challenge_chsh

October 1, 2023

1 Challenge - Creating Entanglement with Qiskit

This challenge demonstrates interesting properties of *entangled* qubits. In particular, we will consider two experiments: - **CHSH Inequality Violation** - this shows that quantum mechanics *cannot* be explained by a local hidden variable theory - **Teleportation** - teleport an arbitrary quantum state using an entangled qubit pair as a resource

Material for this challenge was sourced from the Qiskit Global Summer School 2023.

IMPORTANT: Several of the challenge questions ask for answers to questions or explanations of your answers. To get full credit for you answers, create markdown cells and fill in your answer in words. Clearly indicate where you have provided an answer so that the judges can clearly see it.

1.1 Getting Started

Start by importing some libraries we need, including the `Sampler` and `Estimator` primitives from Qiskit. While the primitives from `qiskit.providers` use a local statevector simulator by default, the syntax within this lab is easily generalizable to running experiments on real systems.

You can choose to run the code using real quantum computers, however this is not required. To run on real hearware requires a Qiskit Runtime service instance. If you haven't done so already, follow the instructions in the Qiskit Getting started guide to set one up. After setup, import the `Sampler` and `Estimator` primitives from `qiskit_ibm_runtime` instead. Additionally we will need `QiskitRuntimeService` and `Session`, which form the interface between Qiskit and Qiskit IBM Runtime. Then the below exercises can be run on real systems by instantiating the primitives in this way (as opposed to from `qiskit.primitives`):

```
from qiskit_ibm_runtime import QiskitRuntimeService, Session, Sampler, Estimator

service = QiskitRuntimeService()
backend = service.get_backend('...')
session = Session(service=service, backend=backend)
sampler = Sampler(session=session)
estimator = Estimator(session=session)
```

where additional options can be specified in the `Sampler` and `Estimator` with the `Options` class. See this how-to for using Primitives with Runtime Sessions.

[1]:
```
from qiskit.circuit import QuantumCircuit
from qiskit.primitives import Estimator, Sampler
from qiskit.quantum_info import SparsePauliOp
```

```
from qiskit.visualization import plot_histogram

import numpy as np
import matplotlib.pyplot as plt
plt.style.use('dark_background') # optional
```

1.2 CHSH Inequality Violation

1.2.1 Warm Up

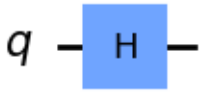
Create circuits that put the qubit in the excited $|1\rangle$ and superposition $|+\rangle$ states, respectivly, and measure them in different bases. This is done first with the `Sampler` primitive, and then with the `Estimator` primitive to show how measurement is abstracted in that we do not need to worry about rotating the qubit into the appropriate measurement basis. The primitives will be executed withing the `Session` context which allows efficiency to optimize workloads.

[2]:
```
# create excited |1> state
qc_1 = QuantumCircuit(1)
qc_1.x(0)
qc_1.draw('mpl')
```

[2]:

[3]:
```
# create superposition |+> state
qc_plus = QuantumCircuit(1)
qc_plus.h(0)
qc_plus.draw('mpl')
```

[3]:



1.2.2 Sampler Primitive

First use the `Sampler` to measure qubits in the Z-basis (the physical basis in which qubits are measured). The `Sampler` will count the number of outcomes of the $|0\rangle$ state and $|1\rangle$ state, normalized by

the number of shots (experiments performed). Results returned are refered to as *quasi-probabilities*.

Measurments must be present in the circuit when using the `Sampler` primitive. Then the `Session` context is opened, the `Sampler` is instantiated, and `sampler.run()` is used to send the circuits to the backend, similar to the `backend.run()` syntax you may already be familiar with.

[4]:
```
qc_1.measure_all()
qc_plus.measure_all()

sampler = Sampler()
job_1 = sampler.run(qc_1)
job_plus = sampler.run(qc_plus)
```

