COMS 4281 - Intro to Quantum Computing

Problem Set 1, Quantum Info Basics

Due: October 6, 11:59pm

Collaboration is allowed and encouraged (teams of at most 3). Please read the syllabus carefully for the guidlines regarding collaboration. In particular, everyone must write their own solutions in their own words.

Write your collaborators here:

Recommended Environment to Run This Notebook

We highly recommend that you use the qBraid platform to run this Jupyter notebook. This supports Qiskit, and furthermore to render your Problem Set solutions to PDF, you have to do the following:

1. File > Save and Export Notebook As > HTML
2. Save the HTML file somewhere on your local computer
3. Open the HTML file using your favorite browser, and Print to PDF. We recommend using Landscape mode so the Python code shows up better.

Click here to collapse these cells after running the first two

These commands will install qiskit and the qiskit simulator on the Jupyter environment (we recommend qBraid). It may take a few minutes.

In [ ]:
```
!pip install qiskit > /dev/null
!pip install qiskit_aer > /dev/null
!pip install qiskit_ibm_runtime > /dev/null
```

The following code are helper routines that will be used throughout this problem set. After running it, you can click the cell called "Click here to collapse..." to hide this.

In [ ]:

```
from qiskit import *
import qiskit
from qiskit.visualization import plot_state_city
from qiskit.compiler import transpile
from qiskit.visualization import plot_histogram
from qiskit.quantum_info.operators import Operator
from qiskit.quantum_info import Statevector
from qiskit.circuit.library import UnitaryGate
from qiskit_aer import AerSimulator
from qiskit_ibm_runtime import SamplerV2
backend = AerSimulator()
sampler = SamplerV2(mode=backend)

import numpy as np
from typing import Callable, List, Tuple
import math
from functools import *
import copy

QuantumClassicalOperator = Callable[[QuantumRegister, ClassicalRegister], QuantumCircuit]
QuantumOperator = Callable[[QuantumRegister], QuantumCircuit]
def append(global_circuit: QuantumCircuit,
             operator: QuantumClassicalOperator,
             quantum_register: List[int],
             classical_register: List[int]) -> QuantumCircuit:
    delegated_qregister = QuantumRegister(len(quantum_register), "quantum_register")
    delegated_cregister = ClassicalRegister(len(classical_register), "classical_register")
    delegated_operation_circuit = operator(delegated_qregister, delegated_cregister)
    global_circuit.append(delegated_operation_circuit,
                          qargs = [global_circuit.qubits[reg] for reg in quantum_register],
                          cargs = [global_circuit.clbits[reg] for reg in classical_register])
    return global_circuit.decompose(delegated_operation_circuit.name)

def append(global_circuit: QuantumCircuit,
             operator: QuantumOperator,
             quantum_register: List[int]) -> QuantumCircuit:
    delegated_qregister = QuantumRegister(len(quantum_register), "quantum_register")
    delegated_operation_circuit = operator(delegated_qregister)
```

```
    global_circuit.append(delegated_operation_circuit,
                          qargs = [global_circuit.qubits[reg] for reg in quantum_register],
                          cargs = [])
    return global_circuit.decompose(delegated_operation_circuit.name)

def append2(global_circuit: QuantumCircuit,
             operator: QuantumClassicalOperator,
             quantum_register: List[int],
             classical_register: List[int]) -> QuantumCircuit:
    delegated_qregister = QuantumRegister(len(quantum_register), "quantum_register")
    delegated_cregister = ClassicalRegister(len(classical_register), "classical_register")
    delegated_operation_circuit = operator(delegated_qregister, delegated_cregister)
    global_circuit.append(delegated_operation_circuit,
                          qargs = [global_circuit.qubits[reg] for reg in quantum_register],
                          cargs = [global_circuit.clbits[reg] for reg in classical_register])
    return global_circuit.decompose(delegated_operation_circuit.name)


def get_basis(n_qubit: int) -> List[str]:
    basis = []
    def helper(n: int, arr: List[int], i: int) -> None:
        if i == n:
            basis.append(''.join(arr))
            return
        arr[i] = '0'
        helper(n, arr, i + 1)
        arr[i] = '1'
        helper(n, arr, i + 1)
