Notebook

October 29, 2023

1 Introduction

1.1 Some lecture notes

One online tool very useful to emulate quantum circuits is the IBM Quantum Learning platform.

It is also available a library which deals with quantum mechanics: QuTiP

1.2 Prerequisistes

Here we will implement our first quantum circuit using the python library `Qiskit`. In order to run the code, remember to install the dependencies:

```
pip install qiskit qiskit[visualization] qiskit-aer
```

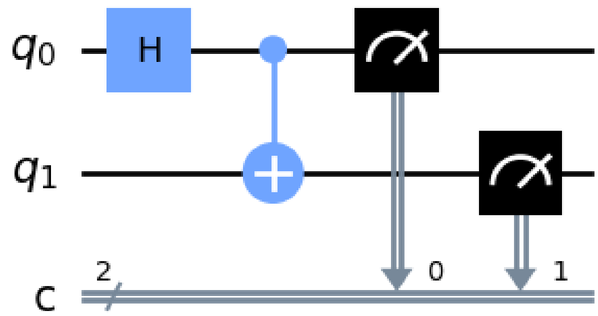
Let's import the function we will use:

[1]:
```python
from qiskit import QuantumCircuit, transpile
from qiskit.providers.aer import QasmSimulator
from qiskit.visualization import plot_histogram
```

[2]:
```python
simulator = QasmSimulator()
# declare a circuit with 2 qubits and 2 classical bits
circuit = QuantumCircuit(2, 2)
```

[3]:
```python
# apply a Hadamard gate to the qubit 0
circuit.h(0)
# apply a C-X gate on quibit 1 using qubit 0 as control
circuit.cx(0, 1)
# measure both qubits, storing values into classical bits
circuit.measure([0, 1], [0, 1])
circuit.draw(output='mpl')
```

[3]:

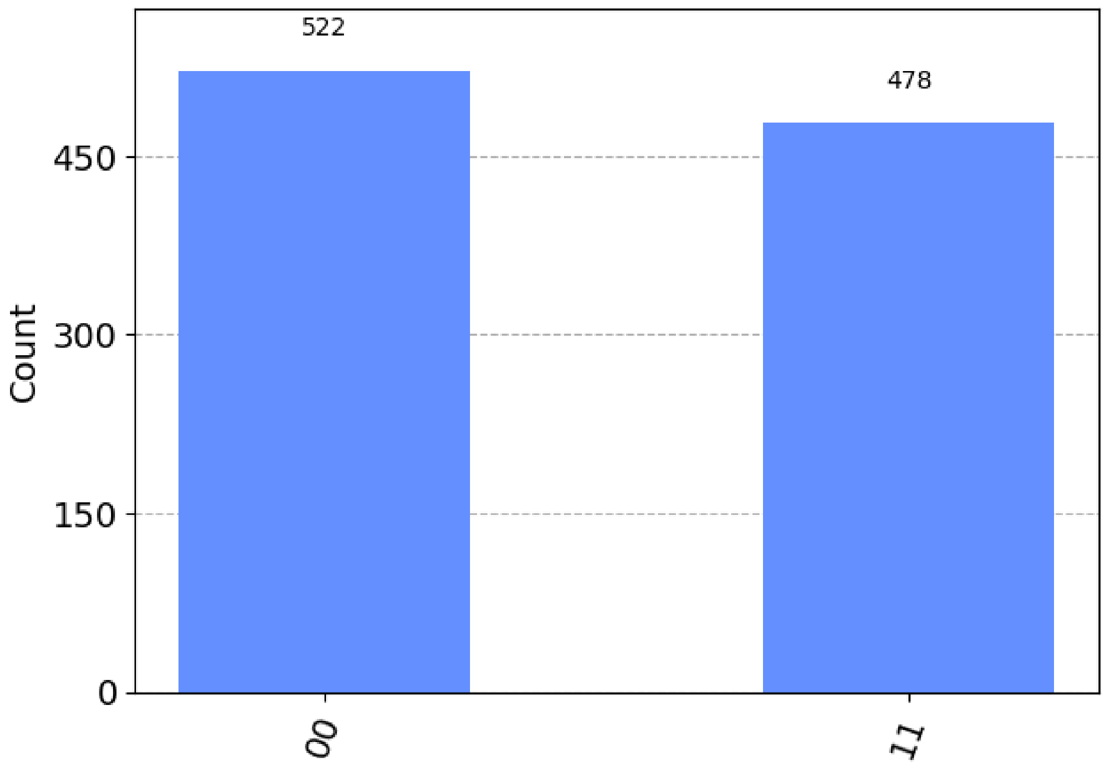


[4]:
```
# transpile (i.e. "compile") the circuit to run on the simulator
compiled_circuit = transpile(circuit, simulator)
# run the circuit on the simulator and get the results
job = simulator.run(compiled_circuit, shots=1000)
result = job.result()
```

[5]:
```
counts = result.get_counts(compiled_circuit)
print("\nTotal count for 00 and 11 are:", counts)

plot_histogram(counts)
```

```
Total count for 00 and 11 are: {'11': 478, '00': 522}
```

[5]:



2 Quantum Teleportation

2.1 The circuit

In this first lab lecture we will see how to simulate quantum teleportation using `Qiskit`.

First, let's create the quantum circuit we need:

[1]:
```
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit

# one register of 3 qubits
qr1 = QuantumRegister(2, name='a')
qr2 = QuantumRegister(1, name='b')
# 2 registers of 1 bit each
cr1 = ClassicalRegister(1, name='cr1')
cr2 = ClassicalRegister(1, name='cr2')
# quantum circuit
teleportation_circuit = QuantumCircuit(qr1, qr2, cr1, cr2)
```

In order to do teleportation we must give Alice and Bob an entangled pair.

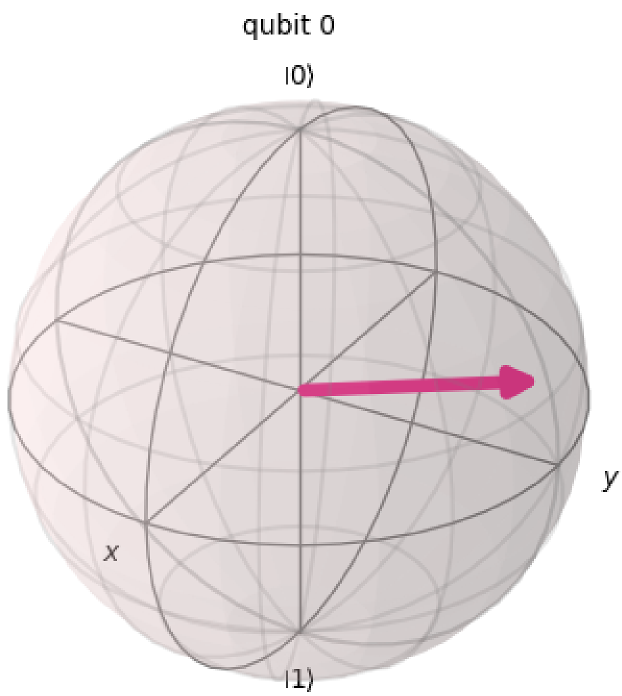
[2]:
```
def bellPair(qc, a, b):
    '''
```

```
    Creates a bell pair in qc using qubits a & b
    '''
    qc.h(a) # Put qubit a into state |+>
    qc.cx(a,b) # CNOT with a as control and b as target
```