[5]:
```
job_1.result().quasi_dists
```

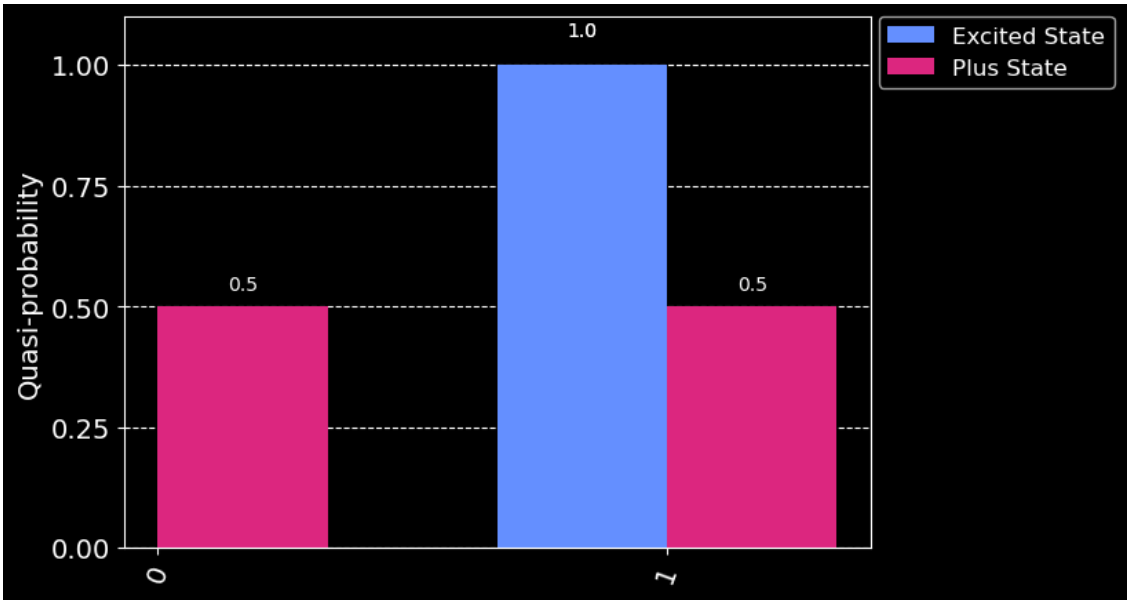
[5]:
```
[{1: 1.0}]
```

[6]:
```
job_plus.result().quasi_dists
```

[6]:
```
[{0: 0.4999999999999999, 1: 0.4999999999999999}]
```

[7]:
```
legend = ["Excited State", "Plus State"]
plot_histogram([job_1.result().quasi_dists[0], job_plus.result().
 ↪quasi_dists[0]], legend=legend)
```

[7]:



The result for the excited state is always $|1\rangle$ wheres it is roughly half $|0\rangle$ and half $|1\rangle$ for the plus superposition state. This is because the $|0\rangle$ and $|1\rangle$ states are *eigenstates* of the Z operator (with $+1$ and -1 eigenvalues, respectively).

Let's switch and measure in the X basis. Using the `Sampler` we must rotate the qubit from the X-basis to the Z-basis for measurement (because that is the only basis we can actually perform measurement in).

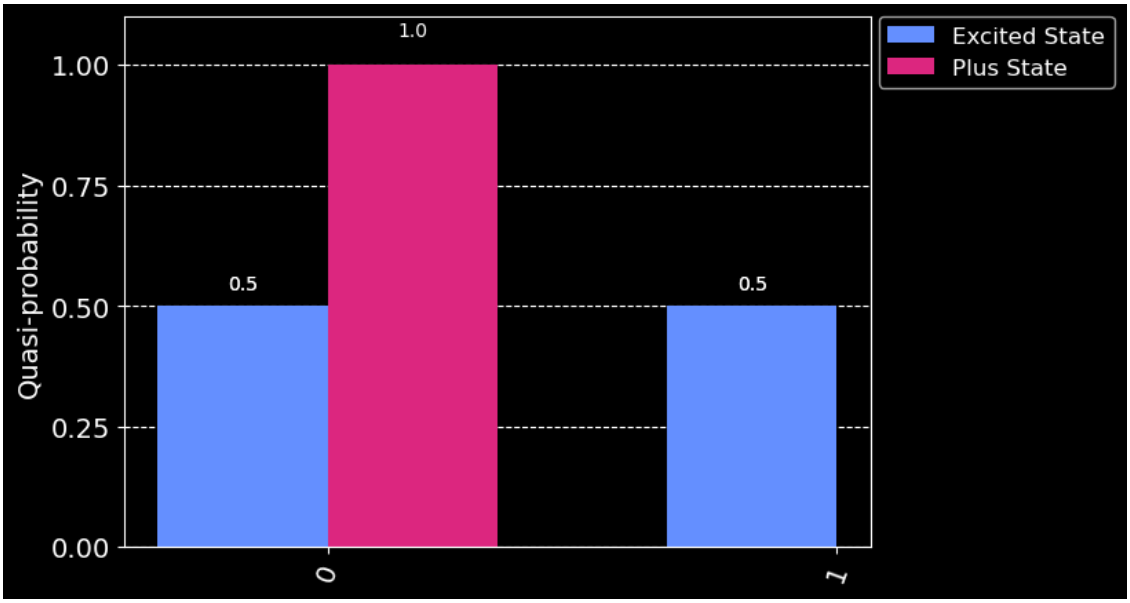
STEP 1: Alter circuits `qc_1` and `qc_plus` to take measurements on each in the X basis.

[8]:
```
# First remove the measurements prior to altering the circuits
qc_1.remove_final_measurements()
qc_plus.remove_final_measurements()

# INSERT ANSWER HERE
# rotate into the X-basis
qc_1.h(0)
qc_plus.h(0)

## END ANSWER

qc_1.measure_all()
qc_plus.measure_all()
```

[9]:
```
sampler = Sampler()
job_1 = sampler.run(qc_1)
job_plus = sampler.run(qc_plus)
```

[10]:
```
plot_histogram([job_1.result().quasi_dists[0], job_plus.result().
↪quasi_dists[0]], legend=legend)
```

[10]:



You should see the opposite to the previous result: the plus superposition always give the 1 result, hence an eigenstate of the X operator, whereas the excited $|1\rangle$ yields a roughly fifty-fifty split.