    helper(n_qubit, ['0']*n_qubit, 0)
    return basis


def apply_oracle_gate(type: str, input: str) -> str:
    a, b, c = input
    a, b, c = int(a), int(b), int(c)
    assert type in ['OR', 'XOR', 'AND']
    if type == 'OR': c = c ^ (a | b)
    elif type == 'XOR': c = c ^ (a ^ b)
    elif type == 'AND': c = c ^ (a & b)
    return f'{a}{b}{c}'

def test_gates(gate_operators: List[QuantumOperator]) -> None:
```

```python
    print("Testing gates...")
    basis = get_basis(3)
    gate_types = ['OR', 'XOR', 'AND']
    qr = QuantumRegister(3, name="input")
    qc = QuantumCircuit(qr)
    for gate_type, gate_operator in zip(gate_types, gate_operators):
        correct = True
        for base in basis:
            oracle_output = apply_oracle_gate(type=gate_type, input=base)
            gate = gate_operator(qr)
            qc0 = copy.deepcopy(qc)
            for i, bit in enumerate(base):
                if bit == '1': qc0.x(i)
            qc0.compose(gate, qubits=[0,1,2], inplace=True)
            state = Statevector(qc0)
            for amp, base0 in zip(state, basis):
                if base0[::-1] == oracle_output: correct = correct and amp==1
                else: correct = correct and amp==0
        msg = 'OK' if correct else 'Error'
        print(f'{gate_type} gate: {msg}.')

def apply_oracle_adder(a: str, b: str) -> str:
    # a, b in little-endian
    # return c in little-endian
    c = int(a[::-1],2) + int(b[::-1], 2)
    return f'{c:03b}'[::-1]

def test_two_bit_adder(adder: QuantumCircuit, num_anc: int, has_scratch: bool) -> None:
    if has_scratch:
        print("Testing two-bit adder with scratch...")
    else:
        print("Testing two-bit adder without scratch...")
    basis = get_basis(2)
    qr = QuantumRegister(7+num_anc, name="input")
    cr = ClassicalRegister(7+num_anc, name="measurement_outcomes")
    qc = QuantumCircuit(qr, cr)
    error = False
    for a in basis:
        for b in basis:
            c = apply_oracle_adder(a, b)
            qc0 = copy.deepcopy(qc)
            for i, bit in zip(range(0,2), a):
```

```python
                if bit == '1': qc0.x(i)
            for i, bit in zip(range(2,4), b):
                if bit == '1': qc0.x(i)
            qc0.compose(adder, qubits=[i for i in range(7+num_anc)], inplace=True)
            qc0.measure([i for i in range(7+num_anc)], cr)
            job_sim = sampler.run([transpile(qc0, backend)], shots = 1000)
            result_sim = job_sim.result()[0]

            measurements = list(result_sim.data.measurement_outcomes.get_counts().keys())
            if len(measurements)!=1:
                print(f'Error: Obtained non-deterministic result for A = {a}, B = {b}.')
                continue
            output = measurements[0][::-1]
            oa, ob, oc, od = output[0:2], output[2:4], output[4:7], output[7:]
            # Note that oa, ob, oc are all expressed as little-endian now
            has_error = True
            if oc!=c:
                print(f'Error (incorrect): A = {a}, B = {b}, expected C = {c}; got {oc}.')
            elif oa != a or ob !=b:
                print(f'Error (A or B modified): A = {a} -> {oa}, B = {b} -> {ob}.')
            elif not has_scratch and od != '0' * num_anc:
                print(f'Error (has scratch): D = {od}.')
            else: has_error = False
            error = has_error or error
    if not error: print('OK.')




def test_noisy_teleportation(noisy_tp_circuit: QuantumCircuit) -> None:
    print("Testing noisy teleportation...")
    qr1 = QuantumRegister(1, name="psi")
    qr2 = QuantumRegister(2, name="theta")
    cr = ClassicalRegister(3, name="m")
    qc0 = QuantumCircuit(qr1, qr2, cr)

    for i in range(4):
        qc = copy.deepcopy(qc0)
        if i==1:
            qc.x(0)
        elif i==2:
            qc.h(0)
```

```
        elif i==3:
            qc.x(0)
            qc.h(0)

        qc.compose(copy.deepcopy(noisy_tp_circuit), qubits=[0,1,2], inplace=True)

        if i==1:
            qc.x(2)
        elif i==2:
            qc.h(2)
        elif i==3:
            qc.h(2)
            qc.x(2)


        qc.measure(qr2[1], cr[2])
        job_sim = sampler.run([transpile(qc, backend)], shots = 1000)
        result_sim = job_sim.result()[0]
        measurements = result_sim.data.m.get_counts()
        for state in measurements.keys():
            if state[0] != '0':
                print('Error.')
                return
    print('OK.')
```

Problem 1: Implement a Quantum Circuit

In this problem, you will implement a simple quantum circuit that constructs the $|\psi\rangle = \frac{1}{\sqrt{2}}(|000\rangle - |111\rangle)$ from the all zeroes state. In other words, you will find a circuit C such that

$$C\,|000\rangle = |\psi\rangle$$

In this problem, you will use the Qiskit library to implement, visualize, and analyze the circuit C.

a). Design a circuit C to prepare the state $|\psi\rangle$, and write the corresponding Qiskit code between the "BEGIN CODE" and "END CODE" delineations below. You may use any of the gates we have learned in class. We've already created the circuit object, you just need to specify what gates to add.

In [ ]:
```python
def create_sym_state_circuit() -> QuantumCircuit:
    qr = QuantumRegister(3, name='x')
    qc = QuantumCircuit(qr)

    # ========= BEGIN CODE =================


    # ========= END CODE =================


    return qc

qc = create_sym_state_circuit()
qc.draw()
```

We can furthermore print out the output state of the circuit you just created:

In [ ]:
```python
ghz_circuit = create_sym_state_circuit()
final_state = Statevector(ghz_circuit)
final_state.draw(output='latex')
```

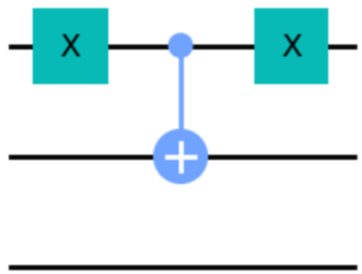
b) Update your circuit to measure $|\psi\rangle$ in the standard basis, and visualize the measurement statistics of 1000 shots using a histogram. We have provided an `sampler` object (of Type `SamplerV2`) to help you simulate the circuit.

In [ ]:
```python
# here's an example of running the circuit using the sampler object
# replace YOUR_CIRCUIT_OBJECT with the variable corresponding to your quantum circuit
#job_sim = sampler.run([transpile(YOUR_CIRCUIT_OBJECT, backend)], shots=1000)
#result_sim = job_sim.result()[0]
#counts = result_sim.data.output.get_counts()
#plot_histogram(counts)

# ========= BEGIN CODE =================


# ========= END CODE =================
```

c) Consider running the following circuit with $|\psi\rangle$ as input. Let $|\theta\rangle$ denote the output state. Calculate the state $|\theta\rangle$ by computing the intermediate states of the circuit, and write it out below in $\LaTeX$ below where it says **Solution**.



Solution

d) Write code to implement a circuit that prepares the state $|\theta\rangle$, measure it in the standard basis and visualize the measurement statistics of 1000 shots using a histogram.

In [ ]:
```
# ========= BEGIN CODE =================


# ========= END CODE =================
```

Problem 2: A Quantum Two-bit Adder

The classical two-bit adder is an **irreversible** function that takes in four bits, (A_0, A_1) and (B_0, B_1), and outputs three bits (C_0, Q_0, Q_1) which is the binary representation of the sum of $2A_1 + A_0$ and $2B_1 + B_0$ (i.e., integers that (A_0, A_1) and (B_0, B_1) represent in binary). For example, on input $(0, 1)$ and $(1, 1)$ the two bit adder should return $(1, 0, 1)$. On input $(1, 1)$ and $(1, 1)$ it should output $(0, 1, 1)$.

You can find a circuit for an irreversible circuit for the two-bit adder here, consisting of XOR, OR, and AND gates (see Wikipedia for gate symbol reference).

In this problem you will implement a **reversible** two-bit adder in Qiskit.

a) First, let's implement reversible versions of the XOR, OR, and AND gates. Recall that every boolean function f can be converted to a reversible transformation T_f using an additional ancilla bit. Since XOR, OR, AND map 2 bits to 1 bit, the reversible functions T_{XOR}, T_{OR}, T_{AND} will map 3 bits to 3 bits. The corresponding matrices are 8×8. Specifically, we want

$$T_f |a, b, c\rangle = |a, b, f(a, b) \oplus c\rangle$$

In the functions below, enter the matrix representations of T_{XOR}, T_{OR}, T_{AND} below (replace the entries with the appropriate values). Be cognizant of the row/column ordering convention!

Your implementations of reversible XOR, OR, and AND will be tested.

In [ ]:
```
def create_Tor(qr: QuantumRegister) -> QuantumCircuit:
    assert len(qr) == 3, 'Tor gate should operate on 3 qubits.'
    qc = QuantumCircuit(qr)
    ##### FILL IN THE MATRIX BELOW FOR THE REVERSIBLE OR GATE ##########
    Tor = Operator([
        [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],
    ])
    ####################################################################
    qc.unitary(Tor, [2, 1, 0], label='Tor')
    return qc

def create_Txor(qr: QuantumRegister) -> QuantumCircuit:
    assert len(qr) == 3, 'Txor gate should operate on 3 qubits.'
    qc = QuantumCircuit(qr)
    ##### FILL IN THE MATRIX BELOW FOR THE REVERSIBLE XOR GATE ##########
    Txor = Operator([
        [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],
    ])
```

```
    #################################################################
    qc.unitary(Txor, [2, 1, 0], label='Txor')
    return qc

def create_Tand(qr: QuantumRegister) -> QuantumCircuit:
    assert len(qr) == 3, 'Tand gate should operate on 3 qubits.'
    qc = QuantumCircuit(qr)
    ##### FILL IN THE MATRIX BELOW FOR THE REVERSIBLE AND GATE ##########
    Tand = Operator([
        [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],
    ])
    #################################################################
    qc.unitary(Tand, [2, 1, 0], label='Tand')
    return qc
```

In [ ]:
```
#Running test cases on your adder....
test_gates([create_Tor,create_Txor,create_Tand])
```

b) You can now put together reversible circuits consisting of T_{XOR}, T_{OR}, and T_{AND} by using the functions `create_Tor`, `create_Txor`, and `create_Tand`, and also a helper function called `append` that allows you to append a gate G to a circuit C. The function takes in a circuit C, a function g constructs the gate G, and a list of bits that G operates on. See the code below as an example.

In [ ]:
```
### EXAMPLE ONLY ###############
qx = QuantumRegister(2, name="x")
qa = QuantumRegister(2, name="a")
qc = QuantumCircuit(qx,qa)                #add the X register and A registers
qc = append(qc, create_Tand, [0, 1, 2])   #reversible AND acting on (x0,x1) and ancilla a0
qc = append(qc, create_Txor, [0,1,3])     #reversible XOR acting on (x0,x1) and ancilla a1
qc.draw()
```

Now, transform the irreversible circuit for the two-bit adder above to a **reversible** circuit C for the two-bit adder. More precisely, the circuit C should act on bits

- (A_0, A_1) representing the first number $A = 2A_1 + A_0$
- (B_0, B_1) representing the second number $B = 2B_1 + B_0$
- (C_0, C_1, C_2) representing the binary representation of $A + B$
- Some number of ancilla bits $(D_0, D_1, \ldots)$

The circuit C should have the behavior: for all inputs $A_0, A_1, B_0, B_1 \in \{0, 1\}$,

$$C\,|A, B, 0, 0 \cdots 0\rangle = \left|\underbrace{A}_{2\text{ bits}}, \underbrace{B}_{2\text{ bits}}, \underbrace{A + B}_{3\text{ bits}}, \underbrace{S_{A,B}}_{\text{ancillas}}\right\rangle$$

where A, B are two bits and $A + B$ is represented by three bits. $S_{A,B}$ corresponds to the bits of the ancilla that depends on the inputs A, B. This data corresponds to the "scratch work" of the computation. We will assume all ancillas are initiated to $|0\rangle$.

Your circuit C can use T_{XOR}, T_{AND}, T_{OR}, as well as $CNOT$, X, Z and H gates. Choose the appropriate number of ancillas, and then implement your circuit where indicated. The code afterwards will visualize your circuit as well run it on several test cases.

In [ ]:

```
# Todo: fill in the number of ancillary qubits for your circuit
num_anc = 1

def create_two_bit_adder_with_scratch(num_anc) -> QuantumCircuit:
    A = QuantumRegister(2, name="a")
    B = QuantumRegister(2, name="b")
    C = QuantumRegister(3, name="c")
    D = QuantumRegister(num_anc, name="d")
    qc = QuantumCircuit(A,B,C,D)

    #### BEGIN YOUR CODE HERE ############################


    ##### END YOUR CODE HERE ###############################
```

```
    return qc

two_bit_adder_with_scratch = create_two_bit_adder_with_scratch(num_anc=num_anc)
two_bit_adder_with_scratch.draw()
```

In [ ]:

```
# Running test cases on your adder....
test_two_bit_adder(two_bit_adder_with_scratch, num_anc, has_scratch=True)
```

c) Now we go one step further to implement a reversible two-bit adder that does the same thing as above except the scratch bits start **and end** in the zero state.

$$C\,|A, B, 0, 0\cdots 0\rangle = |A, B, A+B, 0\cdots 0\rangle$$

In other words, the scratch work is erased.

In [ ]:

```
def create_two_bit_adder() -> QuantumCircuit:
    A = QuantumRegister(2, name="a")
    B = QuantumRegister(2, name="b")
    C = QuantumRegister(3, name="c")
    D = QuantumRegister(num_anc, name="d")
    qc = QuantumCircuit(A,B,C,D)

    #### BEGIN YOUR CODE HERE ############################



    ##### END YOUR CODE HERE ###############################

    return qc

two_bit_adder = create_two_bit_adder()
two_bit_adder.draw()
```

In [ ]:

```
#Running test cases on your adder....
test_two_bit_adder(two_bit_adder, num_anc, has_scratch=False)
```

Problem 3: Non-standard Basis Measurements

a) Consider an orthonormal basis $B = \{|b_1\rangle, \ldots, |b_d\rangle\}$ for $\mathbb{C}^d$. As we learned in class, measuring a quantum state $|\psi\rangle \in \mathbb{C}^d$ according to the basis B yields outcome $|b_j\rangle$ with probability $|\langle b_j|\psi\rangle|^2$.

In class we also learned that this process was equivalent to first applying a unitary U on $|\psi\rangle$, and then measuring the resulting state in the standard basis. In other words, the probability of obtaining standard basis outcome $|j\rangle$ when measuring $U|\psi\rangle$ in the standard basis, equal to $|\langle b_j|\psi\rangle|^2$. What unitary U accomplishes this? Given an algebraic expression for U, such as a sum of outer products, or a description of the rows/columns of U, etc. Then prove that it works.

Your Solution:

write your solution here, using LaTeX and Markdown

b) Now let's implement the unitary for measuring in the following basis B:

$$|\psi_0\rangle = \cos(\pi/8)|0\rangle + \sin(\pi/8)|1\rangle$$

and

$$|\psi_1\rangle = -\sin(\pi/8)|0\rangle + \cos(\pi/8)|1\rangle$$

First, write down the measurement probabilities if we measure the following states in the basis B:

$$|1\rangle, |-\rangle, |+\rangle, \cos(\pi/8)|0\rangle + \sin(\pi/8)|1\rangle$$

c) In the code below, write the matrix U that implements the change of basis from the standard basis to the basis above.

In [ ]:
```
# ========= BEGIN CODE =================

U = [[1, 0], [0, 1]]    # <-- edit this matrix!

# ========= END CODE =================

def perform_basis_measurement(initial_state: List[float]) -> QuantumCircuit:
    qr = QuantumRegister(1, name="input_state")
    cr = ClassicalRegister(1, name="output")
    qc = QuantumCircuit(qr, cr)
    qc.initialize(initial_state)
```

```
    qc.append(UnitaryGate(U), qr)
    qc.measure(qr, cr)
    return qc
```

Now we'll test your basis change on some states and plot their measurement statistics. You should use this to check whether you implemented the right basis change U.

In [ ]:
```
#First, we test it on the |1> state
qc1 = perform_basis_measurement([0.0, 1.0])
qc1.draw()

job_sim = sampler.run([transpile(qc1, backend)], shots=5024)
# Grab the results from the job.
result_sim = job_sim.result()[0]
counts = result_sim.data.output.get_counts()

plot_histogram(counts)
```

In [ ]:
```
#...and the |+> state
qc1 = perform_basis_measurement([1.0/np.sqrt(2), 1.0/np.sqrt(2)])
qc1.draw()

job_sim = sampler.run([transpile(qc1, backend)], shots=5024)
# Grab the results from the job.
result_sim = job_sim.result()[0]
counts = result_sim.data.output.get_counts()

plot_histogram(counts)
```

In [ ]:
```
# Next we try it on the |-> state
qc1 = perform_basis_measurement([1.0/np.sqrt(2), -1.0/np.sqrt(2)])
qc1.draw()

job_sim = sampler.run([transpile(qc1, backend)], shots=5024)
# Grab the results from the job.
result_sim = job_sim.result()[0]
counts = result_sim.data.output.get_counts()

plot_histogram(counts)
```

In [ ]:
```
#and now the cos(pi/8) |0> + sin(pi/8) |1> state
qc1 = perform_basis_measurement([np.cos(math.pi/8), np.sin(math.pi/8)])
qc1.draw()

job_sim = sampler.run([transpile(qc1, backend)], shots=5024)
# Grab the results from the job.
result_sim = job_sim.result()[0]
counts = result_sim.data.output.get_counts()

plot_histogram(counts)
```

Problem 4: EPR Pair Properties

Let's examine properties of the EPR pair

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

In what follows, let's suppose that Alice is given the left qubit of the EPR pair, and Bob is given the right qubit, and they are separated by a large distance.

a) Let $A = \{|a_1\rangle, |a_2\rangle\}$ be some orthonormal basis for $\mathbb{C}^2$. Suppose Alice measures her qubit using basis A. What are the statistics of the measurement outcomes (i.e. what are the probability of $|a_1\rangle$ or $|a_2\rangle$)?

Your Solution:

write your solution here, using LaTeX and Markdown

b) Show that if Alice obtains measurement outcome $|a_i\rangle$ for some $i \in \{1, 2\}$, the post-measurement state of the EPR pair is $|a_i\rangle \otimes |a_i\rangle^*$ where $|a_i\rangle^*$ is the **complex conjugate** of $|a_i\rangle$ (i.e. the j-th entry is the complex conjugate of the j-th entry of $|a_i\rangle$).

This is interesting because Alice might have decided on the basis only after Bob was sent away, yet Alice's measurement causes Bob's qubit to instantaneously collapse into one of the basis states of A (up to complex conjugation). This is a phenomenon called **quantum steering**, because Alice is able to **steer** Bob's qubit, even though she is only acting on **her** qubit.**c)** In the code below, write the matrix U that implements the change of basis from the standard basis to the basis above.

Your Solution:

write your solution here, using LaTeX and Markdown

c) Suppose that Bob then measures his qubit using an orthonormal basis $B = \{|b_1\rangle, |b_2\rangle\}$. What are the statistics of his measurement outcomes, conditioned on Alice's outcome?

Your Solution:

write your solution here, using LaTeX and Markdown

d) Suppose the order of measurements were reversed: Bob measures his qubit first using basis B, and then Alice measures her qubit using basis A. Show that the **joint** probability distribution of their measurement outcomes is the same as before.

Your Solution:

write your solution here, using LaTeX and Markdown

e) What can you conclude about the effectiveness of using quantum entanglement and quantum steering as a method for faster-than-light communication? In other words, can Alice and Bob, by only making local measurements on their entangled state, send information to each other?

Your Solution:

write your solution here, using LaTeX and Markdown

Problem 5: Quantum Teleportation with Noise

We saw how to teleport quantum states in class. Let's consider a twist on the standard teleportation protocol. Let's imagine that when Alice and Bob meet up to create an entangled state, the settings on their lab equipment was screwed up and they accidentally create the following two-qubit entangled state

$$|\theta\rangle = \frac{1}{\sqrt{3}}|00\rangle - \frac{1}{\sqrt{6}}|01\rangle + \frac{1}{\sqrt{6}}|10\rangle + \frac{1}{\sqrt{3}}|11\rangle \ .$$

Only Alice realizes this after they haven each taken a qubit each and gone their separate ways.

Suppose that Alice now gets a gift qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. Is there a way that she can still teleport $|\psi\rangle$ to Bob, using their corrupted entangled state $|\theta\rangle$ and the classical communication channel? Like in the standard teleportation protocol, Alice can only apply unitaries and measurements to her two qubits, and Bob will apply the same corrections as in the standard teleportation protocal (since he's not aware of the corruption).

a) Show how the teleportation protocol can be adapted for the corruption from Alice's side and analyze the correctness of your proposed protocol.

Your Solution:

write your solution here, using LaTeX and Markdown

b) Now let's implement Alice's teleportation protocol using the noisy EPR pair with qiskit.

Write code in `create_alice_noisy_tp_circuit` function below, which takes as as input a QuantumRegister (consisting of two qubits) and a ClassicalRegister (consisting of two 2 bits).

Important Note: the register indices in Alice's and Bob's functions are **local** (0-indexed), meaning that from Alice or Bob's point of view, her zeroth qubit is the gift qubit, and her first qubit is the first half of the EPR pair. From Bob's point of view, he only has the other half of the EPR pair, which he considers his zeroth qubit.

In [ ]:
```
def initialize_noisy_epr_pair(qc: QuantumCircuit, qubits: List[int]) -> QuantumCircuit:
    # For qc.initialize, the ordering of the states are |00>, |01>, |10>, |11>
    #if the top wire corresponds to the rightmost bit (recall the little endian convention of Qiskit)
    qc.initialize([np.sqrt(1/3.0), np.sqrt(1/6.0), -np.sqrt(1/6.0), np.sqrt(1/3.0)], qubits = qubits)
    qc.barrier()
    return qc
```

```python
def create_base_noisy_tp_circuit() -> QuantumCircuit:
    qr1 = QuantumRegister(1, name="psi")
    qr2 = QuantumRegister(2, name="theta")
    cr = ClassicalRegister(2, name="m")
    qc = QuantumCircuit(qr1, qr2, cr)
    return initialize_noisy_epr_pair(qc, [1, 2])

def create_alice_noisy_tp_circuit(qr: QuantumRegister, cr: ClassicalRegister) -> QuantumCircuit:
    qc = QuantumCircuit(qr, cr)
    # Alice has two qubits (index 0,1) and access to two classical registers (index 0,1)
    # ========= BEGIN CODE =================


    # ========= END CODE =================
    return qc

def create_bob_noisy_tp_circuit(qr: QuantumRegister, cr: ClassicalRegister) -> QuantumCircuit:
    qc = QuantumCircuit(qr, cr)
    qc.z(0).c_if(cr[0], 1) # Apply gates if the registers
    qc.x(0).c_if(cr[1], 1) # are in the state '1'
    return qc
```

In [ ]:
```python
noisy_tp_circuit = create_base_noisy_tp_circuit()
noisy_tp_circuit = append2(noisy_tp_circuit, create_alice_noisy_tp_circuit, [0,1], [0,1])
noisy_tp_circuit = append2(noisy_tp_circuit, create_bob_noisy_tp_circuit, [2], [0,1])
noisy_tp_circuit.draw()
```

In [ ]:
```python
test_noisy_teleportation(noisy_tp_circuit)
```