[3]:
```
from qiskit.quantum_info import random_statevector
from qiskit.visualization import array_to_latex, plot_bloch_multivector

# Initialize a random state
psi = random_statevector(2)
# Display it nicely
display(array_to_latex(psi, prefix="|\\psi\\rangle ="))
# Show it on a Bloch sphere
plot_bloch_multivector(psi)
```

$$|\psi\rangle = \begin{bmatrix} 0.2504445959 - 0.7703112963i & 0.5580342499 + 0.1802658791i \end{bmatrix}$$

[3]:



Let's initialize $|\psi\rangle$ starting from $|0\rangle$. Notice that `Initialize` is technically not a gate since it contains a reset operation, so it's not invertible.

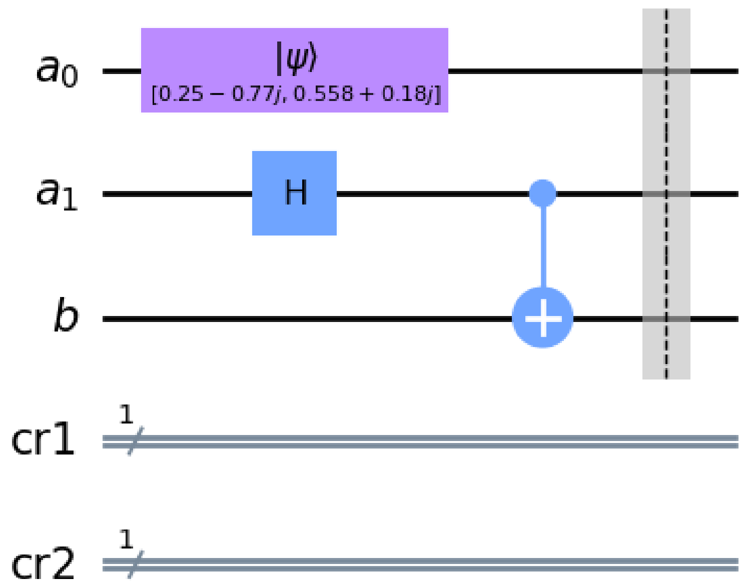
[4]:
```
from qiskit.extensions import Initialize
```

```
teleportation_circuit.append(Initialize(psi), [0])
```

[4]: <qiskit.circuit.instructionset.InstructionSet at 0x7fcb1270e530>

[5]:
```
bellPair(teleportation_circuit, 1, 2)
teleportation_circuit.barrier()
teleportation_circuit.draw(output='mpl')
```

[5]:

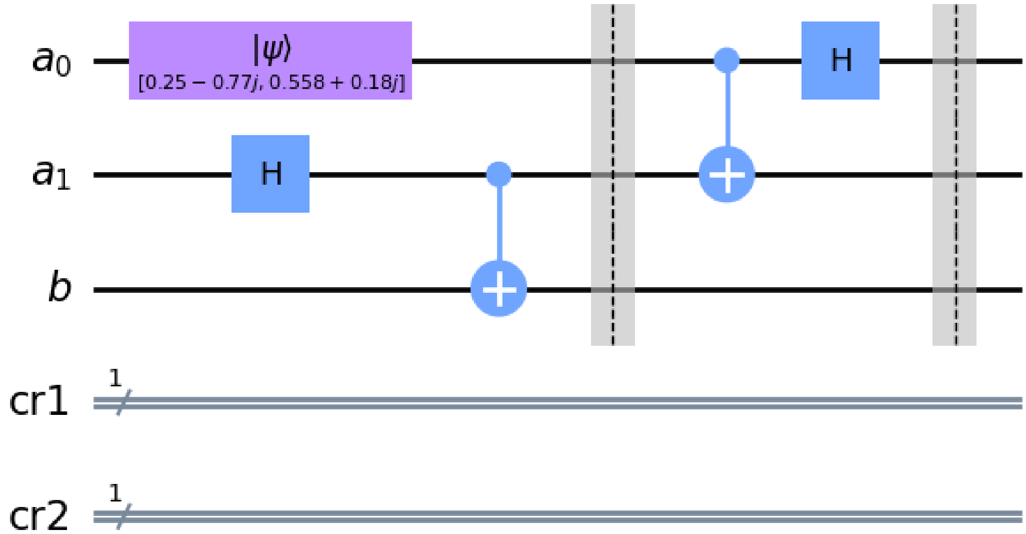


Now Alice and Bob shares the entangled couple $|a_1\rangle$, $|b\rangle$.

The goal of Alice is to send the qubit $|a_0\rangle$ to Bob. In order to do this, Alice will do:

[6]:
```
teleportation_circuit.cx(0,1)
teleportation_circuit.h(0)
teleportation_circuit.barrier()
teleportation_circuit.draw(output='mpl')
```

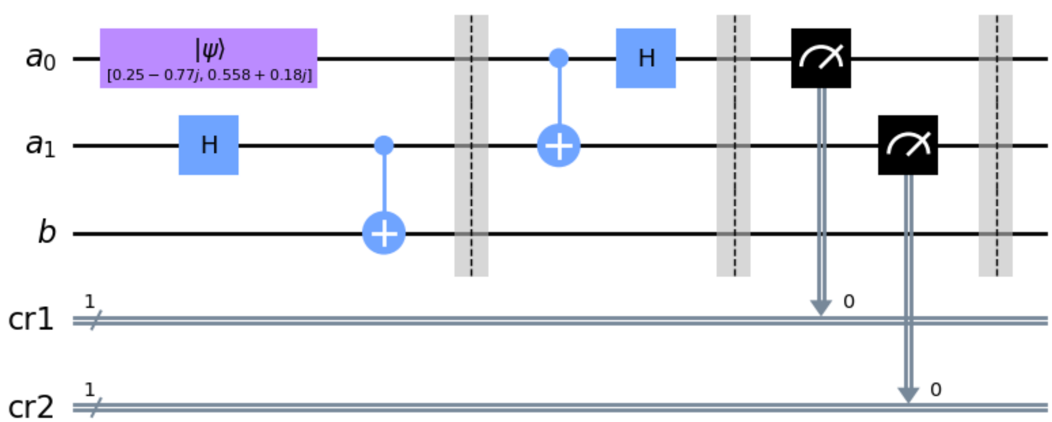
[6]:



Now, Alice needs to send also two classical bits to Bob in order to permit him to adjust the received bit.

