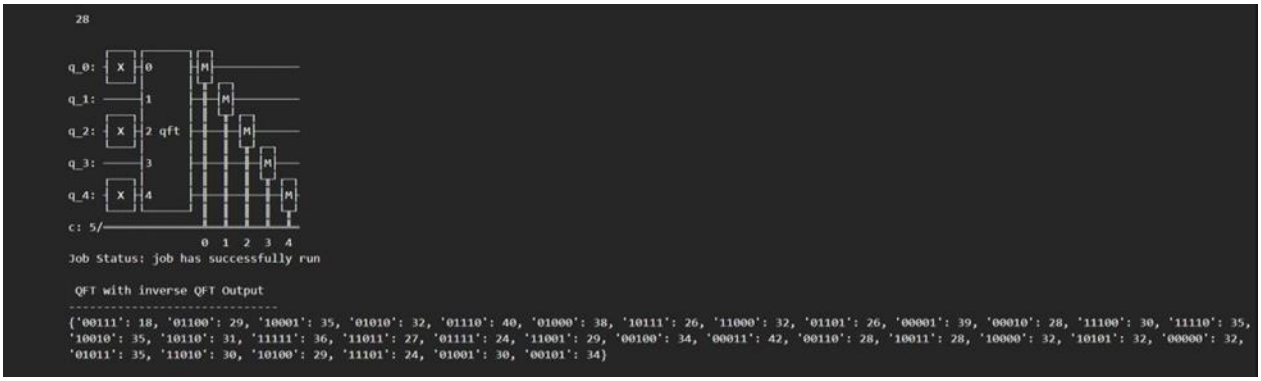


Input:

1. Import the necessary libraries: Import the required libraries for quantum circuit creation and execution
2. Create a quantum circuit with 5 qubits.
3. Apply Hadamard gates to each qubit.
4. Apply controlled phase shift gates to implement the QFT.
5. Measure the qubits to obtain the final result

Output:





Conclusion:

Implementing a 5-qubit Quantum Fourier Transform is a crucial step in unlocking the potential of quantum computing. This algorithm serves as a foundational component for many quantum algorithms with applications in cryptography, optimization, and quantum simulations. By comprehending the principles and techniques behind the QFT, researchers and developers can contribute to the advancement of quantum computing and explore its vast capabilities. The ability to efficiently perform the Quantum Fourier Transform lays the groundwork for solving complex problems that were previously infeasible with classical computers. As quantum computing continues to evolve, the QFT will remain a cornerstone of quantum algorithms and computational techniques.

Outcome:

Upon completion of this experiment, students will be able to:

Experiment level outcome (ELO6): Write a Python program to implement 5 qubit Quantum Fourier Transform .

Questions:

1. Explain the significance of the Quantum Fourier Transform (QFT) in quantum computing. How does it differ from its classical counterpart, the Discrete Fourier Transform (DFT)?

ANS:

- Significance of QFT:

1. Speedup: Exponentially faster than classical DFT.
2. Quantum Speedup: Enables tasks infeasible for classical computers.
3. Essential in many quantum algorithms.
4. Leverages quantum superposition.

- Differences from DFT:

1. Exponential computational speedup.
2. High parallelism through superposition.
3. Manipulates quantum states.
4. Utilizes quantum entanglement in algorithms like Shor's.

2. Describe the step-by-step process of implementing a 5-qubit Quantum Fourier Transform (QFT). Discuss the quantum gates and operations involved in each step and explain their significance. What challenges or potential sources of error might arise during the implementation of a QFT on a quantum computer?

ANS: Steps for implementing a 5-qubit Quantum Fourier Transform (QFT) along with the significance of each step and potential errors:

1. State Preparation using Hadamard gates (H).

2. Apply Phase Shifts with controlled-phase gates (CROT).

3. Swap Operations using swap gates (SWAP).

4. Measurement to obtain the QFT result.

- Significance of Operations:

1. Hadamard Gates (H) create superposition for simultaneous processing.
2. Controlled-Phase Gates (CROT) introduce critical phase shifts.
3. Swap Gates (SWAP) arrange qubits correctly for the desired output.

- Challenges and Potential Errors:

1. Gate Errors: Imperfections impacting accuracy.
2. Decoherence: Environmental factors causing qubit loss.
3. Crosstalk: Unintended qubit interactions.
4. State Preparation Challenges, especially with more qubits.
5. Need for Fault-Tolerance techniques to scale QFT without errors.

3. Quantum computers are known to be exponentially faster than classical computers for certain problems. How does the Quantum Fourier Transform (QFT) exemplify this advantage, and what are the implications for cryptography and factorization problems?

ANS:

The Quantum Fourier Transform (QFT) exemplifies the advantage of quantum computing by being exponentially faster than its classical counterpart, enabling quantum computers to perform complex operations and factorization tasks significantly more efficiently.

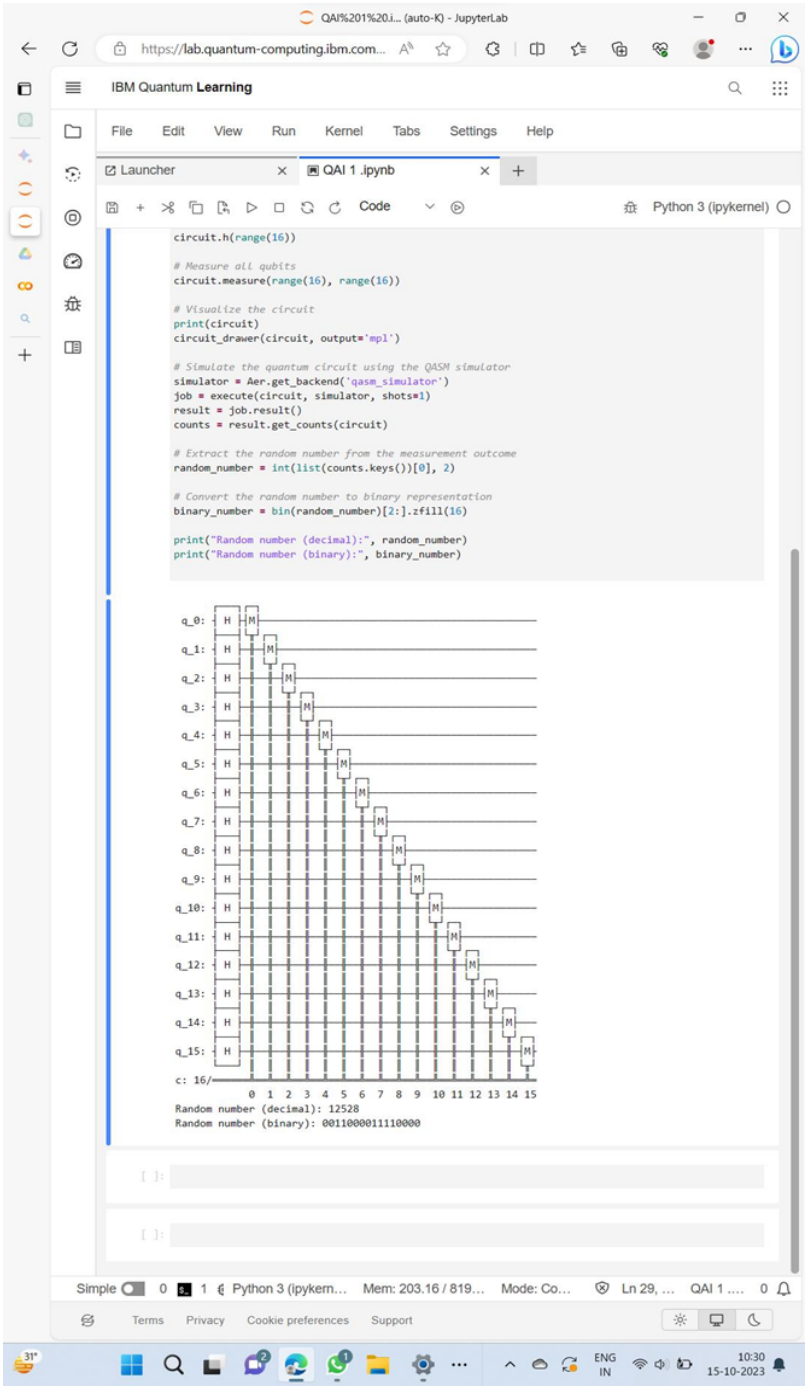
The significance of QFT and its implications for cryptography and factorization problems can be summarized as follows:

1. QFT's Speedup: Quantum Fourier Transform (QFT) is exponentially faster than its classical counterpart (DFT), enabling quantum computers to perform complex operations more efficiently.
2. Cryptography Threat: QFT's speedup, along with quantum algorithms like Shor's, poses a significant threat to classical public-key cryptography (e.g., RSA) by making factorization of large numbers exponentially faster.
3. Factorization Advantage: QFT's efficiency in factorization problems can compromise the security of systems relying on the difficulty of factoring large numbers.

QAI OUTPUT 1

```
# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
# qiskit-ibmq-provider has been deprecated.
# Please see the Migration Guides in https://ibm.biz/provider_migration_guide for more detail.
from qiskit_ibm_runtime import QiskitRuntimeService, Sampler, Estimator, Session, Options
# Loading your IBM Quantum account(s)
service = QiskitRuntimeService(channel="ibm_quantum")
# Invoke a primitive. For more details see
https://qiskit.org/documentation/partners/qiskit_ibm_runtime/tutorials.html
# result = Sampler("ibmq_qasm_simulator").run(circuits).result()


#16qubit random number genertor
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import circuit_drawer
# Create a quantum circuit with 16 qubits
circuit = QuantumCircuit(16, 16)
# Apply Hadamard gates to put all qubits in superposition
circuit.h(range(16))
# Measure all qubits
circuit.measure(range(16), range(16))
# Visualize the circuit
print(circuit)
circuit_drawer(circuit, output='mpl')
# Simulate the quantum circuit using the QASM simulator
simulator = Aer.get_backend('qasm_simulator')
job = execute(circuit, simulator, shots=1)
result = job.result()
counts = result.get_counts(circuit)
# Extract the random number from the measurement outcome
random_number = int(list(counts.keys())[0], 2)
```

```
# Convert the random number to binary representation
binary_number = bin(random_number)[2:].zfill(16)
print("Random number (decimal):", random_number)
print("Random number (binary):", binary_number)
```



QAI OUTPUT 2

```
!pip install qiskit-ignis

from qiskit import QuantumCircuit, assemble, Aer, transpile

from qiskit.visualization import plot_histogram

from qiskit.ignis.mitigation.measurement import CompleteMeasFitter, complete_meas_cal,
tensored_meas_cal

# Define the quantum circuit

qc = QuantumCircuit(3, 3)

# Apply gates and operations to the circuit

qc.h(0)

qc.cx(0, 1)

qc.cx(0, 2)

qc.measure([0, 1, 2], [0, 1, 2])

# Transpile the circuit

backend = Aer.get_backend('qasm_simulator')

transpiled_qc = transpile(qc, backend)

# Simulate the noisy circuit

qobj = assemble(transpiled_qc, shots=1000)

job = backend.run(qobj)

result = job.result()

counts = result.get_counts()

# Perform error mitigation

cal_circuits, state_labels = complete_meas_cal(qubit_list=[0, 1, 2])

cal_job = backend.run(assemble(cal_circuits, backend=backend))

cal_results = cal_job.result()

meas_fitter = CompleteMeasFitter(cal_results, state_labels)

mitigated_counts = meas_fitter.filter.apply(counts)

# Print the original counts

print("Original counts:")

print(counts)
```

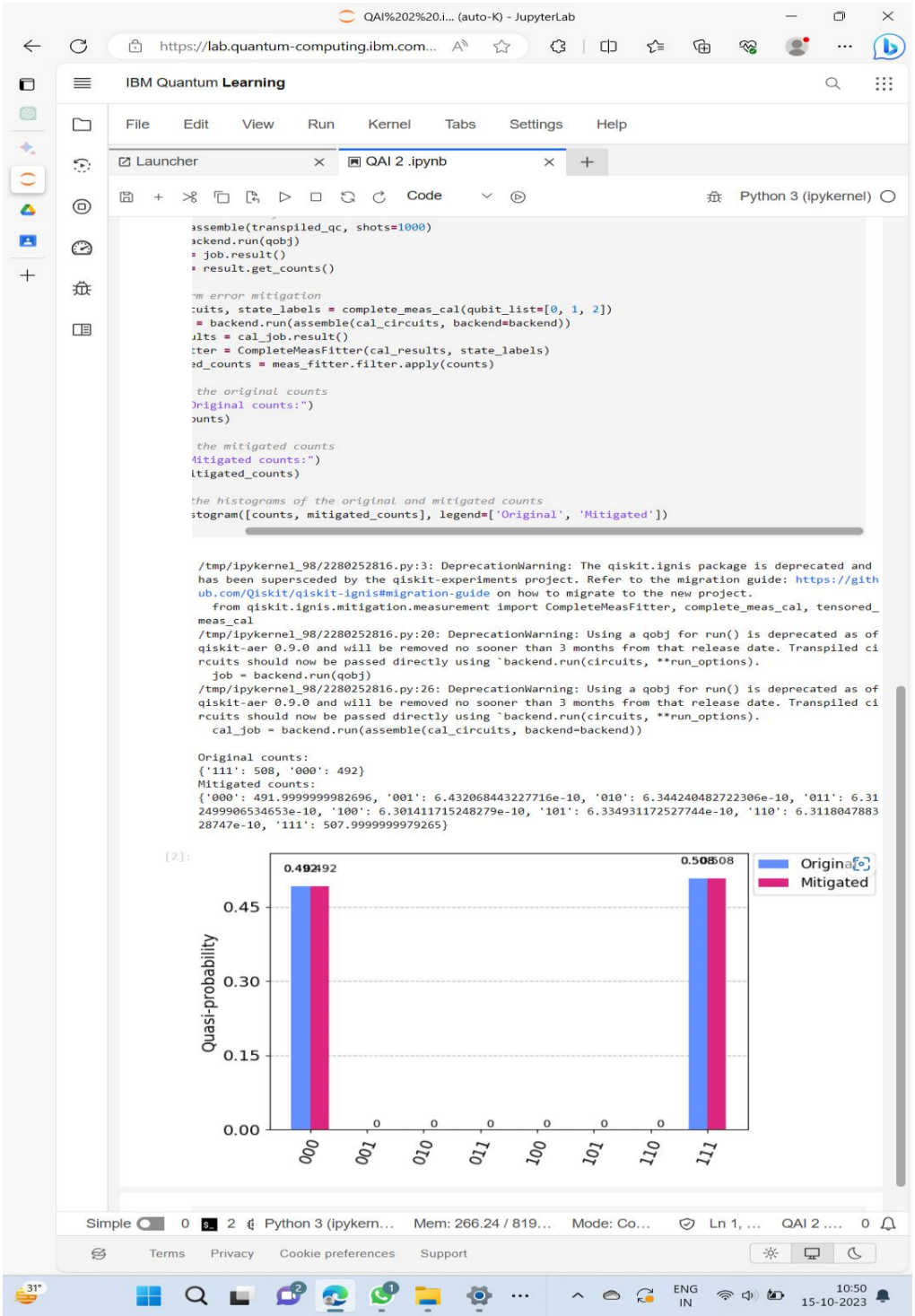
Print the mitigated counts

print("Mitigated counts:")

print(mitigated_counts)

Plot the histograms of the original and mitigated counts

plot_histogram([counts, mitigated_counts], legend=['Original', 'Mitigated'])



QAI OUTPUT 3

```
pip install qiskit –upgrade
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram
# Define the initial state
initial_state = [2, 5, 3, 1, 8, 6, 4, 7, None]
# Define the goal state
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, None]
# Define the quantum circuit
qc = QuantumCircuit(18, 18)
# Initialize the circuit with the initial state
for i, tile in enumerate(initial_state):
    if tile is not None:
        qc.x(i)
# Perform swaps to reach the goal state
for i, tile in enumerate(goal_state):
    if tile is not None:
        initial_index = initial_state.index(tile)
        target_index = goal_state.index(tile)
        qc.swap(i, initial_index + 9)
        qc.swap(i, target_index + 9)
# Measure the final state
qc.measure(range(9), range(9))
# Simulate the circuit
backend = Aer.get_backend('qasm_simulator')
job = execute(qc, backend, shots=1024)
result = job.result()
counts = result.get_counts(qc)
# Find the most probable state (solution)
max_count = max(counts.values())
solution = [state for state, count in counts.items() if count == max_count][0]
```

```
# Print the solution
print("Solution found!")
for i in range(0, 9, 3):
    print(solution[i:i + 3])
```

OUTPUT :

Solution found!
000
000
000

```
from qiskit import Aer, execute, QuantumCircuit
# Define the quantum circuit
qc = QuantumCircuit(1, 1)
qc.h(0)  # Apply Hadamard gate for superposition
qc.measure(0, 0)  # Measure qubit and store result in classical bit
# Use the Aer simulator
simulator = Aer.get_backend('qasm_simulator')
# Set the number of shots (measurements)
num_shots = 1000
# Execute the quantum circuit on the simulator with multiple shots
job = execute(qc, simulator, shots=num_shots)
# Get the result of the measurements
result = job.result()
counts = result.get_counts()
# Print the coin flip results
print("Coin Flip Results:")
for outcome, count in counts.items():
    print(f"{outcome}: {count} ({count/num_shots*100:.2f}%)")
```

OUTPUT:

Coin Flip Results:
0: 538 (53.80%)
1: 462 (46.20%)

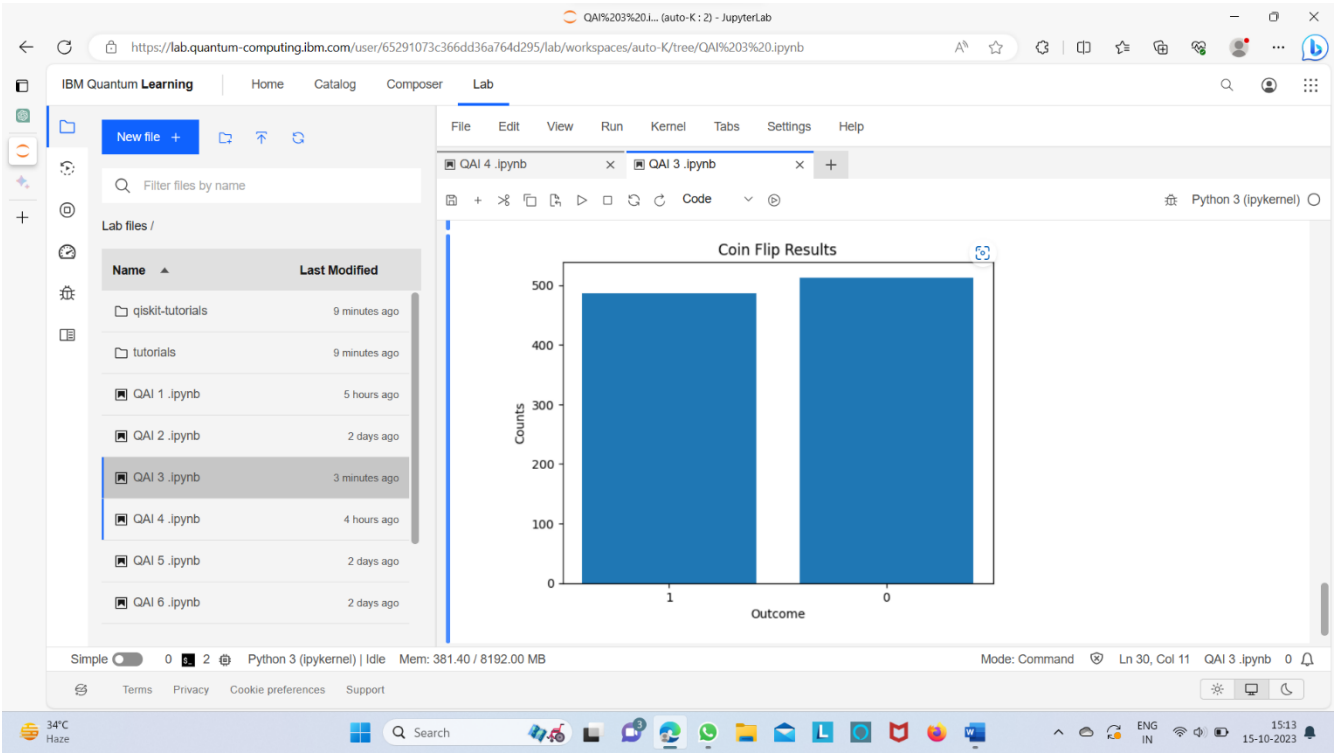
```
from qiskit import Aer, execute, QuantumCircuit
# Define the quantum circuit
qc = QuantumCircuit(1, 1)
qc.h(0)  # Apply Hadamard gate for superposition
qc.measure(0, 0)  # Measure qubit and store result in classical bit
# Use the Aer simulator
simulator = Aer.get_backend('qasm_simulator')
# Set the number of shots (measurements)
num_shots = 1000
# Execute the quantum circuit on the simulator with multiple shots
job = execute(qc, simulator, shots=num_shots)
# Get the result of the measurements
result = job.result()
counts = result.get_counts()
# Print the coin flip results
print("Coin Flip Results:")
for outcome, count in counts.items():
    print(f"{outcome}: {count} ({count/num_shots*100:.2f}%)")
```

OUTPUT:

Coin Flip Results:
1: 493 (49.30%)
0: 507 (50.70%)

```
import matplotlib.pyplot as plt
from qiskit import Aer, execute, QuantumCircuit
# Define the quantum circuit
qc = QuantumCircuit(1, 1)
qc.h(0)  # Apply Hadamard gate for superposition
qc.measure(0, 0)  # Measure qubit and store result in classical bit
# Use the Aer simulator
```

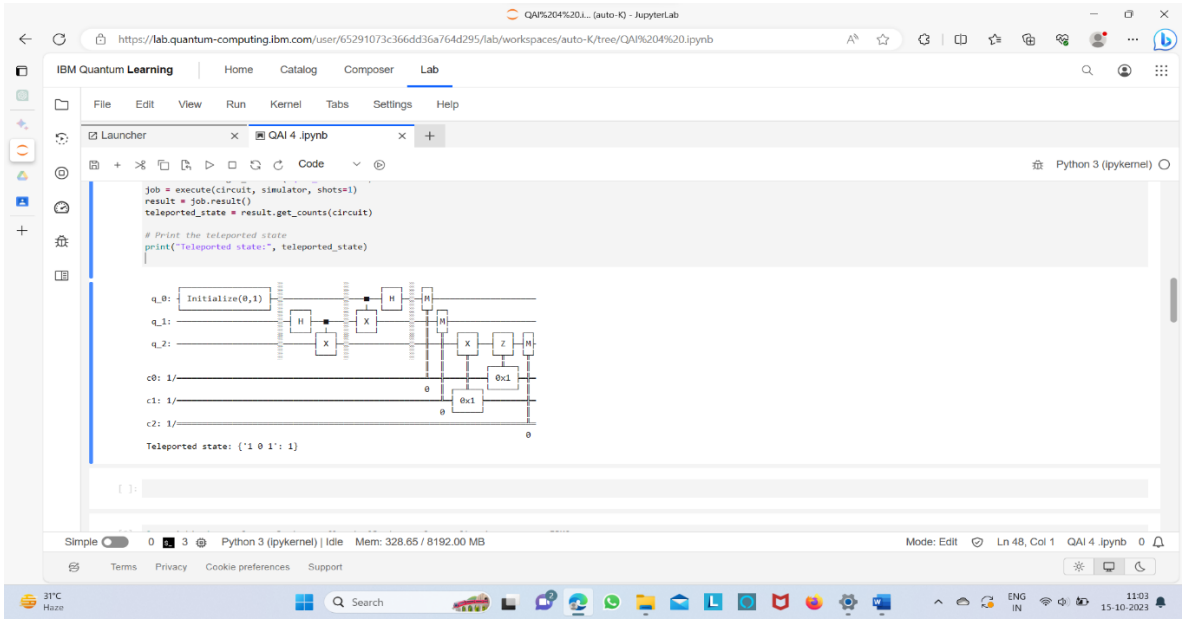
```
simulator = Aer.get_backend('qasm_simulator')
# Set the number of shots (measurements)
num_shots = 1000
# Execute the quantum circuit on the simulator with multiple shots
job = execute(qc, simulator, shots=num_shots)
# Get the result of the measurements
result = job.result()
counts = result.get_counts()
# Plot the coin flip results
outcomes = list(counts.keys())
counts_list = list(counts.values())
plt.bar(outcomes, counts_list)
plt.xlabel('Outcome')
plt.ylabel('Counts')
plt.title('Coin Flip Results')
plt.show()
```



QAI OUTPUT 4

```
# Importing standard Qiskit libraries
from qiskit import QuantumCircuit, transpile
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
# qiskit-ibmq-provider has been deprecated.
# Please see the Migration Guides in https://ibm.biz/provider_migration_guide for more detail.
from qiskit_ibm_runtime import QiskitRuntimeService, Sampler, Estimator, Session, Options
# Loading your IBM Quantum account(s)
service = QiskitRuntimeService(channel="ibm_quantum")
# Invoke a primitive. For more details see
https://qiskit.org/documentation/partners/qiskit_ibm_runtime/tutorials.html
# result = Sampler("ibmq_qasm_simulator").run(circuits).result()
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, execute, Aer
from qiskit.visualization import circuit_drawer
# Create the quantum circuit with 3 qubits and 3 classical bits
q = QuantumRegister(3, 'q')  # Quantum register
c0 = ClassicalRegister(1, 'c0')  # Classical register for Alice's qubit
c1 = ClassicalRegister(1, 'c1')  # Classical register for Bob's qubit
c2 = ClassicalRegister(1, 'c2')  # Classical register for the result
circuit = QuantumCircuit(q, c0, c1, c2)
# Prepare the initial state to be teleported
circuit.initialize([0, 1], q[0])  # Apply X gate to put in state |1>
circuit.barrier()
# Create an entanglement between Alice's and Bob's qubits
circuit.h(q[1])
circuit.cx(q[1], q[2])
circuit.barrier()# Teleportation process
circuit.cx(q[0], q[1])
circuit.h(q[0])
circuit.barrier()
# Measure Alice's qubits and send the measurement results to Bob
```

```
circuit.measure(q[0], c0[0])

circuit.measure(q[1], c1[0])

# Apply corrective operations on Bob's qubit based on the measurement results

circuit.x(q[2]).c_if(c1, 1)

circuit.z(q[2]).c_if(c0, 1)

# Measure the teleported qubit

circuit.measure(q[2], c2[0])

# Visualize the circuit

print(circuit)

circuit_drawer(circuit, output='mpl')

# Simulate the circuit using the QASM simulator

simulator = Aer.get_backend('qasm_simulator')

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

result = job.result()

teleported_state = result.get_counts(circuit)

# Print the teleported state

print("Teleported state:", teleported_state)
```




```
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit, execute, IBMQ

from qiskit.tools.monitor import job_monitor

from qiskit.circuit.library import QFT

import numpy as np

# Load your IBM Quantum account using your API token
```

IBMQ.save_account('bd38c78f4f634e5195dc1ead3874476cd0c044b321bcd0dce58d2c6fabca26bb095e9c46db
d1e3354cbaee5c341900b7d3cbfdadf773c7c71099c9c3ae001c31', overwrite=True) # Replace
'YOUR_API_TOKEN' with your actual API token

IBMQ.load_account()

Choose the provider and backend

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.append(QFT(num_qubits=5, approximation_degree=0, do_swaps=True, inverse=False,
insert_barriers=False), q)

circuit.measure(q, c)

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

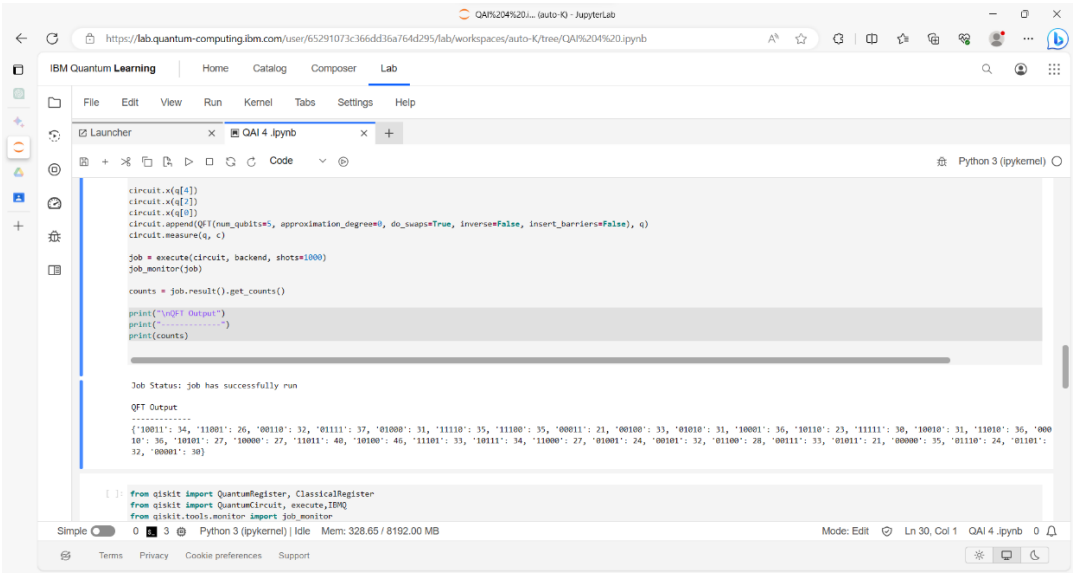
job_monitor(job)

counts = job.result().get_counts()

print("\nQFT Output")

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

print(counts)

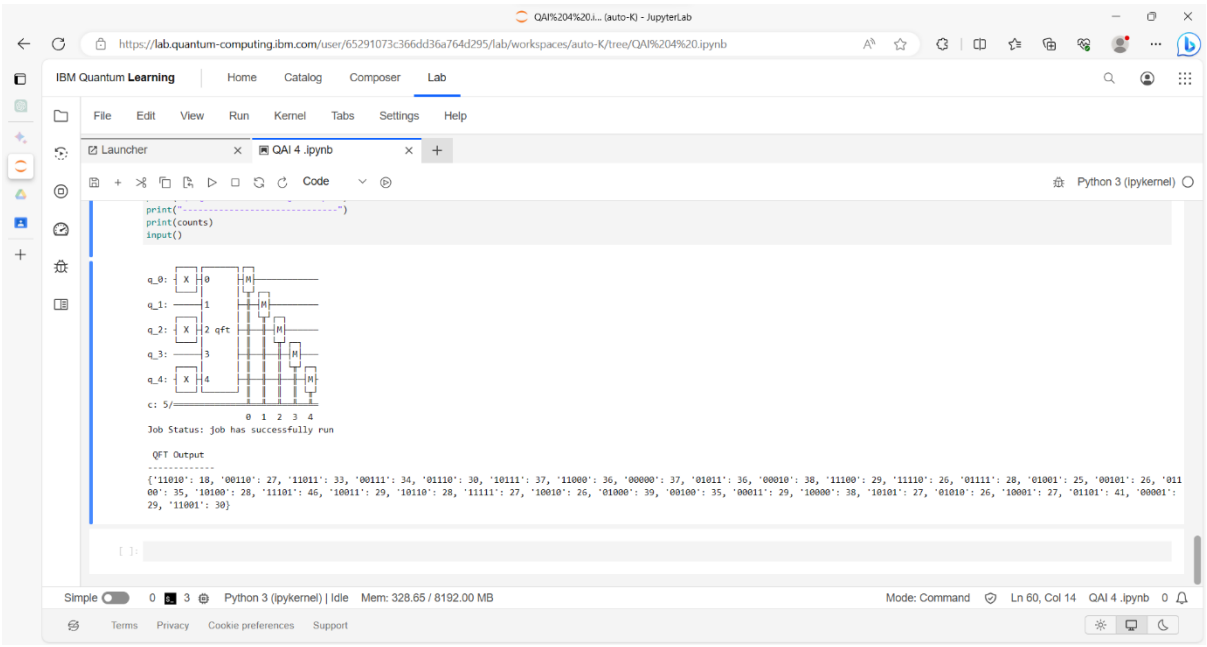


from qiskit import QuantumRegister, ClassicalRegister

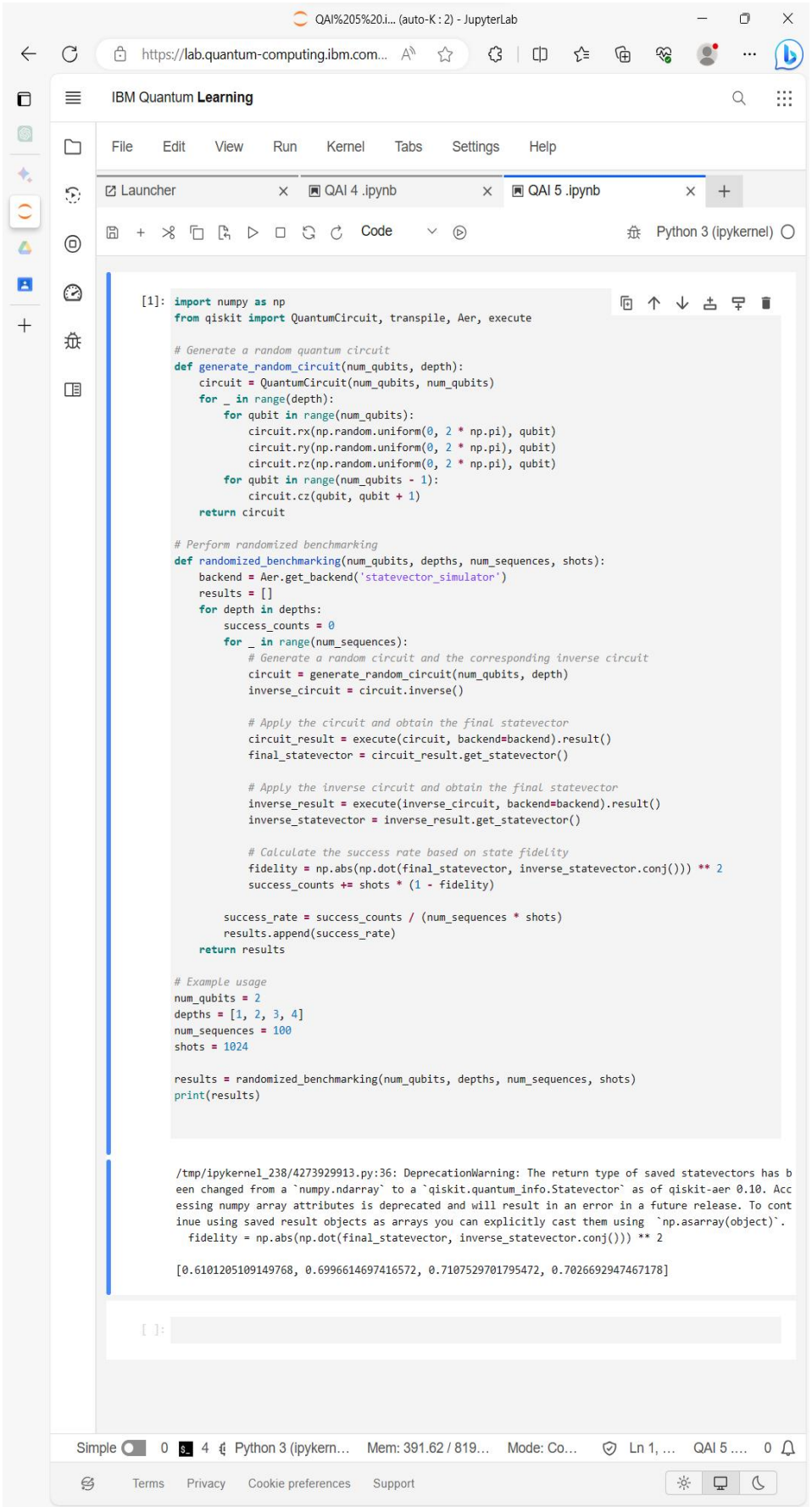
from qiskit import QuantumCircuit, execute,IBMQ

```
from qiskit.tools.monitor import job_monitor

from qiskit.circuit.library import QFT

import numpy as np

pi = np.pi

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.append(QFT(num_qubits=5, approximation_degree=0, do_swaps=True, inverse=False,
insert_barriers=False, name='qft'), q)

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()

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.append(QFT(num_qubits=5, approximation_degree=0, do_swaps=True, inverse=False,
insert_barriers=False, name='qft'), q)

circuit.measure(q,c)
```

```
circuit.draw(output='mpl',filename='qft2.png')
print(circuit)
job = execute(circuit, backend, shots=1000)
job_monitor(job)
counts = job.result().get_counts()
print("\n QFT with inverse QFT Output")
print("------------------------------")
print(counts)
input()
```



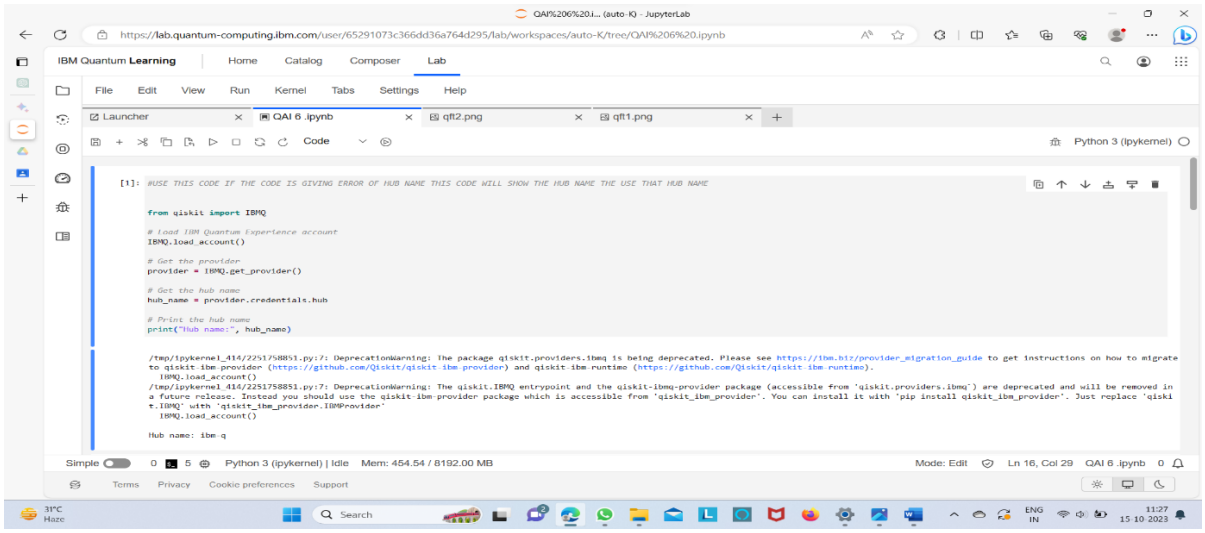
QAI OUTPUT 5


QAI%205%20.i... (auto-K : 2) - JupyterLab
https://lab.quantum-computing.ibm.com...
IBM Quantum Learning
File Edit View Run Kernel Tabs Settings Help
Launcher
QAI 4 .ipynb
QAI 5 .ipynb
Code
Python 3 (ipykernel)
[1]:
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)
/tmp/ipykernel_238/4273929913.py:36: DeprecationWarning: The return type of saved statevectors has b
een changed from a `numpy.ndarray` to a `qiskit.quantum_info.Statevector` as of qiskit-aer 0.10. Acc
essing numpy array attributes is deprecated and will result in an error in a future release. To cont
inue using saved result objects as arrays you can explicitly cast them using `np.asarray(object)`.
fidelity = np.abs(np.dot(final_statevector, inverse_statevector.conj())) ** 2
[0.6101205109149768, 0.6996614697416572, 0.7107529701795472, 0.7026692947467178]
[ ]:
Simple 0 4 Python 3 (ipykern... Mem: 391.62 / 819... Mode: Co... Ln 1, ... QAI 5 0
Terms Privacy Cookie preferences Support

QAI OUTPUT 6

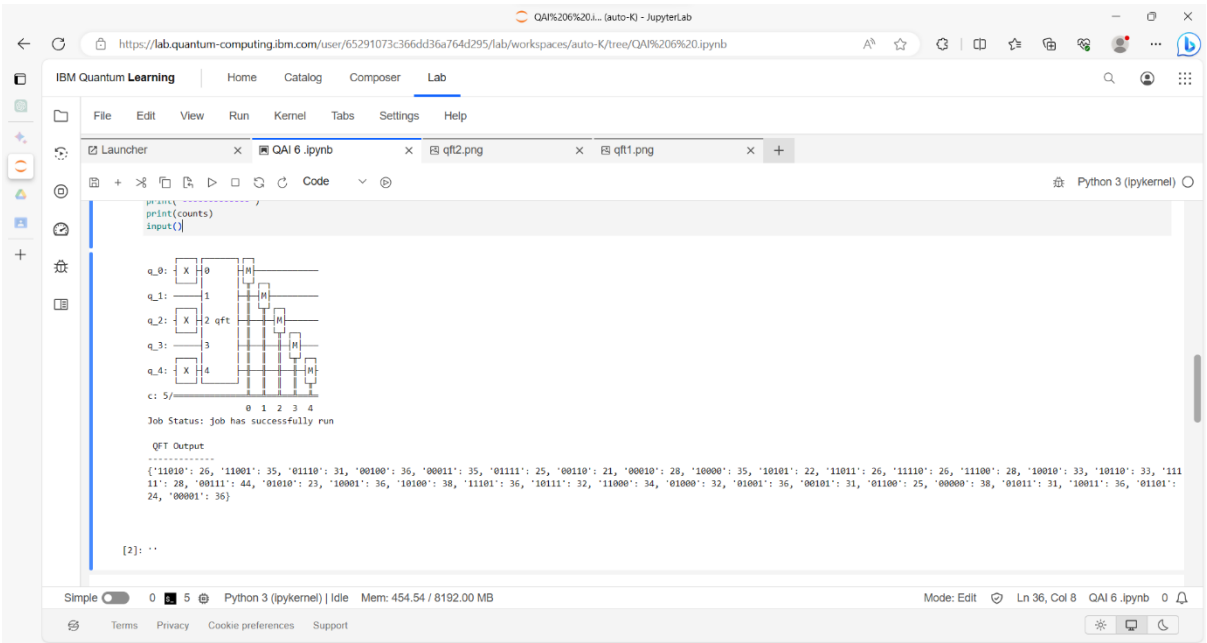
#USE THIS CODE IF THE CODE IS GIVING ERROR OF HUB NAME THIS CODE WILL SHOW THE HUB NAME THE USE THAT HUB NAME

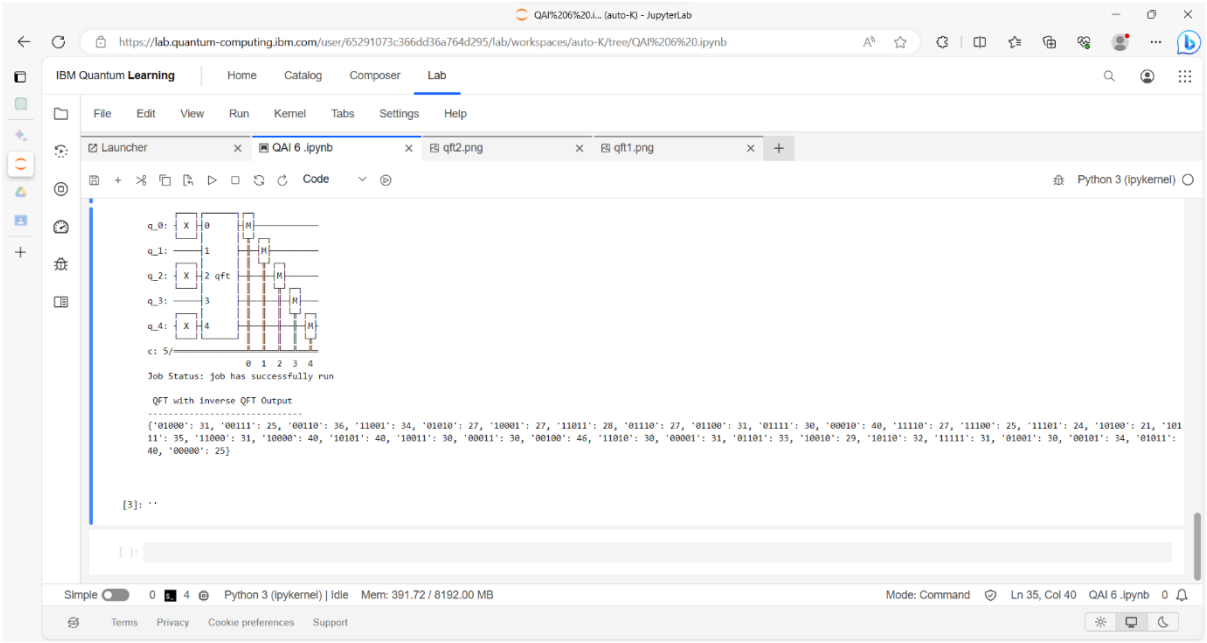
```
from qiskit import IBMQ

# Load IBM Quantum Experience account

IBMQ.load_account()

# Get the provider

provider = IBMQ.get_provider()

# Get the hub name

hub_name = provider.credentials.hub

# Print the hub name

print("Hub name:", hub_name)
```

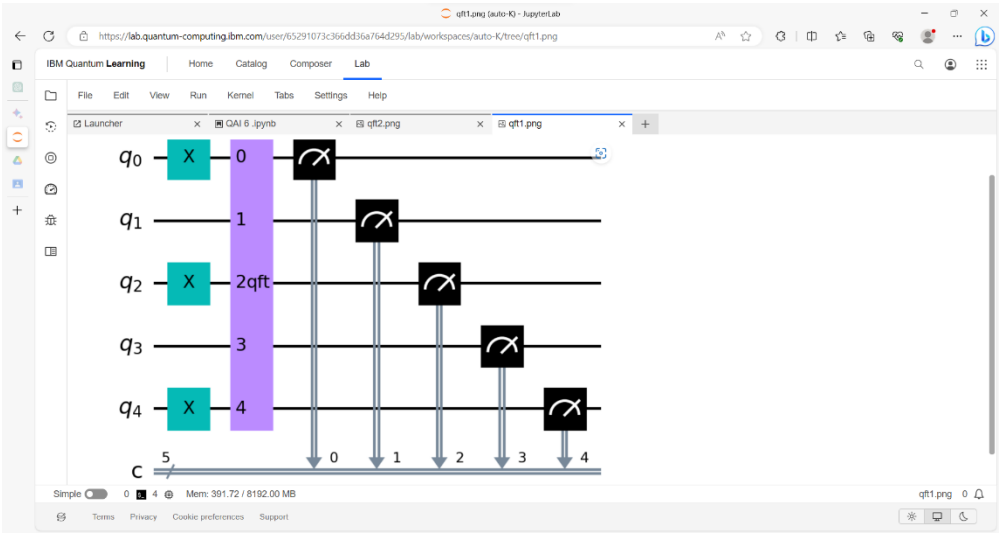


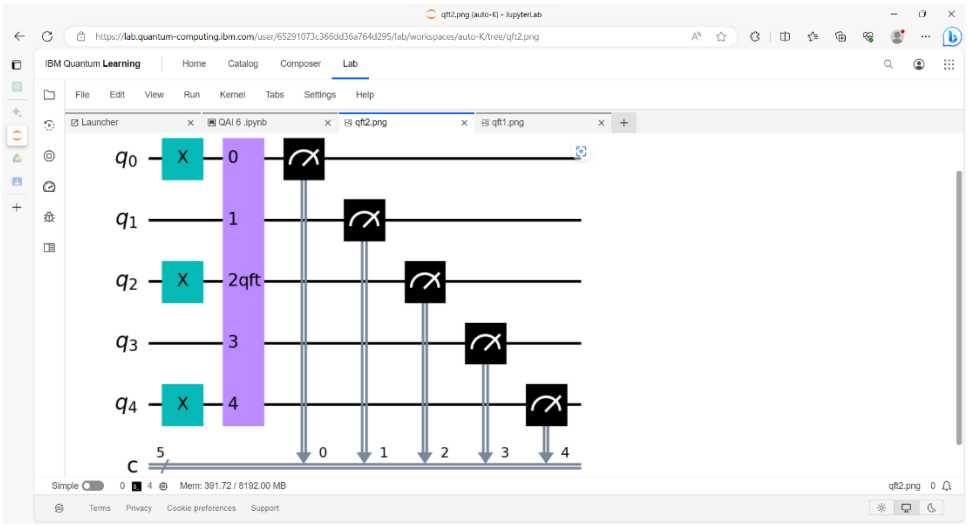

```
from qiskit import QuantumRegister, ClassicalRegister

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

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.append(QFT(num_qubits=5, approximation_degree=0, do_swaps=True, inverse=False,
insert_barriers=False, name='qft'), q)

circuit.measure(q,c)

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

print(circuit)

circuit.append(QFT(num_qubits=5, approximation_degree=0, do_swaps=True, inverse=True,
insert_barriers=False, name='qft'), q)

circuit.measure(q,c)

circuit.draw(output='mpl',filename='qft2.png')

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

job_monitor(job)

counts = job.result().get_counts()

print("\n QFT Output")

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

print("\n QFT with inverse QFT Output")

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

print(counts)

input()
```




QAI%206%20.i... (auto-K) - JupyterLab
https://lab.quantum-computing.ibm.com/user/65291073c366dd36a764d295/lab/workspaces/auto-K/tree/QAI%206%20.ipynb
IBM Quantum Learning
Home
Catalog
Composer
Lab
File
Edit
View
Run
Kernel
Tabs
Settings
Help
Launcher
QAI 6 .ipynb
qft2.png
qft1.png
Code
Python 3 (ipykernel)
q_0: X 0 M
q_1: 1 M
q_2: X 2 qft M
q_3: 3 M
q_4: X 4 M
c: 5/ 0 1 2 3 4
Job Status: job has successfully run
QFT with inverse QFT Output
{'01000': 31, '00111': 25, '00110': 36, '11001': 34, '01010': 27, '10001': 27, '11011': 28, '01110': 27, '01100': 31, '01111': 30, '00010': 40, '11110': 27, '11100': 25, '11101': 24, '10100': 21, '10111': 35, '11000': 31, '10000': 40, '10101': 40, '10011': 30, '00011': 30, '00100': 46, '11010': 30, '00001': 31, '01101': 33, '10010': 29, '10110': 32, '11111': 31, '01001': 30, '00101': 34, '01011': 40, '00000': 25}
[3]:
[ ]:
Simple
Python 3 (ipykernel) | Idle
Mem: 391.72 / 8192.00 MB
Mode: Command
Ln 35, Col 40
QAI 6 .ipynb
Terms
Privacy
Cookie preferences
Support


qft1.png (auto-K) - JupyterLab
https://lab.quantum-computing.ibm.com/user/65291073c366dd36a764d295/lab/workspaces/auto-K/tree/qft1.png
IBM Quantum Learning
Home
Catalog
Composer
Lab
File
Edit
View
Run
Kernel
Tabs
Settings
Help
Launcher
QAI 6 .ipynb
qft2.png
qft1.png
q0
q1
q2
q3
q4
X
0
1
2qft
3
4
c
5
0 1 2 3 4
Simple
Mem: 391.72 / 8192.00 MB
qft1.png
Terms
Privacy
Cookie preferences
Support


qft2.png (auto-K) - JupyterLab
https://lab.quantum-computing.ibm.com/user/65291073c366dd36a764d295/lab/workspaces/auto-K/tree/qft2.png
IBM Quantum Learning
Home
Catalog
Composer
Lab
File
Edit
View
Run
Kernel
Tabs
Settings
Help
Launcher
QAI 6 .ipynb
qft2.png
qft1.png
q0
q1
q2
q3
q4
X
0
1
2qft
3
4
c
5
0 1 2 3 4
Simple
Mem: 391.72 / 8192.00 MB
qft2.png
Terms
Privacy
Cookie preferences
Support

MINI PROJECT REPORT

Title: Implement Grover's Search Algorithm.

Introduction:

- *Grover's algorithm is a quantum algorithm that solves the unstructured search problem. In an unstructured search problem, we are given a set of N elements and we want to find a single marked element. A classical computer would need to search through all N elements in order to find the marked element, which would take time O(N). Grover's algorithm, on the other hand, can find the marked element in time O(√N).*
- *Grover's algorithm is a powerful tool that can be used to solve a variety of problems. For example, it can be used to find patterns in data, break cryptographic keys, and solve optimization problems. As quantum computers become more powerful, Grover's algorithm will become increasingly important.*

Algorithm:

The algorithm works by applying a series of quantum operations to the input state, which is initialized as a superposition of all possible search states. The key idea behind Grover's algorithm is to amplify the amplitude of the marked state (i.e., the state containing the item that we are searching for) by iteratively applying a quantum operation known as the Grover operator.

The Grover operator has two quantum operations:

- The reflection on the mean
- The inversion of the marked state.

Here is a more detailed explanation of how Grover's algorithm works:

1. Initial state:

The algorithm starts in a state that is a superposition of all *N* elements. This state can be written as:

$$|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$$

where |*x*☐ is the state corresponding to the element *x*.

2. Diffusion operator:

The diffusion operator is a quantum operation that amplifies the amplitudes of the states that correspond to the marked element. The diffusion operator can be written as

$$U = \frac{1}{2}I - \frac{1}{2}|\psi\rangle\langle\psi|$$

where *I* is the identity operator.
3. Measurement:
The algorithm measures the state of the system. This collapses the superposition and gives us the marked element. The repeated use of this operator increases the scope of the specified condition, making it easier to measure. Once the specified state is reached, the algorithm returns the index of the object corresponding to that state.

Pseudo Code:
Input: N: number of items in the list, oracle(x): a function that returns true if x is the target item, and false otherwise
Step 1: Initialize state

- *Hadamard transform on all qubits*

Step 2: Iterate over Grover's algorithm
for k = 1 to sqrt(N) do

 # Step 2a: Apply the oracle

- *Apply the oracle to the state*

 # Step 2b: Apply the diffusion operator

- *Hadamard transform on all qubits*
- *Apply an X gate on all qubits*
- *Apply a multi-controlled Z gate (which flips the sign of the state only if all qubits are in the state |1>)*
- *Apply an X gate on all qubits*
- *Hadamard transform on all qubits*

end for

Step 3: Measure the state and output the result

- *Measure the state and output the result*

Below is the Implementation of the above Code:

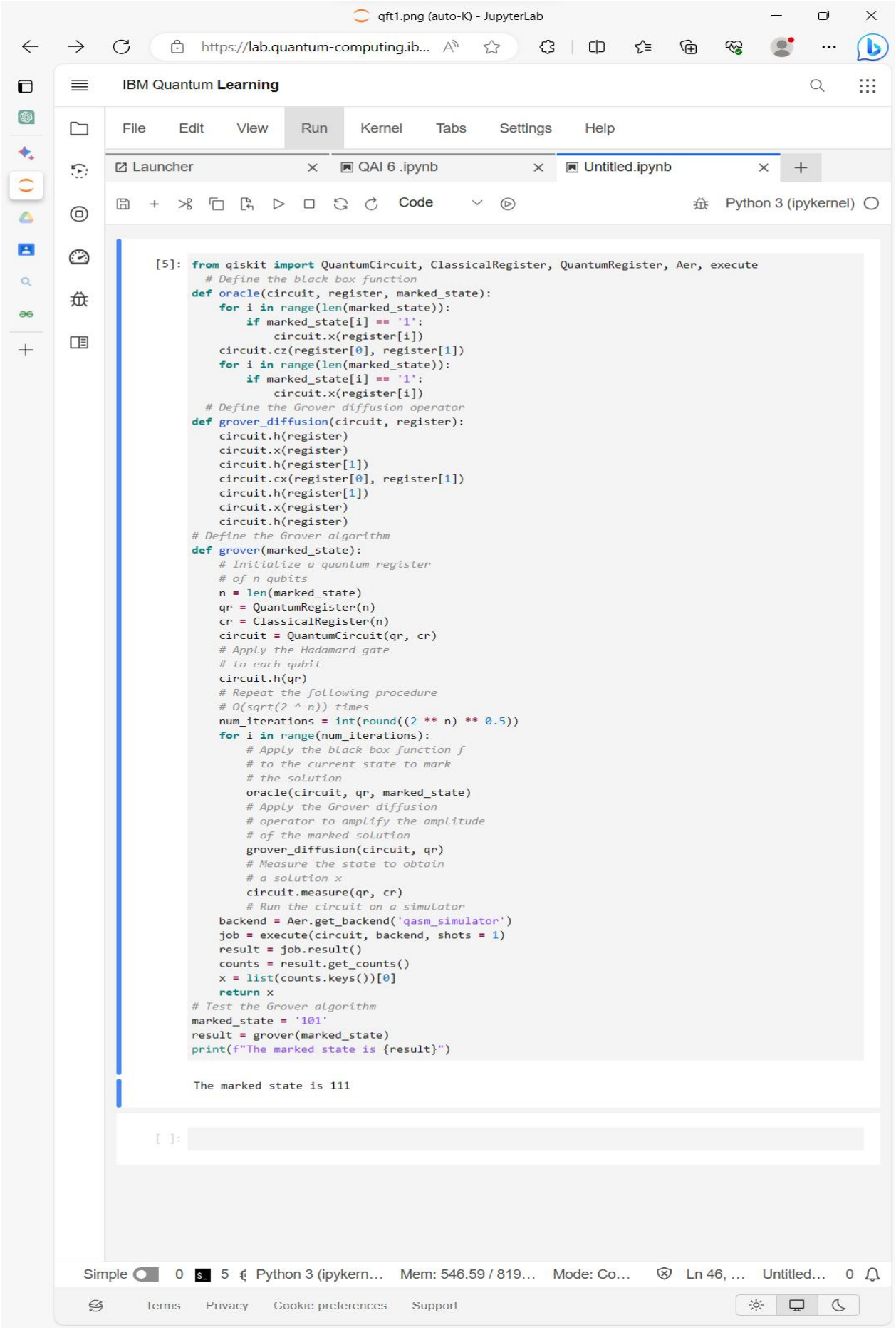
This code describes the Oracle function and the Grover diffusion operator and then uses it to implement the Grover algorithm for a given specified situation. The algorithm uses the Qiskit framework to define and run a quantum circuit in a simulator and returns the results of the measurements as a solution x.

Code:

```
from qiskit import QuantumCircuit, ClassicalRegister,
QuantumRegister, Aer, execute

# Define the black box function


def oracle(circuit, register, marked_state):
    for i in range(len(marked_state)):
        if marked_state[i] == '1':
            circuit.x(register[i])
    circuit.cz(register[0], register[1])
    for i in range(len(marked_state)):
        if marked_state[i] == '1':
            circuit.x(register[i])

# Define the Grover diffusion operator


def grover_diffusion(circuit, register):
    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)

# Define the Grover algorithm


def grover(marked_state):

    # Initialize a quantum register
    # of n qubits
    n = len(marked_state)
    qr = QuantumRegister(n)
    cr = ClassicalRegister(n)
    circuit = QuantumCircuit(qr, cr)

    # Apply the Hadamard gate
    # to each qubit
    circuit.h(qr)

    # Repeat the following procedure
    # O(sqrt(2 ^ n)) times
    num_iterations = int(round((2 ** n) ** 0.5))
```

```
    for i in range(num_iterations):
        # Apply the black box function f
        # to the current state to mark
        # the solution
        oracle(circuit, qr, marked_state)

        # Apply the Grover diffusion
        # operator to amplify the amplitude
        # of the marked solution
        grover_diffusion(circuit, qr)

    # Measure the state to obtain
    # a solution x
    circuit.measure(qr, cr)

    # Run the circuit on a simulator
    backend = Aer.get_backend('qasm_simulator')
    job = execute(circuit, backend, shots = 1)
    result = job.result()
    counts = result.get_counts()
    x = list(counts.keys())[0]

    return x


# Test the Grover algorithm
marked_state = '101'
result = grover(marked_state)
print(f"The marked state is {result}")
```

Output:


qft1.png (auto-K) - JupyterLab
https://lab.quantum-computing.ib...
IBM Quantum Learning
File Edit View Run Kernel Tabs Settings Help
Launcher
QAI 6 .ipynb
Untitled.ipynb
Code
Python 3 (ipykernel)
[5]:
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, Aer, execute
# Define the black box function
def oracle(circuit, register, marked_state):
for i in range(len(marked_state)):
if marked_state[i] == '1':
circuit.x(register[i])
circuit.cz(register[0], register[1])
for i in range(len(marked_state)):
if marked_state[i] == '1':
circuit.x(register[i])
# Define the Grover diffusion operator
def grover_diffusion(circuit, register):
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)
# Define the Grover algorithm
def grover(marked_state):
# Initialize a quantum register
# of n qubits
n = len(marked_state)
qr = QuantumRegister(n)
cr = ClassicalRegister(n)
circuit = QuantumCircuit(qr, cr)
# Apply the Hadamard gate
# to each qubit
circuit.h(qr)
# Repeat the following procedure
# O(sqrt(2 ^ n)) times
num_iterations = int(round((2 ** n) ** 0.5))
for i in range(num_iterations):
# Apply the black box function f
# to the current state to mark
# the solution
oracle(circuit, qr, marked_state)
# Apply the Grover diffusion
# operator to amplify the amplitude
# of the marked solution
grover_diffusion(circuit, qr)
# Measure the state to obtain
# a solution x
circuit.measure(qr, cr)
# Run the circuit on a simulator
backend = Aer.get_backend('qasm_simulator')
job = execute(circuit, backend, shots = 1)
result = job.result()
counts = result.get_counts()
x = list(counts.keys())[0]
return x
# Test the Grover algorithm
marked_state = '101'
result = grover(marked_state)
print(f"The marked state is {result}")
The marked state is 111
[ ]:
Simple 0 5 Python 3 (ipykern... Mem: 546.59 / 819... Mode: Co... Ln 46, ... Untitled... 0
Terms Privacy Cookie preferences Support

Applications of Grover's Algorithm:

- Data mining: Grover's algorithm can be used to find patterns in large datasets that would be impossible to find with a classical computer. For example, it could be used to find fraudulent transactions in a financial database or to identify cancer cells in a medical image.
- Cryptography: Grover's algorithm can be used to break cryptographic keys that are currently considered secure. This could have a major impact on the security of online communications and transactions.
- Optimization: Grover's algorithm can be used to solve optimization problems that are difficult or impossible to solve with a classical computer. For example, it could be used to find the shortest route between two points or to find the optimal investment portfolio..

Limitations of Grover's Algorithm:

- Quadratic speedup: Grover's algorithm gives a quadratic speedup compared to classical algorithms, which means that it can only provide a significant speedup for some problems that have a large search space. For problems with smaller search spaces, the speedup may not be good enough to justify the use of this algorithm.
- Limited by hardware: Grover's algorithm requires the use of a large number of qubits to achieve a significant speedup, which is currently beyond the capabilities of most quantum hardware. This means that the algorithm may not be practical for many real-world applications until more powerful quantum hardware is developed.
- Limited to unstructured search: Grover's algorithm is designed for unstructured search problems, which means that it may not be applicable to other types of problems such as optimization or simulation.
- Error-prone: Grover's algorithm, like all quantum algorithms, is susceptible to errors due to noise and decoherence. These errors can affect the accuracy and reliability of the algorithm, which can limit its practical use in real-world applications.
- Limited applicability in cryptography: Although Grover's algorithm can be used to break certain cryptographic algorithms, it is not applicable to all types of encryption. For example, public-key encryption is not vulnerable to Grover's algorithm due to its use of mathematical problems that are not efficiently solvable by quantum computers.

Conclusion:

Overall, Grover's algorithm is a powerful tool that can be used to solve a variety of problems. For example, it can be used to find patterns in data, break cryptographic keys, and solve optimization problems. As quantum computers become more powerful, Grover's algorithm will become increasingly important.