The $|+\rangle$ and $|-\rangle$ states are eigenstates of the X operator, with eigenvalues $+1$ and -1, respectively. This is good to remember when considering how the `Estimator` works in the next subsection.

1.2.3 Estimator Primitive

The Qiskit Runtime Primitives allow us to abstract measurement into the `Estimator` primitive, where it is specified as an *observable*. In particular, we can construct the same circuits, the excited $|1\rangle$ and superposition $|+\rangle$ as before. However, in the case of the `Estimator`, we *do not* add measurements to the circuit. Instead, specify a list of observables which take the form of Pauli strings. In our case for a measurement of a single qubit, we specify `'Z'` for the Z-basis and `'X'` for the X-basis.

[11]:
```
qc2_1 = QuantumCircuit(1)
qc2_1.x(0)

qc2_plus = QuantumCircuit(1)
qc2_plus.h(0)

obsvs = list(SparsePauliOp(['Z', 'X']))
```

[12]:
```
estimator = Estimator()
job2_1 = estimator.run([qc2_1]*len(obsvs), observables=obsvs)
job2_plus = estimator.run([qc2_plus]*len(obsvs), observables=obsvs)
```

[13]:
```
job2_1.result()
```

[13]:
```
EstimatorResult(values=array([-1.,  0.]), metadata=[{}, {}])
```

[14]:
```
print(f'    |   <Z>   |   <X> ')
print(f'----|-----------------')
print(f'|1> | {job2_1.result().values[0]}    | {job2_1.result().values[1]}')
print(f'|+> | {job2_plus.result().values[0]} | {job2_plus.result().values[1]}')
```

```
    |   <Z>   |   <X>
----|-----------------
|1> | -1.0    | 0.0
|+> | 0.0 | 0.9999999999999998
```

Just as before, we see the $|1\rangle$ state expectation in the Z-basis is -1 (corresponding to its eigenvalue) and around zero in the X-basis (average over $+1$ and -1 eigenvalues), and vice-versa for the $|+\rangle$ state (although its eigenvalue of the X operators is $+1$).

1.3 CHSH Inequality

Imagine Alice and Bob are given each one part of a bipartite entangled system. Each of them then performs two measurements on their part in two different bases. Let's call Alice's bases A and a and Bob's B and b. What is the expectation value of the quantity

$$\langle CHSH \rangle = \langle AB \rangle - \langle Ab \rangle + \langle aB \rangle + \langle ab \rangle ?$$

Now, Alice and Bob have one qubit each, so any measurement they perform on their system (qubit) can only yield one of two possible outcomes: +1 or -1. Note that whereas we typically refer to the two qubit states as $|0\rangle$ and $|1\rangle$, these are eigenstates, and a projective measurement will yield their eigenvalues, +1 and -1, respectively.

Therefore, if any measurement of A, a, B, and b can only yield ± 1, the quantities $(B-b)$ and $(B+b)$ can only be 0 or ± 2. And thus, the quantity $A(B-b)+a(B+b)$ can only be either +2 or -2, which means that there should be a bound for the expectation value of the quantity we have called

$$|\langle CHSH\rangle| = |\langle AB\rangle - \langle Ab\rangle + \langle aB\rangle + \langle ab\rangle| \leq 2.$$

Now, the above discussion is oversimplified, because we could consider that the outcome on any set of measurements from Alice and Bob could depend on a set of local hidden variables, but it can be shown with some math that, even when that is the case, the expectation value of the quantity $CHSH$ should be bounded by 2 if local realism held.

But what happens when we do these experiments with an entangled system? Let's try it!

The first step is to build the observable

$$CHSH = A(B-b) + a(B+b) = AB - Ab + aB + ab$$

where A, a are each one of $\{IX, IZ\}$ for qubit 0 and B, b are each one of $\{XI, ZI\}$ for qubit 1 (corresponding to little-endian notation). Paulis on different qubits can be composed by specifying order with a Pauli string, for example instantiating a `SparsePauliOp` with the `'ZX'` argument implies a measurement of $\langle X\rangle$ on `q0` and $\langle Z\rangle$ on `q1` . This *tensor* product (combining operations on *different* qubits) can be explicitly stated using the `.tensor()` method. Additionally, combining operations on the *same* qubit(s) uses the *compositional* product with the `.compose()` method. For example, all these statements create the same Pauli operator:

```
from qiskit.quantum_info import SparsePauliOp

ZX = SparsePauliOp('ZX')
ZX = SparsePauliOp(['ZX'], coeffs=[1.]) # extendable to a sum of Paulis
ZX = SparsePauliOp('Z').tensor(SparsePauliOp('X')) # extendable to a tensor product of Paulis
ZX = SparsePauliOp('XZ').compose(SparsePauliOp('YY')) # extendable to a compositional product
```

