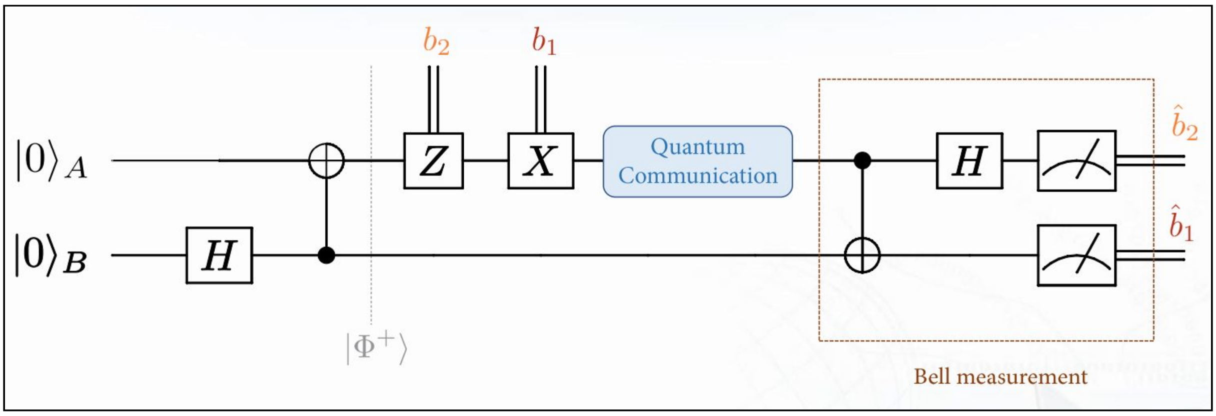
通信實驗 Lab 2

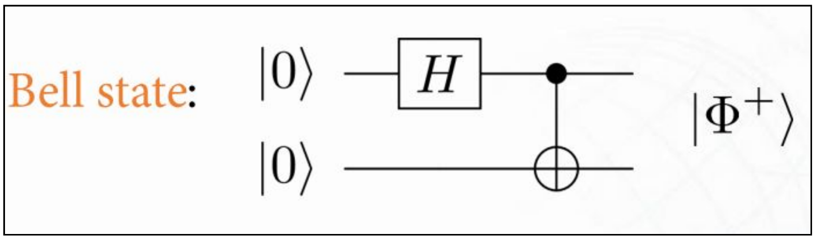
B07901103 電機三 陳孟宏

<Problem 1> Superdense Coding

(a)



- Step 1: Start with Charlie, a third party, to prepare an entangled state.




```
def create_bell_pair(qc, a, b):
    qc.h(a) # Apply a h-gate to the first qubit
    qc.cx(a,b) # Apply a CNOT, using the first qubit as the control
```

- Step 2: Charlie sends the first qubit to Alice and the second qubit to Bob. Before sending 2 classical bits of information to Bob, Alice needs to apply a set of quantum gates to her qubit depending on the 2 bits of information she wants to send.

Intended Message	Applied Gate	Resulting State ($\cdot\sqrt{2}$)
00	I	$\|00\rangle + \|11\rangle$
10	X	$\|01\rangle + \|10\rangle$
01	Z	$\|00\rangle - \|11\rangle$
11	ZX	$\|10\rangle - \|01\rangle$

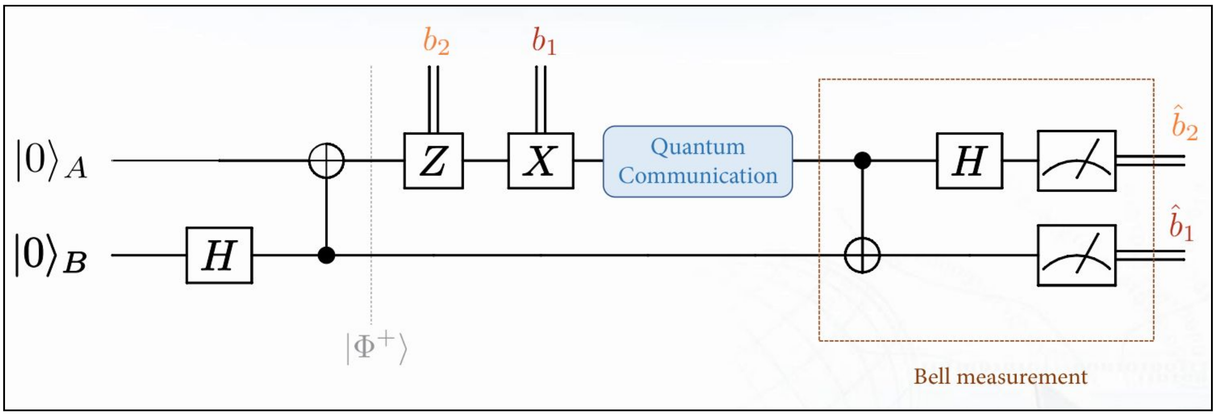
```python
def encode_message(qc, qubit, msg):
    if msg == "00":
        pass    # To send 00 we do nothing
    elif msg == "10":
        qc.z(qubit) # To send 10 we apply an Z-gate
    elif msg == "01":
        qc.x(qubit) # To send 01 we apply a X-gate
    elif msg == "11":
        qc.x(qubit) # To send 11, we apply a Z-gate
        qc.z(qubit) # followed by an X-gate
    else:
        print("Failed")
```

- Step 3: Bob receives Alice's qubit and uses his qubit to decode Alice's message.

Bob Receives:	After CNOT-gate:	After H-gate:
$\|00\rangle + \|11\rangle$	$\|00\rangle + \|01\rangle$	$\|00\rangle$
$\|01\rangle + \|10\rangle$	$\|11\rangle + \|10\rangle$	$\|10\rangle$
$\|00\rangle - \|11\rangle$	$\|00\rangle - \|01\rangle$	$\|01\rangle$
$\|10\rangle - \|01\rangle$	$\|10\rangle - \|11\rangle$	$\|11\rangle$

```python
def decode_message(qc, a, b):
    qc.cx(a,b)
    qc.h(a)
```

- So we first randomly generate Alice's bits (4-classical-bits). Note that the list of "alice_bit" will be like [0, 1, 1, 1], based on the model, the "measured" output from my 4-bits circuit will be [q3, q2, q1, q0] respectively. (top qubit = q0). Hence, I need to input a "message" that is reversed with "alice_bit", which will be [1, 1, 1, 0], and put them into q0's b2、b1 and q2's b2、b1 in order.



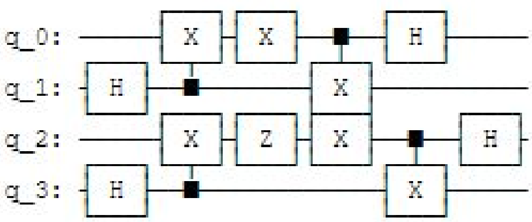
```
from numpy.random import randint
import numpy as np

# Alice's bit = b4b3b2b1
n = 4
a_bits = randint(2, size=n) #alice_bits
message = str(a_bits[3])+str(a_bits[2])+str(a_bits[1])+str(a_bits[0])
print(a_bits)
print("Message: ", message)
```

```
[0 1 1 1]
Message:  1110
```

- Then, I create a 4-qubits quantum circuit to carry |00>, |01>, |10>, |11>, etc, which can transmit 4 classical bits. Here, we input a “message = 1110” (Only extend 2-qubits quantum circuit.)

```
# Parameter
measure = 0 #record the length of measurement_result
bob_s = 0 #record the symbol error
bob_b = 0 #record the bit error
first_part = str(a_bits[0]) + str(a_bits[1])
second_part = str(a_bits[2]) + str(a_bits[3])

# Superdense Coding
qc = QuantumCircuit(4)

create_bell_pair(qc, 1, 0)
create_bell_pair(qc, 3, 2)

encode_message(qc, 0, first_part)
encode_message(qc, 2, second_part)

# After recieving qubit 0, Bob applies the recovery protocol:
decode_message(qc, 0, 1)
decode_message(qc, 2, 3)

print("First part: ", first_part)
print("Second part: ", second_part)

qc.draw()
```

```
First part:  01
Second part:  11
```



- At last, I compare the measured bits with the input bits to calculate bit / symbol error rate. Because it is 4 bits, one bit for ¼ probability and one symbol (2 bits) for ½ probability.

```python
# Finally, Bob measures his qubits to read Alice's message
qc.measure_all()

backend = Aer.get_backend('qasm_simulator')
job_sim = execute(qc, backend, shots=1024)
sim_result = job_sim.result()

measurement_result = sim_result.get_counts(qc)
print(measurement_result)

# Calculate the symbol/bit error rate
for key in measurement_result:
    measure += measurement_result[key]
    # first part
    if (first_part == key[3]+key[2]):
        bob_s += measurement_result[key]/2
    if (first_part[0] == key[3]):
        bob_b += measurement_result[key]/4
    if (first_part[1] == key[2]):
        bob_b += measurement_result[key]/4
    # second_part
    if (second_part == key[1]+key[0]):
        bob_s += measurement_result[key]/2
    if (second_part[0] == key[1]):
        bob_b += measurement_result[key]/4
    if (second_part[1] == key[0]):
        bob_b += measurement_result[key]/4

# Symbol Error Rate
s_rate = (1 - (bob_s/measure))*100
print("Symbol Error Rate: %2.f%%" % s_rate)

# Bit Error Rate
b_rate = (1 - (bob_b/measure))*100
print("Bit Error Rate: %2.f%%" % b_rate)
```

```
{'1110': 1024}
Symbol Error Rate:  0%
Bit Error Rate:  0%
```

(b)

- First, log in and query for the least busy IBMQ backend

```python
from qiskit import IBMQ
from qiskit.providers.ibmq import least_busy
shots = 256

# Load local account information
#IBMQ.save_account('cdad9ef01fbad88f5ee99fa7fc4cfb7b5eb8755c52fd1697690c6b680797a57384a80d5a6bfd5e99332b8ee8a58e74e3903034a7f4c1
IBMQ.load_account()

# Get the least busy backend
provider = IBMQ.get_provider(hub='ibm-q')
backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits >= 2 and not x.configuration().simulator and
print("least busy backend: ", backend)

# Run our circuit
job = execute(qc, backend=backend, shots=shots)
```

```
C:\Users\user\Downloads\anaconda\envs\IBMQ\lib\site-packages\qiskit\providers\ibmq\ibmqfactory.py:192: UserWarning: Timestamps
in IBMQ backend properties, jobs, and job results are all now in local time instead of UTC.
  warnings.warn('Timestamps in IBMQ backend properties, jobs, and job results '

least busy backend:  ibmq_santiago
```

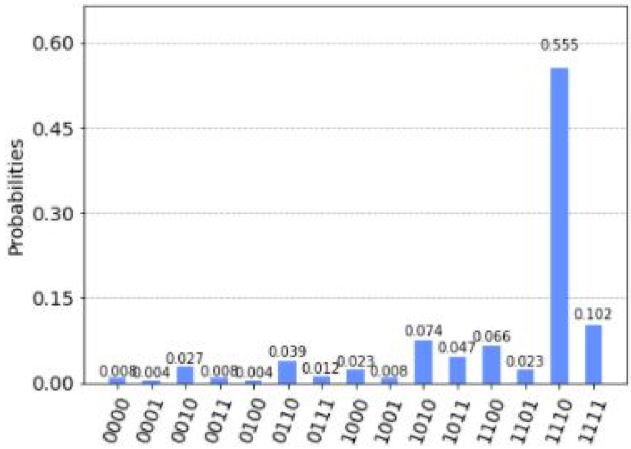
- I use a monitor to see when will turn to me.

```python
# Monitoring our job
from qiskit.tools.monitor import job_monitor
job_monitor(job)
```

```
Job Status: job has successfully run
```

- We can see that the message is “1110”, but in the real IBMQ, “1110” does not get the probability of 100%.

```python
# Plotting our result
result = job.result()
plot_histogram(result.get_counts(qc))
```

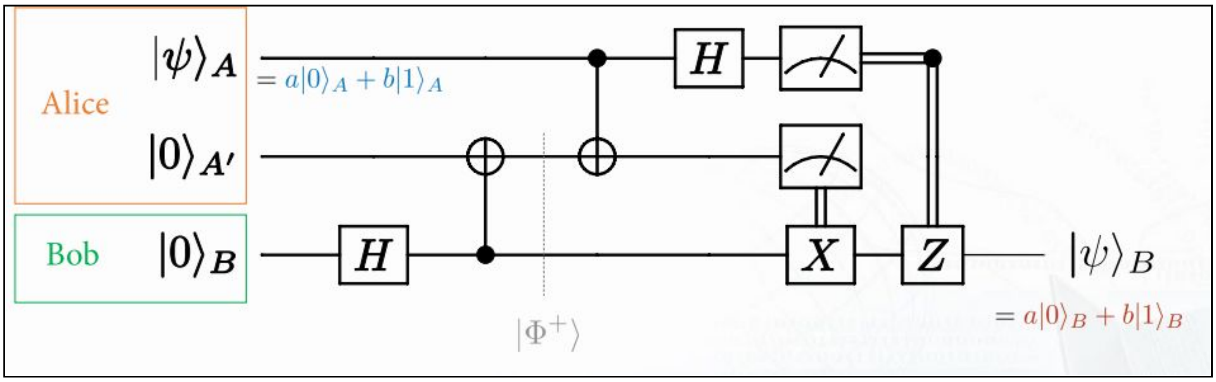


- Hence, it still has symbol and bit error.

```python
correct_results = result.get_counts(qc)
print("Message: ", message)
print("Measurement result: ", correct_results)
# Parameter
measure = 0 #record the length of measurement_result
bob_s = 0 #record the symbol error
bob_b = 0 #record the bit error
first_part = str(message[0]) + str(message[1])
second_part = str(message[2]) + str(message[3])

# Calculate the symbol/bit error rate
for key in correct_results:
    measure += correct_results[key]
    # first part
    if (first_part == key[0:2]):
        bob_s += correct_results[key]/2
    if (first_part[0] == key[0]):
        bob_b += correct_results[key]/4
    if (first_part[1] == key[1]):
        bob_b += correct_results[key]/4
    # second_part
    if (second_part == key[2:4]):
        bob_s += correct_results[key]/2
    if (second_part[0] == key[2]):
        bob_b += correct_results[key]/4
    if (second_part[1] == key[3]):
        bob_b += correct_results[key]/4

# Symbol Error Rate
s_rate = (1 - (bob_s/measure))*100
print("Symbol Error Rate: %.3f%%" % s_rate)

# Bit Error Rate
b_rate = (1 - (bob_b/measure))*100
print("Bit Error Rate: %.3f%%" % b_rate)
```

```
Message:  1110
Measurement result:  {'0000': 2, '0001': 1, '0010': 7, '0011': 2, '0100': 1, '0110': 10, '0111': 3, '1000': 6, '1001': 2, '101
0': 19, '1011': 12, '1100': 17, '1101': 6, '1110': 142, '1111': 26}
Symbol Error Rate: 27.930%
Bit Error Rate: 16.016%
```

<Problem 2> Quantum Teleportation



(a)-(1) Use the “statevector_simulator”

- Step 1: A third party, Charlie, creates an entangled pair of qubits and gives one to Bob and one to Alice.

```
def create_bell_pair(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
```

- Step 2: I create a “alice_gate”, use a CNOT controlled by |ψ⟩ (the qubit she is trying to send Bob), and then use an “H-gate”.

```
def alice_gates(qc, psi, a):
    qc.cx(psi, a)
    qc.h(psi)
```

- Step 3: Alice applies a measurement to both qubits that she owns, q1 and |ψ⟩, and stores this result in two classical bits. She then sends these two bits to Bob.

```
def measure_and_send(qc, a, b):
    #Measures qubits a & b and 'sends' the results to Bob
    qc.barrier()
    qc.measure(a,0)
    qc.measure(b,1)
```

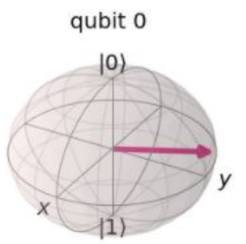
- Step 4: Then Bob applies the following gates depending on the state of the classical bits.
    - |00> : Do nothing
    - |01> : Apply X-gate
    - |10> : Apply Z-gate
    - |11> : Apply ZX-gate

```
def bob_gates(qc, qubit, crz, crx):
    # Here we use c_if to control our gates with a classical
    # bit instead of a qubit
    qc.x(qubit).c_if(crx, 1) # Apply gates if the registers
    qc.z(qubit).c_if(crz, 1) # are in the state '1'
```

- So first, I use random.uniform() and u3-gate to generate a random |ψ⟩ state. (I cannot pip install “qiskit.textbook”)

```python
# Create one qubits and one classical bits
import random
q_init = QuantumRegister(1)
c_init = ClassicalRegister(1)

# Create a quantum circuit
cir_init = QuantumCircuit(q_init, c_init)
cir_init.u3(random.uniform(0, 2*pi), random.uniform(0, 2*pi), random.uniform(0, 2*pi), q_init[0])

sim_init = Aer.get_backend('statevector_simulator')
job_init = execute(cir_init, sim_init)
result_init = job_init.result()
statevector_init = result_init.get_statevector()
print(statevector_init)

init_gate = Initialize(statevector_init)
init_gate.label = "init"

# Show it on a Bloch sphere
plot_bloch_multivector(statevector_init)
```

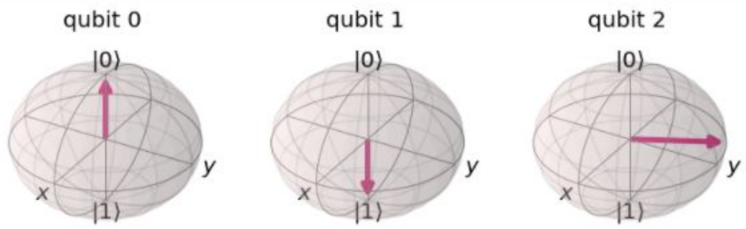
```
[-0.71656205+0.j         0.2901672 -0.6343042j]
```



- Then, I use “statevector_simulator” to create the above quantum circuit and observe that the message Bob gets (qubit 2), is the same as that of Alice sends (the above random state)

```python
## SETUP
qr = QuantumRegister(3, name="q")   # Protocol uses 3 qubits
crz = ClassicalRegister(1, name="crz") # and 2 classical registers
crx = ClassicalRegister(1, name="crx")
qc = QuantumCircuit(qr, crz, crx)

## STEP 0
# First, let's initialize Alice's q0
qc.append(init_gate, [0])
qc.barrier()

## STEP 1
# Now begins the teleportation protocol
create_bell_pair(qc, 1, 2)
qc.barrier()

## STEP 2
# Send q1 to Alice and q2 to Bob
alice_gates(qc, 0, 1)

## STEP 3
# Alice then sends her classical bits to Bob
measure_and_send(qc, 0, 1)

## STEP 4
# Bob decodes qubits
bob_gates(qc, 2, crz, crx)
```

```python
backend = BasicAer.get_backend('statevector_simulator')
out_vector = execute(qc, backend).result().get_statevector()
plot_bloch_multivector(out_vector)
```



(a)-(2) Use the “qasm_simulator”

- Quantum teleportation

$$|0\rangle \xrightarrow{\text{Initialize}} |\psi\rangle$$

- Inverse initialization

$$|\psi\rangle \xrightarrow{\text{Inverse Initialize}} |0\rangle$$

- Hence, after inverse initializing, we measure q[2] and it will be ‘0’ for all time, no matter q[1]、q[0] are (00 / 01 / 10 / 11)

```
inverse_init_gate = init_gate.gates_to_uncompute()

## SETUP
qr = QuantumRegister(3, name="q")   # Protocol uses 3 qubits
crz = ClassicalRegister(1, name="crz") # and 2 classical registers
crx = ClassicalRegister(1, name="crx")
qc = QuantumCircuit(qr, crz, crx)

## STEP 0
# First, let's initialize Alice's q0
qc.append(init_gate, [0])
qc.barrier()

## STEP 1
# Now begins the teleportation protocol
create_bell_pair(qc, 1, 2)
qc.barrier()

## STEP 2
# Send q1 to Alice and q2 to Bob
alice_gates(qc, 0, 1)

## STEP 3
# Alice then sends her classical bits to Bob
measure_and_send(qc, 0, 1)

## STEP 4
# Bob decodes qubits
bob_gates(qc, 2, crz, crx)

## STEP 5
# reverse the initialization process
qc.append(inverse_init_gate, [2])
```

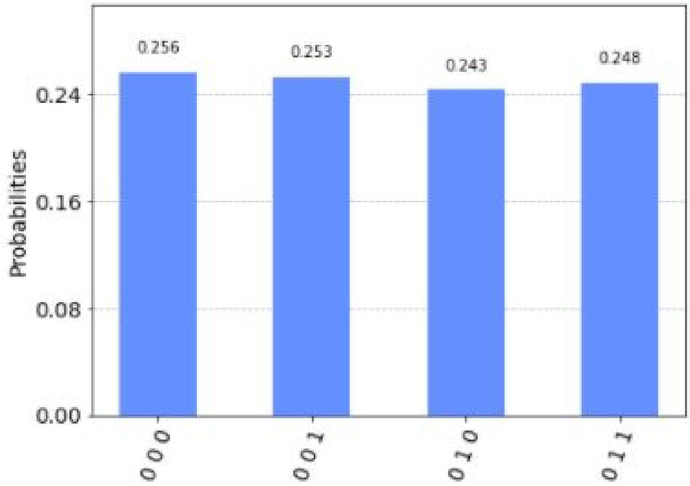
```
<qiskit.circuit.instructionset.InstructionSet at 0x1a222e3c940>
```

(Initialization)

```python
# Need to add a new ClassicalRegister to see the result
cr_result = ClassicalRegister(1)
qc.add_register(cr_result)
qc.measure(2,2)
```

```
<qiskit.circuit.instructionset.InstructionSet at 0x1a22279f9a0>
```

```python
backend = BasicAer.get_backend('qasm_simulator')
counts = execute(qc, backend, shots=1024).result().get_counts()
plot_histogram(counts)
```

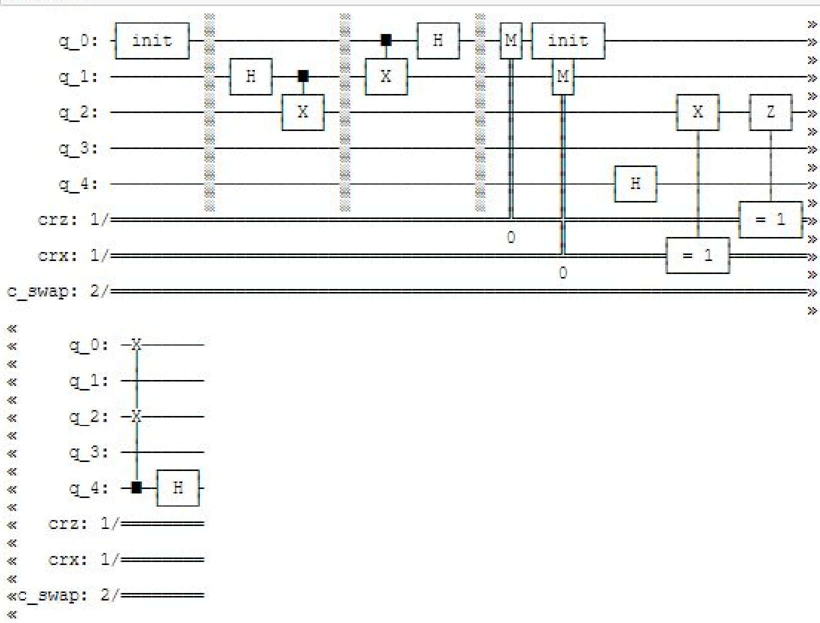


(q[2], at the bottom, is always ‘0’)

(a)-(3) Use the “Swap Test”

- I create a 5-qubits quantum register to use “cswap” on q[4], and compare Alice’s and Bob’s quantum state.

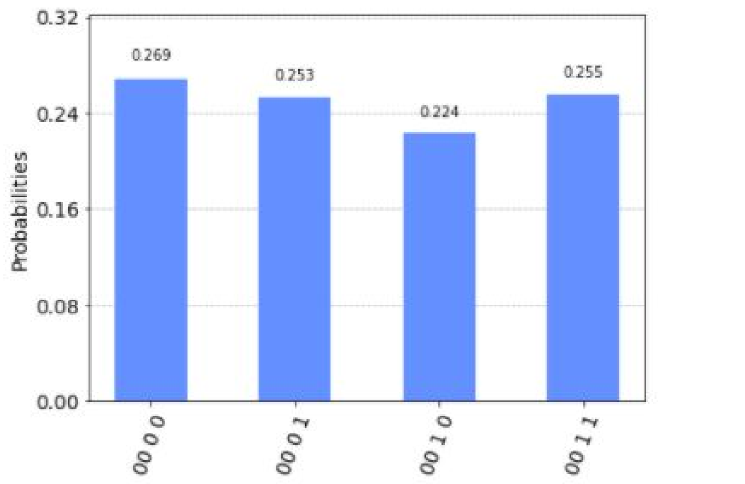
```python
# states are the same --> output: '100'
qc.append(init_gate, [0])
qc.h(qr[4])
qc.cswap(qr[4], qr[2], qr[0])
qc.h(qr[4])

# Draw circuit
qc.draw()
```



- I use the “qasm_simulator” and find that the q[2] is always ‘0’, so Alice and Bob has the same quantum state.

```
# Measure > {'01': 1024}
qc.measure(qr[4], c_swap[1])
sim_swap = Aer.get_backend('qasm_simulator')
job_swap = execute(qc, sim_swap, shots=1024)
result_swap = job_swap.result()
counts = result_swap.get_counts()
print(counts)

backend = BasicAer.get_backend('qasm_simulator')
counts = execute(qc, backend, shots=1024).result().get_counts()
plot_histogram(counts)
```

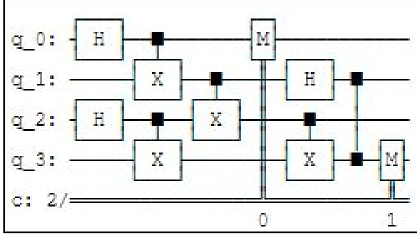
```
{'00 0 0': 252, '00 0 1': 245, '00 1 0': 259, '00 1 1': 268}
```



(b) Entanglement Swapping (Quantum Repeater)

- I create a 4-bits quantum circuit, q[0] is Alice’s, q[1] and q[2] is Chalie’s, while q[3] is Bob’s. Entangle them.

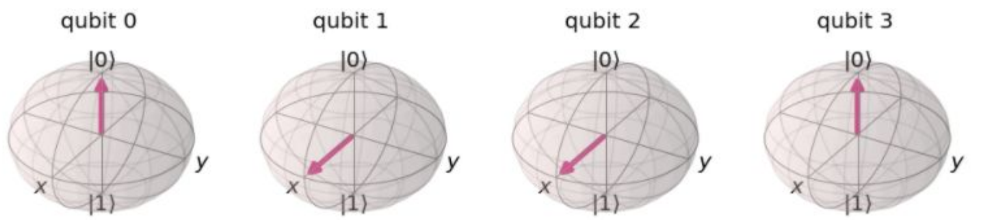
```
# Problem 2-(b) Entanglement Swapping
q_es = QuantumRegister(4, 'q')
c_es = ClassicalRegister(2, 'c')
cir_es = QuantumCircuit(q_es, c_es)

cir_es.h(q_es[0])
cir_es.h(q_es[2])
cir_es.cx(q_es[0], q_es[1])
cir_es.cx(q_es[2], q_es[3])
cir_es.cx(q_es[1], q_es[2])
cir_es.h(q_es[1])
cir_es.cx(q_es[2], q_es[3])
# Control-Z (cz) gate
cir_es.cz(q_es[1], q_es[3])

#Measure
cir_es.measure([0, 3], [0, 1])

cir_es.draw()
```



- Measuring Alice’s (q[0]) and Bob’s (q[3]) state at the end, we will find that they are always the same, either “00” or “11”.

```
# Measure > {'01': 1024}

sim1 = Aer.get_backend('qasm_simulator')
job1 = execute(cir_es, sim1, shots=1024)
result1 = job1.result()
counts1 = result1.get_counts()
print(counts1)

backend1 = BasicAer.get_backend('statevector_simulator')
out_vector1 = execute(cir_es, backend1).result().get_statevector()
plot_bloch_multivector(out_vector1)
```

```
{'00': 490, '11': 534}
```



<Problem 3> BB84 Protocol

(a) Run the sample codes to calculate the probability of Eve guessing Alice’s bit correctly.

- Alice generates bits.

```
n = 100; # number of qubits used in the BB84
np.random.seed(seed=3)
## Step 1
# Alice generates bits
alice_bits = randint(2, size=n)
print(alice_bits)
print("Alice's first bit = %i" % alice_bits[0])
```

```
[0 0 1 1 0 0 0 1 1 1 0 1 1 1 0 1 1 0 0 0 0 1 1 0 0 0 1 0 0 0 0 1 0 1 1 0 1
 0 0 1 1 0 0 1 0 1 0 1 1 1 1 0 1 0 0 1 1 1 0 0 0 1 0 0 0 1 0 0 1 1 0 0 1 1
 1 0 1 1 1 1 1 1 0 1 0 0 1 1 0 1 0 0 0 0 0 1 1 0 1 1]
Alice's first bit = 0
```

- Alice encodes her bits.

```
def encode_message(bits, bases):
    message = []
    for i in range(n):
        qc = QuantumCircuit(1,1)
        if bases[i] == 0: # Prepare qubit in X-basis
            if bits[i] == 0:
                pass
            else:
                qc.x(0)
        else: # Prepare qubit in Z-basis
            if bits[i] == 0:
                qc.h(0)
            else:
                qc.x(0)
                qc.h(0)
        qc.barrier()
        message.append(qc)
    return message
```

- Eve might intercept the message.

```python
def intercept_message(message, bases):
    backend = Aer.get_backend('qasm_simulator')
    measurements = []
    for q in range(n):
        if bases[q] == 0: # measuring in Z-basis
            message[q].measure(0,0)
        if bases[q] == 1: # measuring in X-basis
            message[q].h(0)
            message[q].measure(0,0)
            message[q].h(0) # preparing the post-measurement state
        result = execute(message[q], backend, shots=1, memory=True).result()
        measured_bit = int(result.get_memory()[0])
        measurements.append(measured_bit)
    return measurements
```

- Bob finally gets message and need to measure them.

```python
def measure_message(message, bases):
    backend = Aer.get_backend('qasm_simulator')
    measurements = []
    for q in range(n):
        if bases[q] == 0: # measuring in Z-basis
            message[q].measure(0,0)
        if bases[q] == 1: # measuring in X-basis
            message[q].h(0)
            message[q].measure(0,0)
        result = execute(message[q], backend, shots=1, memory=True).result()
        measured_bit = int(result.get_memory()[0])
        measurements.append(measured_bit)
    return measurements
```

```python
## Step 3
# Decide which basis to measure in:
bob_bases = randint(2, size=n)
bob_results = measure_message(message, bob_bases)
print(bob_bases)
print("Bob's first chosen basis = %i" % bob_bases[0])
```

```
[1 0 0 0 1 0 1 1 1 1 1 0 0 1 0 1 1 0 0 1 0 1 0 0 0 0 1 1 0 1 0 0 0 1 0 0 0
 1 1 1 0 0 0 1 0 1 0 0 0 1 0 1 1 1 0 0 0 1 1 0 0 0 0 0 0 1 0 1 0 1 1 1 1 0
 0 0 0 1 0 0 1 0 1 0 0 1 0 0 1 1 1 0 0 1 1 1 1 1 1 0]
Bob's first chosen basis = 1
```

- To avoid message being intercepted, some changed bits should be removed.

```python
list_index = []
def remove_garbage(a_bases, b_bases, bits):
    good_bits = []
    for q in range(n):
        if a_bases[q] == b_bases[q]:
            # If both used the same basis, add
            # this to the list of 'good' bits
            good_bits.append(bits[q])
            list_index.append(q)
    return good_bits
```

- Using the length of original message and that of after removing, to calculate the probability Eve guessing correctly, approximately 55%, a bit higher than 50% (guess randomly)

```python
## Step 4
# Remove bits where Alice's chosen bases are not equal to that of Bob's
alice_key = remove_garbage(alice_bases, bob_bases, alice_bits)

Eve_remove = []
for i in list_index:
    Eve_remove.append(intercepted_message[i])

bob_key = remove_garbage(alice_bases, bob_bases, bob_results)

ab_same = 0
# Compare the bits between Alice's and Bob's
for i in range(len(alice_key)):
    if (alice_key[i] == bob_key[i]): ab_same += 1

# Compare the bits between Alice's and Eve's
ae_same = 0
for i in range(len(alice_key)):
    if (alice_key[i] == Eve_remove[i]): ae_same += 1

ab_same_pb = (ab_same/len(alice_key))*100
ae_same_pb = (ae_same / len(alice_key))*100
print("Alice's key: ", alice_key)
print("Bob's key: ", bob_key)
print("The probability that Alice's = Bob's bits after removing: %2.f%%"%ab_same_pb)
print("The probability that Alice's = Eve's bits after removing: %2.f%%"%ae_same_pb)
```

```
Alice's key:  [0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1,
0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1]
Bob's key:  [0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0,
1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1]
The probability that Alice's = Bob's bits after removing: 79%
The probability that Alice's = Eve's bits after removing: 69%
```

```python
# Problem 3-(a)
p = (ab_same_pb*ae_same_pb)/100
print("Probability of Eve guessing Alice’s bit correctly = %2.f%%" % p)
```

```
Probability of Eve guessing Alice’s bit correctly = 55%
```

(b) Eve fixed basis

$$\left\{\cos\tfrac{\pi}{8}|0\rangle + \sin\tfrac{\pi}{8}|1\rangle, \sin\tfrac{\pi}{8}|0\rangle - \cos\tfrac{\pi}{8}|1\rangle\right\}$$

- I use U3 gate at “Eve’s basis” (theta = pi/4, lambda = pi) and get the result, approximately 60%, a bit higher than XZ-bases.

$$U3(\theta,\phi,\lambda) = \begin{pmatrix} \cos(\frac{\theta}{2}) & -e^{i\lambda}\sin(\frac{\theta}{2}) \\ e^{i\phi}\sin(\frac{\theta}{2}) & e^{i(\phi+\lambda)}\cos(\frac{\theta}{2}) \end{pmatrix}$$

```python
def intercept_message(message, bases):
    backend = Aer.get_backend('qasm_simulator')
    measurements = []
    for q in range(n):
        if bases[q] == 0: # measuring in Ry-basis
            message[q].u3(pi/4, 0, pi, 0).inverse()
            message[q].measure(0,0)
        if bases[q] == 1: # measuring in Ry-basis
            message[q].u3(pi/4, 0, pi, 0).inverse()
            message[q].measure(0,0)
        result = execute(message[q], backend, shots=1, memory=True).result()
        measured_bit = int(result.get_memory()[0])
        measurements.append(measured_bit)
    return measurements
```

```python
## Step 4
# Remove bits where Alice's chosen bases are not equal to that of Bob's
alice_key = remove_garbage(alice_bases, bob_bases, alice_bits)

Eve_remove = []
for i in list_index:
    Eve_remove.append(intercepted_message[i])

bob_key = remove_garbage(alice_bases, bob_bases, bob_results)

ab_same = 0
# Compare the bits between Alice's and Bob's
for i in range(len(alice_key)):
    if (alice_key[i] == bob_key[i]): ab_same += 1

# Compare the bits between Alice's and Eve's
ae_same = 0
for i in range(len(alice_key)):
    if (alice_key[i] == Eve_remove[i]): ae_same += 1

ab_same_pb = (ab_same / len(alice_key))*100
ae_same_pb = (ae_same / len(alice_key))*100
print("Alice's key: ", alice_key)
print("Bob's key: ", bob_key)
print("The probability that Alice's = Bob's bits after removing: %2.f%%"%ab_same_pb)
print("The probability that Alice's = Eve's bits after removing: %2.f%%"%ae_same_pb)
```

```
Alice's key:  [0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1,
0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1]
Bob's key:  [1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0,
0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1]
The probability that Alice's = Bob's bits after removing: 73%
The probability that Alice's = Eve's bits after removing: 83%
```

```python
# Problem 3-(b)
p = (ab_same_pb*ae_same_pb)/100
print("Probability of Eve guessing Alice’s bit correctly = %2.f%%" % p)
```

```
Probability of Eve guessing Alice’s bit correctly = 60%
```