[7]:
```
teleportation_circuit.measure(0,0)
teleportation_circuit.measure(1,1)
teleportation_circuit.barrier()
teleportation_circuit.draw(output='mpl')
```

[7]:

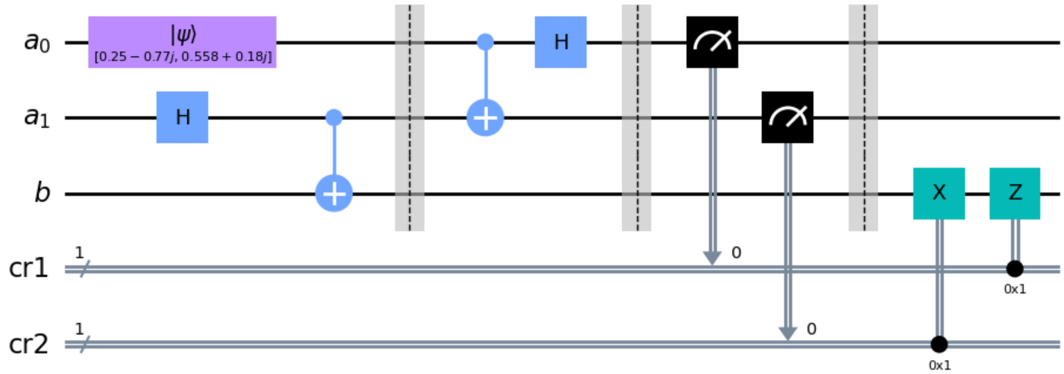


At this moment Bob has all he needs in order to get the original state $|a_0\rangle$.

[8]:

```
teleportation_circuit.x(2).c_if(cr2, 1)
teleportation_circuit.z(2).c_if(cr1, 1)
teleportation_circuit.draw(output='mpl')
```

[8]:

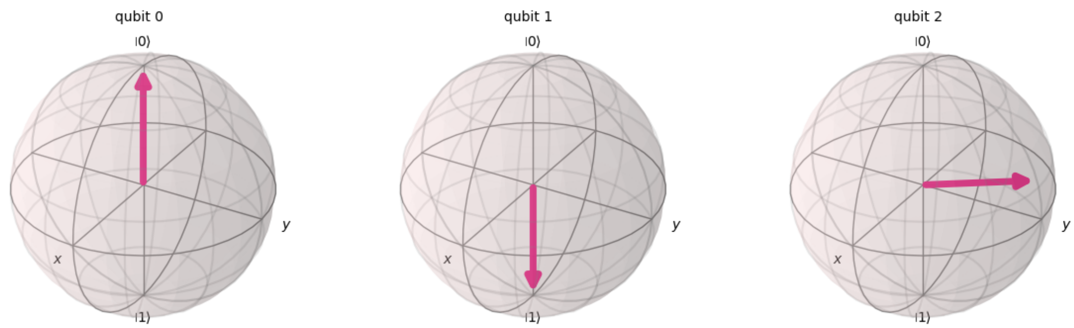


[9]:

```
from qiskit import Aer
import numpy as np

simulator = Aer.get_backend('statevector_simulator')
result = simulator.run(teleportation_circuit).result()
psi_out = result.get_statevector()

display(array_to_latex(psi_out, prefix="|\\psi\\rangle ="))
plot_bloch_multivector(psi_out)
```

$$|\psi\rangle = \begin{bmatrix} 0 & 0 & 0.2504445959 - 0.7703112963i & 0 & 0 & 0 & 0.5580342499 + 0.1802658791i & 0 \end{bmatrix}$$

[9]:



The circuit is complete: Alice's qubit has correctly been sent to Bob.

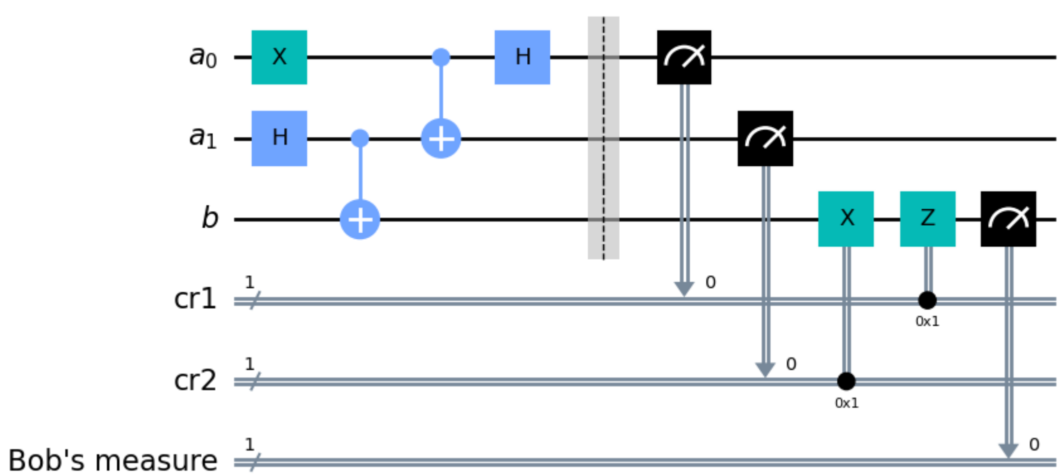
Let's save the circuit in a function, in order to reuse it.

[10]:
```
def tpCircuit(initState = None):
    '''
    Returns a teleportation circuit with the given initial state
    '''
    qr1 = QuantumRegister(2, name='a')
    qr2 = QuantumRegister(1, name='b')
    cr1 = ClassicalRegister(1, name='cr1')
    cr2 = ClassicalRegister(1, name='cr2')
    cr3 = ClassicalRegister(1, name='Bob\'s measure')
    qc = QuantumCircuit(qr1, qr2, cr1, cr2, cr3)
    if initState is not None:
        qc.append(initState, [0])
    else:
        qc.x(0)
    bellPair(qc, 1, 2)
    qc.cx(0,1)
    qc.h(0)
    qc.barrier()
    qc.measure(0,0)
    qc.measure(1,1)
    qc.x(2).c_if(cr2, 1)
    qc.z(2).c_if(cr1, 1)
    return qc
```

2.2 Simulate the teleportation protocol

[11]:
```
qc = tpCircuit()
qc.measure(2, 2)
qc.draw(output='mpl')
```

[11]:



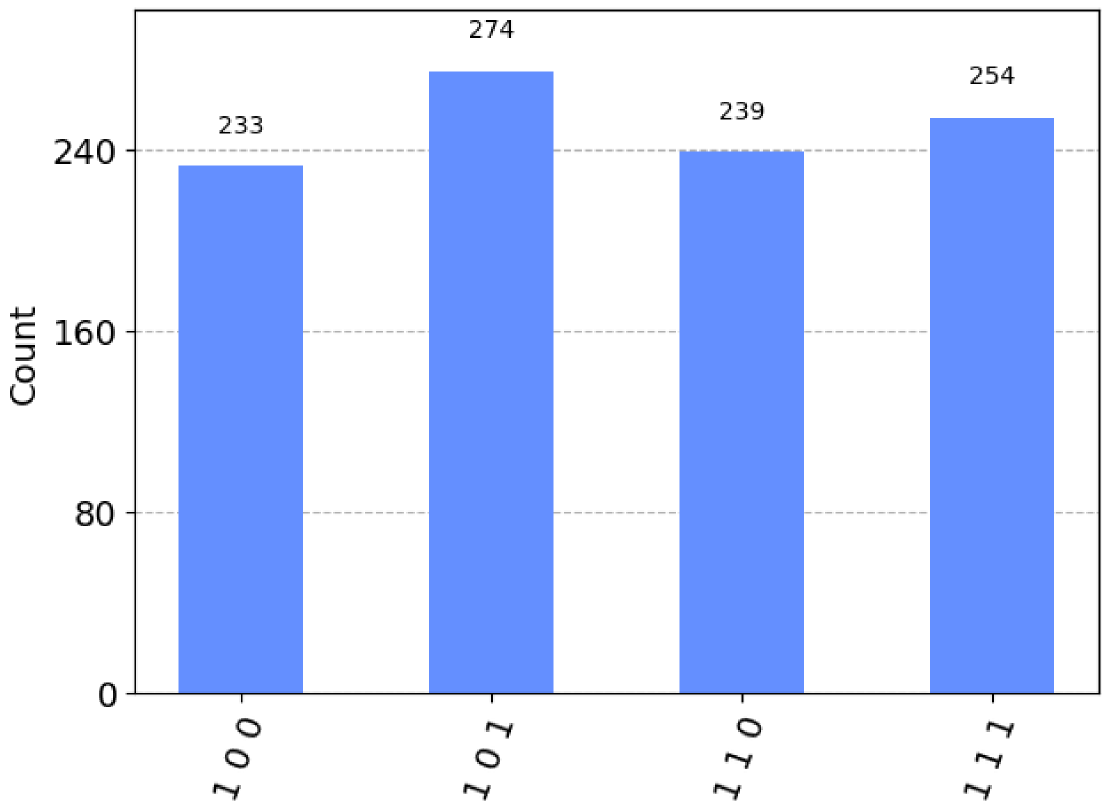
[12]:

```
simulator = Aer.get_backend('aer_simulator')
N = 10**3 # Number of shots
job = simulator.run(qc, shots=N)
```

[13]:

```
from qiskit.visualization import plot_histogram

counts = job.result().get_counts()
plot_histogram(counts)
```

[13]:



As we can see, each string of bits begins with a 1, which means that we have 100% probability to get the one state.

2.3 Teleportation Circuit on a real quantum device

For this last part we are going to use the IBM hardware. In order to do this we should ask pip

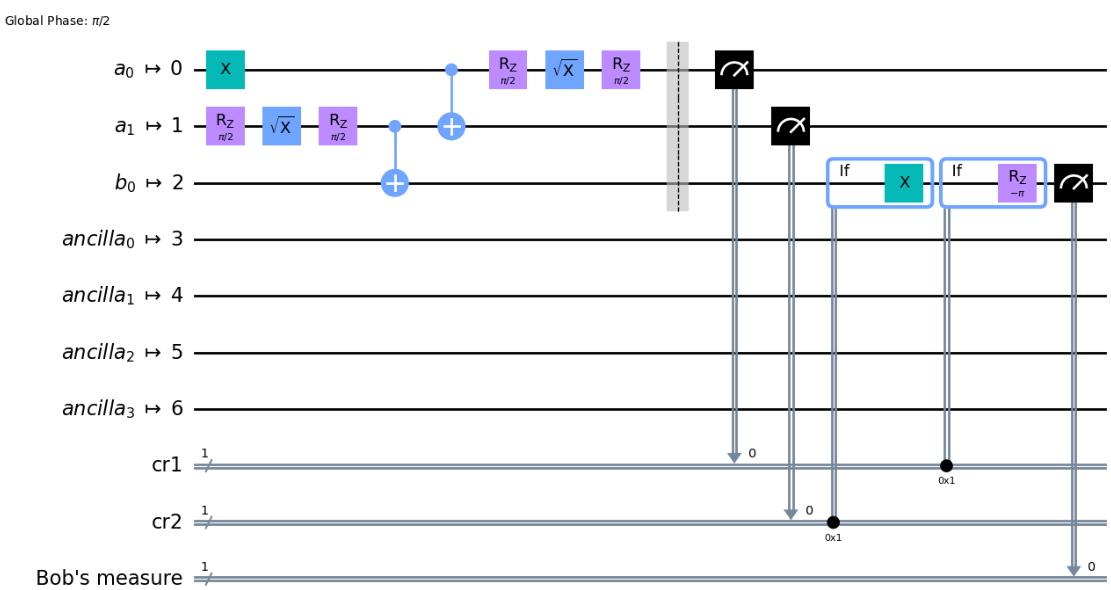
`pip install qiskit-ibm-provider`

[14]:
```
from qiskit_ibm_provider import IBMProvider

with open('token.txt', 'r') as f:
    token = f.read()
IBMProvider.save_account(token=token, overwrite=True)

provider = IBMProvider()
```

Before pushing the code into the IBM device we need to transpile it in order to make it compatible.

[15]:
```
from qiskit import transpile
from qiskit_ibm_provider import least_busy

backend = least_busy(provider.backends(
    simulator=False, operational=True, dynamic_circuits=True, n_qubits=7))


tqc = transpile(qc, backend)
tqc.draw(output='mpl')
```

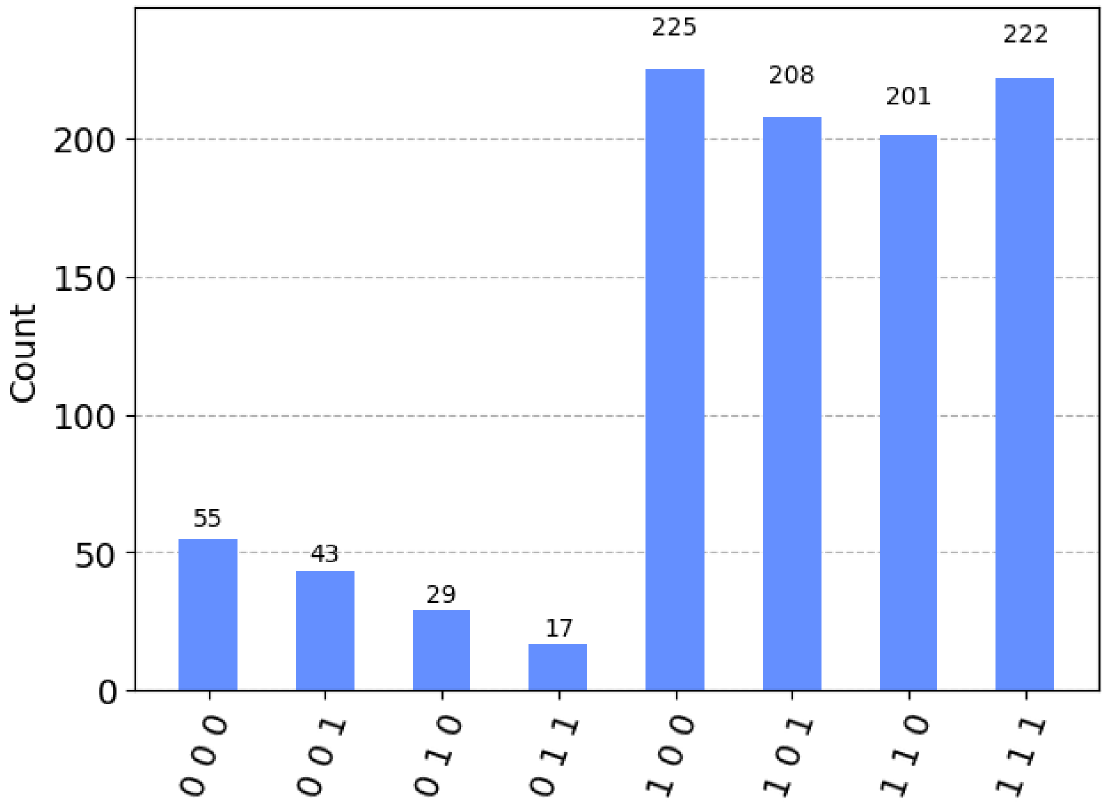
[15]:



Now we can run the job

[16]:
```
job = backend.run(tqc, shots=N, dynamic=True)
result = job.result()

plot_histogram(result.get_counts())
```

[16]:



We can see that sometimes the qubit 2 is measured in $|0\rangle$: this is due to errors in gates/qubits.

3 Quantum Phase Estimation

3.1 The circuit

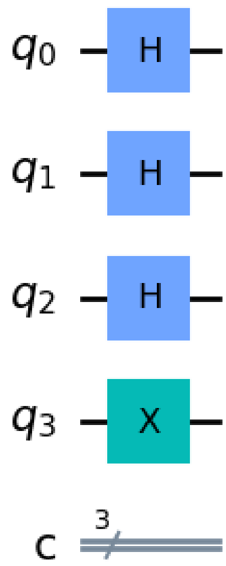
We have a goal: given a unitary operator U, we want to estimate θ from $U|\psi\rangle = e^{2\pi\theta}|\psi\rangle$.

Notice that in this case $|\psi\rangle$ is an eigenvector of eigenvalue $e^{2\pi\theta}$. Since U is unitary, every eigenvalue has norm 1.

We will use three qubits as *counting qubits*, and a fourth one as eigenstate of the unitary operator T. We initialize the last one in $|1\rangle$ by applying the X gate.

[1]:
```
from qiskit import QuantumCircuit

qpe_circuit = QuantumCircuit(4, 3)
for qbit in range(3):
    qpe_circuit.h(qbit)
qpe_circuit.x(3)
qpe_circuit.draw(output='mpl')
```

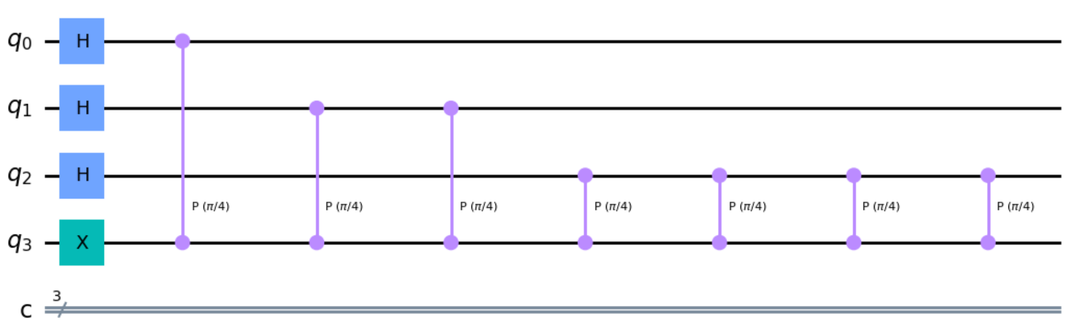
[1]:



Now we want to perform controlled unitary operations

[2]:
```
import numpy as np

repetitions = 1
for counting_qubit in range(3):
    for i in range(repetitions):
        qpe_circuit.cp(np.pi/4, counting_qubit, 3) # This is the Controlled U␣
↪gate
    repetitions *= 2
qpe_circuit.draw(output='mpl')
```

[2]:



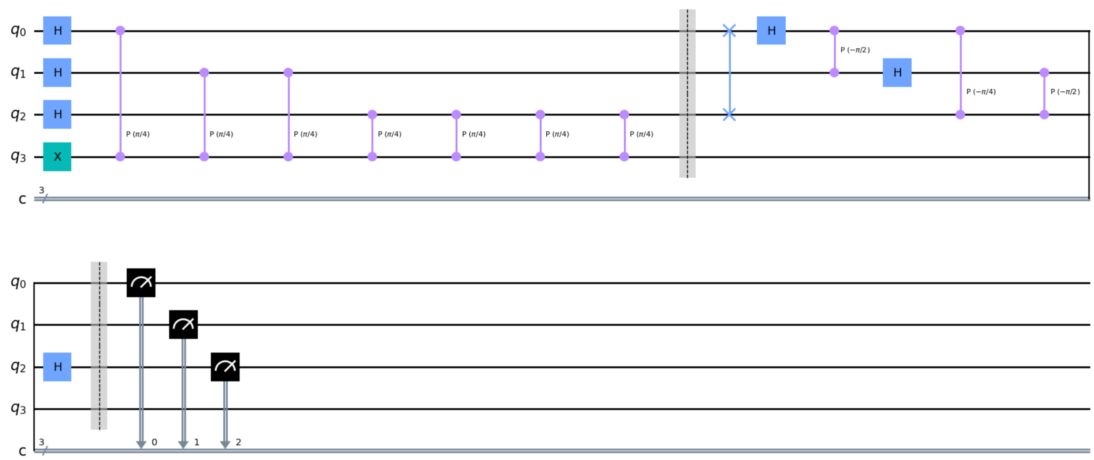
Now we apply the **Inverse Quantum Fourier Transform** in order to convert the counting register state.

The code for the IQFT is:

[3]:
```
def iqft(qCircuit, n):
    '''
    Apply the inverse quantum Fourier transform to the first n qubits in␣
    ↪qCircuit
    '''
    for qubit in range(n//2):
        qCircuit.swap(qubit, n-qubit-1)
    for j in range(n):
        for m in range(j):
            qCircuit.cp(-np.pi/float(2**(j-m)), m, j)
        qCircuit.h(j)
```

[4]:
```
qpe_circuit.barrier()
iqft(qpe_circuit, 3)
qpe_circuit.barrier()

for n in range(3):
    qpe_circuit.measure(n,n)

qpe_circuit.draw(output='mpl')
```

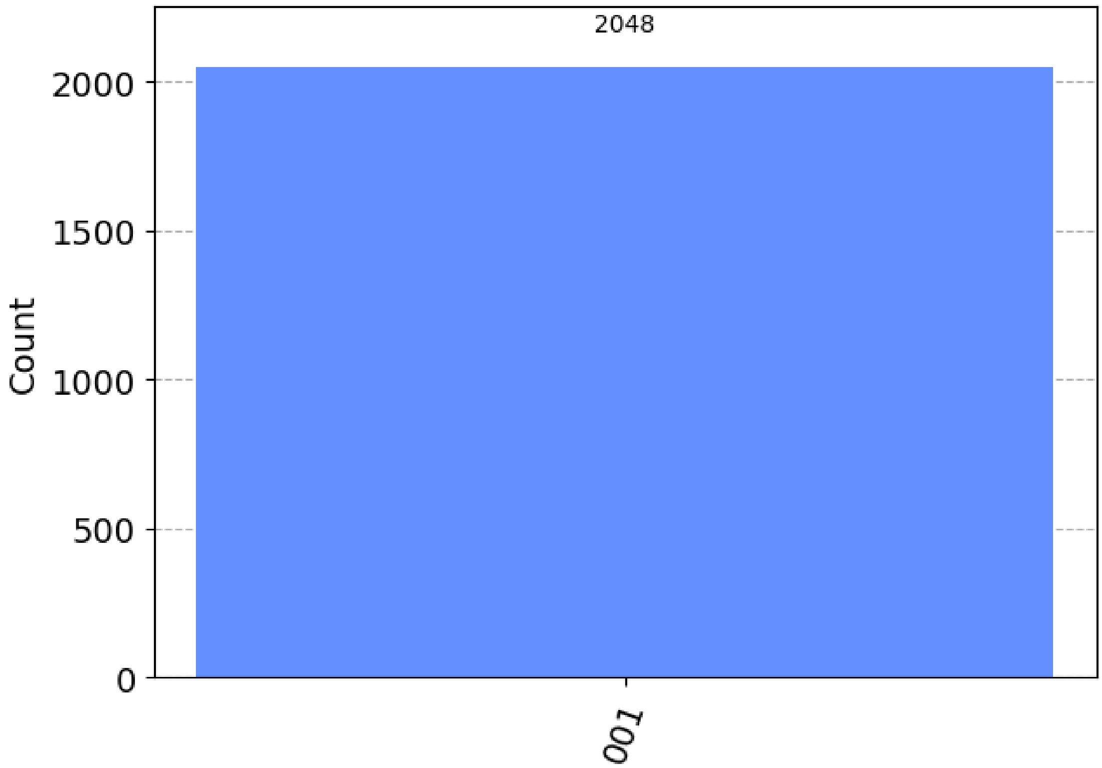
[4]:



3.2 Simulate the IQFT

[5]:
```
from qiskit import Aer, transpile
from qiskit.visualization import plot_histogram

simulator = Aer.get_backend('aer_simulator')
N = 2**11
t_qpe = transpile(qpe_circuit, simulator)

result = simulator.run(t_qpe, shots=N).result()
plot_histogram(result.get_counts())
```

[5]:



Notice that we always get (001) as result: it translates into decimal 1.

In order to get θ we need to divide our result by 2^n, i.e.

$$\theta = \frac{1}{2^3} = \frac{1}{8}$$

as expected.

3.3 Increasing precision

We may use, instead of a T-gate, a gate with $\theta = \frac{1}{3}$.

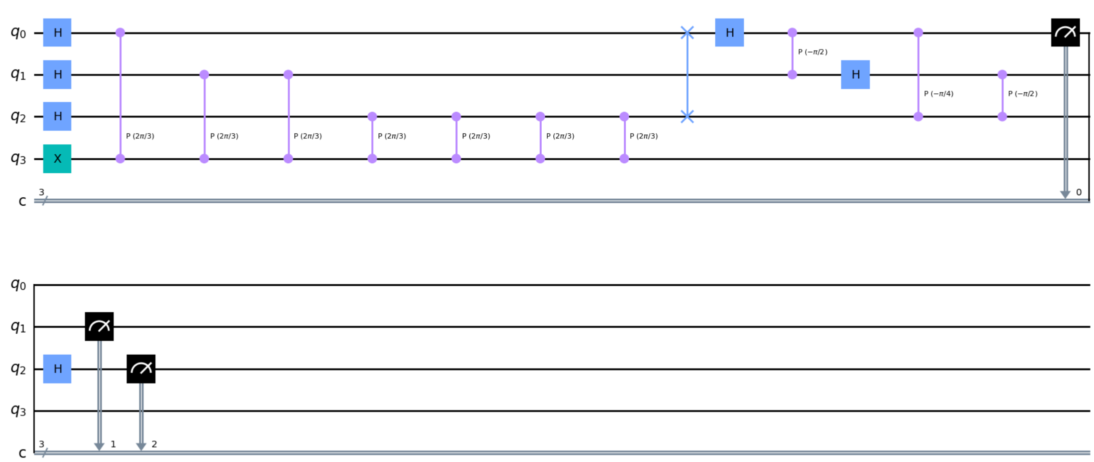
[6]:

```
def build_qpe(qubits, bits):
    # Create and set up circuit
    qpe = QuantumCircuit(qubits, bits)

    # Apply H-Gates to counting qubits:
    for qubit in range(bits):
        qpe.h(qubit)

    # Prepare our eigenstate |psi>:
    qpe.x(bits)

    # Do the controlled-U operations:
    angle = 2*np.pi/3
    repetitions = 1
    for counting_qubit in range(bits):
        for i in range(repetitions):
            qpe.cp(angle, counting_qubit, bits);
        repetitions *= 2

    # Do the inverse QFT:
    iqft(qpe, bits)

    # Measure of course!
    for n in range(bits):
        qpe.measure(n,n)

    return qpe
```

[7]:

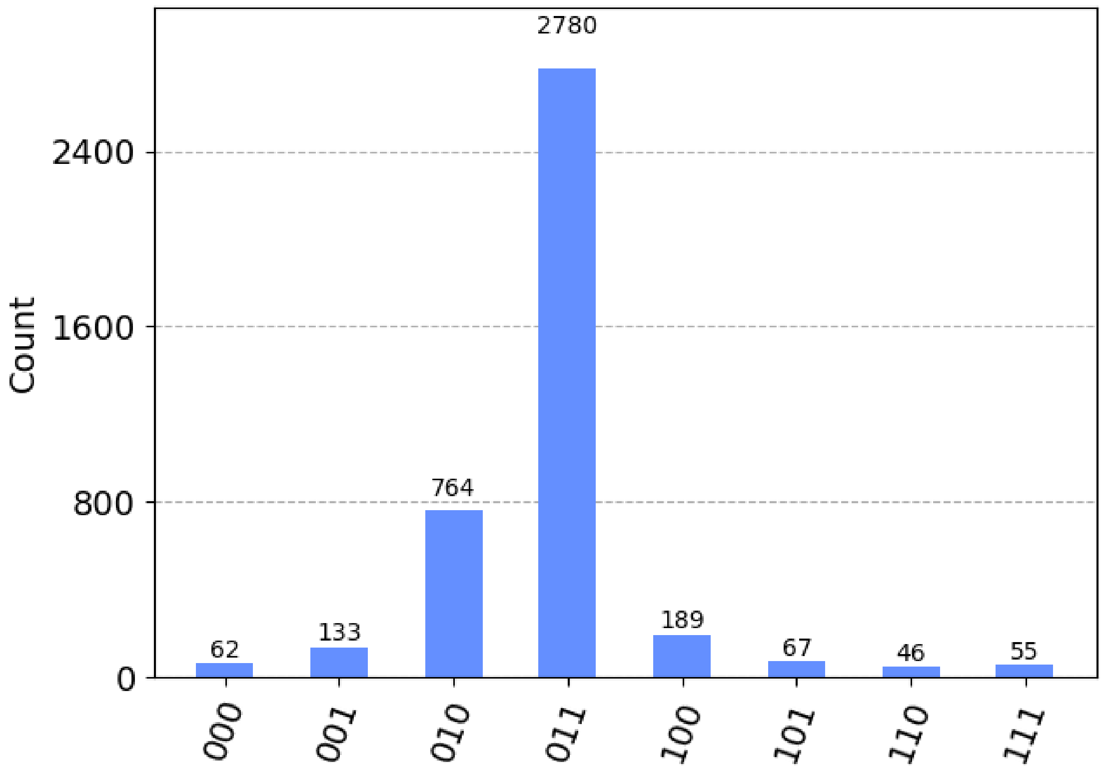
```
qpe_circuit = build_qpe(4, 3)
qpe_circuit.draw(output='mpl')
```

[7]:



[8]:
```
simulator = Aer.get_backend('aer_simulator')
N = 2**12
t_qpe = transpile(qpe_circuit, simulator)
results = simulator.run(t_qpe, shots=N).result()
plot_histogram(results.get_counts())
```

[8]:

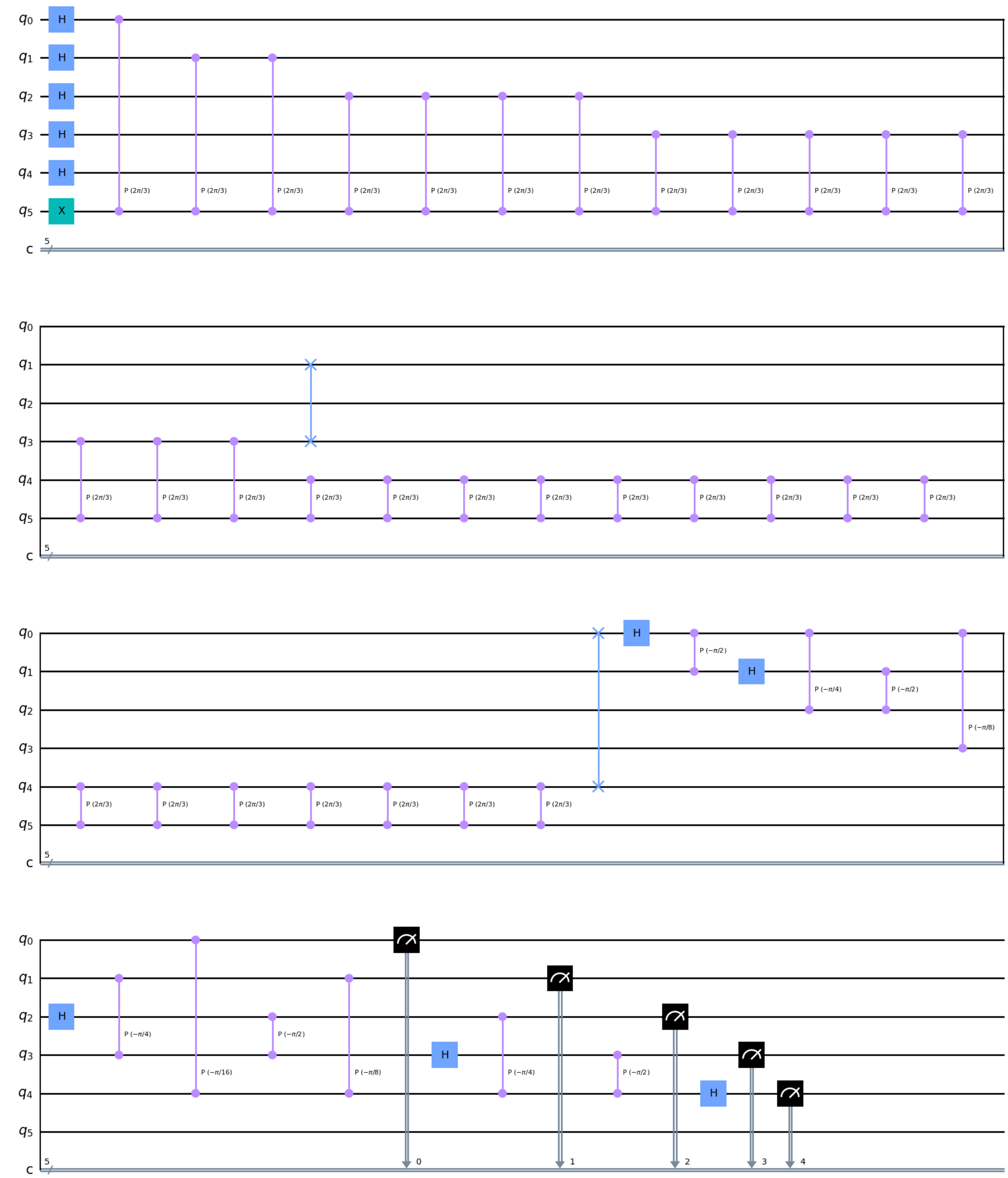


We expect as result $\theta = 0.333...$. We see that the most likely results are $(010) = 2$ and $(011) = 3$, which imply $\theta = 0.25$ or $\theta = 0.375$. The true value lies between two of our possible values... how to solve this problem?

Well we can, for instance, add more counting qubits:

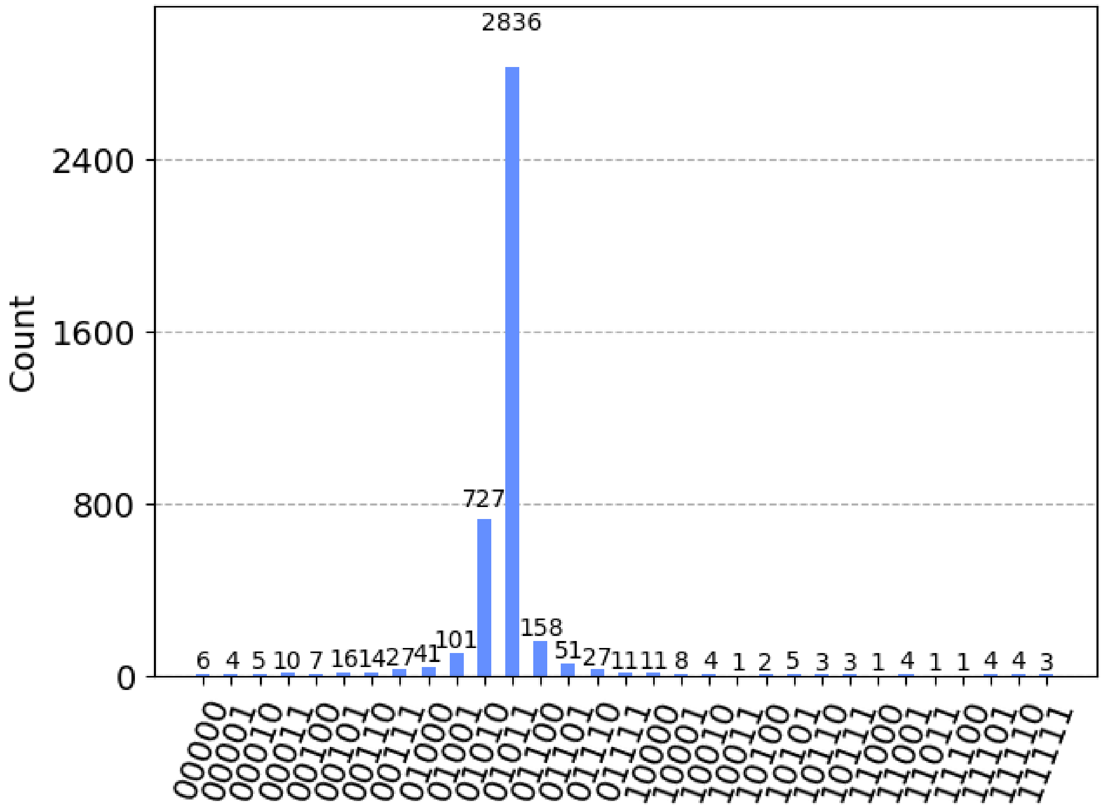
[9]:
```
qpe_circuit = build_qpe(6, 5)
qpe_circuit.draw(output='mpl')
```

[9]:



[10]:
```
simulator = Aer.get_backend('aer_simulator')
N = 2**12
t_qpe = transpile(qpe_circuit, simulator)
results = simulator.run(t_qpe, shots=N).result()
plot_histogram(results.get_counts())
```

[10]:



Now we measure (01011) = 11 and (01010) = 10. So

$$\theta = \frac{11}{2^5} = 0.34 \quad \text{or} \quad \theta = \frac{10}{2^5} = 0.31$$

and the precision has been increased.

3.4 Phase estimation on a Real Device

Now we want to reproduce on the IBM device the previous circuit. In order to do it we need to install

`pip install qiskit-ibm-runtime`

First, we try it without error correction.

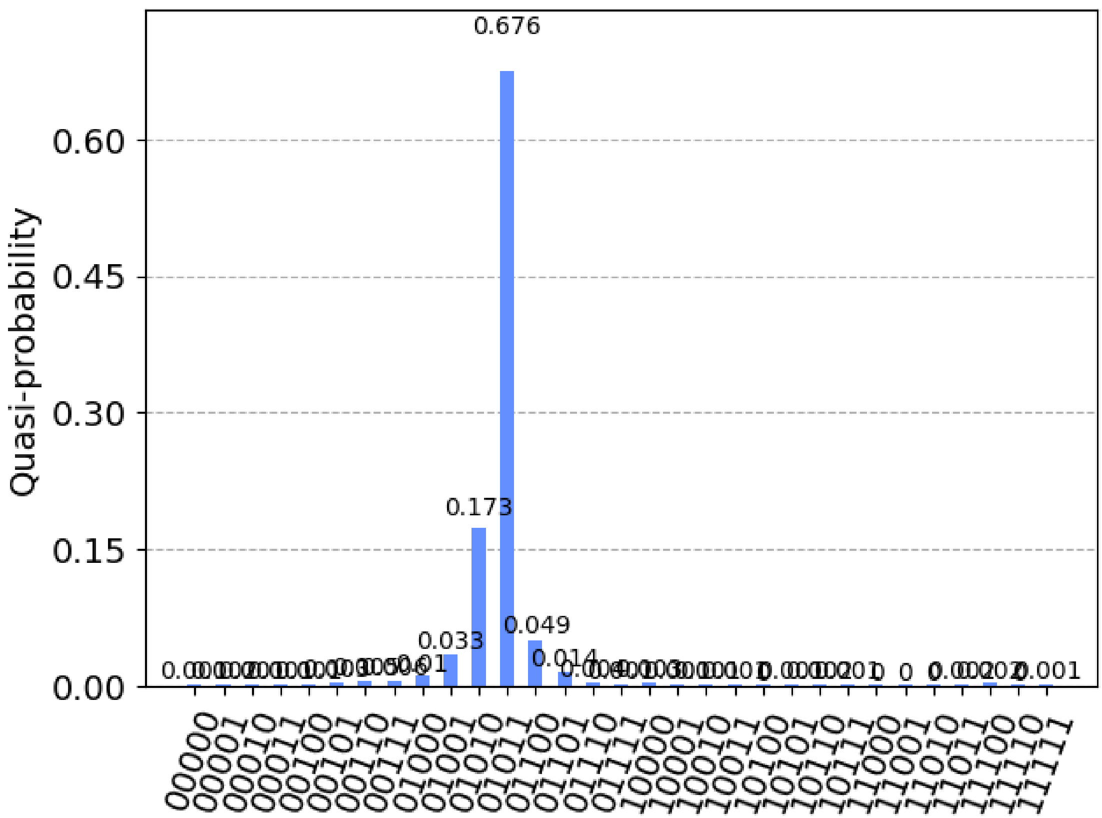
[11]:
```python
from qiskit_ibm_runtime import QiskitRuntimeService

with open('token.txt', 'r') as f:
    APItoken = f.read()

QiskitRuntimeService.save_account(
    channel='ibm_quantum', token=APItoken, overwrite=True)
```

```
service = QiskitRuntimeService(channel="ibm_quantum")

backend = service.backend("ibmq_qasm_simulator")  # or ibmq_lagos
```

[12]:
```
from qiskit_ibm_runtime import Sampler, Options

options = Options()
options.resilience_level = 0
options.execution.shots = 2**11

sampler = Sampler(backend=backend, options=options)

job = sampler.run(qpe_circuit)
```

[14]:
```
from qiskit.visualization import plot_distribution

result = job.result()

# Collect quasiprobability distribution
distribution = result.quasi_dists[0].binary_probabilities()
plot_distribution(distribution)
```

[14]:



Then we try to mitigate error by using T-REx mitigation, which stands for Twirled Readout Error extinction.

[16]:
```
options = Options()
options.resilience_level = 1  # T-REx mitigation
options.execution.shots = 2**11

sampler = Sampler(backend=backend, options=options)

job = sampler.run(qpe_circuit)
print(f">>> Job Status: {job.status()}")
```

```
>>> Job Status: JobStatus.RUNNING
```

[17]:
```
result = job.result()

# Collect quasiprobability distribution
distribution = result.quasi_dists[0].binary_probabilities()
plot_distribution(distribution)
```

[17]: