CSCI 4140 Communications Algorithms

Mark Green

Faculty of Science

Ontario Tech

Introduction

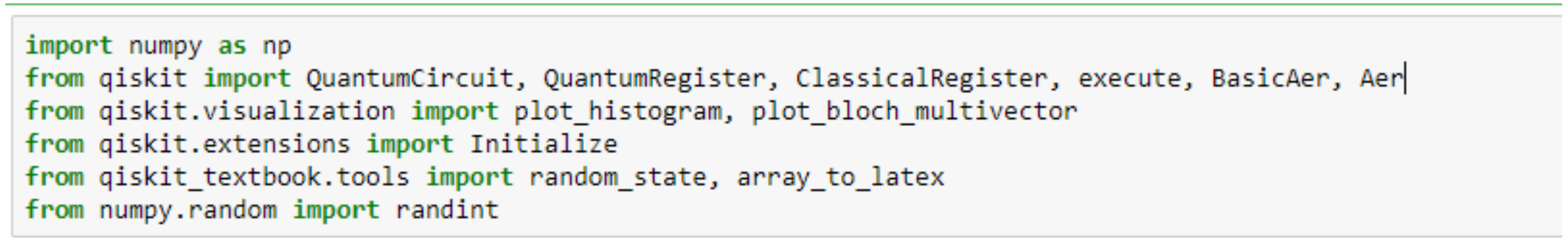
- In the fundamental algorithms lecture we saw three communications oriented algorithms:
  - Quantum teleportation
  - Superdense coding
  - Quantum key distribution
- In this lecture we will examine them in more detail including their implementation in Qiskit

Quantum Teleportation

- Recall: Alice wants to send a qubit to Bob, but only has two classical bits
- They share a pair of entangled qubits
- Alice performs a CNOT and a Hadamard on her two qubits, measures the results and sends the two bits to Bob
- Bob then then applies an X and Z gate to his entangled qubit to retrieve the original qubit from Alice
- Build the circuit and simulate it

Quantum Teleportation

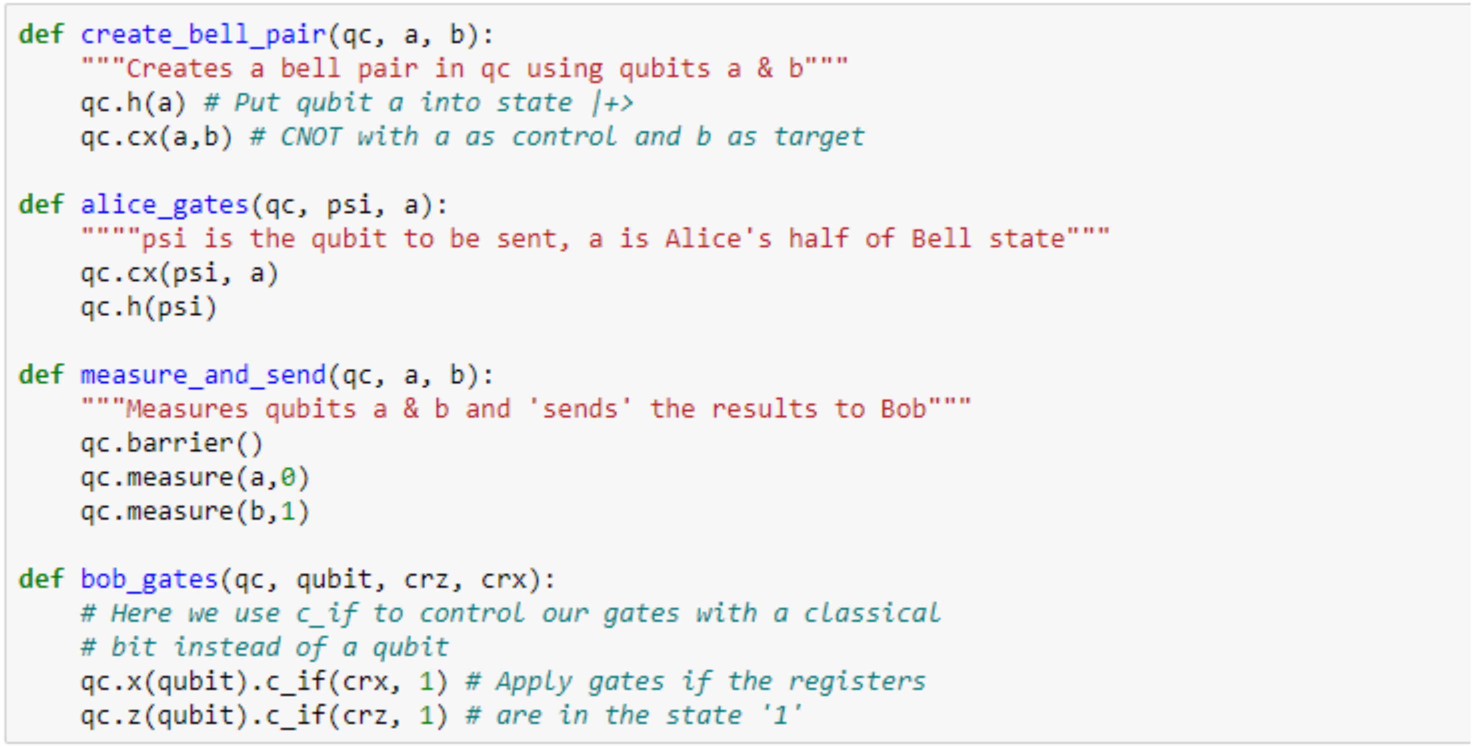
- Start with some import statements:



```
import numpy as np
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister, execute, BasicAer, Aer
from qiskit.visualization import plot_histogram, plot_bloch_multivector
from qiskit.extensions import Initialize
from qiskit_textbook.tools import random_state, array_to_latex
from numpy.random import randint
```

- Next create four procedures that contain the key parts of the circuit, see the next slide
- The first three are straight forward, create an entangled pair, add Alice's gate and measure Alice's qubits

Quantum Teleportation

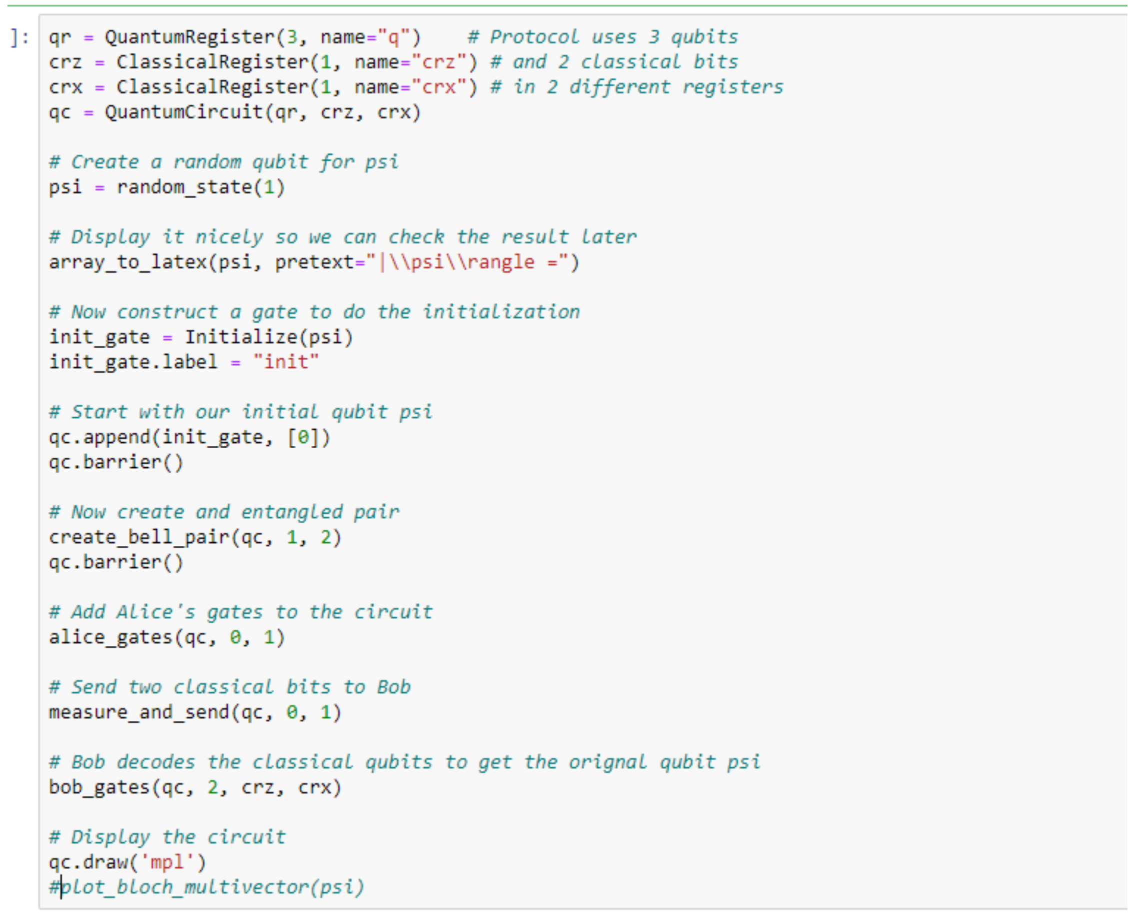


```
def create_bell_pair(qc, a, b):
    """Creates a bell pair in qc using qubits a & b"""
    qc.h(a) # Put qubit a into state |+>
    qc.cx(a,b) # CNOT with a as control and b as target

def alice_gates(qc, psi, a):
    """"psi is the qubit to be sent, a is Alice's half of Bell state"""
    qc.cx(psi, a)
    qc.h(psi)

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)

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'
```

Quantum Teleportation

- The fourth procedure has something we have seen before
- We need to select the quantum gates based on the values of classical bits
- We have a different version of the controlled gates, where the control is a classical bit instead of a qubit
- With this we can put together the code on the next slide
- As an extra test we use random_state() to create a random qubit and then use initialize() to convert that into a gate that we can add to our circuit

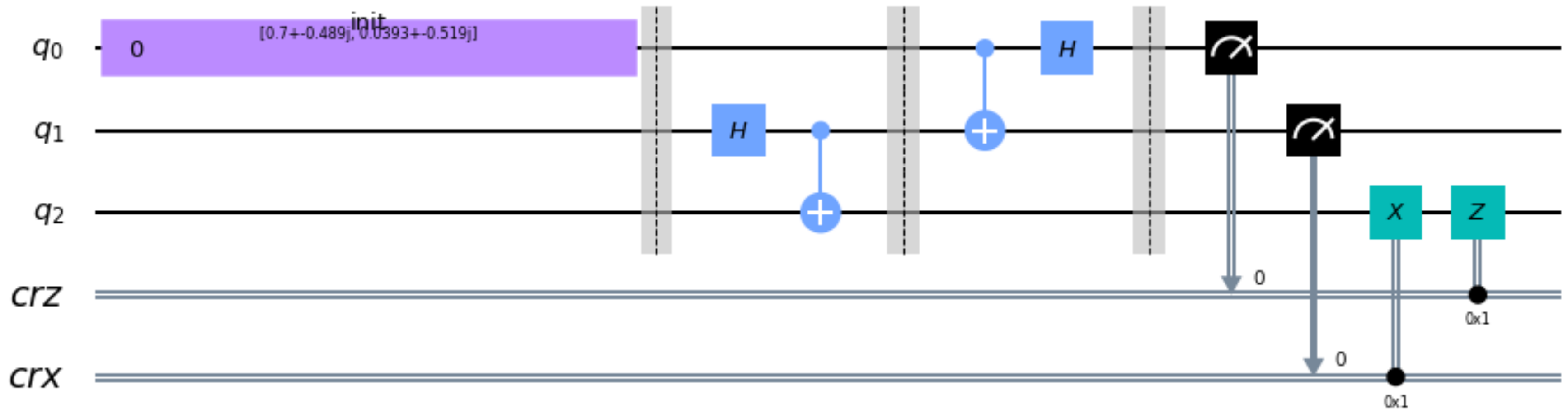
Quantum Teleportation



```
]: qr = QuantumRegister(3, name="q")    # Protocol uses 3 qubits
   crz = ClassicalRegister(1, name="crz") # and 2 classical bits
   crx = ClassicalRegister(1, name="crx") # in 2 different registers
   qc = QuantumCircuit(qr, crz, crx)

   # Create a random qubit for psi
   psi = random_state(1)

   # Display it nicely so we can check the result later
   array_to_latex(psi, pretext="|\\psi\\rangle =")

   # Now construct a gate to do the initialization
   init_gate = Initialize(psi)
   init_gate.label = "init"

   # Start with our initial qubit psi
   qc.append(init_gate, [0])
   qc.barrier()

   # Now create and entangled pair
   create_bell_pair(qc, 1, 2)
   qc.barrier()

   # Add Alice's gates to the circuit
   alice_gates(qc, 0, 1)

   # Send two classical bits to Bob
   measure_and_send(qc, 0, 1)

   # Bob decodes the classical qubits to get the orignal qubit psi
   bob_gates(qc, 2, crz, crx)

   # Display the circuit
   qc.draw('mpl')
   #plot_bloch_multivector(psi)
```

Quantum Teleportation

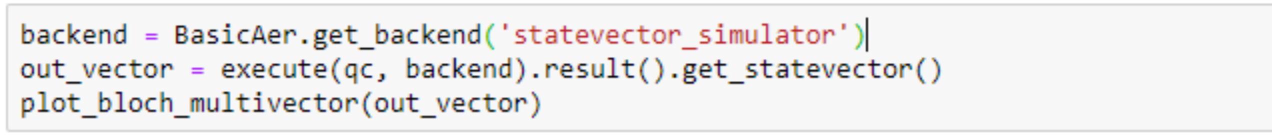
- As a check we will display the circuit:



- We can also display the initial qubit in a Bloch sphere, which we will see later

Quantum Teleportation

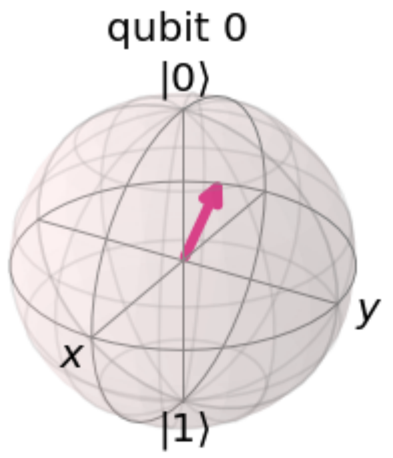
- Next we want to simulate the circuit and see if it works
- To do this we construct a new cell and add the following code:



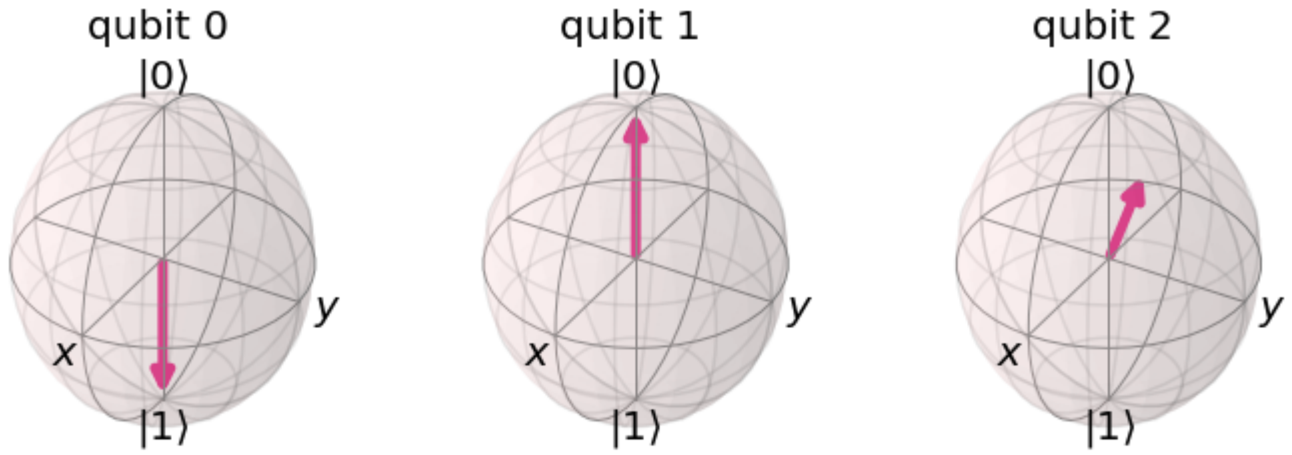
```
backend = BasicAer.get_backend('statevector_simulator')
out_vector = execute(qc, backend).result().get_statevector()
plot_bloch_multivector(out_vector)
```

- On the next slide we show the Bloch sphere for the original qubit and the result of the simulation
- Note that qubit 2 from the simulation is the same as our original qubit

Quantum Teleportation



Original qubit

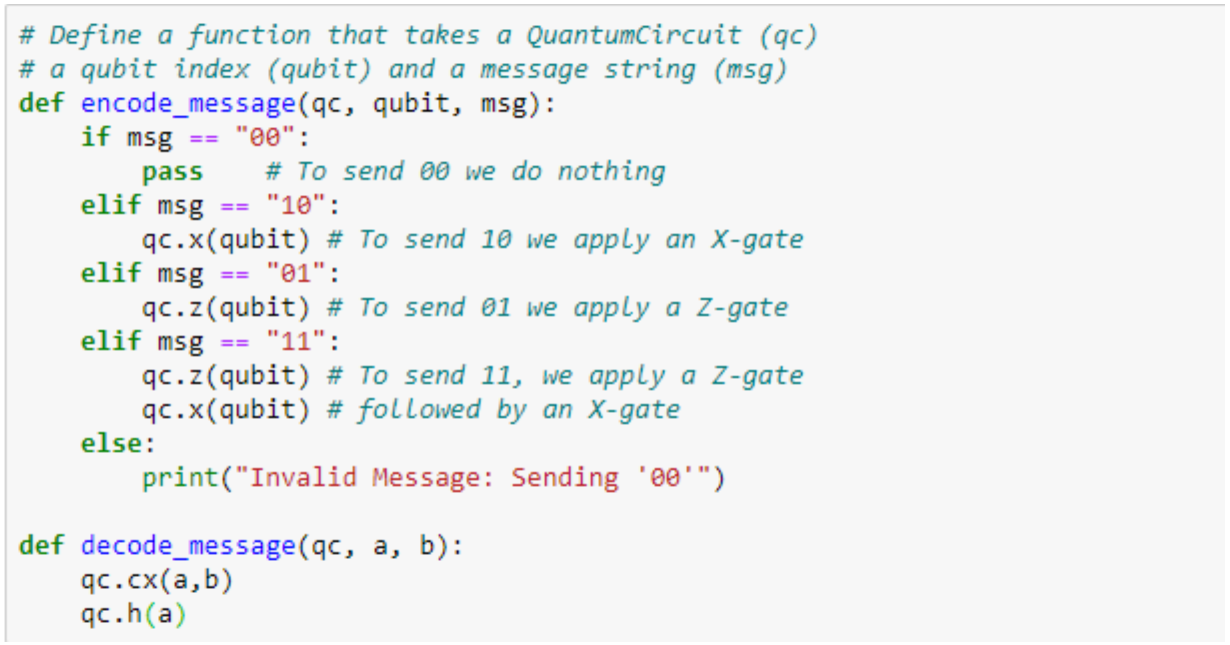


Result of simulation

Superdense Coding

- We can reuse some of our code from quantum teleportation for this circuit, just continue in the same Jupyter notebook
- We need to add two new procedures:
  - Encode the message sent by Alice as a qubit
  - The gates Bob uses to decode the message
- There functions are shown on the next slide, there is nothing really new here

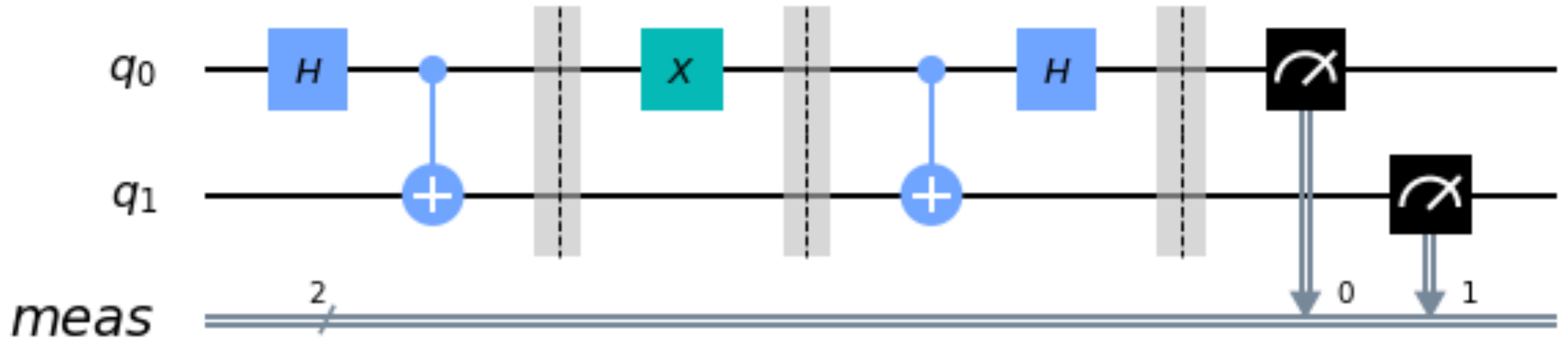
Superdense Coding



```
# Define a function that takes a QuantumCircuit (qc)
# a qubit index (qubit) and a message string (msg)
def encode_message(qc, qubit, msg):
    if msg == "00":
        pass    # To send 00 we do nothing
    elif msg == "10":
        qc.x(qubit) # To send 10 we apply an X-gate
    elif msg == "01":
        qc.z(qubit) # To send 01 we apply a Z-gate
    elif msg == "11":
        qc.z(qubit) # To send 11, we apply a Z-gate
        qc.x(qubit) # followed by an X-gate
    else:
        print("Invalid Message: Sending '00'")

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

Superdense Coding

- With this in hand we can create the circuit, the code is shown on the next slide, the circuit we want is:



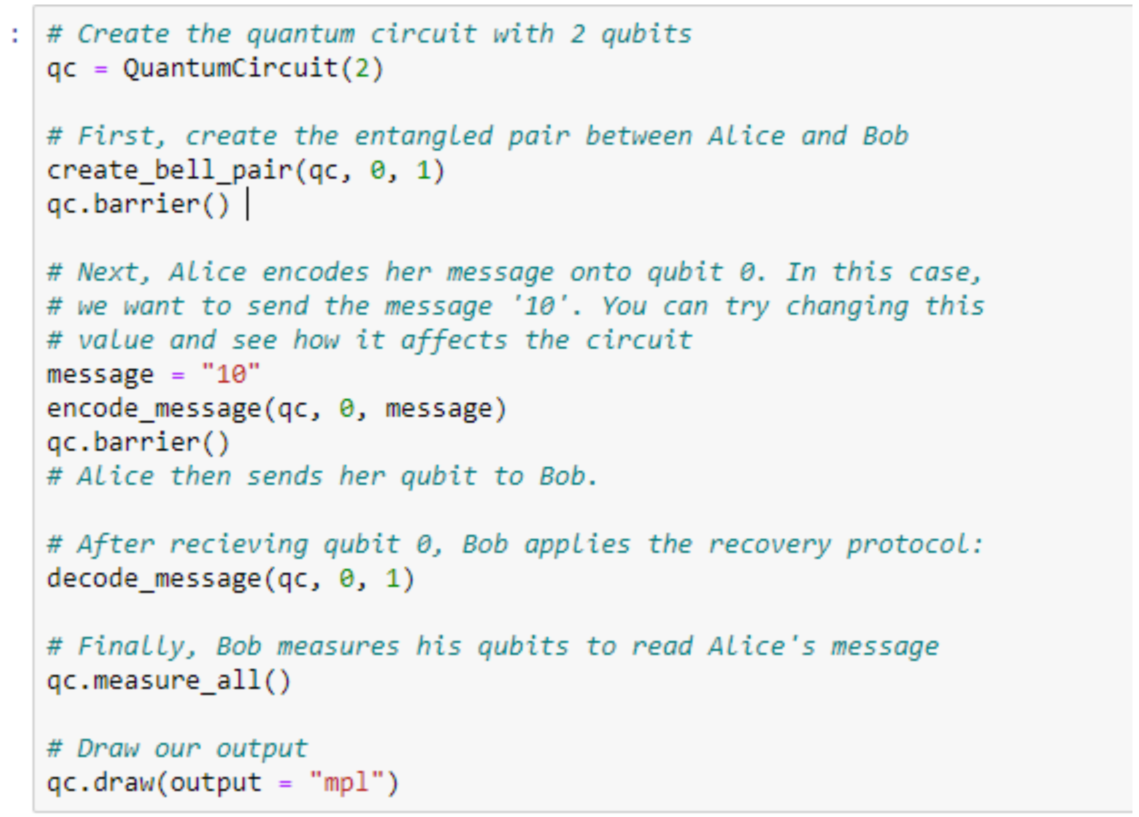
Entanglement

Create the message

Decode the message

Convert to Bits

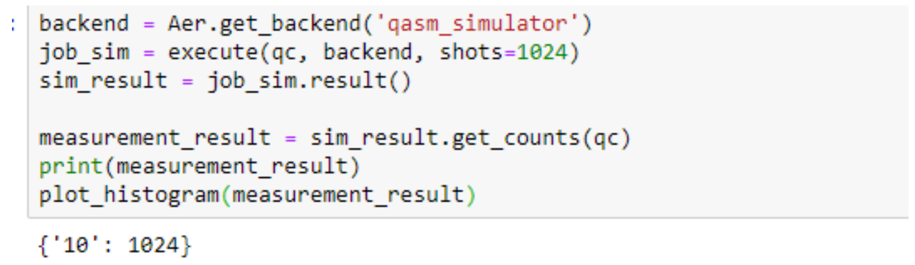
Superdense Coding



```
# Create the quantum circuit with 2 qubits
qc = QuantumCircuit(2)

# First, create the entangled pair between Alice and Bob
create_bell_pair(qc, 0, 1)
qc.barrier()

# Next, Alice encodes her message onto qubit 0. In this case,
# we want to send the message '10'. You can try changing this
# value and see how it affects the circuit
message = "10"
encode_message(qc, 0, message)
qc.barrier()
# Alice then sends her qubit to Bob.

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

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

# Draw our output
qc.draw(output = "mpl")
```

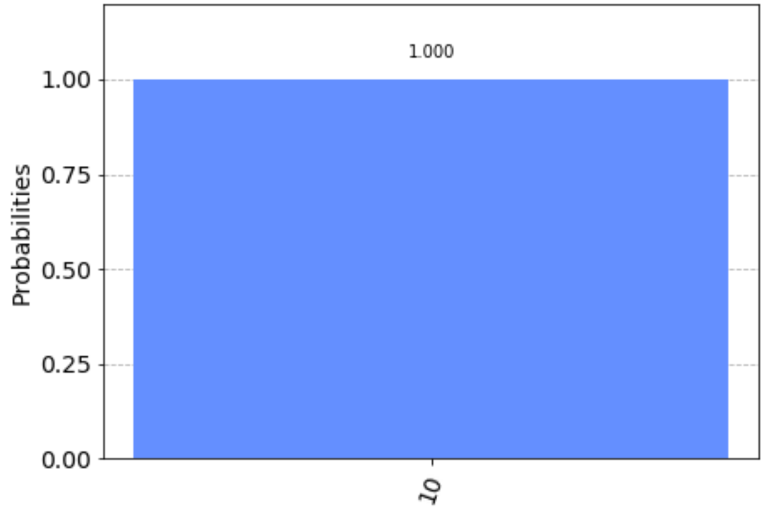
Superdense Coding

- Finally we can simulate the circuit and view the results



```
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)
plot_histogram(measurement_result)
```

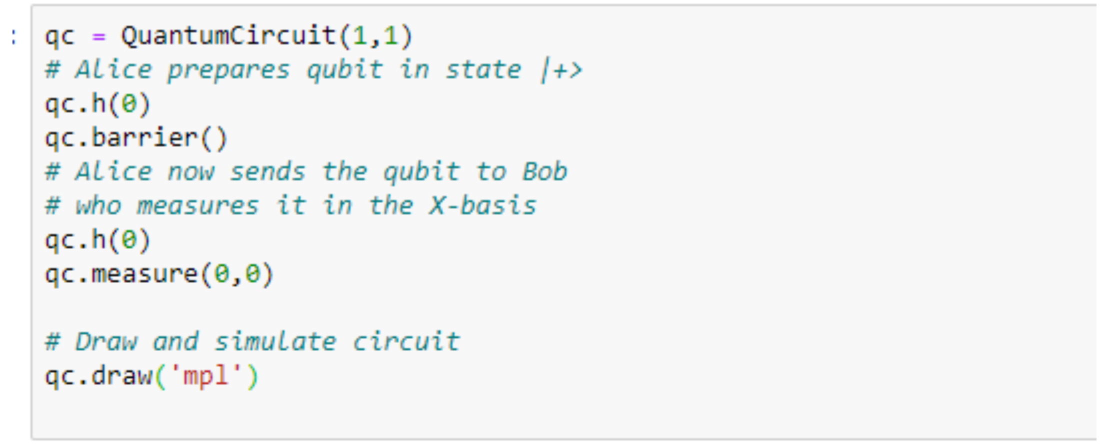
```
{'10': 1024}
```



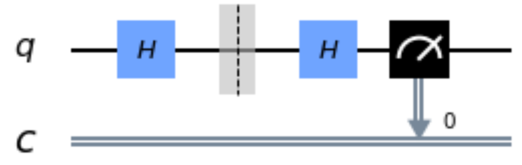
Quantum Key Distribution

- Before we get into the circuit for QKD it’s worth looking at some single bit results, shows how things work
- Start with Alice preparing a qubit in state |+>, this is a 0 bit encoded in the X axis
- Bob then measures this qubit in the X axis
- Bob should now be able to retrieve a 0 bit
- The circuit is shown on the next slide

Quantum Key Distribution

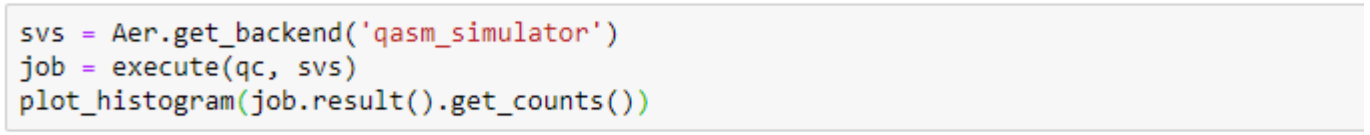


```
qc = QuantumCircuit(1,1)
# Alice prepares qubit in state |+>
qc.h(0)
qc.barrier()
# Alice now sends the qubit to Bob
# who measures it in the X-basis
qc.h(0)
qc.measure(0,0)

# Draw and simulate circuit
qc.draw('mpl')
```

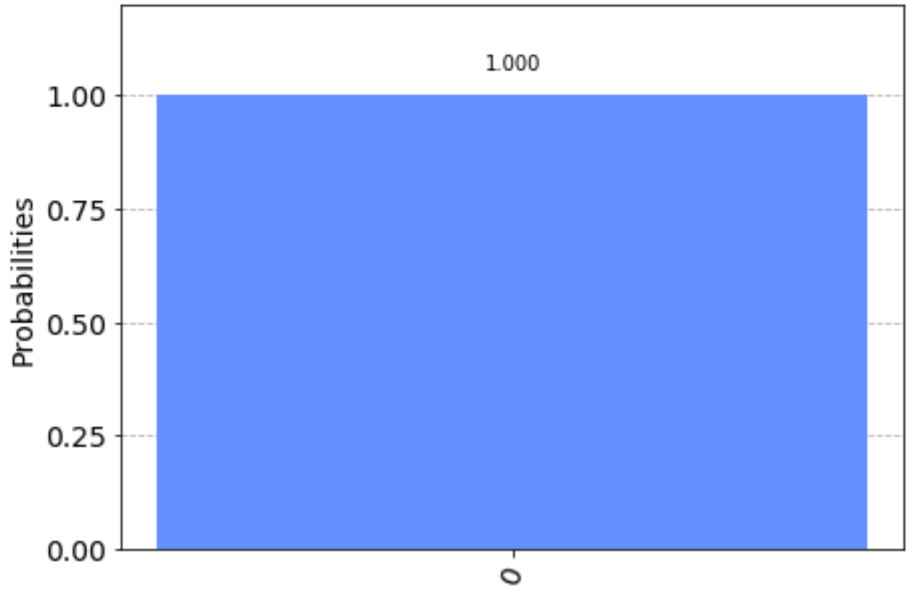


Quantum Key Distribution

- We can now simulate it to show that we get the correct result:

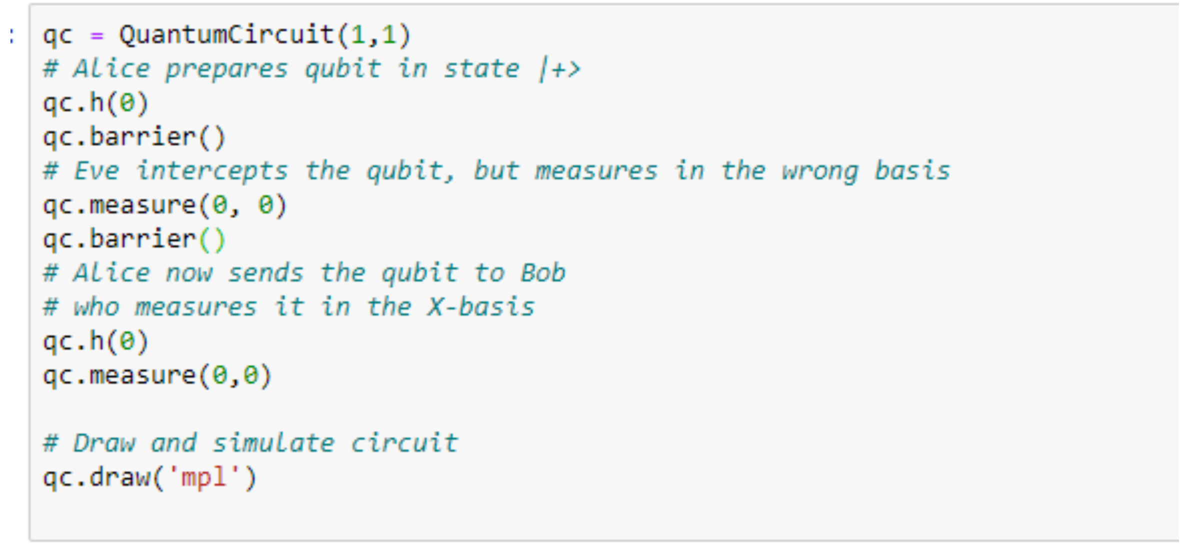


```
svs = Aer.get_backend('qasm_simulator')
job = execute(qc, svs)
plot_histogram(job.result().get_counts())
```

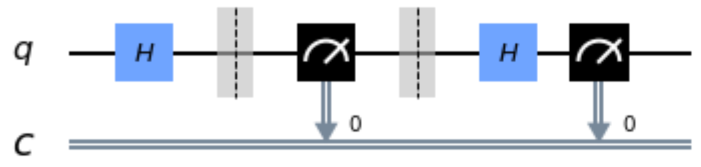


Quantum Key Distribution

- Now add Eve to the circuit, who intercepts the message and reads it in the wrong basis

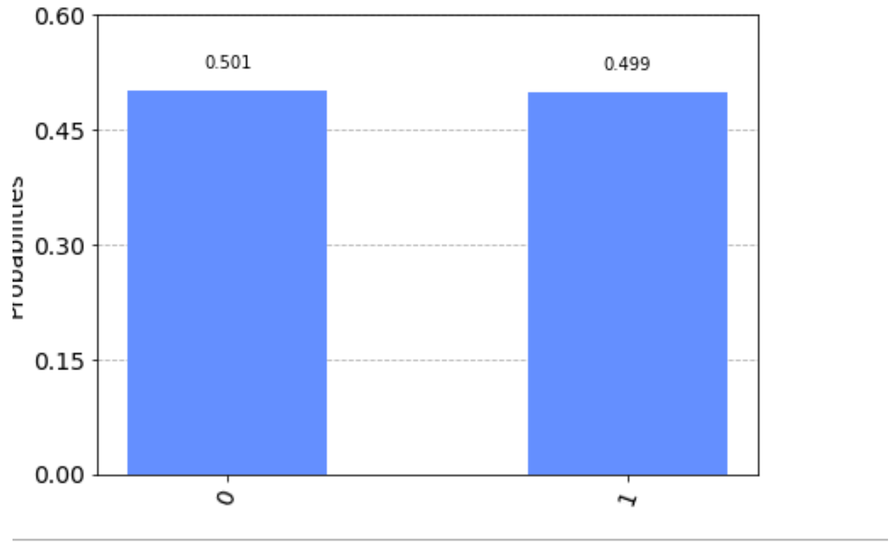


```
qc = QuantumCircuit(1,1)
# Alice prepares qubit in state |+>
qc.h(0)
qc.barrier()
# Eve intercepts the qubit, but measures in the wrong basis
qc.measure(0, 0)
qc.barrier()
# Alice now sends the qubit to Bob
# who measures it in the X-basis
qc.h(0)
qc.measure(0,0)

# Draw and simulate circuit
qc.draw('mpl')
```



Quantum Key Distribution

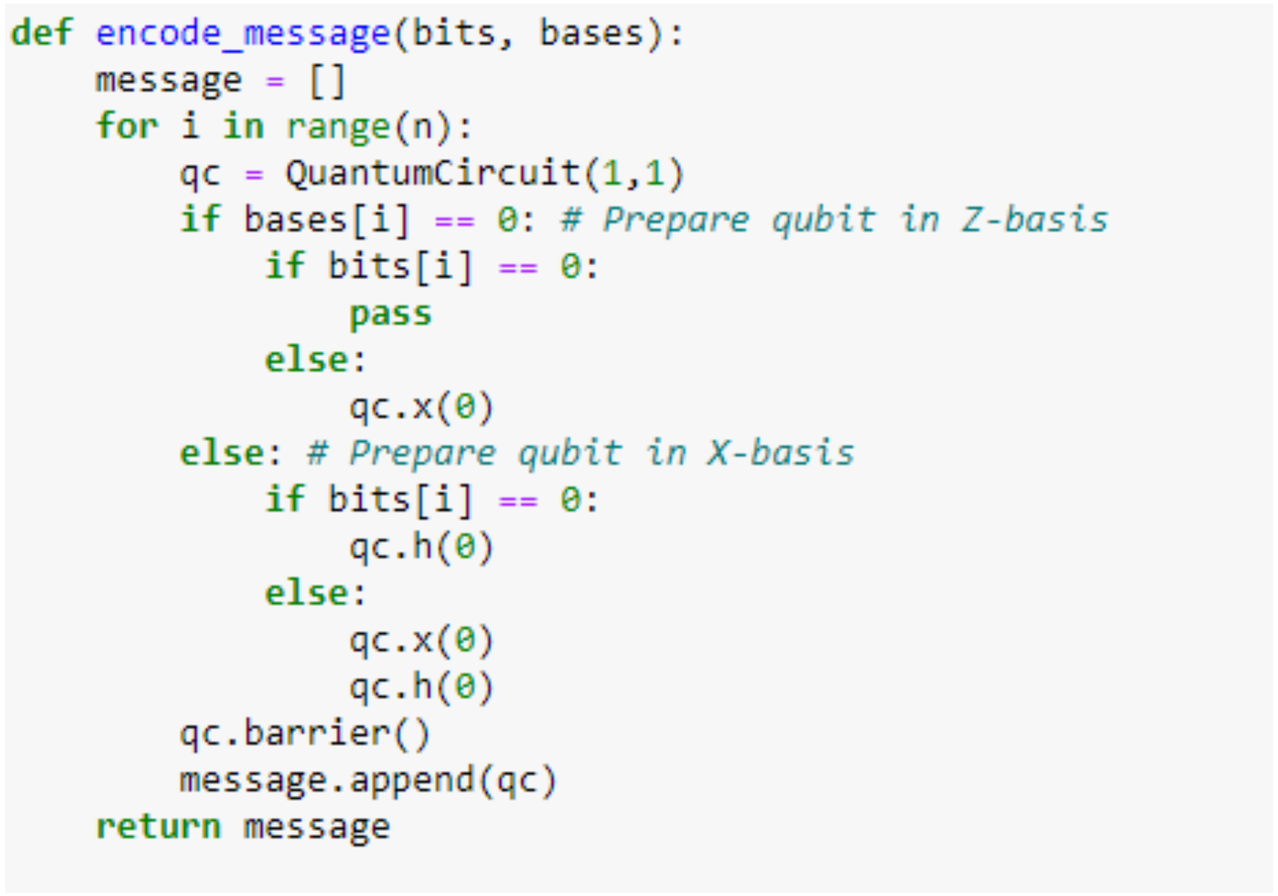
- Now when we run the simulation we see that there is a 50% chance of Bob reading the wrong value
- Clearly with a single bit this isn't overly reliable, this is why we need to go to a large number of bits



Quantum Key Distribution

- Now that we’ve seen what happens with a single bit, it’s time to build a circuit that does an arbitrary number of bits
- Alice will create a list of random bits that will be the basis for the key
- Alice also generates a list of basis that will be used to encode these bits
- Each key bit requires 0, 1 or 2 gates to encode it, this will end up being a fairly large circuit if we have hundreds of bits
- The procedure on the next slide is used to construct this circuit for an arbitrary number of bits

Quantum Key Distribution

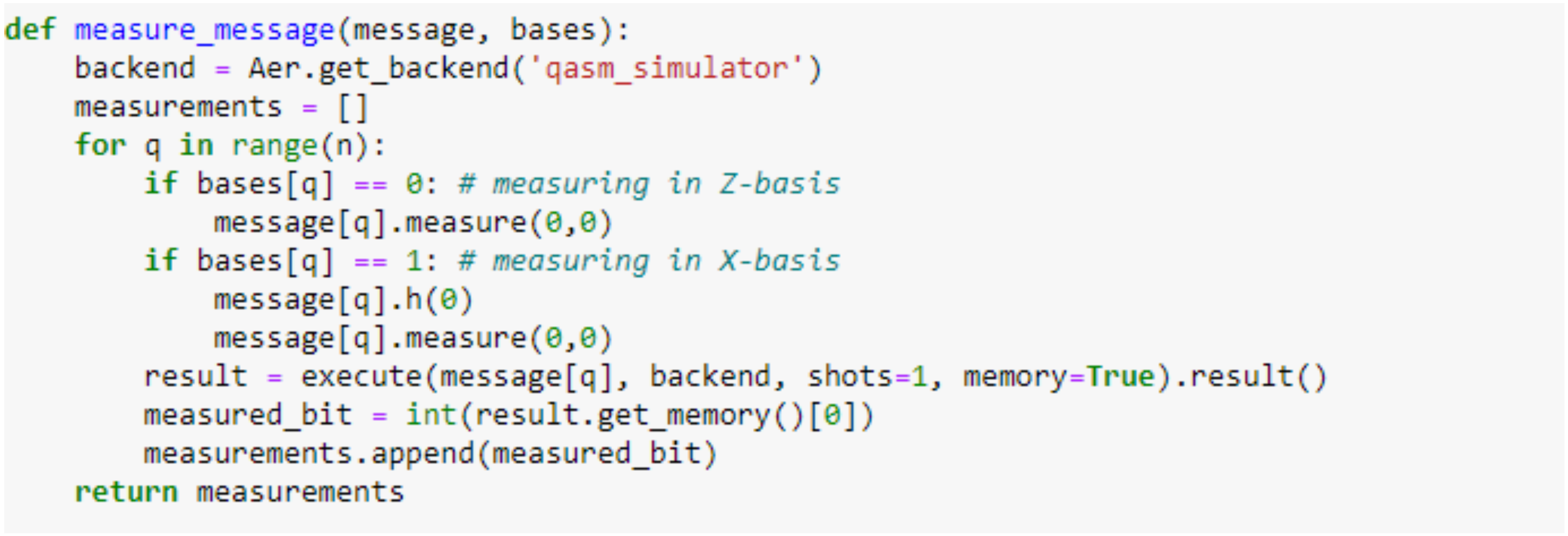


```python
def encode_message(bits, bases):
    message = []
    for i in range(n):
        qc = QuantumCircuit(1,1)
        if bases[i] == 0: # Prepare qubit in Z-basis
            if bits[i] == 0:
                pass
            else:
                qc.x(0)
        else: # Prepare qubit in X-basis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

Quantum Key Distribution

- Note that the result of this procedure is a list of circuits
- At the other end Bob will generate a random list of basis
- Bob will then measure each of the qubits with respect to these basis
- Since we have an arbitrary number of bits, we also construct a procedure to do this
- This procedure adds the measurement gates to the gates for each bit and then simulates the circuit
- Bob saves the result of the simulation in a list

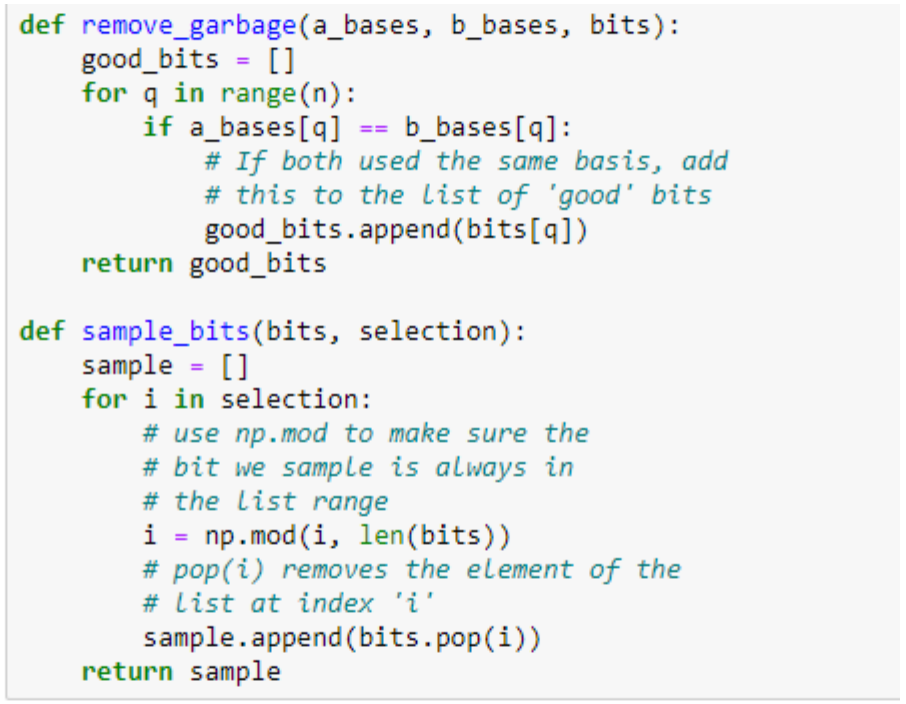
Quantum Key Distribution



```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
```

Quantum Key Distribution

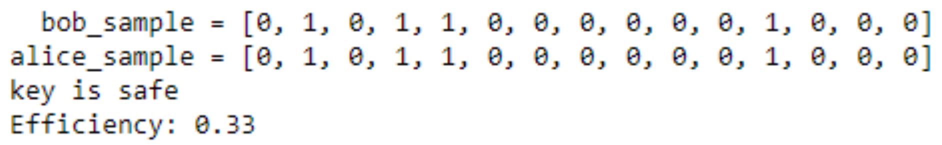
- We have two smaller procedures that remove the bits where the two basis lists don’t agree and randomly samples the good bits



```python
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])
    return good_bits

def sample_bits(bits, selection):
    sample = []
    for i in selection:
        # use np.mod to make sure the
        # bit we sample is always in
        # the list range
        i = np.mod(i, len(bits))
        # pop(i) removes the element of the
        # list at index 'i'
        sample.append(bits.pop(i))
    return sample
```

Quantum Key Distribution

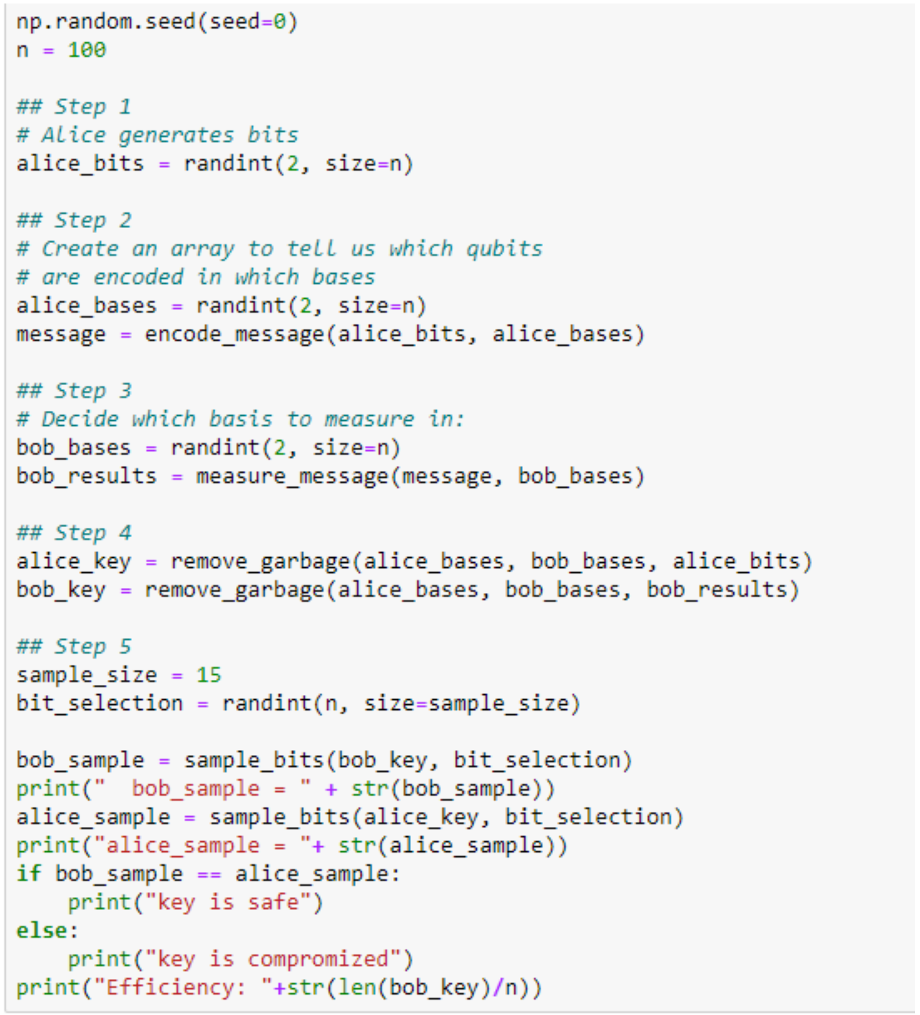
- We are now ready to put everything together and run the program, shown on the next slide
- There is nothing particularly tricky in this code, when we run it we get the following:



```
  bob_sample = [0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
alice_sample = [0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
key is safe
Efficiency: 0.33
```

- Note that only 33% of the original bits end up in the final key, we will come back to that after we deal with a third party intercepting the message

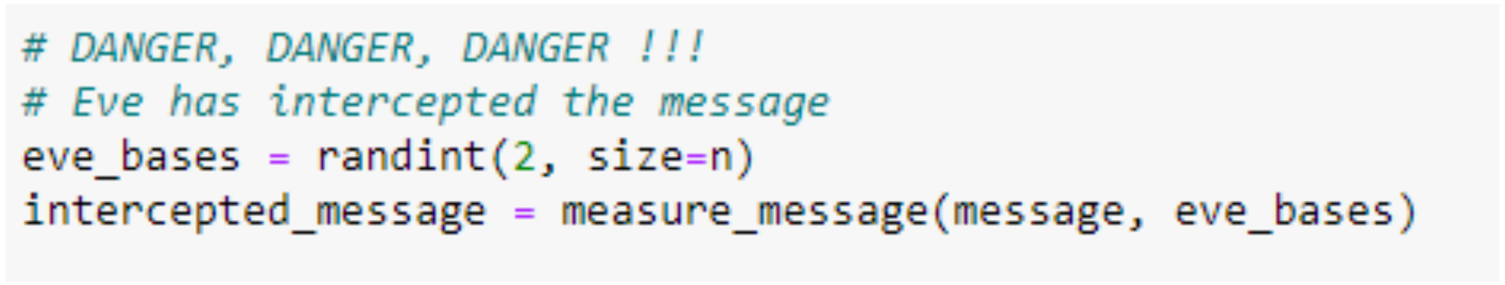
Quantum Key Distribution



```python
np.random.seed(seed=0)
n = 100

## Step 1
# Alice generates bits
alice_bits = randint(2, size=n)

## Step 2
# Create an array to tell us which qubits
# are encoded in which bases
alice_bases = randint(2, size=n)
message = encode_message(alice_bits, alice_bases)

## Step 3
# Decide which basis to measure in:
bob_bases = randint(2, size=n)
bob_results = measure_message(message, bob_bases)

## Step 4
alice_key = remove_garbage(alice_bases, bob_bases, alice_bits)
bob_key = remove_garbage(alice_bases, bob_bases, bob_results)

## Step 5
sample_size = 15
bit_selection = randint(n, size=sample_size)

bob_sample = sample_bits(bob_key, bit_selection)
print("  bob_sample = " + str(bob_sample))
alice_sample = sample_bits(alice_key, bit_selection)
print("alice_sample = "+ str(alice_sample))
if bob_sample == alice_sample:
    print("key is safe")
else:
    print("key is compromized")
print("Efficiency: "+str(len(bob_key)/n))
```

Quantum Key Distribution

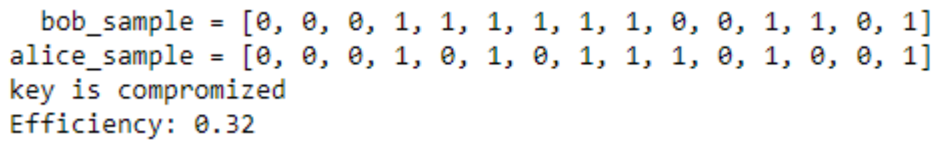
- No let’s add a third party, Eve who is intercepting our message and see if we can detect her
- Eve will measure the qubits, with a random list of basis, before they reach Bob
- We only need to add two lines before Bob measures the qubits:



```python
# DANGER, DANGER, DANGER !!!
# Eve has intercepted the message
eve_bases = randint(2, size=n)
intercepted_message = measure_message(message, eve_bases)
```

Quantum Key Distribution

- When we run the program we now get these results:



```
  bob_sample = [0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1]
alice_sample = [0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1]
key is compromized
Efficiency: 0.32
```

- The samples differ on 4 bits in this case, we could try different random numbers and we will get slightly different results
- If we change the seed to 3 there are 3 different bits
- This doesn't seem like much of a margin of safety, but it should be okay

Quantum Key Distribution

- In this example we generate 100 bits, but only 33 remain in the key, this isn’t particularly efficient
- Statistically the best we can do is 50%, since on average half the basis pairs will be different
- We then need to subtract off the bits we use for the random sample, we can’t use them in the key
- As we increase the number of bits the efficiency does go up, but we will rarely get more than 50%

Quantum Key Distribution

- What we’ve seen is an issue with QKD algorithms
- We have the algorithms, we have hardware that implements them, but they aren’t efficient
- They are good enough to encode keys, which are relatively short and not sent very often
- They don’t work for whole messages, the best research systems are now approaching 1Mbps, with typical speeds in the Kbps range
- This is much slower than the underlying network technology

Summary

- Produced implementations of the three communications based algorithms presented in class
- Concentrated on QKD, since it’s important in practice
- Shown that this algorithm doesn’t make efficient use of network bandwidth, which is typical of all QKD algorithms