CSBB 311 : QUANTUM COMPUTING

LAB PRACTICAL FILE

Submitted By:

Name: KARTIK MITTAL

Roll No: 221210056

Branch: CSE

Semester: 5th Sem

Group: 2

Submitted To: Dr. VS Pandey

Department of Computer Science and Engineering

NATIONAL INSTITUTE OF TECHNOLOGY DELHI

2024

INDEX TABLE

ASSIGNMENT - 1

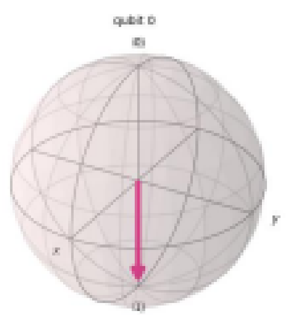
Aim: Develop circuits to execute on them with Python and Qiskit.

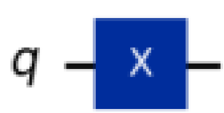
Source Code:

```
# Import necessary libraries
from qiskit import QuantumCircuit, transpile, assemble
from qiskit_aer import Aer
from qiskit.visualization import plot_bloch_vector, plot_histogram
from qiskit.visualization import plot_bloch_multivector
from qiskit.quantum_info import Statevector
import matplotlib.pyplot as plt
import numpy as np

# Function to display a quantum circuit
def show_circuit(qc):
    qc.draw(output='mpl')
    plt.show()
```

1. Pauli-X (NOT) Gate

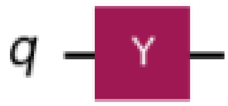
```
# 1. Pauli-X (NOT) Gate
description_x = "Pauli-X (NOT) Gate: Flips the qubit state from |0⟩ to |1⟩ or vice versa."
qc_x = QuantumCircuit(1)
qc_x.x(0)  # Apply X gate
explain_gate(qc_x, description_x)
```

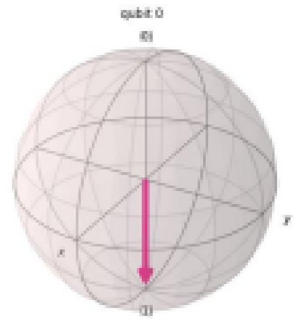




2. Pauli-Y (NOT) Gate

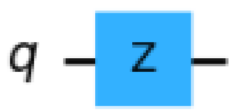
```
# 2. Pauli-Y Gate
description_y = "Pauli-Y Gate: Rotates the qubit around the Y-axis of the Bloch sphere by π radians."
qc_y = QuantumCircuit(1)
qc_y.y(0)  # Apply Y gate
explain_gate(qc_y, description_y)
```

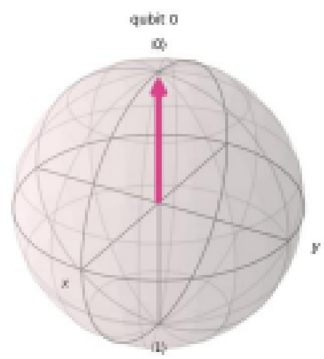




3. Pauli-Z (NOT) Gate

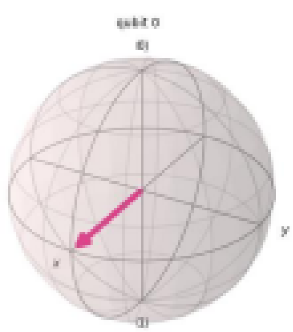
```
# 3. Pauli-Z Gate
description_z = "Pauli-Z Gate: Applies a phase flip to the |1⟩ state, leaving |0⟩ unchanged."
qc_z = QuantumCircuit(1)
qc_z.z(0)  # Apply Z gate
explain_gate(qc_z, description_z)
```

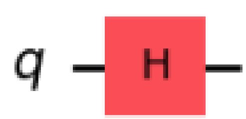




4. Hadamard Gate

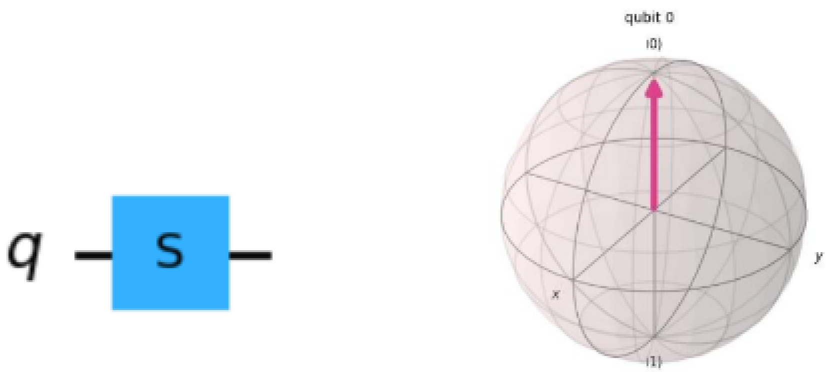
```
# 4. Hadamard Gate
description_h = "Hadamard Gate: Puts the qubit into a superposition, equally likely to be measured as |0⟩ or |1⟩."
qc_h = QuantumCircuit(1)
qc_h.h(0)  # Apply H gate
explain_gate(qc_h, description_h)
```





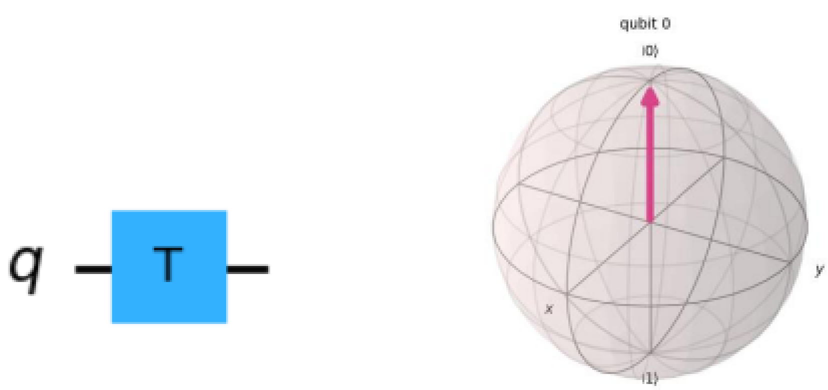
5. Phase (S) Gate

```
# 5. Phase (S) Gate
description_s = "Phase (S) Gate: Adds a phase of π/2 to the qubit's |1⟩ state."
qc_s = QuantumCircuit(1)
qc_s.s(0)  # Apply S gate
explain_gate(qc_s, description_s)
```



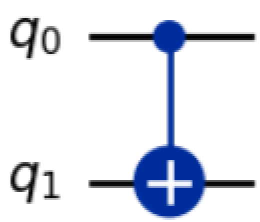
6. T Gate

```
# 6. T Gate
description_t = "T Gate: Adds a phase of π/4 to the |1⟩ state."
qc_t = QuantumCircuit(1)
qc_t.t(0)  # Apply T gate
explain_gate(qc_t, description_t)
```



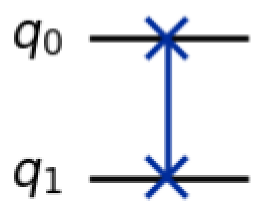
7. Controlled-NOT (CNOT) Gate (2 qubits: control and target)

```python
# 7. Controlled-NOT (CNOT) Gate (2 qubits: control and target)
description_cnot = "CNOT Gate: Flips the target qubit if the control qubit is |1⟩."
qc_cnot = QuantumCircuit(2)
qc_cnot.cx(0, 1)  # Apply CNOT gate
explain_gate(qc_cnot, description_cnot)
```



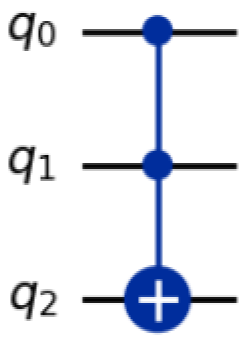
8. SWAP Gate (2 qubits)

```python
# 8. SWAP Gate (2 qubits)
description_swap = "SWAP Gate: Swaps the states of two qubits."
qc_swap = QuantumCircuit(2)
qc_swap.swap(0, 1)  # Apply SWAP gate
explain_gate(qc_swap, description_swap)
```



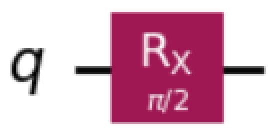
9. Toffoli (CCNOT) Gate (3 qubits: 2 control, 1 target)

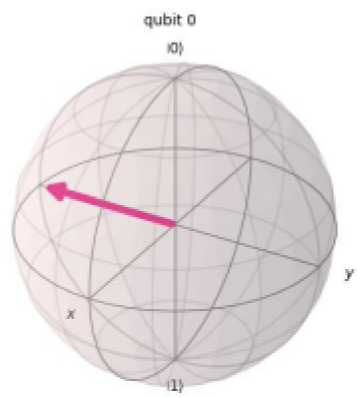
```python
# 9. Toffoli (CCNOT) Gate (3 qubits: 2 control, 1 target)
description_toffoli = "Toffoli (CCNOT) Gate: Flips the target qubit if both control qubits are in the |1⟩ state."
qc_toffoli = QuantumCircuit(3)
qc_toffoli.ccx(0, 1, 2)  # Apply Toffoli gate
explain_gate(qc_toffoli, description_toffoli)
```



10. Rotation-X Gate (Rx)

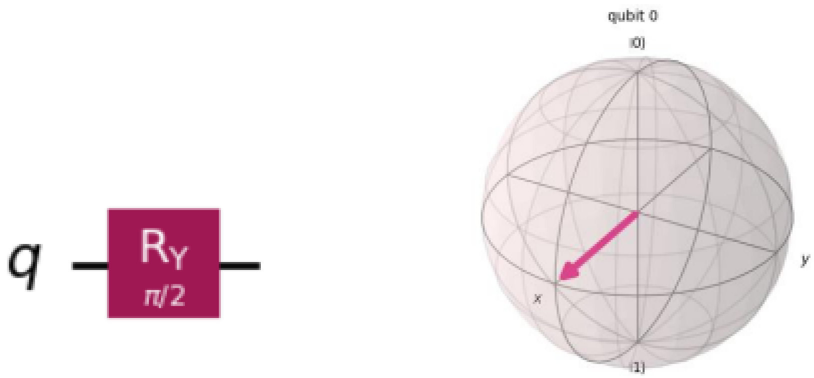
```
# 10. Rotation-X Gate (Rx)
description_rx = "Rotation-X Gate: Rotates the qubit around the X-axis by a given angle (here, π/2)."
qc_rx = QuantumCircuit(1)
qc_rx.rx(np.pi / 2, 0)  # Rotate around X-axis by π/2
explain_gate(qc_rx, description_rx)
```





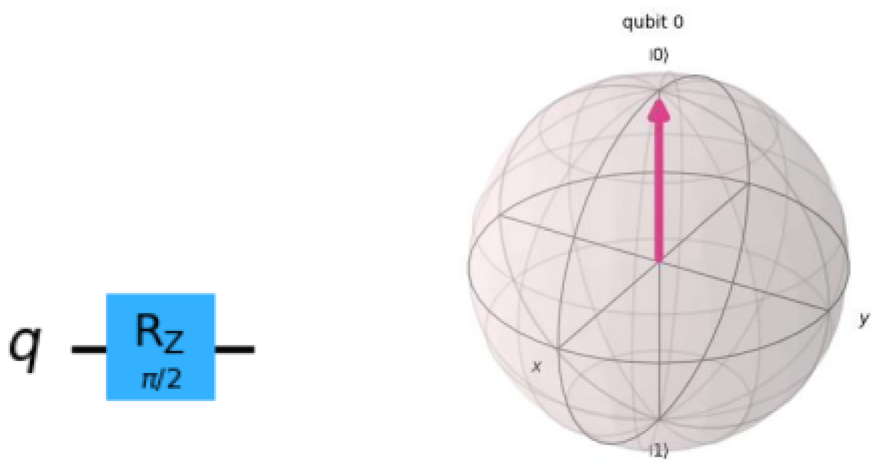
11. Rotation-Y Gate (Ry)

```
# 11. Rotation-Y Gate (Ry)
description_ry = "Rotation-Y Gate: Rotates the qubit around the Y-axis by a given angle (here, π/2)."
qc_ry = QuantumCircuit(1)
qc_ry.ry(np.pi / 2, 0)  # Rotate around Y-axis by π/2
explain_gate(qc_ry, description_ry)
```



12. Rotation-Z Gate (Rz)

```
# 12. Rotation-Z Gate (Rz)
description_rz = "Rotation-Z Gate: Rotates the qubit around the Z-axis by a given angle (here, π/2)."
qc_rz = QuantumCircuit(1)
qc_rz.rz(np.pi / 2, 0)  # Rotate around Z-axis by π/2
explain_gate(qc_rz, description_rz)
```



ASSIGNMENT - 2

Theory -

1. **Introduction to Quantum Measurement**

- **Quantum Measurement** is the process of observing a qubit's state, collapsing it from superposition to a classical state (|0⟩ or |1⟩).
- This process is essential for extracting output from quantum algorithms, as qubits exist in superposition before measurement.

2. Superposition and Quantum Gates

- A **qubit** can exist in superposition, represented as a linear combination of |0⟩ and |1⟩.
- The **Hadamard gate (H)** is commonly used to create superposition, applied via qc.h(qubit_index).

3. Basic Workflow for Quantum Measurement in Qiskit

- **Create a quantum circuit**: Initialize with qubits and classical bits.
- **Measure the qubits**: Collapse quantum states into classical bits.
- **Simulate the circuit**: Use AerSimulator for execution.
- **Retrieve and visualize results**: Use histograms to display measurement outcomes.

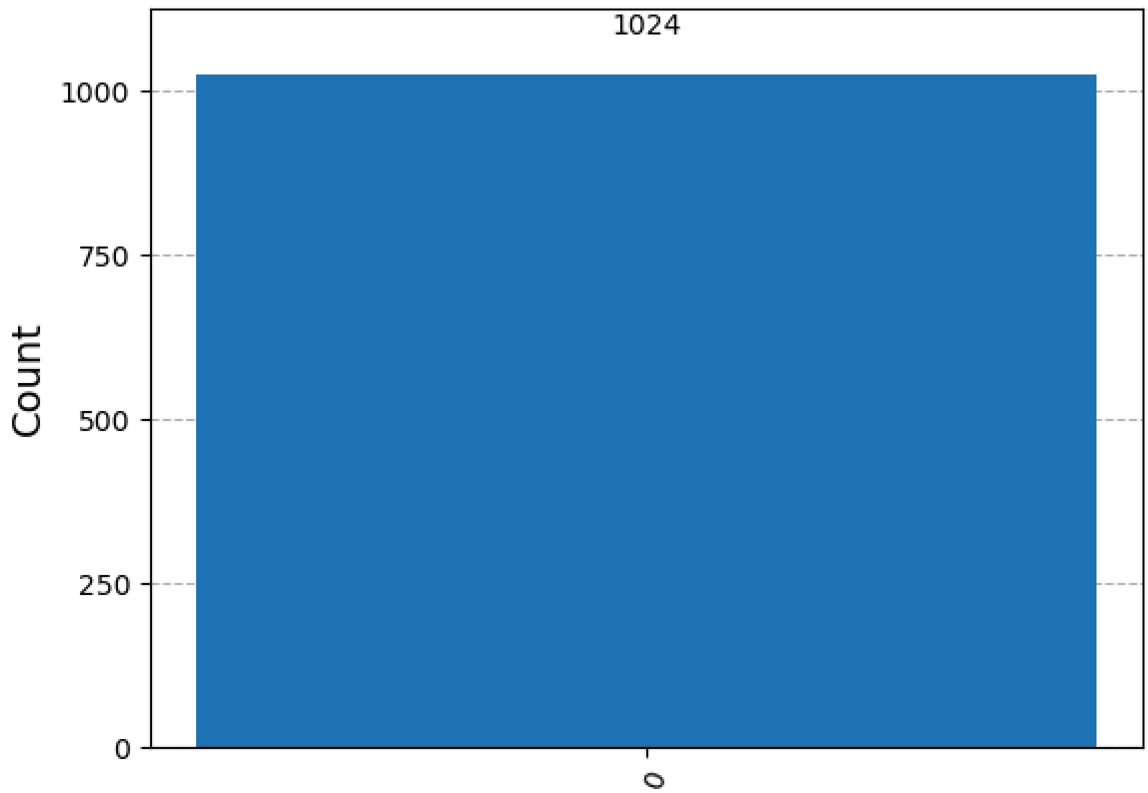
4. Importance of Quantum Measurement

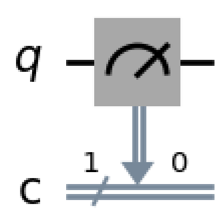
- Quantum measurement bridges the quantum and classical worlds, making it essential for any quantum computing task.
- The probabilistic outcomes underscore the distinction between classical and quantum computation.

Code -

```
from qiskit import QuantumCircuit
from qiskit_aer import Aer
from qiskit import transpile
from qiskit.visualization import plot_histogram
from qiskit_aer import AerSimulator
import matplotlib.pyplot as plt

# Create a quantum circuit with 1 qubit and 1 classical bit
qc = QuantumCircuit(1, 1)

# Measure the qubit in its initial |0⟩ state
qc.measure(0, 0)

# Draw the circuit
qc.draw('mpl')

# Simulate the circuit
backend = AerSimulator()
compiled_circuit = transpile(qc, backend)
result = backend.run(compiled_circuit, shots=1024).result()
counts = result.get_counts()

# Plot the measurement result
plot_histogram(counts)
plt.show()
```

Output -

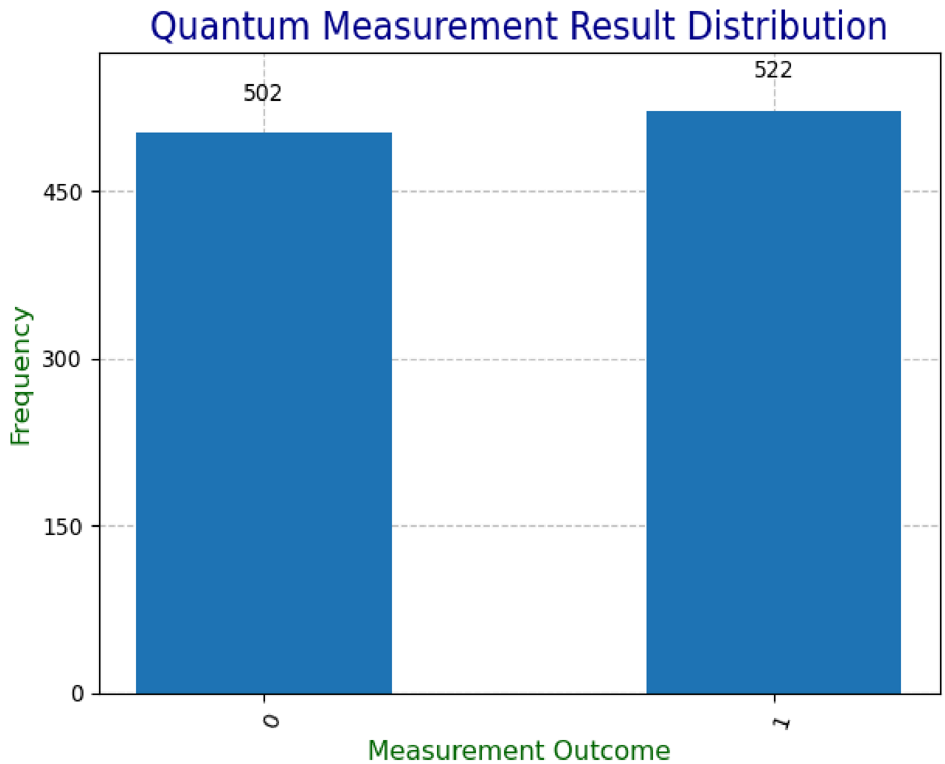


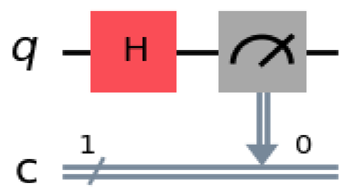


Code -

```
from qiskit import QuantumCircuit, transpile
from qiskit.visualization import plot_histogram
from qiskit_aer import AerSimulator
import matplotlib.pyplot as plt

# Step 1: Initialize a quantum circuit with 1 qubit and 1 classical bit
circuit = QuantumCircuit(1, 1)

# Step 2: Introduce superposition by applying a Hadamard gate to the qubit
circuit.h(0)

# Step 3: Measure the qubit and map the result to the classical bit
circuit.measure(0, 0)

# Step 4: Visualize the quantum circuit (updated object name for uniqueness)
circuit.draw(output='mpl')
plt.show()

# Step 5: Use the AerSimulator to simulate the circuit with 1024 shots
simulator_backend = AerSimulator()
transpiled_circuit = transpile(circuit, simulator_backend)
simulation_result = simulator_backend.run(transpiled_circuit, shots=1024).result()

# Step 6: Retrieve and store the measurement outcomes
measurement_counts = simulation_result.get_counts()

# Step 7: Visualize the measurement results in a histogram (with adjusted variable names)
plot_histogram(measurement_counts)
plt.show()
```

Output -





Conclusion -

- Qiskit enables the construction and simulation of quantum circuits , allowing experimentation with quantum phenomena.
- Quantum measurement converts quantum information into classical information, vital for practical applications.

ASSIGNMENT - 3

Theory -

1. Introduction to Quantum Phase Estimation

- Quantum Phase Estimation (QPE) is a fundamental quantum algorithm used to estimate the phase (θ) of an eigenvalue associated with an eigenstate of a unitary operator.
- It is a critical algorithm for many applications in quantum computing, including factoring, Shor's algorithm, and quantum simulations

2. Phase Estimation and Accuracy

- The accuracy of QPE depends on the number of qubits used for estimation. More qubits allow for a finer resolution of the estimated phase, leading to higher precision.
- The algorithm determines θ by estimating the binary fraction of the phase, where the precision improves exponentially with the number of qubits.

3. Basic Workflow for Quantum Phase Esimation

- **Create the quantum circuit:** Initialize qubits for phase estimation and one target qubit for the unitary operation.
- **Apply Hadamard gates:** Prepare the qubits in superposition using Hadamard gates.
- **Controlled Unitary operations:** Apply the controlled-U operations based on the unitary operator associated with the phase θ.
- **Simulate and analyze:** Use simulators to execute the circuit and retrieve the measurement results.

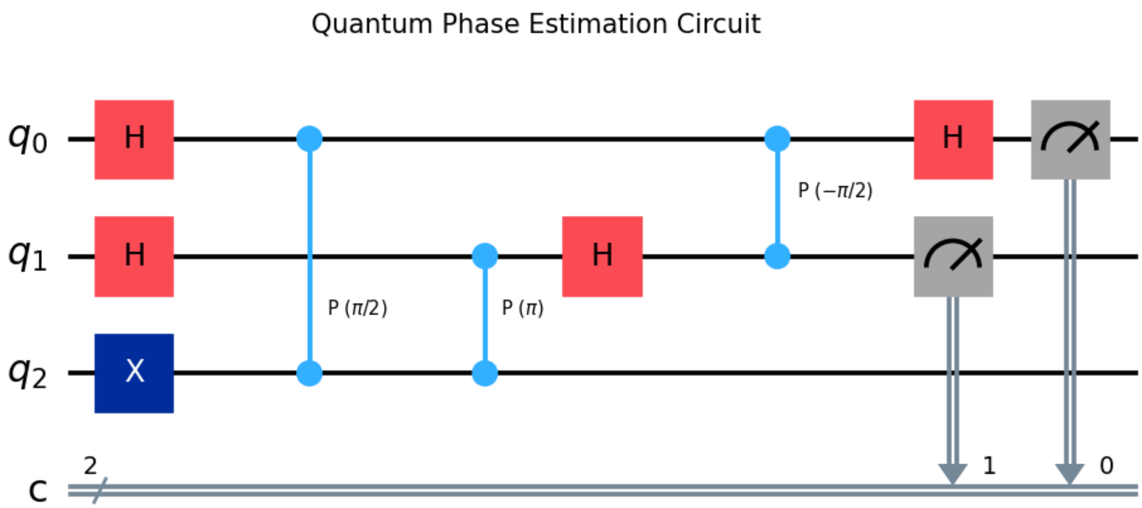
4. Importance of Accuracy in Quantum Phase Estimation

- The accuracy of QPE plays a vital role in determining the effectiveness of algorithms relying on phase estimation, such as quantum simulations and cryptographic algorithms.

Code -

```
# Import required libraries
from qiskit import QuantumCircuit, transpile
from qiskit_aer import Aer
from numpy import pi
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt

# Define the unitary operator (U)
theta = 1/4  # The phase theta we want to estimate (e.g., 1/4)

# Create quantum circuit for QPE with 2 qubits for estimation and 1 target qubit
qpe_circuit = QuantumCircuit(3, 2)

# Prepare the eigenvector |psi> (the last qubit)
qpe_circuit.h([0, 1])  # Apply Hadamard gates to the first 2 qubits
qpe_circuit.x(2)       # Set the target qubit to |1>

# Apply controlled-U gates
qpe_circuit.cp(2 * pi * theta, 0, 2)  # Controlled-U with theta applied to qubit 0
qpe_circuit.cp(4 * pi * theta, 1, 2)  # Controlled-U^2 with theta applied to qubit 1

# Inverse Quantum Fourier Transform (simplified for 2 qubits)
qpe_circuit.h(1)
qpe_circuit.cp(-pi/2, 0, 1)  # Controlled Phase shift between qubit 0 and qubit 1
qpe_circuit.h(0)

# Measure the first two qubits
qpe_circuit.measure([0, 1], [0, 1])

# Transpile the circuit for the 'qasm_simulator' backend
simulator = Aer.get_backend('qasm_simulator')
transpiled_circuit = transpile(qpe_circuit, simulator)

# Plot the quantum circuit using matplotlib
fig, ax = plt.subplots(figsize=(10, 5))
qpe_circuit.draw(output='mpl', ax=ax)  # Draw the circuit on the specified axes
plt.title('Quantum Phase Estimation Circuit')
plt.show()
```

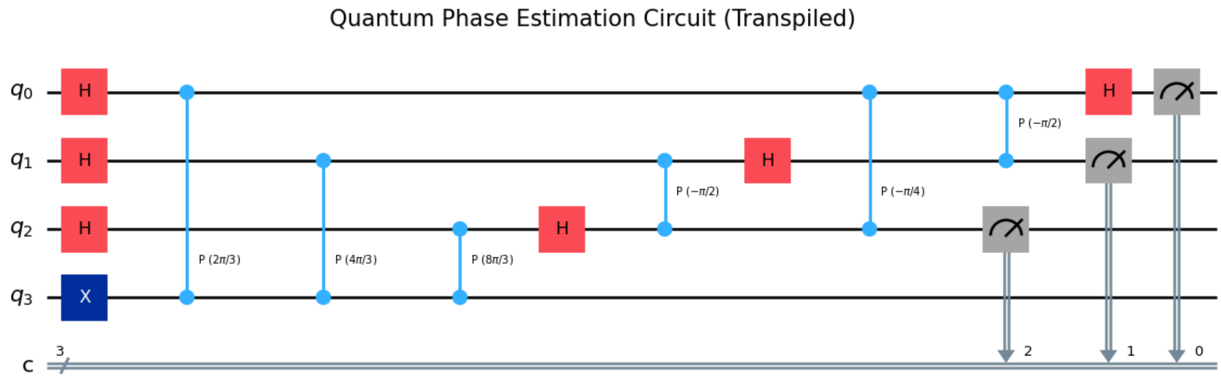
Output -


Quantum Phase Estimation Circuit
q_0
q_1
q_2
c
H
H
X
P (π/2)
P (π)
H
P (−π/2)
H
2
1
0

Code -

```python
# Import required libraries
from qiskit import QuantumCircuit, transpile
from qiskit_aer import Aer
from numpy import pi
import matplotlib.pyplot as plt

# Define the phase theta we want to estimate
theta = 1/3  # Let's assume we want to estimate theta = 1/3

# Create quantum circuit for QPE with 3 qubits for estimation and 1 target qubit
qpe_circuit = QuantumCircuit(4, 3)

# Prepare the eigenvector |psi> (the last qubit)
qpe_circuit.h([0, 1, 2])  # Apply Hadamard gates to the first 3 qubits
qpe_circuit.x(3)          # Set the target qubit to |1>

# Apply controlled-U gates
qpe_circuit.cp(2 * pi * theta, 0, 3)  # Controlled-U with theta applied to qubit 0
qpe_circuit.cp(4 * pi * theta, 1, 3)  # Controlled-U^2 with theta applied to qubit 1
qpe_circuit.cp(8 * pi * theta, 2, 3)  # Controlled-U^4 with theta applied to qubit 2

# Inverse Quantum Fourier Transform (simplified for 3 qubits)
qpe_circuit.h(2)
qpe_circuit.cp(-pi/2, 1, 2)  # Controlled Phase shift between qubit 1 and qubit 2
qpe_circuit.h(1)

qpe_circuit.cp(-pi/4, 0, 2)  # Controlled Phase shift between qubit 0 and qubit 2
qpe_circuit.cp(-pi/2, 0, 1)  # Controlled Phase shift between qubit 0 and qubit 1
qpe_circuit.h(0)

# Measure the first three qubits
qpe_circuit.measure([0, 1, 2], [0, 1, 2])

# Transpile the circuit for the 'qasm_simulator' backend
simulator = Aer.get_backend('qasm_simulator')
transpiled_circuit = transpile(qpe_circuit, simulator)

# Plot the transpiled quantum circuit using matplotlib
fig, ax = plt.subplots(figsize=(12, 6))
qpe_circuit.draw(output='mpl', ax=ax)
plt.title('Quantum Phase Estimation Circuit (Transpiled)')
plt.show()
```

Output -



Conclusion -

- **Precision-Quibit Trade-off:** The accuracy of Quantum Phase Estimation improves with the number of qubits, as more qubits allow for finer phase resolution..
- **Algorithmic Impact:** The accuracy of QPE is crucial for algorithms like Shor's and quantum simulations, as more precise phase estimations lead to better performance and more accurate results in these applications.

ASSIGNMENT - 4

Theory -

1. Introduction to Iterative Quantum Phase Estimation

- Iterative Quantum Phase Estimation (IQPE) is a variant of the Quantum Phase Estimation (QPE) algorithm, designed to determine the phase (θ) of an eigenvalue corresponding to an eigenstate of a given unitary operator with reduced qubit requirements.

2. Iterative Phase Estimation and Precision

- In IQPE, precision is achieved by sequentially estimating the binary digits of the phase θ, from the most significant to the least significant bit. The algorithm accomplishes this through controlled unitary operations and adaptive rotations on a single qubit.

3. Step-by-Step Process in Iterative Quantum Phase Estimation

- **Initialize the Circuit:**Prepare the quantum circuit with a single qubit in the initial state. This qubit is the primary computational resource in IQPE.
- **Apply a Hadamard Gate:** The Hadamard gate creates a superposition, preparing the qubit for iterative phase measurement.
- **Controlled Unitary Operations**: For each bit of precision, apply a controlled unitary operation corresponding to the eigenvalue's unitary matrix raised to powers of two (U, U^2, U^4, etc.)
- **Adaptive Rotation and Measurement**: After each controlled operation, rotate the qubit by an angle proportional to the current phase estimate. Measure the qubit, and based on the result, update the phase estimate for the next iteration.
- **Iterate and Refine the Phase Estimate**: Repeat the steps, updating the phase estimate bit-by-bit until the desired precision is reached.

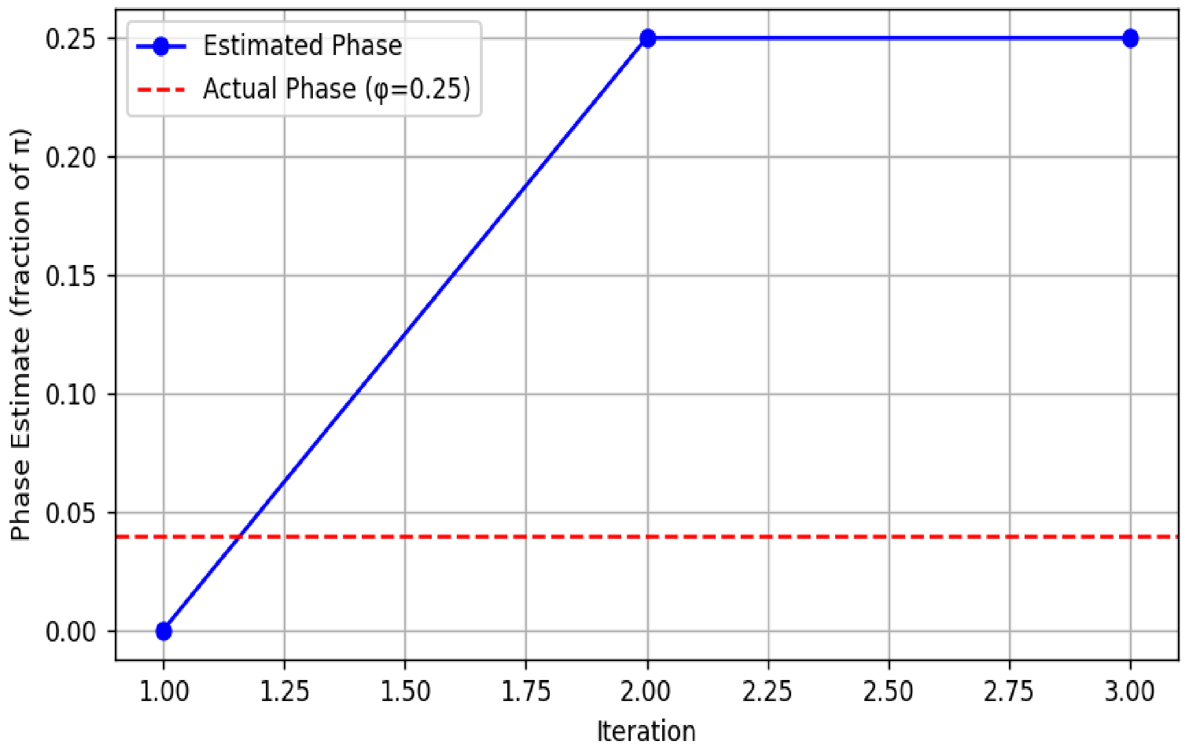
Code -

```
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
import numpy as np
import matplotlib.pyplot as plt

# Define the unitary operator (e.g., Pauli-Z gate with phase)
phi = 0.25  # Known phase to estimate (should be between 0 and 1)
unitary = QuantumCircuit(1)
unitary.p(2 * np.pi * phi, 0)  # Phase gate with known phase

# Iterative Quantum Phase Estimation
def iqpe(unitary, num_iterations):
    phase_estimate = 0
    estimated_phases = []  # List to track phase estimates across iterations
    actual_phase_fraction = phi / (2 * np.pi)  # Actual phase as a fraction of π

    for k in range(num_iterations):
        qc = QuantumCircuit(1, 1)

        # Step 1: Apply Hadamard to prepare superposition
        qc.h(0)

        # Step 2: Apply controlled unitary (U^(2^k))
        power_unitary = unitary.power(2 ** k)
        qc.append(power_unitary.to_instruction(), [0])

        # Step 3: Rotate ancilla qubit by phase angle and measure
        qc.p(-2 * np.pi * phase_estimate * (2 ** k), 0)
        qc.h(0)
        qc.measure(0, 0)

        # Transpile the circuit before running (optimizing for the simulator backend)
        qc_transpiled = transpile(qc, backend=AerSimulator())
```

```python
        # Run the transpiled circuit on the simulator
        simulator = AerSimulator()
        result = simulator.run(qc_transpiled, shots=1024).result()  # Manual run without execute
        counts = result.get_counts()

        # Update phase estimate based on measurement result
        if '1' in counts and counts['1'] > counts.get('0', 0):
            phase_estimate += 1 / (2 ** (k + 1))

        # Append the estimated phase for this iteration
        estimated_phases.append(phase_estimate)

    return estimated_phases, actual_phase_fraction

# Run IQPE with 3 iterations to estimate phase
estimated_phases, actual_phase = iqpe(unitary, 3)

# Plotting the results
iterations = range(1, len(estimated_phases) + 1)
plt.figure(figsize=(8, 5))

# Plot the estimated phases
plt.plot(iterations, estimated_phases, marker='o', label="Estimated Phase", color='b')

# Plot the actual phase (constant line)
plt.axhline(y=actual_phase, color='r', linestyle='--', label="Actual Phase (ϕ=0.25)")

# Formatting the plot
plt.xlabel("Iteration")
plt.ylabel("Phase Estimate (fraction of π)")
plt.title("Convergence of Phase Estimate in IQPE")
plt.legend()
plt.grid(True)
plt.show() # Show the plot

# Print the final results
print(f"Estimated Phase after {len(estimated_phases)} iterations: {estimated_phases[-1]}")
print(f"Actual Phase (as fraction of π): {actual_phase}")
```

Output -

```
Estimated Phase after 3 iterations: 0.25
Actual Phase (as fraction of π): 0.039788735772973836
```



Convergence of Phase Estimate in IQPE

Conclusion -

- **Precision-Iteration Trade-off**: In Iterative Quantum Phase Estimation (IQPE), the accuracy of phase estimation improves with the number of iterations, as each iteration refines the phase estimate with greater precision.
- **Applications and Practical Benefits**: The iterative nature of IQPE makes it well-suited for practical quantum applications, including cryptographic algorithms and quantum simulations, where precise phase estimation enhances the accuracy and performance of results.

ASSIGNMENT - 5

Theory -

1. Introduction to Shor's Algorithm

- Shor's Algorithm is a quantum algorithm for integer factorization, which efficiently finds the prime factors of a given integer N. Its efficiency comes from its ability to solve problems that are considered intractable on classical computers, making it a foundational quantum algorithm with implications for cryptography.

2. Period Finding in Shor's Algorithm

- A central component of Shor's Algorithm is its reliance on quantum period finding, which utilizes Quantum Phase Estimation (QPE) to determine the period of a specific modular exponentiation function.

3. Workflow of Scalable Shor's Algorithm

- **Quantum Circuit Preparation**: Initialize a circuit with minimal qubits for quantum period finding and apply controlled operations.
- **Modular Exponentiation**: Compute the modular exponentiation values on the quantum circuit, which represent the periodic function for factorization.
- **Iterative Phase Estimation**: Use IQPE or a reduced-qubit phase estimation approach to find the period accurately while minimizing qubit resources.
- **Classical Computation**: After retrieving the measurement outcomes, perform classical post-processing to identify factors based on the computed period.

4. Importance of Scalability in Shor's Algorithm

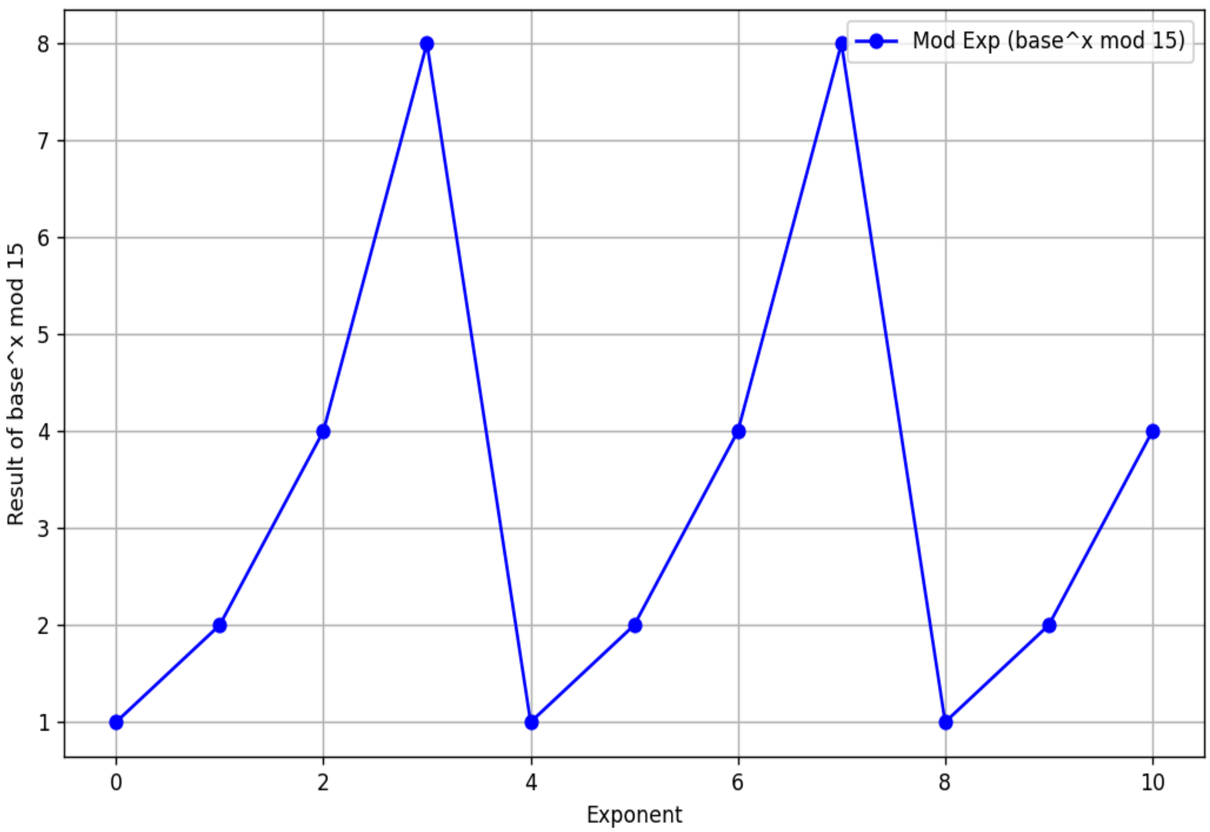
- The scalability of Shor's Algorithm is essential for making it applicable on practical quantum hardware with limited qubit resources.

Code (Modular Exponentiation) -

```
from qiskit import QuantumCircuit
import numpy as np
import matplotlib.pyplot as plt

# Function to perform modular exponentiation
def modular_exponentiation(base, exponent, modulus):
    return pow(base, exponent, modulus)

# Function to demonstrate modular exponentiation and plot the results
def visualize_modular_exponentiation(base, max_exponent, modulus):
    # Initialize a sample quantum circuit (for illustration only, no quantum operations)
    quantum_circuit = QuantumCircuit(4)

    # Compute modular exponentiation for each exponent value from 0 to max_exponent
    results = [modular_exponentiation(base, exp, modulus) for exp in range(max_exponent + 1)]
    print(f"Results of modular exponentiation (base={base}, modulus={modulus}): {results}")

    # Visualize results with a plot
    plt.figure(figsize=(10, 6))
    plt.plot(range(max_exponent + 1), results, marker='o', linestyle='-', color='blue',
             label=f"Mod Exp (base^x mod {modulus})")
    plt.xlabel("Exponent")
    plt.ylabel(f"Result of base^x mod {modulus}")
    plt.legend()
    plt.grid()

    # Show the plot
    plt.show()
    # Adding title below the plot
    plt.figtext(0.5, -0.05, f"Modular Exponentiation for base = {base}, modulus = {modulus}",
                wrap=True, horizontalalignment='center', fontsize=12)

# Example usage
visualize_modular_exponentiation(base=2, max_exponent=10, modulus=15)
```

Output -

```
Results of modular exponentiation (base=2, modulus=15): [1, 2, 4, 8, 1, 2, 4, 8, 1, 2, 4]
```

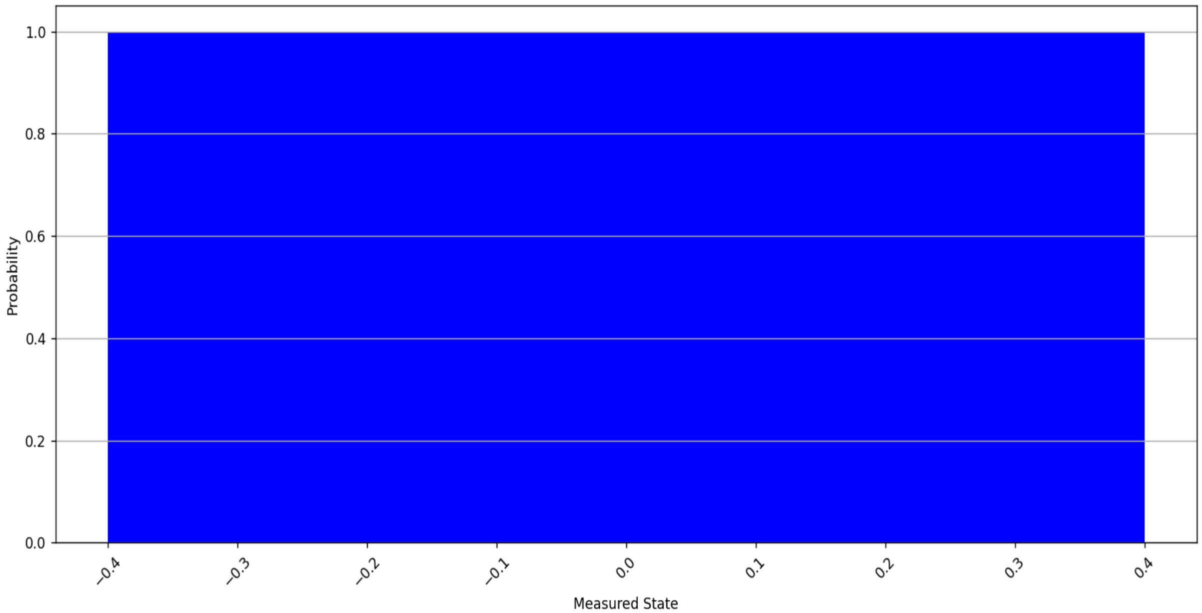


Code (Period Finding) -

```python
from qiskit import QuantumCircuit
from qiskit.primitives import Sampler
import numpy as np
import matplotlib.pyplot as plt

# Define the Quantum Fourier Transform (QFT) function
def apply_qft(circuit, num_qubits):
    for qubit in range(num_qubits):
        circuit.h(qubit)
        for target in range(qubit + 1, num_qubits):
            circuit.cp(np.pi / 2 ** (target - qubit), target, qubit)
    circuit.barrier()

# Construct the quantum circuit for period finding
def perform_quantum_period_finding():
    num_qubits = 3
    circuit = QuantumCircuit(num_qubits)

    # Apply Hadamard gates to all qubits
    circuit.h(range(num_qubits))

    # Apply Quantum Fourier Transform
    apply_qft(circuit, num_qubits)

    # Measure all qubits
    circuit.measure_all()

    # Run the circuit using the Sampler to get the results
    sampler = Sampler()
    job = sampler.run(circuit)
    result = job.result()
    measurement_counts = result.quasi_dists[0]  # Retrieve measurement probabilities

    return measurement_counts

# Execute the period finding circuit
measurement_counts = perform_quantum_period_finding()
print("Measurement results for period finding:", measurement_counts)

# Plot the measurement outcomes
plt.figure(figsize=(8, 5))
plt.bar(measurement_counts.keys(), measurement_counts.values(), color='blue')
plt.xlabel("Measured State")
plt.ylabel("Probability")
plt.title("Measurement Results for Quantum Period Finding Circuit")
plt.xticks(rotation=45)
plt.grid(axis='y')
plt.show()
```

Output -

```
Measurement counts from period finding: {0: 0.9999999999999999}
```



Measurement Results for Quantum Period Finding

Conclusion -

- **Efficiency-Scalability Trade-off**: Scalable Shor's Algorithm optimizes quantum resources, making it feasible to factorize larger integers by balancing quantum gate depth with the number of qubits. This scalability is essential for real-world applications, where efficient use of limited quantum resources is crucial.
- **Algorithmic Significance**:By enhancing factorization capabilities on quantum devices, Scalable Shor's Algorithm serves as a significant step toward practical applications in cryptography and computational number theory, offering a tangible impact on industries relying on large-scale factorization and secure encryption.