STEP 2: create an operator for CHSH witness

[15]:
```
# FILL IN CODE HERE

'''
obsv = # CREATE OPERATOR FOR CHSH WITNESS
'''

#### ANSWER
obsv = SparsePauliOp('XX') + SparsePauliOp('YY') + SparsePauliOp('ZX') -␣
 ↪SparsePauliOp('ZY')# create operator for chsh witness
print(obsv)
#### END ANSWER
```

```
SparsePauliOp(['XX', 'YY', 'ZX', 'ZY'],
              coeffs=[ 1.+0.j,  1.+0.j,  1.+0.j, -1.+0.j])
```

1.3.1 Create Entangled Qubit Pair

Next we want to test the $CHSH$ observable on an entangled pair, for example the maximally-entangled Bell state

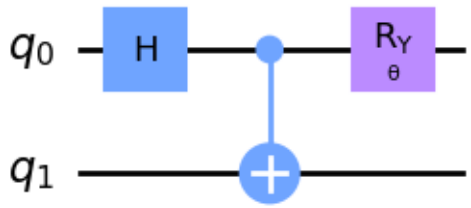
$$|\Phi\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right)$$

which is created with a Hadamard gate followed by a CNOT with the target on the same qubit as the Hadamard. Due to the simplifaction of measuring in just the X- and Z-bases as discussed above, we will *rotate* the Bell state around the Bloch sphere which is equivalant to changing the measurement basis as demonstrated in the Warmup section. This can be done by applying an $R_y(\theta)$ gate where θ is a `Parameter` to be specified at the `Estimator` API call. This produces the state

$$|\psi\rangle = \frac{1}{\sqrt{2}}\left(\cos(\theta/2)|00\rangle + \sin(\theta/2)|11\rangle\right)$$

[16]:
```
from qiskit.circuit import Parameter

theta = Parameter(' ')

qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
qc.ry(theta, 0)

qc.draw('mpl')
```

[16]:



Next we need to specify a `Sequence` of `Parameters` that show a clear violation of the CHSH Inequality, namely

$$|\langle CHSH\rangle| > 2.$$

Let's make sure we have at least three points in violation.

CHALLENGE 3: Create a Parameterization (i.e., list, array) of the angle in the above circuit (in radians)

Hint: Note the `type` for the `parameter_values` argument is `Sequence[Sequence[float]]`.

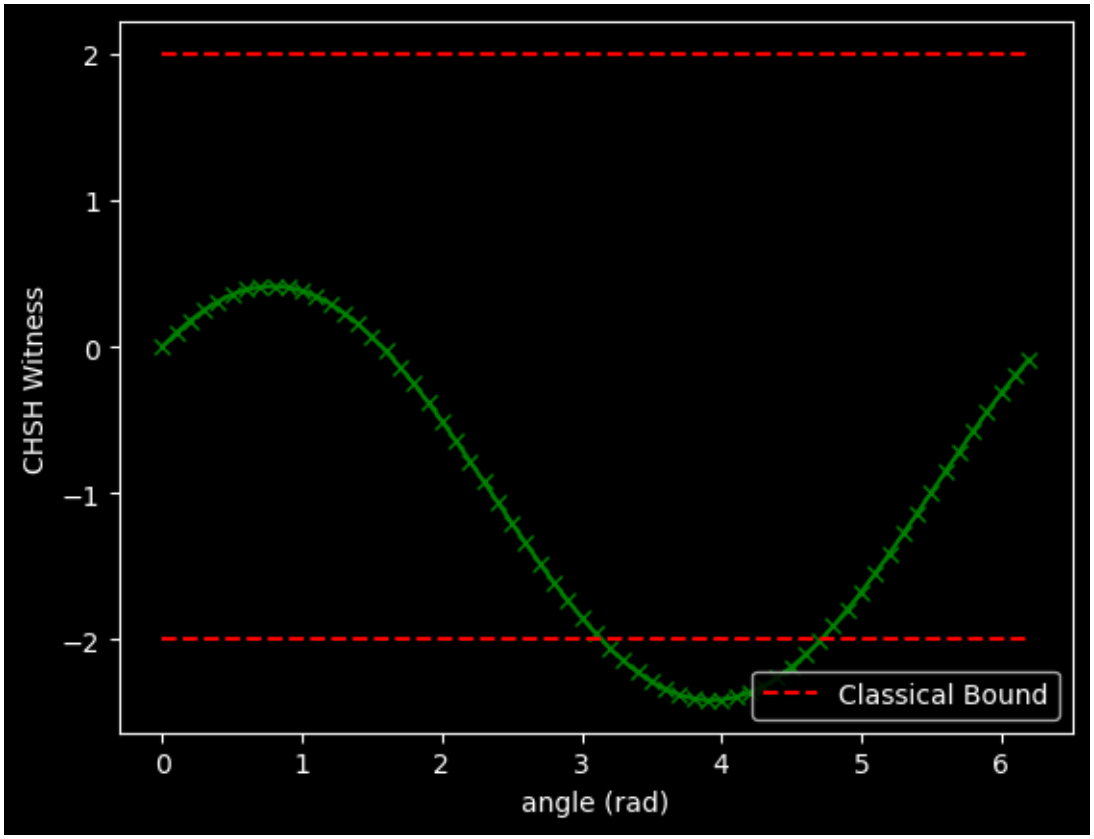
[17]:
```
# FILL IN CODE HERE

'''
angles = # create a parameterization of angles that will violate the inequality
'''

#### ANSWER
angles =  [[i * 0.1] for i in range(int(2 * np.pi / 0.1) + 1)]
# create a parameterization of angles that will violate the inequality
#### END ANSWER
print(angles)
```

```
[[0.0], [0.1], [0.2], [0.30000000000000004], [0.4], [0.5], [0.6000000000000001],
[0.7000000000000001], [0.8], [0.9], [1.0], [1.1], [1.2000000000000002], [1.3],
[1.4000000000000001], [1.5], [1.6], [1.7000000000000002], [1.8],
[1.9000000000000001], [2.0], [2.1], [2.2], [2.3000000000000003],
[2.4000000000000004], [2.5], [2.6], [2.7], [2.8000000000000003],
[2.9000000000000004], [3.0], [3.1], [3.2], [3.3000000000000003],
[3.4000000000000004], [3.5], [3.6], [3.7], [3.8000000000000003],
[3.9000000000000004], [4.0], [4.1000000000000005], [4.2], [4.3], [4.4], [4.5],
[4.6000000000000005], [4.7], [4.800000000000001], [4.9], [5.0],
[5.1000000000000005], [5.2], [5.300000000000001], [5.4], [5.5],
[5.6000000000000005], [5.7], [5.800000000000001], [5.9], [6.0],
[6.1000000000000005], [6.2]]
```

