

EXPERIMENT NO 1

TITLE: Installation and Configuring Qiskit in Windows Machine

AIM: To install and configuring qiskit for Windows

OBJECTIVE: To enable learners to successfully install and configure Qiskit on a Windows machine, empowering them to create and run basic quantum circuits. This objective emphasizes practical skills in utilizing quantum programming environments, fostering foundational knowledge in quantum computing concepts and Qiskit functionalities.

THEORY:

Step 1.1: Install Python Qiskit requires Python 3.7 or later. This can be downloaded from the official Python website if Python is not pre-installed on the system.

The following instructions are crucial:

- Download the Python installer from the Python website.
- Run the installer and make sure to check the box that says "Add Python to PATH."
- Follow the installation prompts.

Step 1.2: Install a Virtual Environment (Optional but Recommended) It's recommended to install Qiskit within a virtual environment to avoid conflicts with other packages.

Create a Virtual Environment: *python -m venv qiskit-env*

Activate the Virtual Environment: *.\qiskit-env\Scripts\activate*

Step 1.3: Install Qiskit With your environment set up, you can now install Qiskit using pip.

Install Qiskit: pip install Qiskit[all]

This command installs the basic Qiskit package along with its dependencies.

Step 1.4: Verify the Installation After the installation is complete, you can verify it by running a simple Python script.

Create a Test Script:

Create a file named test_qiskit.py with the following content:

```
from qiskit import QuantumCircuit, Aer, execute

Circuit = QuantumCircuit(1, 1) circuit.h(0) circuit.measure(0, 0)

Simulator = Aer.get_backend('qasm_simulator')

result = execute(circuit, simulator).result()
```

Run the Script:

```
python test_qiskit.py
```

If the installation was successful, you should see output similar to `{'0': 512, '1': 512}`, indicating that the Hadamard gate created an equal superposition of states.

Step 1.5: Install Optional Components (If needed)

If you want to use advanced features, such as running circuits on IBM Quantum's real hardware or using additional simulators, you might need to install extra packages.

Install Qiskit Terra (for advanced quantum algorithms):

```
pip install qiskit-terra
```

Install Qiskit Aer (for additional simulators):

```
pip install qiskit-aer
```

Install Qiskit Ignis (for noise modeling and error correction):

```
pip install qiskit-ignis
```

Install Qiskit IBMQ Provider (to access IBM Quantum hardware):

```
pip install qiskit-ibmq-provider
```

Step 1.6: Access IBM Quantum Devices (Optional)

To run your quantum circuits on real quantum devices provided by IBM, you need to create an IBM Quantum account.

- Sign up for IBM Quantum: Go to IBM Quantum and create an account.
- Get Your API Token: After logging in, go to your account settings to find your API token.
- Save Your API Token:

Use the following commands in a Python script or directly in a Python shell:

```
from qiskit import IBMQ
```

```
IBMQ.save_account('YOUR_API_TOKEN')
```

After saving your token, you can load your IBMQ account and access IBM's quantum devices directly from your Python scripts.

Step 1.7: Keeping Qiskit Updated Qiskit is frequently updated with new features and improvements. To update Qiskit to the latest version,

```
run: pip install -U qiskit
```

This command ensures that all Qiskit components are up to date.

CONCLUSION: We have successfully installed Qiskit and run a simple quantum circuit on your Windows machine! This process lays the groundwork for exploring quantum computing further.

EXPERIMENT NO 2

TITLE: Write Python codes to demonstrate the concepts of Basic Linear algebra, Vector Operations, Vector Multiplications, and Tensor Products

AIM: To demonstrate the concepts of “Basic Linear algebra, Vector Operations, Vector Multiplications, and Tensor Products”

OBJECTIVE: To enable learners to implement and understand basic linear algebra concepts, vector operations, vector multiplications, and tensor products using Python.

2.1: Machine Learning and AI

- **Data Representation:** Vectors are used to represent data points, features, and weights in models.
- **Neural Networks:** Tensor products are essential in the architecture of neural networks, especially in deep learning, where multi-dimensional data needs to be processed.
- **Optimization:** Linear algebra is crucial for optimization algorithms used in training models.

2.2: Computer Graphics

- **Transformations:** Vector operations are fundamental in performing transformations (translation, rotation, scaling) in 2D and 3D graphics.
- **Rendering Techniques:** Tensors are used in shading, lighting calculations, and simulations of physical properties.

2.3: Physics and Engineering

- **Mechanics:** Vectors represent forces, velocities, and accelerations, while tensors can describe stress and strain in materials.
- **Electromagnetism:** The behavior of electric and magnetic fields can be described using vector fields and tensors.

2.4: Robotics

- **Kinematics and Dynamics:** Vectors are used to represent the position and orientation of robots, while tensors can describe complex interactions and transformations.
- **Control Systems:** Linear algebra is essential for modeling and controlling robotic systems.

THEORY:

Basic Linear Algebra: Linear algebra is a branch of mathematics that studies vectors, vector spaces (or linear spaces), linear transformations, and systems of linear equations. It provides a framework for modeling and solving problems involving linear relationships.

Vectors: A vector is an ordered list of numbers, which can represent a point in space or a direction and magnitude. Vectors can be of different dimensions (e.g., 2D, 3D) and are typically denoted in bold (e.g., \mathbf{v}).

Vector Operations

- **Vector Addition:**
 - **Definition:** The operation of adding two vectors by adding their corresponding components.
- **Vector Subtraction:**
 - **Definition:** The operation of subtracting one vector from another by subtracting their corresponding components.
- **Scalar Multiplication:**
 - **Definition:** The operation of multiplying a vector by a scalar (a single number), which scales each component of the vector.

Vector Multiplications

- **Dot Product:**
 - **Definition:** The dot product (or scalar product) of two vectors results in a scalar value, calculated as the sum of the products of their corresponding components.]
- **Cross Product:**
 - **Definition:** The cross product of two vectors in three-dimensional space results in a vector that is perpendicular to both.

Tensor Products: The tensor product of two vectors produces a matrix (or a higher-dimensional tensor), where each element is the product of elements from the original vectors.

IMPLEMENTATION:

Code 1: MATRIX OPERATIONS

```
import numpy as np
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])
b = np.array([1, 0, 1])
result = np.dot(A, b)
print("Matrix A:")
print(A)
print("\nVector b:")
print(b)
print("\nResult of A * b:")
print(result)
# Determinant of matrix A
det_A = np.linalg.det(A)
print("\nDeterminant of A:")
print(det_A)
try:
    inv_A = np.linalg.inv(A)
    print("\nInverse of A:")
    print(inv_A)
except np.linalg.LinAlgError:
    print("\nMatrix A is singular
and cannot be inverted.")
```

Code 2: VECTOR OPERATIONS

```
import numpy as np
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])
add = np.add(v1, v2)
sub = np.subtract(v1, v2)
dot_product = np.dot(v1, v2)
cross_product = np.cross(v1, v2)
print("Vector v1:")
print(v1)
print("\nVector v2:")
print(v2)
print("\nAddition of v1 and v2:")
print(add)
print("\nSubtraction of v2 from
v1:")
print(sub)
print("\nDot Product of v1 and v2:")
print(dot_product)
print("\nCross Product of v1 and
v2:")
print(cross_product)
```

Code 3: DOT PRODUCT

```
import numpy as np
v1 = np.array([2, 3, 4])
v2 = np.array([1, 0, -1])
element_wise_mult = np.multiply(v1, v2)
dot_product = np.dot(v1, v2)
print("Vector v1:")
print(v1)
print("\nVector v2:")
print(v2)
print("\nElement-wise
multiplication of v1 and v2:")
print(element_wise_mult)
print("\nDot Product of v1 and
v2:")
print(dot_product)
```

Code 4: TENSOR PRODUCT

```
import numpy as np
v1 = np.array([1, 2])
v2 = np.array([3, 4])
tensor_product = np.outer(v1, v2)

print("Vector v1:")
print(v1)
print("\nVector v2:")
print(v2)
print("\nTensor Product (Outer
Product) of v1 and v2:")
print(tensor_product)
A = np.array([[1, 2], [3, 4]])
B = np.array([[0, 5], [6, 7]])
kronecker_product = np.kron(A, B)

print("\nMatrix A:")
```

	<pre> print(A) print("\nMatrix B:") print(B) print("\nKronecker Product of A and B:") print(kronecker_product) </pre>
--	---

RESULT:

<p>OUTPUT1 MATRIX OPERATIONS</p> <pre> ➡ Matrix A: [[1 2 3] [4 5 6] [7 8 9]] Vector b: [1 0 1] Result of A * b: [4 10 16] Determinant of A: 0.0 Matrix A is singular and cannot be inverted. </pre>	<p>OUTPUT2 VECTOR OPERATIONS</p> <pre> ➡ Vector v1: [1 2 3] Vector v2: [4 5 6] Addition of v1 and v2: [5 7 9] Subtraction of v2 from v1: [-3 -3 -3] Dot Product of v1 and v2: 32 Cross Product of v1 and v2: [-3 6 -3] </pre>
<p>OUTPUT3 DOT PRODUCT</p> <pre> ➡ Vector v1: [2 3 4] Vector v2: [1 0 -1] Element-wise multiplication of v1 and v2: [2 0 -4] Dot Product of v1 and v2: -2 </pre>	<p>OUTPUT4: TENSOR PRODUCT</p> <pre> ➡ Vector v1: [1 2] Vector v2: [3 4] Tensor Product (Outer Product) of v1 and v2: [[3 4] [6 8]] Matrix A: [[1 2] [3 4]] Matrix B: [[0 5] [6 7]] Kronecker Product of A and B: [[0 5 0 10] [6 7 12 14] [0 15 0 20] [18 21 24 28]] </pre>

CONCLUSION: By running this code, you will see how to perform basic linear algebra operations, including vector addition, subtraction, dot products, scalar multiplications, and tensor products. These concepts form the foundational building blocks for more advanced topics in linear algebra and quantum computing.

EXPERIMENT NO 3

TITLE: Write Python code to Implement the Identity Matrix for 1Qubit, 2 Qubits, 3 Qubits

AIM: To Implement the Identity Matrix for 1Qubit, 2 Qubits, 3 Qubits

OBJECTIVE: To implement the identity matrix for qubits, we can represent the identity matrix for 1, 2, and 3 qubits. The identity matrix is used in quantum computing to indicate a state that is unchanged by the operation.

- **Quantum Gates:** The identity matrix is fundamental in defining quantum gates, particularly when combined with other operations to ensure that certain qubits maintain their states.
- **Quantum Circuit Design:** In designing quantum circuits, identity operations can be employed to maintain the integrity of qubits while other gates are applied.
- **Error Correction:** In quantum error correction schemes, the identity matrix is used to isolate qubits that are not part of the error correction operation.
- **Quantum Algorithms:** Many quantum algorithms utilize identity matrices to perform conditional operations, especially in algorithms like Grover's or Shor's, where some qubits may not need to be altered.

THEORY:

3.1: 1 Qubit

For a single qubit, the identity matrix I is represented as:

$$I_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

3.2: 2 Qubits

For two qubits, the identity matrix I acts on a 4-dimensional state space. The 2-qubit identity matrix is:

$$I_2 = I \otimes I = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

3.3: 3 Qubits

For three qubits, the identity matrix acts on an 8-dimensional state space. The 3-qubit identity matrix is:

$$I_3 = I \otimes I \otimes I = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

IMPLEMENTATION:

IDENTITY MATRIX

```
import numpy as np
def identity_matrix(n_qubits):
    dim = 2 ** n_qubits
    return np.eye(dim)
def apply_identity(state_vector, n_qubits):
    identity = identity_matrix(n_qubits)
    return np.dot(identity, state_vector)
def example_state_vectors(n_qubits):
    dim = 2 ** n_qubits
    state_vector = np.zeros(dim, dtype=complex)
    state_vector[0] = 1 # Initialize to |0> state
    return state_vector
def demonstrate_identity_operations():
    identities = {}
    states = {}
    for n_qubits in range(1, 4):
        identity = identity_matrix(n_qubits)
        state_vector = example_state_vectors(n_qubits)
        new_state_vector = apply_identity(state_vector,
n_qubits)
        identities[n_qubits] = identity
        states[n_qubits] = {
            'Original State': state_vector,
            'New State after Identity': new_state_vector
        }
    return identities, states
identities, states = demonstrate_identity_operations()
for n_qubits in identities:
    print(f"Identity Matrix for {n_qubits} Qubits:")
    print(identities[n_qubits])
    print("\nState Vectors:")
    for key, value in states[n_qubits].items():
        print(f"{key}: {value}")
    print("\n" + "=" * 40)
```


RESULT: IDENTITY MATRIX

```
⇒ Identity Matrix for 1 Qubits:
[[1. 0.]
 [0. 1.]]

State Vectors:
Original State: [1.+0.j 0.+0.j]
New State after Identity: [1.+0.j 0.+0.j]

=====
Identity Matrix for 2 Qubits:
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]

State Vectors:
Original State: [1.+0.j 0.+0.j 0.+0.j 0.+0.j]
New State after Identity: [1.+0.j 0.+0.j 0.+0.j 0.+0.j]

=====
Identity Matrix for 3 Qubits:
[[1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0.]
```

CONCLUSION:

Matrices are foundational in quantum algorithms and operations, serving as a benchmark for transformations. They help maintain the integrity of qubit states during quantum computations, facilitating the study of more complex operations such as gates and measurements.

EXPERIMENT NO 4

TITLE: Write Python code to Implement Pauli Gates

AIM: To Implement Pauli Gates

OBJECTIVE: The objective of implementing Pauli gates in quantum computing is to demonstrate their role as fundamental quantum operations that manipulate the states of qubits. The three Pauli gates—Pauli-X, Pauli-Y, and Pauli-Z—are essential for quantum state manipulation, error correction, and creating complex quantum algorithms.

4.1: Quantum Computing Algorithms

- **Quantum Search Algorithms:** In algorithms like Grover's search, Pauli gates are used to manipulate qubit states efficiently, helping to enhance the search speed compared to classical algorithms.
- **Quantum Cryptography:** Pauli gates can be utilized in protocols like BB84 to ensure the security of quantum key distribution by manipulating qubit states during the transmission process.

4.2: Quantum Error Correction

- **Error Correction Codes:** Pauli gates are integral in various quantum error correction schemes (e.g., surface codes, stabilizer codes), where they help correct errors introduced during quantum computation.
- **Fault-Tolerant Quantum Computing:** The ability to implement Pauli gates accurately is essential for building fault-tolerant quantum circuits, ensuring reliable quantum operations despite the presence of noise.

THEORY:

Pauli Gates

4.3: Pauli-X Gate (NOT Gate)

The Pauli-X gate flips the state of a qubit. It is represented as:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

- **Action:**
 - $|0\rangle$ becomes $|1\rangle$
 - $|1\rangle$ becomes $|0\rangle$

4.4: Pauli-Y Gate

The Pauli-Y gate combines a bit flip and a phase flip. It is represented as:

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

- Action:
 - $|0\rangle$ becomes $-i|1\rangle$
 - $|1\rangle$ becomes $i|0\rangle$

4.5: Pauli-Z Gate

The Pauli-Z gate applies a phase flip. It is represented as:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

- Action:
 - $|0\rangle$ remains $|0\rangle$
 - $|1\rangle$ becomes $-|1\rangle$

IMPLEMENTATION:

PAULI GATES:

```
import numpy as np
def pauli_x():
    """Pauli-X (NOT) gate"""
    return np.array([[0, 1],
                     [1, 0]])
def pauli_y():
    """Pauli-Y gate"""
    return np.array([[0, -1j],
                     [1j, 0]])
def pauli_z():
    """Pauli-Z gate"""
    return np.array([[1, 0],
                     [0, -1]])
def apply_pauli_gate(gate, state_vector):
    """Apply the Pauli gate to the given state vector"""
    return np.dot(gate, state_vector)
def example_state_vector():
    """Example state vector  $|0\rangle$ """
    return np.array([1, 0])
def demonstrate_pauli_gates():
    # Get Pauli gates
    x_gate = pauli_x()
    y_gate = pauli_y()
    z_gate = pauli_z()
    state_vector = example_state_vector()
    x_result = apply_pauli_gate(x_gate, state_vector)
    y_result = apply_pauli_gate(y_gate, state_vector)
    z_result = apply_pauli_gate(z_gate, state_vector)
    return {
        'Pauli-X': x_result,
        'Pauli-Y': y_result,
        'Pauli-Z': z_result
    }
results = demonstrate_pauli_gates()
for gate, result in results.items():
    print(f"{gate} Gate applied to  $|0\rangle$ : {result}")
```

RESULT: PAULI GATES



```
Pauli-X Gate applied to |0>: [0 1]
Pauli-Y Gate applied to |0>: [0.+0.j 0.+1.j]
Pauli-Z Gate applied to |0>: [1 0]
```

CONCLUSION: The Pauli gates serve as fundamental building blocks in quantum computing, allowing for the manipulation of qubit states essential for constructing quantum algorithms.

- **Pauli-X** functions as a classical NOT gate, enabling state flipping.
- **Pauli-Y** combines both bit and phase flips, introducing complex amplitude factors.
- **Pauli-Z** provides a simple phase shift without changing the state vector's probabilities.

EXPERIMENT NO 5

TITLE: Write Python code to Implement Hadamard Gates

AIM: To Implement Hadamard Gates

OBJECTIVE: The **Hadamard gate** is a fundamental quantum gate used in quantum computing. Its primary objectives include: superposition, interference and entanglement

5.1: Quantum Teleportation

- **State Preparation:** In quantum teleportation protocols, the Hadamard gate helps prepare the entangled state necessary for the successful teleportation of quantum information.

5.2: Quantum Cryptography

- **Quantum Key Distribution (QKD):** The Hadamard gate is used in protocols like BB84, where it helps encode classical bits into quantum states, ensuring secure communication.

THEORY:

HADMARD GATE:

The Hadamard gate (often denoted as H) is a fundamental single-qubit gate in quantum computing that creates superposition. When applied to a qubit, it transforms the basis states $|0\rangle$ and $|1\rangle$ into equal superpositions of both states.

The matrix representation of the Hadamard gate is given by:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

The Hadamard gate is a powerful tool in quantum computing, enabling the creation of superposition states that are crucial for many quantum algorithms. Its ability to transform basis states into equal superpositions plays a significant role in quantum algorithms, allowing for parallelism and interference effects.

IMPLEMENTATION:

```
import numpy as np
def hadamard_gate():
    """Define the Hadamard matrix."""
    H = np.array([[1, 1], [1, -1]]) / np.sqrt(2)
    return H
def apply_hadamard(qubit_state):
    """Apply the Hadamard gate to a given qubit state."""
    H = hadamard_gate()
    return np.dot(H, qubit_state)
def main():
    state_0 = np.array([1, 0]) # |0>
    state_1 = np.array([0, 1]) # |1>
    hadamard_0 = apply_hadamard(state_0)
    hadamard_1 = apply_hadamard(state_1)
    print("Hadamard applied to |0> results in:", hadamard_0)
    print("Hadamard applied to |1> results in:", hadamard_1)
if __name__ == "__main__":
    main()
```

RESULT: HADAMARD GATE



```
Hadamard applied to |0> results in: [0.70710678 0.70710678]
Hadamard applied to |1> results in: [ 0.70710678 -0.70710678]
```

CONCLUSION: The Hadamard gate plays a crucial role in quantum computing by enabling the creation of superposition states and facilitating quantum interference and entanglement. These transformations allow quantum computers to explore multiple solutions simultaneously, offering a significant advantage over classical computation in specific scenarios. The Hadamard gate is fundamental to various quantum algorithms and protocols, making it an essential component in the toolbox of quantum computing.

EXPERIMENT NO 6

TITLE: Write Python code to Implement Two Qubit Gates

AIM: To Implement Two Qubit Gates

OBJECTIVE: The objective of implementing two-qubit gates, such as the CNOT gate and the Hadamard gate, is to deepen understanding of several key concepts in quantum computing.

6.1: Quantum Teleportation

- **State Transfer:** In quantum teleportation protocols, the CNOT gate is crucial for transferring the state of a qubit from one location to another, enabling efficient communication.

6.2: Quantum Computing Architectures

- **Building Quantum Circuits:** The CNOT gate is a key component in quantum circuits, used to connect qubits in larger computational architectures, thereby enhancing computational power.
- **Quantum Gate Sets:** CNOT is part of universal gate sets, allowing the construction of any quantum operation when combined with single-qubit gates.

THEORY:

The CNOT (Controlled-NOT) gate is a fundamental two-qubit gate in quantum computing. It operates on two qubits: a control qubit and a target qubit. The CNOT gate flips the state of the target qubit if the control qubit is in the $|1\rangle$ state. This gate is essential for creating entanglement and performing conditional operations in quantum circuits.

The matrix representation of the CNOT gate is as follows:

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

IMPLEMENTATION:

```
import numpy as np
def cnot_gate():
    """Define the CNOT matrix."""
    CNOT = np.array([[1, 0, 0, 0],
                     [0, 1, 0, 0],
                     [0, 0, 0, 1],
                     [0, 0, 1, 0]])

    return CNOT
def apply_cnot(control, target):
    """Apply the CNOT gate to a control and target qubit."""
    combined_state = np.kron(control, target)
    cnot = cnot_gate()
    return np.dot(cnot, combined_state)

def main():
    state_0 = np.array([1, 0]) # |0>
    state_1 = np.array([0, 1]) # |1>
    control_qubit = state_0 # Control qubit |0>
    target_qubit = state_0 # Target qubit |0>
    result = apply_cnot(control_qubit, target_qubit)
    print("CNOT applied to |00> results in:", result)
    control_qubit = state_1 # Control qubit |1>
    target_qubit = state_0 # Target qubit |0>
    result = apply_cnot(control_qubit, target_qubit)
    print("CNOT applied to |10> results in:", result)
    target_qubit = state_1 # Target qubit |1>
    result = apply_cnot(control_qubit, target_qubit)
    print("CNOT applied to |11> results in:", result)
if __name__ == "__main__":
    main()
```

RESULT: TWO QUBIT GATES (CNOT GATE)

```
⇒ CNOT applied to |00> results in: [1 0 0 0]
   CNOT applied to |10> results in: [0 0 0 1]
   CNOT applied to |11> results in: [0 0 1 0]
```

CONCLUSION: The study and implementation of two-qubit gates are pivotal for anyone seeking to understand and innovate in the field of quantum computing, providing essential tools for harnessing the unique capabilities of quantum mechanics.

EXPERIMENT NO 7

TITLE: Write Python code to Implement Three Qubit Gates

AIM: To Implement Three Qubit Gates

OBJECTIVE: The objective of implementing three-qubit gates is to explore the more complex interactions and manipulations of quantum states that arise from operations on three qubits. This understanding is essential for developing advanced quantum algorithms and protocols.

7.1: Quantum Cryptography

- **Secure Communication Protocols:** Three-qubit gates enhance the security of quantum key distribution schemes by enabling the manipulation of multiple entangled states, increasing resistance to eavesdropping.
- **Multi-Party Computation:** These gates facilitate secure computations among multiple parties, allowing them to collaboratively perform calculations without revealing their inputs.

7.2: Quantum Benchmarking and Characterization

- **Performance Evaluation:** Three-qubit gates can be used to benchmark and characterize quantum devices, providing insights into their operational fidelity and coherence properties.
- **Error Characterization:** They enable detailed studies of error rates and types in quantum circuits, contributing to the development of more robust quantum technologies.

THEORY:

7.3 Toffoli Gate:

The Toffoli gate, also known as the Controlled-Controlled-NOT (CCNOT) gate, is a three-qubit gate that flips the state of a target qubit if and only if both control qubits are in the $|1\rangle$. This gate is crucial for creating entangled states and is often used in quantum error correction and quantum algorithms.

The matrix representation of the Toffoli gate is an 8×8 matrix, as it operates on three qubits:

$$\text{Toffoli} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

7.4 Fredkin Gate

The Fredkin gate, also known as the Controlled-SWAP gate, is a three-qubit gate that swaps the states of two target qubits based on the state of a control qubit. If the control qubit is in the $|1\rangle$ state, the states of the two target qubits are swapped. If the control qubit is in the $|0\rangle$ state, the target qubits remain unchanged.

The matrix representation of the Fredkin gate is also an 8x8 matrix:

$$\text{Fredkin} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

IMPLEMENTATION:

CODE 1: TOFFOLI GATE

```
import numpy as np

def toffoli_gate():
    TOFFOLI = np.array([
        [1, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 0, 0, 1]
    ])
    return TOFFOLI

def apply_toffoli(control1, control2, target):
    # Combine the state vectors
    # into a single state vector
    combined_state = np.kron(np.kron(control1, control2), target)
    toffoli = toffoli_gate()
    return np.dot(toffoli, combined_state)

def main():
    state_0 = np.array([1, 0]) # |0>
    state_1 = np.array([0, 1]) # |1>

    # Test case 1: Toffoli applied
    # to |000>
    control1_qubit = state_0
    control2_qubit = state_0
    target_qubit = state_0
    result = apply_toffoli(control1_qubit, control2_qubit, target_qubit)
    print("Toffoli applied to |000> results in:", result)
```

CODE 2: FREDKIN GATE

```
import numpy as np

def fredkin_gate():
    FREDKIN = np.array([
        [1, 0, 0, 0, 0, 0, 0, 0],
        [0, 1, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0],
        [0, 0, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, 0],
        [0, 0, 0, 0, 0, 0, 0, 1]
    ])
    return FREDKIN

def apply_fredkin(control, target1, target2):
    combined_state = np.kron(np.kron(control, target1), target2)
    fredkin = fredkin_gate()
    return np.dot(fredkin, combined_state)

def main():
    state_0 = np.array([1, 0])
    state_1 = np.array([0, 1])



    control_qubit = state_1
    target1_qubit = state_0
    target2_qubit = state_1

    result = apply_fredkin(control_qubit, target1_qubit, target2_qubit)
    print("Fredkin applied to |110> results in:", result)

    control_qubit = state_0
    result = apply_fredkin(control_qubit, target1_qubit, target2_qubit)
```

<pre> # Test case 2: Toffoli applied to 110> control1_qubit = state_1 control2_qubit = state_1 target_qubit = state_0 result = apply_toffoli(control1_qubit, control2_qubit, target_qubit) print("Toffoli applied to 110> results in:", result) if __name__ == "__main__": main() </pre>	<pre> print("Fredkin applied to 010> results in:", result) if __name__ == "__main__": main() </pre>
---	---

RESULT:

<p>TOFFOLI GATE:</p>  <pre> Toffoli applied to 000> results in: [1 0 0 0 0 0 0 0] Toffoli applied to 110> results in: [0 0 0 0 0 0 0 1] </pre>
<p>FREDKIN GATE</p>  <pre> Fredkin applied to 110> results in: [0 0 0 0 0 0 1 0] Fredkin applied to 010> results in: [0 1 0 0 0 0 0 0] </pre>

CONCLUSION: The implementation of three-qubit gates, particularly the Toffoli gate, is crucial for understanding the principles of quantum computing and the interactions of multiple qubits. The study and implementation of three-qubit gates are essential for anyone interested in exploring the depths of quantum computing, equipping them with the necessary tools to harness the power of multi-qubit systems.

EXPERIMENT NO 8

TITLE: Write Python code to Implement Circuit Formation-1

AIM: To Implement Circuit Formation-1

OBJECTIVE: The objective of this implementation is to create a quantum circuit that demonstrates the principles of superposition and entanglement using basic quantum gates.

- To understand and simulate different quantum circuits in Python using NumPy.
- To explore the behavior of qubits through quantum gate applications.
- To simulate entangled states (Bell and GHZ states).
- To implement Quantum Fourier Transform (QFT) and Quantum Phase Estimation (QPE).

THEORY:

8.1: Quantum Circuits:

Quantum circuits consist of qubits and quantum gates. A quantum gate alters the state of a qubit, and a quantum circuit involves applying multiple gates sequentially to qubits.

8.2: Sequential Circuit:

A sequential circuit applies quantum gates in a specific order to qubits, changing the

quantum state step-by-step. These gates may include Hadamard, Pauli-X, and Controlled-NOT (CNOT).

8.3: Bell State:

The Bell state is an entangled state involving two qubits, representing maximal entanglement between them. The most common Bell state is $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, which means both qubits are in superposition and their states are correlated.

8.4: GHZ (Greenberger-Horne-Zeilinger) State:

GHZ states are entangled states of three or more qubits, where all qubits are either in $|0\rangle$ or $|1\rangle$ with equal probability. It generalizes the concept of the Bell state to more than two qubits.

8.5: Quantum Fourier Transform (QFT):

QFT is the quantum version of the classical Fourier transform. It transforms a quantum state into its Fourier components. QFT is widely used in many quantum algorithms, such as Shor's algorithm for factoring integers.

8.6: Quantum Phase Estimation (QPE):

QPE is used to estimate the phase of an eigenvalue associated with a unitary operator. It is crucial in quantum algorithms, especially in factoring large numbers and solving linear systems.

IMPLEMENTATION:

```
import numpy as np

# Define basic gates
def hadamard_gate():
    return np.array([[1, 1], [1, -1]]) / np.sqrt(2)

def pauli_x_gate():
    return np.array([[0, 1], [1, 0]])

def cnot_gate():
    return np.array([[1, 0, 0, 0],
                     [0, 1, 0, 0],
                     [0, 0, 0, 1],
                     [0, 0, 1, 0]])

def phase_gate(phase):
    return np.array([[1, 0],
                     [0, np.exp(2 * np.pi * phase)]])

# Initialize and apply gate
def apply_gate(gate, state, qubits, n_qubits):
    full_gate = np.eye(2**n_qubits)
    if len(qubits) == 1:
        idx = qubits[0]
        full_gate = np.kron(np.eye(2**idx), np.kron(gate,
np.eye(2**(n_qubits - idx - 1))))
    elif len(qubits) == 2:
        idx1, idx2 = qubits
        if idx1 > idx2:
            idx1, idx2 = idx2, idx1
        pre_gate = np.eye(2**idx1)
        middle_gate = np.kron(np.eye(2**(idx2 - idx1 - 1)), gate)
        post_gate = np.eye(2**(n_qubits - idx2 - 1))
        full_gate = np.kron(pre_gate, np.kron(middle_gate,
post_gate))
    return np.dot(full_gate, state)

# Print quantum state
def print_state(state, n_qubits):
```

```

        for i in range(2**n_qubits):
            print(f"|{i:0{n_qubits}b}>: {state[i]:.4f}")

# Quantum Circuits: Bell State, GHZ, QFT, QPE
def bell_state():
    state = np.array([1, 0, 0, 0])
    state = apply_gate(hadamard_gate(), state, [0], 2)
    state = apply_gate(cnot_gate(), state, [0, 1], 2)
    print_state(state, 2)

def ghz_state():
    state = np.array([1] + [0]*7)
    state = apply_gate(hadamard_gate(), state, [0], 3)
    state = apply_gate(cnot_gate(), state, [0, 1], 3)
    state = apply_gate(cnot_gate(), state, [0, 2], 3)
    print_state(state, 3)

def qft():
    state = np.array([1] + [0]*7)
    state = apply_gate(hadamard_gate(), state, [0], 3)
    state = apply_gate(phase_gate(0.5), state, [1], 3)
    state = apply_gate(hadamard_gate(), state, [1], 3)
    print_state(state, 3)




def qpe():
    state = np.array([1] + [0]*15)
    for i in range(3):
        state = apply_gate(hadamard_gate(), state, [i], 4)
    state = apply_gate(phase_gate(0.5), state, [1], 4)
    print_state(state, 4)

def main():
    print("Bell State:")
    bell_state()
    print("\nGHZ State:")
    ghz_state()
    print("\nQFT:")
    qft()
    print("\nQPE:")
    qpe()

if __name__ == "__main__":
    main()

```


RESULT: FORMATION CIRCUIT 1

 Bell State:			
00>: 0.7071			011>: 0.0000
01>: 0.0000			100>: 0.5000
10>: 0.0000			101>: 0.0000
11>: 0.7071			110>: 0.5000
			111>: 0.0000
GHZ State:			QPE:
000>: 0.7071			0000>: 0.3536
001>: 0.0000			0001>: 0.0000
010>: 0.0000			0010>: 0.3536
011>: 0.0000			0011>: 0.0000
100>: 0.0000			0100>: 8.1815
101>: 0.0000			0101>: 0.0000
110>: 0.0000			0110>: 8.1815
111>: 0.7071			0111>: 0.0000
QFT:			
000>: 0.5000			1000>: 0.3536
001>: 0.0000			1001>: 0.0000
010>: 0.5000			1010>: 0.3536
011>: 0.0000			1011>: 0.0000
100>: 0.5000			1100>: 8.1815
101>: 0.0000			1101>: 0.0000
110>: 0.5000			1110>: 8.1815
			1111>: 0.0000

CONCLUSION: In this implementation, we successfully demonstrated a basic quantum circuit formation .This simple circuit forms the foundation for understanding more complex quantum circuits and algorithms, illustrating key principles such as superposition and entanglement, which are fundamental in quantum computing.

EXPERIMENT NO 9

TITLE: Write Python code to Implement Circuit Formation-2

AIM: To Implement Circuit Formation-2

OBJECTIVE: -

- To create a quantum circuit with multiple qubits and apply various quantum gates.
- To simulate quantum state evolution using Python's `qiskit` library.

THEORY:

Quantum computing uses quantum bits (qubits), which differ from classical bits as they can represent both 0 and 1 simultaneously, thanks to the principle of superposition. Another crucial concept is entanglement, where the state of one qubit is dependent on the state of another, even at large distances.

Key Concepts:

9.1. Qubit: A qubit is a two-level quantum system that can exist in a linear combination of the basis states $|0\rangle$ and $|1\rangle$. The state of a qubit is represented as:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where α and β are complex numbers, and $|\alpha|^2 + |\beta|^2 = 1$

9.2: Superposition: A qubit can be in a superposition of both 0 and 1 states. When a Hadamard gate is applied, it creates an equal probability of being measured in either $|0\rangle$ or $|1\rangle$.

- Hadamard Gate (H): The Hadamard gate creates superposition:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

9.3: Entanglement: When two qubits become entangled, the state of one qubit directly affects the other. The CNOT gate (controlled-NOT gate) is used to create this entanglement between qubits.- CNOT Gate: The CNOT gate flips the state of the target qubit if the control qubit is in state $|1\rangle$.

The transformation is given by:

$$\text{CNOT}(|00\rangle) = |00\rangle, \text{CNOT}(|01\rangle) = |01\rangle$$

$$\text{CNOT}(|10\rangle) = |11\rangle, \text{CNOT}(|11\rangle) = |10\rangle$$

When combined with the Hadamard gate, it creates an entangled Bell state:

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

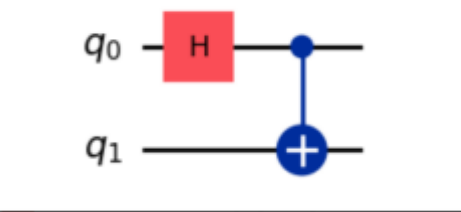
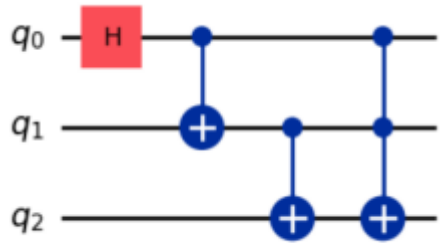

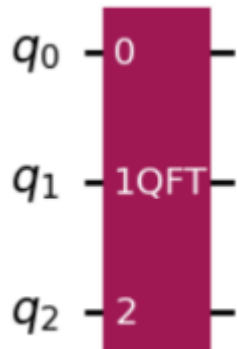
9.4: Measurement: Measuring a qubit collapses its quantum state into one of the basis states $|0\rangle$ or $|1\rangle$ with probabilities related to the coefficients

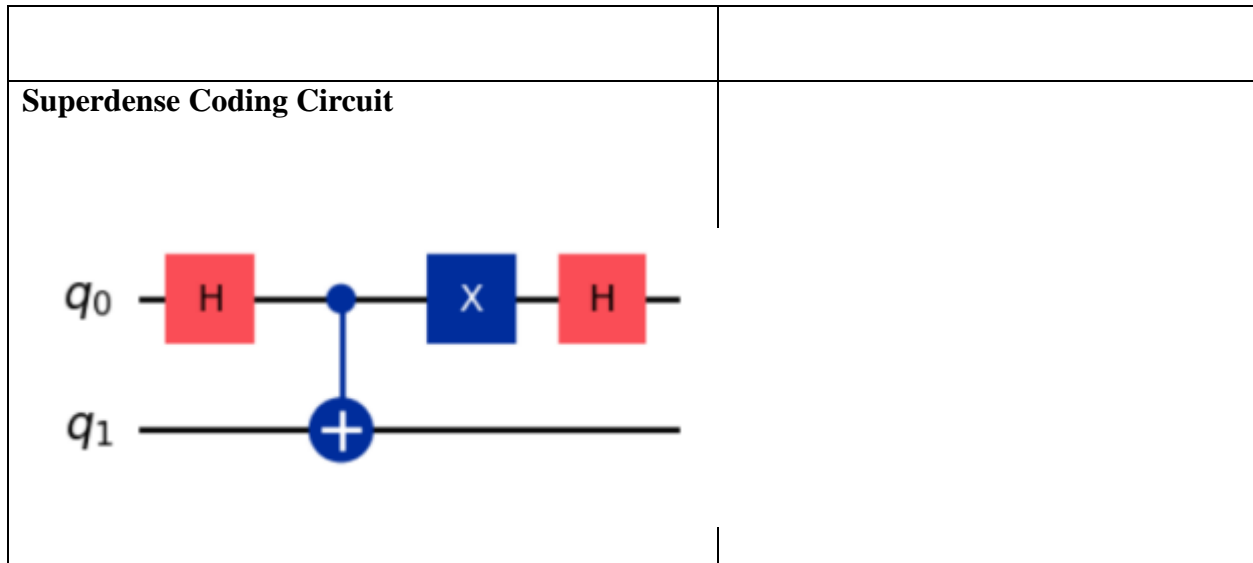
IMPLEMENTATION:

<p>Basic Quantum Circuit with Hadamard and CNOT Gates</p> <p>Basic Quantum Circuit with Hadamard and CNOT Gates:</p> <pre> from qiskit import QuantumCircuit circuit = QuantumCircuit(2) circuit.h(0) # Apply Hadamard gate to qubit 0 circuit.cx(0, 1) # Apply CNOT gate with qubit 0 as control and qubit 1 as target circuit.draw('mpl') </pre>	<p>Quantum Circuit with Toffoli Gate (CCX Gate)</p> <p>Quantum Circuit with Toffoli Gate (CCX Gate):</p> <pre> from qiskit import QuantumCircuit circuit = QuantumCircuit(3) circuit.h(0) circuit.cx(0, 1) circuit.cx(1, 2) circuit.ccx(0, 1, 2) # Apply Toffoli gate circuit.draw('mpl') </pre>
<p>Quantum Circuit with All Pauli Gates:</p> <p>Quantum Circuit with All Pauli Gates:</p> <pre> from qiskit import QuantumCircuit circuit = QuantumCircuit(2) circuit.h(0) circuit.x(0) # Apply X gate circuit.y(1) # Apply Y gate circuit.z(0) # Apply Z gate </pre>	<p>Quantum Circuit with Quantum Fourier Transform</p> <p>Quantum Circuit with Quantum Fourier Transform:</p> <pre> from qiskit import QuantumCircuit from qiskit.circuit.library import QFT circuit = QuantumCircuit(3) circuit.append(QFT(3), range(3)) </pre>

<code>circuit.draw('mpl')</code>	<code>circuit.draw('mpl')</code>
<p>Superdense Coding Circuit</p> <p>Superdense Coding Circuit:</p> <pre> from qiskit import QuantumCircuit circuit = QuantumCircuit(2) circuit.h(0) circuit.cx(0, 1) circuit.x(0) circuit.h(0) circuit.draw('mpl') </pre>	

RESULT:

<p>Basic Quantum Circuit with Hadamard and CNOT Gates</p> 	<p>Quantum Circuit with Toffoli Gate (CCX Gate)</p> 
<p>Quantum Circuit with All Pauli Gates:</p> 	<p>Quantum Circuit with Quantum Fourier Transform</p> 



CONCLUSION: Quantum circuits can be implemented in Python using the `qiskit` library to simulate quantum operations. In this experiment, the Hadamard and CNOT gates were used to create an entangled state between two qubits, showing the fundamental behavior of quantum entanglement.

EXPERIMENT NO 10

TITLE: Case study: Ibm cloud-related quantum computing

AIM: To analyze and evaluate IBM's role in the development and deployment of quantum computing technologies through its cloud services. Specifically, it seeks to understand how IBM integrates quantum computing into its cloud infrastructure, the implications for various industries, and the challenges and opportunities that arise from this integration.

OBJECTIVE: IBM has emerged as a key player in the development and deployment of quantum computing technologies, particularly through its cloud services. Below is an analysis of IBM's role, the integration of quantum computing into its cloud infrastructure, implications for various industries, and the challenges and opportunities that arise from this integration

- **Pioneering Research:**

IBM has been at the forefront of quantum computing research for decades, developing superconducting qubits and quantum algorithms. The IBM Quantum Experience, launched in 2016, allowed researchers and developers to access quantum computers via the cloud, fostering collaboration and innovation.

- **IBM Quantum System One:**

This was one of the first commercially available quantum computers, designed for both research and commercial use. It highlights IBM's commitment to creating scalable quantum systems.

- **IBM Quantum Network:**

A collaboration network that connects academic institutions, startups, and corporations, allowing them to work on quantum applications and share insights.

THEORY:

- **IBM Cloud Quantum Services:**

- IBM has integrated quantum computing into its cloud infrastructure through IBM Quantum, which provides access to quantum processors, simulators, and tools for developing quantum applications.
- Users can run experiments on real quantum hardware or simulate quantum circuits on classical computers, facilitating research and development.

- **Qiskit:**

- IBM's open-source quantum computing framework, Qiskit, allows users to create quantum programs and run them on IBM's quantum devices. This platform supports various tasks, from basic circuit creation to complex algorithms.
- **Hybrid Quantum-Classical Models:**
 - IBM promotes hybrid quantum-classical computing, where quantum algorithms work alongside classical ones to solve complex problems more efficiently. This approach leverages existing classical infrastructure while transitioning to quantum capabilities

IMPLEMENTATION (Basic Linear Algebra Operations in Python using NumPy)

```
import numpy as np

# Create matrices

A = np.array([[1, 2], [3, 4]])

B = np.array([[5, 6], [7, 8]])

# Matrix Addition

C = A + B

print("Matrix Addition:\n", C)

# Matrix Multiplication

D = A @ B # or np.dot(A, B)

print("Matrix Multiplication:\n", D)

# Transpose of A

A_transpose = A.T

print("Transpose of A:\n", A_transpose)

# Determinant of A

det_A = np.linalg.det(A)

print("Determinant of A:", det_A)

# Inverse of A
```

```

A_inverse = np.linalg.inv(A)

print("Inverse of A:\n", A_inverse)

# Pauli-X Gate example

X_gate = np.array([[0, 1], [1, 0]])

initial_state = np.array([[1], [0]]) # |0>

# Apply the Pauli-X gate

final_state = X_gate @ initial_state

print("Final State after Pauli-X Gate:\n", final_state)

```

RESULT: (Basic Linear Algebra Operations in Python using NumPy)

```

⇒ Matrix Addition:
  [[ 6  8]
  [10 12]]
Matrix Multiplication:
  [[19 22]
  [43 50]]
Transpose of A:
  [[1 3]
  [2 4]]
Determinant of A: -2.0000000000000004
Inverse of A:
  [[-2.   1. ]
  [ 1.5 -0.5]]
Final State after Pauli-X Gate:
  [[0]
  [1]]

```

CONCLUSION: IBM is a leading force in the development and deployment of quantum computing through its cloud services. By integrating quantum capabilities into its infrastructure, IBM opens up opportunities for various industries to leverage this technology, while also facing challenges that need to be addressed. As quantum computing matures, its potential applications are vast, and IBM's proactive approach positions it well for future advancements in this exciting field.