Test your angles and observable by running with the `Estimator`.

[18]:
```
estimator = Estimator()
job = estimator.run([qc]*len(angles), observables=[obsv]*len(angles),␣
 ↪parameter_values=angles)
exps = job.result().values

plt.plot(angles, exps, marker='x', ls='-', color='green')
plt.plot(angles, [2]*len(angles), ls='--', color='red', label='Classical Bound')
plt.plot(angles, [-2]*len(angles), ls='--', color='red')
plt.xlabel('angle (rad)')
plt.ylabel('CHSH Witness')
plt.legend(loc=4)
```

[18]:
```
<matplotlib.legend.Legend at 0x7fbd98df0310>
```



Did you see at least 3 points outside the red dashed lines? What does this mean? Provide an explanation below:

EXPLAIN THE INTERPRETATION OF THE RESULTS HERE:The CHSH inequality (Clauser-Horne-Shimony-Holt) is a critical test in quantum physics to discriminate between classical and quantum connections in entangled systems. This means that the behaviour of the entangled particles cannot be explained by classical physics alone and that quantum entanglement is a necessary component of the description if this inequality fails (i.e., the left-hand side exceeds 2). It sets a bound on the correlations observed between measurements on entangled particles within a classical framework. The CHSH inequality is a mathematical expression that tests whether a given physical system can be described by classical physics or if it exhibits quantum entanglement. In simple terms, the CHSH inequality defines the correlation between measurements of two entangled particles. This correlation has a limit in a classical system but can go beyond it in a quantum system with entanglement. So, observing at least 3 points outside the red dashed lines in your plot means that the experiment you conducted with your entangled qubit pair and the chosen measurement settings has produced results that classical physics cannot explain. It signifies a clear violation of the CHSH inequality. This is an essential illustration of the non-classical character of the prepared entangled state.

2 Teleportation

Quantum information cannot be copied due to the *No Cloning Theorem*, however it can be "teleported" in the sense that a qubit can be entangled with a quantum resource, and via a protocol of measurements and *classical communication* of their results, the original quantum state can be reconstructed on a different qubit. This process destroys the information in the original qubit via measurement.

In this challenge, we will construct a particular qubit state and then transfer that state to another qubit using the teleportation protocol. Here we will be looking at specific classical and quantum registers, so we need to import those.

[20]:
```
from qiskit.circuit import ClassicalRegister, QuantumRegister
```

2.0.1 Create the circuit

Define an angle θ to rotate our qubit by. This will allow us to easily make comparisons for the original state and the teleported state.

[21]:
```
theta = Parameter(' ')

qr = QuantumRegister(1, 'q')
qc = QuantumCircuit(qr)
qc.ry(theta, 0)
qc.draw('mpl')
```

[21]:

Alice possesses the quantum information $|\psi\rangle$ in the state of q and wishes to transfer it to Bob. The resource they share is a special entangled state called a Bell state

$$|\Phi^+\rangle = \frac{1}{2}\left(|00\rangle + |11\rangle\right)$$

with the first of the pair going to Alice and the second to Bob. Hence Alice has a 2-qubit register (q and $Bell_0$) and Bob has a single-qubit register ($Bell_1$). We will construct the circuit by copying the original `qc` and adding the appropriate registers.

[22]:
```
tele_qc = qc.copy()
bell = QuantumRegister(2, 'Bell')
alice = ClassicalRegister(2, 'Alice')
bob = ClassicalRegister(1, 'Bob')
tele_qc.add_register(bell, alice, bob)
```

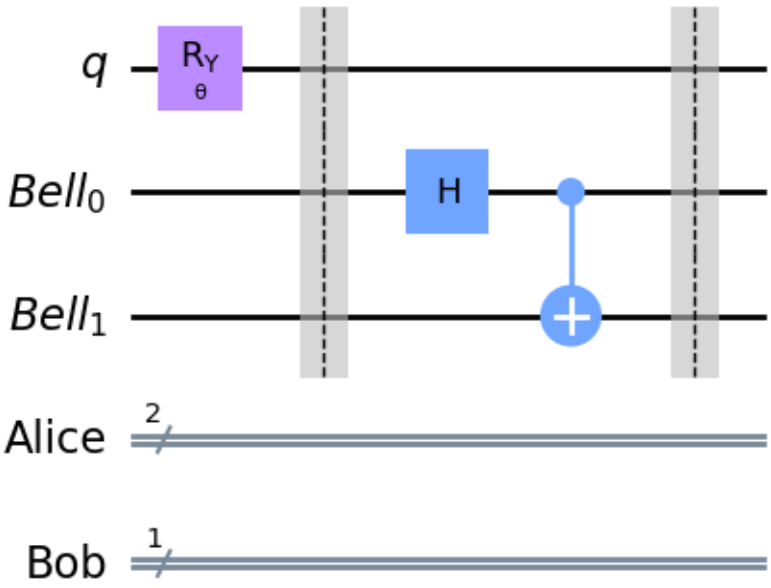
```
tele_qc.draw('mpl')
```

[22]:

Now create the Bell pair with $Bell_0$ going to Alice and $Bell_1$ going to Bob. This is done by using a Hadamard gate to put $Bell_0$ in the $|+\rangle$ state and then performing a CNOT with the same qubit as the control. After they receive their respective qubit, they part ways.

[23]:
```
# create Bell state with other two qubits
tele_qc.barrier()
tele_qc.h(1)
tele_qc.cx(1, 2)
tele_qc.barrier()
tele_qc.draw('mpl')
```

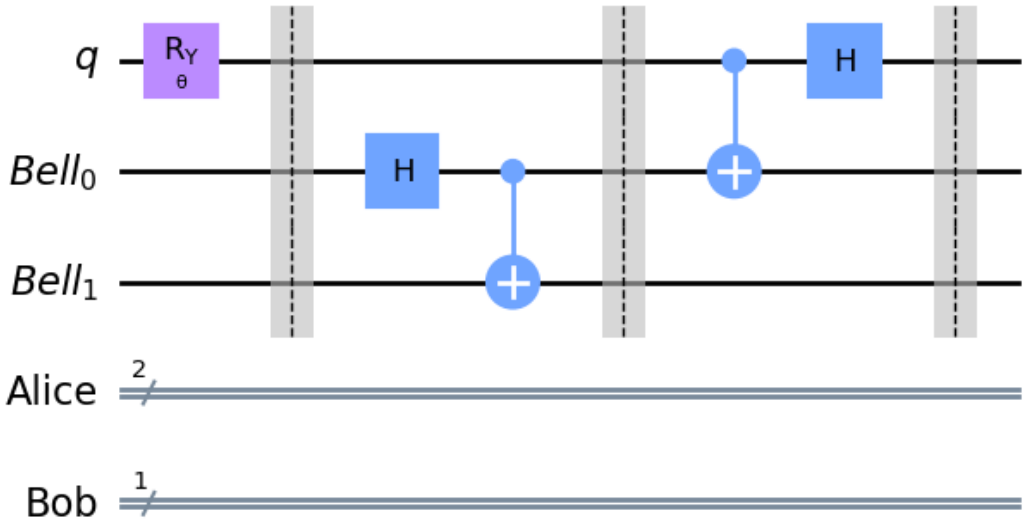
[23]:



Next, Alice performs a CNOT controlled by q on $Bell_0$, which maps information about the state onto it. She then applies a Hadamard gate on q.

[24]:
```
# alice operates on her qubits
tele_qc.cx(0, 1)
tele_qc.h(0)
tele_qc.barrier()
tele_qc.draw('mpl')
```

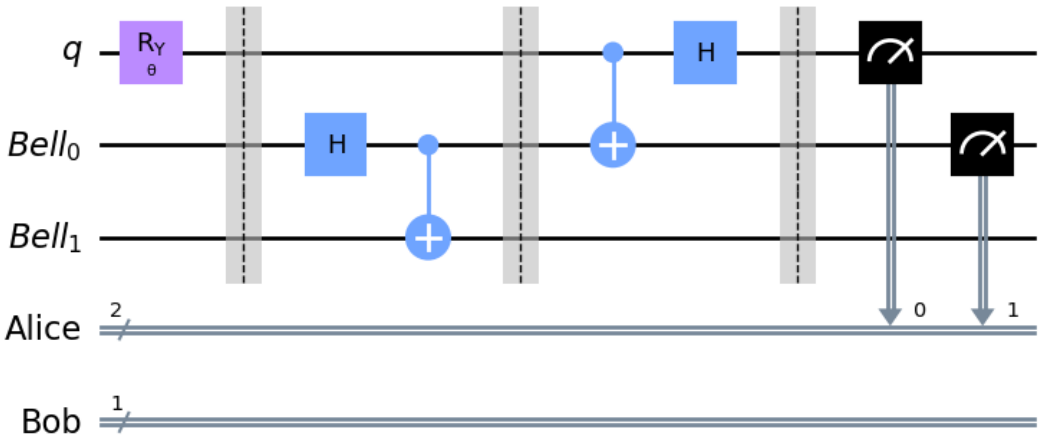
[24]:



Now Alice measures her qubits and saves the results to her register.

[25]:
```
tele_qc.measure([qr[0], bell[0]], alice)
tele_qc.draw('mpl')
```

[25]:



Bob's qubit now has the information $|\psi\rangle$ from Alice's qubit q encoded in $Bell_1$, but he does not know what basis to measure in to extract it. Accordingly, Alice must share the information in her register over a *classical* communication channel (this is why teleportation does not violate special relativity, no matter how far Alice and Bob are apart). She instructs Bob to perform an X gate on

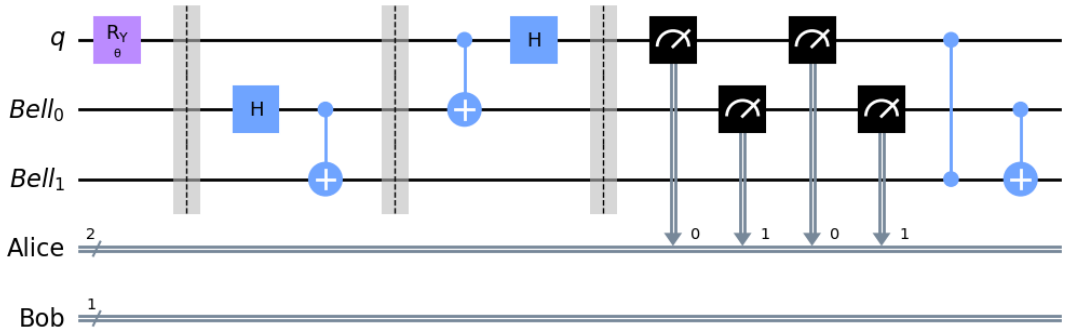
his qubit if her measurement of $Bell_0$ yields a 1 outcome, followed by a Z gate if her measurement of q yields a 1.

The applications of these gates can be conditioned on the measurement outcomes in two ways: - the `.c_if()` instruction, which applies the gate it modifies if the value of the `ClassicalRegister` index is equal to the value specified. Note that this works **only** on simulators. - the `.if_test()` context which operates similarly, but generalizes the syntax to allow for nested conditionals. This works on both simulators and actual hardware.

STEP 4: Add approriate conditional gates to transform Bob's qubit into the Z-basis.

[26]:
```
full_qc = tele_qc.copy()


#### ANSWER
# add gates to full_qc here
full_qc.measure(0, 0)
full_qc.measure(1, 1)

# Apply correction gates to Bob's qubit
full_qc.cz(0, 2)  # Apply a Z gate if the result of the first measurement was 1
full_qc.cx(1, 2)


#### END ANSWER


full_qc.draw('mpl')
```

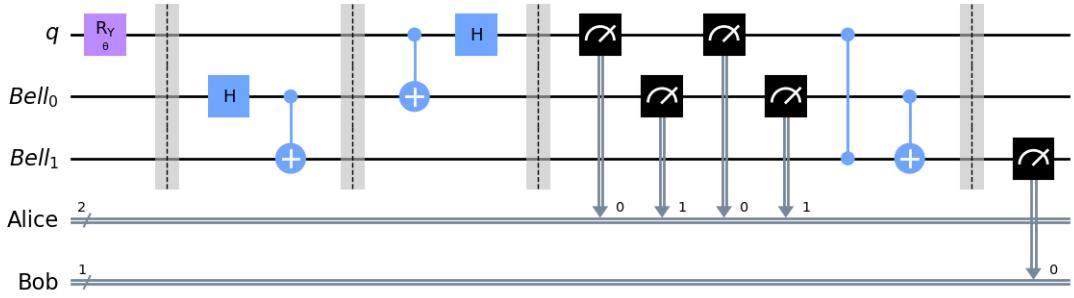
[26]:



Finally, Bob can measure his qubit, which would yield results with the same probabilities as had Alice measured it originally.

[27]:
```
full_qc.barrier()
full_qc.measure(bell[1], bob)
full_qc.draw('mpl')
```

[27]:



The statevector simulator cannot work with dynamic circuits because measurement is not a unitary operation. Therefore we import the `Sampler` primitive from `qiskit_aer` to use the `AerSimulator`. We choose our angle to be $5\pi/7$, which will yield a 1 result about 80% of the time and 0 result about 20% of the time. Then we run both circuits: the original one Alice had and the teleported one Bob receives.

[28]:
```
from qiskit_aer.primitives import Sampler

angle = 5*np.pi/7

sampler = Sampler()
qc.measure_all()
job_static = sampler.run(qc.bind_parameters({theta: angle}))
job_dynamic = sampler.run(full_qc.bind_parameters({theta: angle}))

print(f"Original Dists: {job_static.result().quasi_dists[0].
  ↪binary_probabilities()}")
print(f"Teleported Dists: {job_dynamic.result().quasi_dists[0].
  ↪binary_probabilities()}")
```

```
Original Dists: {'0': 0.1884765625, '1': 0.8115234375}
Teleported Dists: {'001': 0.0263671875, '000': 0.0556640625, '010': 0.0546875,
'110': 0.1943359375, '100': 0.203125, '101': 0.2236328125, '111': 0.1923828125,
'011': 0.0498046875}
```

Wait, we see different results! While measuring Alice's original q yields the expected ratio of outcomes, the teleported distributions have many more values. This is because the teleported circuit includes Alice's measurements of q and $Bell_0$, whereas we only wish to see Bob's measurements of $Bell_1$ yield the same distribution.

In order to rectify this, we must take the *marginal* counts, meaning we combine results in which Bob measures a 0 and all the results in which Bob measures a 1 over all the possible combinations. This is done with the `marginal_counts` method from `qiskit.result`, which combines results over measurement indices.

STEP 5: Marginalize the teleported counts

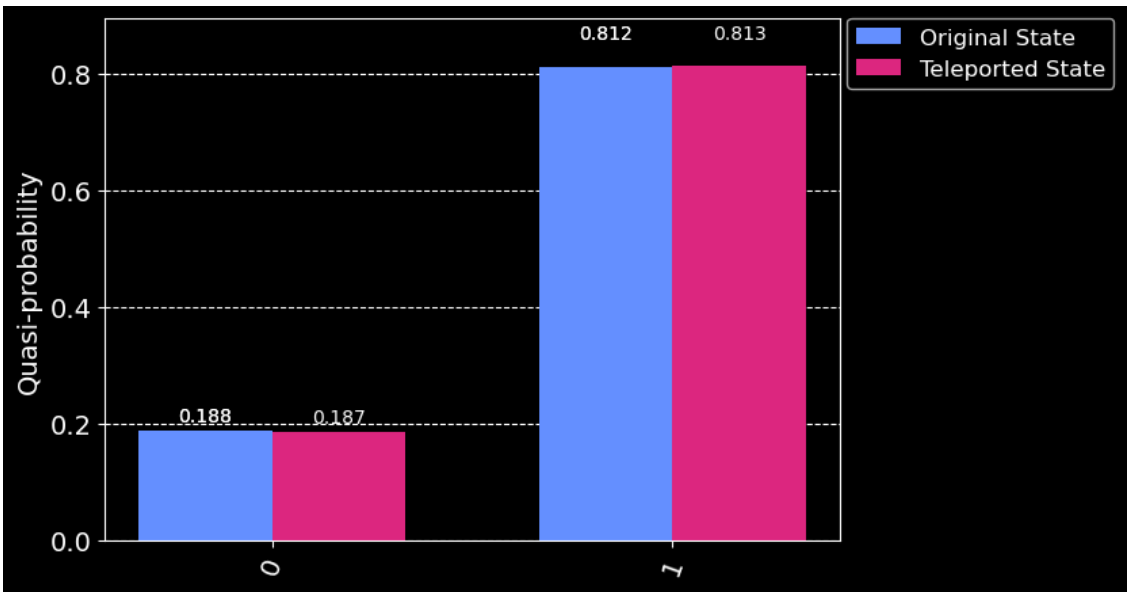
Hint: Remember that bit strings are reported in the little-endian convention.

[29]:
```
from qiskit.result import marginal_counts

# FILL IN CODE HERE
'''
tele_counts = # marginalize counts
'''

#### ANSWER
# Get the probabilities from the Result object
quasi_probs = job_dynamic.result().quasi_dists[0].binary_probabilities()

# Get the marginal counts for the third qubit and put it into tele_counts
tele_counts = marginal_counts(quasi_probs, [2])
#### END ANSWER
```

If we marginalized correctly, we will see that the quasi-distributions from Alice's measurement and Bob's measurement are nearly identical, demonstrating that teleportation was successful!

[30]:
```
legend = ['Original State', 'Teleported State']
plot_histogram([job_static.result().quasi_dists[0].binary_probabilities(),␣
↪tele_counts], legend=legend)
```

[30]:



[31]:
```
import qiskit.tools.jupyter
%qiskit_version_table
```

```
<IPython.core.display.HTML object>
```

[ ]: