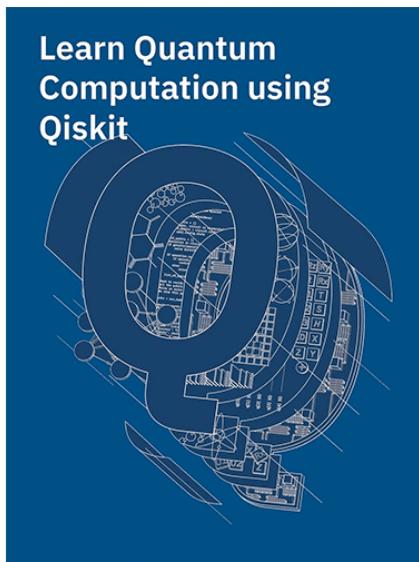


Learn Quantum Computing using Qiskit



Greetings from the Qiskit Community team! We initiated this open-source textbook in collaboration with IBM Research as a university quantum algorithms/computation course supplement based on Qiskit.

Traditional Quantum Computation Course



Learn Quantum Computation using Qiskit Textbook



The goal of the textbook is to develop skills in the following areas.

1. The mathematics behind quantum algorithms
2. Details about today's non-fault-tolerant quantum devices
3. Writing code in Qiskit to implement quantum algorithms on IBM's cloud quantum systems

While this textbook does not attempt to be an expansive survey of the field, it does attempt to be as self-contained as possible.

If you have any questions or suggestions about the textbook or would like to incorporate it into your curriculum, please contact Abraham Asfaw (abraham.asfaw@ibm.com). In the true spirit of open-source, any chapter contributions are welcome in [this GitHub repository](#).

Contributors in Alphabetical Order

Learn Quantum Computation using Qiskit is the work of several individuals. If you use it in your work, cite it using this [bib file](#) or directly as:

Abraham Asfaw, Luciano Bello, Yael Ben-Haim, Sergey Bravyi, Lauren Capelluto, Almudena Carrera Vazquez, Jack Ceroni, Jay Gambetta, Shelly Garion, Leron Gil, Salvador De La Puente Gonzalez, David McKay, Zlatko Minev, Paul Nation, Anna Phan, Arthur Rattew, Javad Shabani, John Smolin, Kristan Temme, Madeleine Tod, James Wootton.

Upcoming topics

The following topics are currently being developed for addition to the textbook.

1. Shor's Algorithm
2. Overview of Quantum Algorithms for NISQ Hardware
3. Mapping the Ising Model onto a Superconducting Quantum Computer
4. Solving Combinatorial Optimization Problems using QAOA
5. Solving Linear Systems of Equations using HHL
6. Securing Communications using BB84
7. Decoherence and Energy Relaxation: Measuring T2 and T1
8. Optimizing Microwave Pulses for High-Fidelity Qubit Operations
9. State and Process Tomography

Introduction to Python and Jupyter notebooks

Python is a programming language where you don't need to compile. You can just run it line by line (which is how we can use it in a notebook). So if you are quite new to programming, Python is a great place to start. The current version is Python 3, which is what we'll be using here.

One way to code in Python is to use a Jupyter notebook. This is probably the best way to combine programming, text and images. In a notebook, everything is laid out in cells. Text cells and code cells are the most common. If you are viewing this section as a Jupyter notebook, the text you are now reading is in a text cell. A code cell can be found just below.

To run the contents of a code cell, you can click on it and press Shift + Enter. Or if there is a little arrow thing on the left, you can click on that.

```
1 + 1
```

```
2
```

If you are viewing this section as a Jupyter notebook, execute each of the code cells as you read through.

```
a = 1
b = 0.5
a + b
```

```
1.5
```

Above we created two variables, which we called `a` and `b`, and gave them values. Then we added them. Simple arithmetic like this is pretty straightforward in Python.

Variables in Python come in many forms. Below are some examples.

```
an_integer = 42 # Just an integer
a_float = 0.1 # A non-integer number, up to a fixed precision
```

```
a_boolean = True # A value that can be True or False  
a_string = '''just enclose text between two 's, or two "s, or do what we did for this s  
none_of_the_above = None # The absence of any actual value or variable type
```

As well as numbers, another data structure we can use is the *list*.

```
a_list = [0,1,2,3]
```



Lists in Python can contain any mixture of variable types.

```
a_list = [ 42, 0.5, True, [0,1], None, 'Banana' ]
```



Lists are indexed from 0 in Python (unlike languages such as Fortran). So here's how you access the 42 at the beginning of the above list.

```
a_list[0]
```



```
42
```

A similar data structure is the *tuple*.

```
a_tuple = ( 42, 0.5, True, [0,1], None, 'Banana' )  
a_tuple[0]
```



```
42
```

A major difference between the list and the tuple is that list elements can be changed

```
a_list[5] = 'apple'  
  
print(a_list)
```



```
[42, 0.5, True, [0, 1], None, 'apple']
```

whereas tuple elements cannot

```
a_tuple[5] = 'apple'
```



```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-9-42d08f1e5606> in <module>  
----> 1 a_tuple[5] = 'apple'
```

```
TypeError: 'tuple' object does not support item assignment
```

Also we can add an element to the end of a list, which we cannot do with tuples.

```
a_list.append( 3.14 )
```



```
print(a_list)
```

```
[42, 0.5, True, [0, 1], None, 'apple', 3.14]
```

Another useful data structure is the *dictionary*. This stores a set of *values*, each labeled by a unique *key*.

Values can be any data type. Keys can be anything sufficiently simple (integer, float, Boolean, string). It cannot be a list, but it *can* be a tuple.

```
a_dict = { 1:'This is the value, for the key 1', 'This is the key for a value 1':1, 1:
```



The values are accessed using the keys

```
a_dict['This is the key for a value 1']
```



```
1
```

New key/value pairs can be added by just supplying the new value for the new key

```
a_dict['new key'] = 'new_value'
```



To loop over a range of numbers, the syntax is

```
for j in range(5):
    print(j)
```



```
0
1
2
3
4
```

Note that it starts at 0 (by default), and ends at n-1 for `range(n)`.

You can also loop over any 'iterable' object, such as lists

```
for j in a_list:
    print(j)
```



```
42
0.5
True
[0, 1]
None
apple
3.14
```

or dictionaries

```
for key in a_dict:
    value = a_dict[key]
    print('key =',key)
    print('value =',value)
    print()
```



```
key = 1
value = This is the value, for the key 1

key = This is the key for a value 1
value = 1

key = False
value = :)
```

```
key = (0, 1)
value = 256

key = new_key
value = new_value
```

Conditional statements are done with `if`, `elif` and `else` with the following syntax.

```
if 'strawberry' in a_list:
    print('We have a strawberry!')
elif a_list[5]=='apple':
    print('We have an apple!')
else:
    print('Not much fruit here!')
```

We have an apple!

Importing packages is done with a line such as

```
import numpy
```

The `numpy` package is important for doing maths

```
numpy.sin( numpy.pi/2 )
```

1.0

We have to write `numpy.` in front of every numpy command so that it knows to find that command defined in `numpy`. To save writing, it is common to use

```
import numpy as np

np.sin( np.pi/2 )
```

1.0

Then you only need the shortened name. Most people use `np`, but you can choose what you like.

You can also pull everything straight out of `numpy` with

```
from numpy import *
```



Then you can use the commands directly. But this can cause packages to mess with each other, so use with caution.

```
sin( pi/2 )
```



```
1.0
```



If you want to do trigonometry, linear algebra, etc, you can use `numpy`. For plotting, use `matplotlib`. For graph theory, use `networkx`. For quantum computing, use `qiskit`. For whatever you want, there will probably be a package to help you do it.

A good thing to know about in any language is how to make a function.

Here's a function, whose name was chosen to be `do_some_maths`, whose inputs are named `Input1` and `Input2` and whose output is named `the_answer`.

```
def do_some_maths ( Input1, Input2 ):
    the_answer = Input1 + Input2
    return the_answer
```



It's used as follows

```
x = do_some_maths(1,72)
print(x)
```



```
73
```

If you give a function an object, and the function calls a method of that object to alter its state, the effect will persist. So if that's all you want to do, you don't need to `return` anything. For example, let's do it with the `append` method of a list.

```
def add_sausages ( input_list ):
    if 'sausages' not in input_list:
```



```
input_list.append('sausages')
```

```
print('List before the function')
print(a_list)

add_sausages(a_list) # function called without an output

print('\nList after the function')
print(a_list)
```



```
List before the function
[42, 0.5, True, [0, 1], None, 'apple', 3.14]
```

```
List after the function
[42, 0.5, True, [0, 1], None, 'apple', 3.14, 'sausages']
```

Randomness can be generated using the `random` package.

```
import random
```



```
for j in range(5):
    print('* Results from sample',j+1)
    print('\n    Random number from 0 to 1:', random.random() )
    print("\n    Random choice from our list:", random.choice( a_list ) )
    print('\n')
```



```
* Results from sample 1
```

```
    Random number from 0 to 1: 0.24483110888696868
```

```
    Random choice from our list: [0, 1]
```

```
* Results from sample 2
```

```
    Random number from 0 to 1: 0.7426371646254912
```

```
    Random choice from our list: [0, 1]
```

```
* Results from sample 3
```

```
Random number from 0 to 1: 0.7269519228900921
```

```
Random choice from our list: 42
```

* Results from sample 4

```
Random number from 0 to 1: 0.8707823815722878
```

```
Random choice from our list: apple
```

* Results from sample 5

```
Random number from 0 to 1: 0.2731676546693854
```

```
Random choice from our list: True
```

You know have the basics. Now all you need is a search engine, and the intuition to know who is worth listening to on Stack Exchange. Then you can do anything with Python. Your code might not be the most 'Pythonic', but only Pythonistas really care about that.

Basic Qiskit Syntax

Installation

Qiskit is a package in Python for doing everything you'll ever need with quantum computing.

If you don't have it already, you need to install it. Once it is installed, you need to import it.

There are generally two steps to installing Qiskit. The first one is to install Anaconda, a python package that comes with almost all dependencies that you will ever need. Once you've done this, Qiskit can then be installed by running the command

```
pip install qiskit
```

in your terminal. For detailed installation instructions, refer to [the documentation page here](#).

Note: The rest of this section is intended for people who already know the fundamental concepts of quantum computing. It can be used for readers who wish to skip straight to the later chapters in which those concepts are put to use. All other readers should read the [Introduction to Python and Jupyter notebooks](#), and then move on directly to the start of [Chapter 1](#).

Quantum circuits

```
from qiskit import *
```

The object at the heart of Qiskit is the quantum circuit. Here's how we create one, which we will call qc

```
qc = QuantumCircuit()
```

This circuit is currently completely empty, with no qubits and no outputs.

Quantum registers

To make the circuit less trivial, we need to define a register of qubits. This is done using a `QuantumRegister` object. For example, let's define a register consisting of two of qubits and call it `qr`.

```
qr = QuantumRegister(2, 'qreg')
```



Giving it a name like '`qreg`' is optional.

Now we can add it to the circuit using the `add_register` method, and see that it has been added by checking the `qregs` variable of the circuit object.

```
qc.add_register( qr )
```



```
qc.qregs
```

```
[QuantumRegister(2, 'qreg')]
```

Now our circuit has some qubits, we can use another attribute of the circuit to see what it looks like: `draw()`.

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



qreg₀ —

qreg₁ —

Our qubits are ready to begin their journey, but are currently just sitting there in state $|0\rangle$.

Applying Gates

To make something happen, we need to add gates. For example, let's try out `h()`.

```
qc.h()
```



```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-6-68b196ebf214> in <module>  
----> 1 qc.h()  
  
TypeError: h() missing 1 required positional argument: 'q'
```

Here we got an error, because we didn't tell the operation which qubit it should act on. The two qubits in our register `qr` can be individually addressed as `qr[0]` and `qr[1]`.

```
qc.h( qr[0] )
```



```
<qiskit.circuit.instructionset.InstructionSet at 0x1232eddd8>
```

Ignore the output in the above. When the last line of a cell has no `=`, Jupyter notebooks like to print out what is there. In this case, it's telling us that there is a Hadamard as defined by Qiskit. To suppress this output, we could use a `;`.

We can also add a controlled-NOT using `cx`. This requires two arguments: control qubit, and then target qubit.

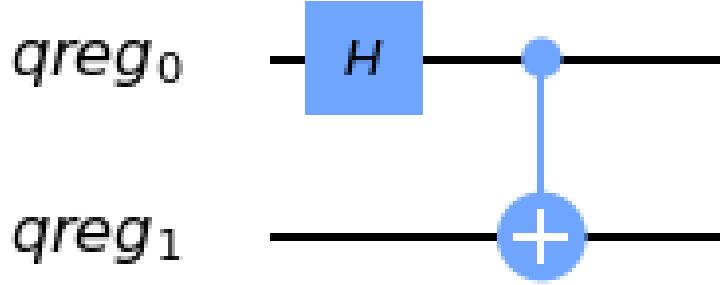
```
qc.cx( qr[0], qr[1] );
```



Now our circuit has more to show

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





Statevector simulator

We are now at the stage that we can actually look at an output from the circuit. Specifically, we will use the 'statevector simulator' to see what is happening to the state vector of the two qubits.

To get this simulator ready to go, we use the following line.

```
vector_sim = Aer.get_backend('statevector_simulator')
```



In Qiskit, we use *backend* to refer to the things on which quantum programs actually run (simulators or real quantum devices). To set up a job for a backend, we need to set up the corresponding backend object.

The simulator we want is defined in the part of qiskit known as `Aer`. By giving the name of the simulator we want to the `get_backend()` method of `Aer`, we get the backend object we need. In this case, the name is '`statevector_simulator`' .

A list of all possible simulators in `Aer` can be found using

```
Aer.backends()
```



```
[<QasmSimulator('qasm_simulator') from AerProvider(),  
<StatevectorSimulator('statevector_simulator') from AerProvider(),  
<UnitarySimulator('unitary_simulator') from AerProvider()>]
```

All of these simulators are 'local', meaning that they run on the machine on which Qiskit is installed. Using them on your own machine can be done without signing up to the IBMQ user agreement.

Running the simulation is done by Qiskit's `execute` command, which needs to be provided with the circuit to be run and the 'backend' to run it on (in this case, a simulator).

```
job = execute( qc, vector_sim )
```



This creates an object that handles the job, which here has been called `job`. All we need from this is to extract the result. Specifically, we want the statevector.

```
ket = job.result().get_statevector()  
for amplitude in ket:  
    print(amplitude)
```

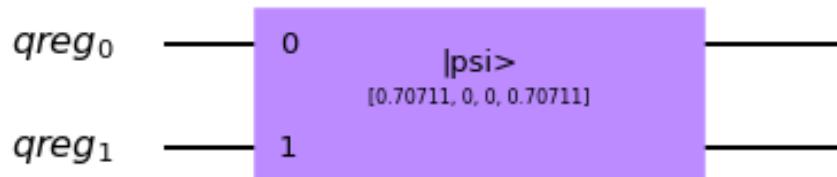


```
(0.7071067811865476+0j)  
0j  
0j  
(0.7071067811865475+0j)
```

This is the vector for a Bell state $(|00\rangle + |11\rangle)/\sqrt{2}$ ($|00\rangle + |11\rangle)/2$, which is what we'd expect given the circuit.

While we have a nicely defined state vector, we can show another feature of Qiskit: it is possible to initialize a circuit with an arbitrary pure state.

```
new_qc = QuantumCircuit( qr )  
  
new_qc.initialize( ket, qr )  
  
new_qc.draw(output='mpl')
```



Classical registers and the qasm simulator

In the above simulation, we got out a statevector. That's not what we'd get from a real quantum computer. For that we need measurement. And to handle measurement we need to define where

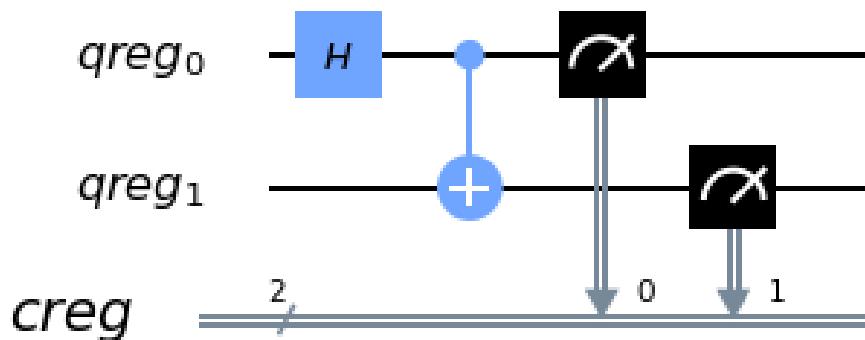
the results will go. This is done with a `ClassicalRegister`. Let's define a two bit classical register, in order to measure both of our two qubits.

```
cr = ClassicalRegister(2, 'creg')  
qc.add_register(cr)
```

Now we can use the `measure` method of the quantum circuit. This requires two arguments: the qubit being measured, and the bit where the result is written.

Let's measure both qubits, and write their results in different bits.

```
qc.measure(qr[0], cr[0])  
qc.measure(qr[1], cr[1])  
  
qc.draw(output='mpl')
```



Now we can run this on a local simulator whose effect is to emulate a real quantum device. For this we need to add another input to the `execute` function, `shots`, which determines how many times we run the circuit to take statistics. If you don't provide any `shots` value, you get the default of 1024.

```
emulator = Aer.get_backend('qasm_simulator')  
  
job = execute( qc, emulator, shots=8192 )
```

The result is essentially a histogram in the form of a Python dictionary.

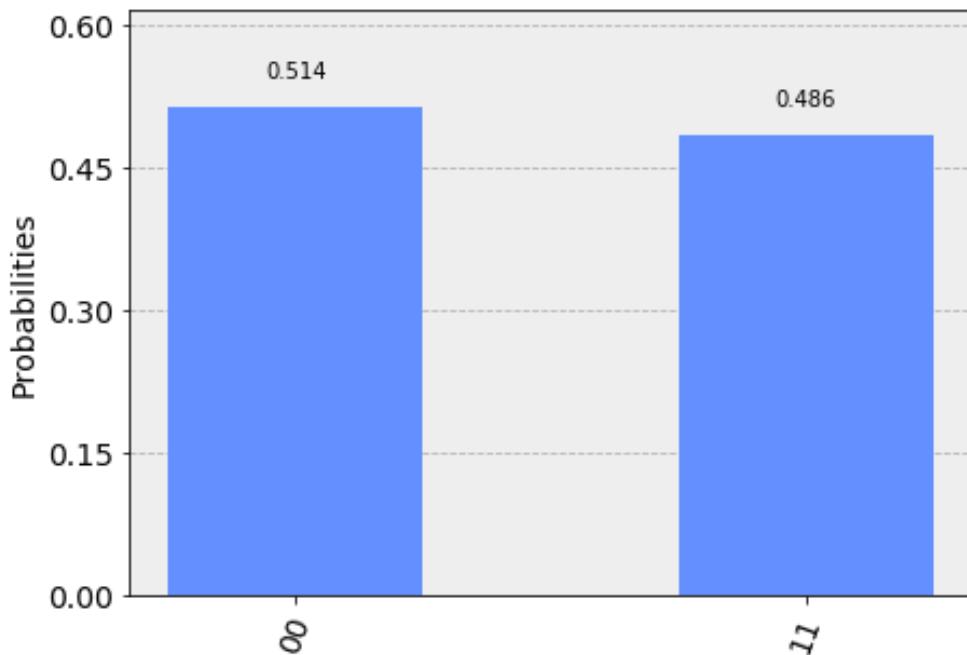
```
hist = job.result().get_counts()  
print(hist)
```



```
{'00': 4214, '11': 3978}
```

We can even get qiskit to plot it as a histogram.

```
from qiskit.tools.visualization import plot_histogram  
  
plot_histogram( hist )
```



For compatible backends we can also ask for and get the ordered list of results.

```
job = execute( qc, emulator, shots=10, memory=True )  
samples = job.result().get_memory()  
print(samples)
```



```
['11', '11', '00', '00', '00', '11', '11', '11', '11', '11']
```

Note that the bits are labelled from right to left. So `cr[0]` is the one to the furthest right, and so on. As an example of this, here's an 8 qubit circuit with a Pauli XX on only the qubit numbered 7, which has its output stored to the bit numbered 7.

```
qubit = QuantumRegister(8)
bit = ClassicalRegister(8)
circuit = QuantumCircuit(qubit,bit)

circuit.x(qubit[7])
circuit.measure(qubit,bit) # this is a way to do all the qc.measure(qr8[j],cr8[j]) at once

execute( circuit, emulator, shots=8192 ).result().get_counts()
```

```
{'10000000': 8192}
```

The 1 appears at the left.

This numbering reflects the role of the bits when they represent an integer.

$$b_{n-1} \ b_{n-2} \ \dots \ b_1 \ b_0 = \sum_j b_j 2^j$$

So the string we get in our result is the binary for $2^7 27$ because it has a 1 for the bit numbered 7.

Simplified notation

Multiple quantum and classical registers can be added to a circuit. However, if we need no more than one of each, we can use a simplified notation.

For example, consider the following.

```
qc = QuantumCircuit(3)
```

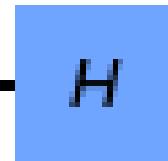
The single argument to `QuantumCircuit` is interpreted as the number of qubits we want. So this circuit is one that has a single quantum register consisting of three qubits, and no classical register.

When adding gates, we can then refer to the three qubits simply by their index: 0, 1 or 2. For example, here's a Hadamard on qubit 1.

```
qc.h(1)
```

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

*q*₀ —————

*q*₁ —————  —————

*q*₂ —————

To define a circuit with both quantum and classical registers, we can supply two arguments to `QuantumCircuit`. The first will be interpreted as the number of qubits, and the second will be the number of bits. For example, here's a two qubit circuit for which we'll take a single bit of output.

```
qc = QuantumCircuit(2,1)
```

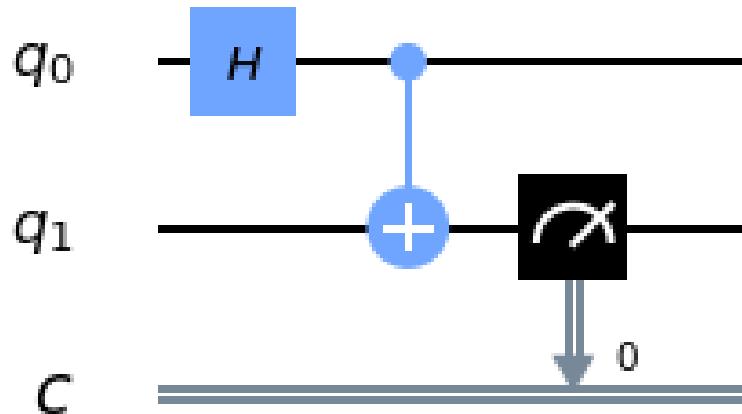


To see this in action, here is a simple circuit. Note that, when making a measurement, we also refer to the bits in the classical register by index.

```
qc.h(0)
qc.cx(0,1)
qc.measure(1,0)

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



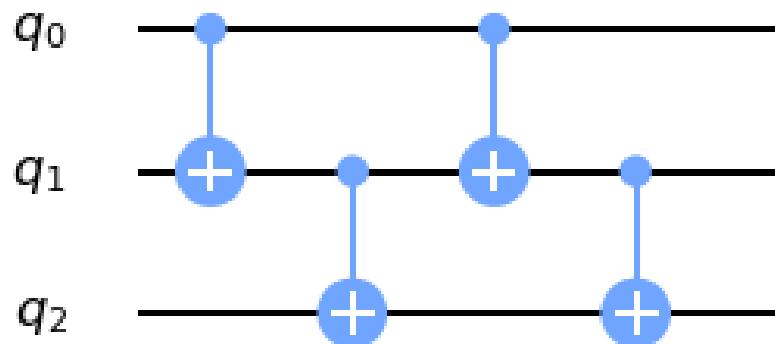


Creating custom gates

As we've seen, it is possible to combine different circuits to make bigger ones. We can also use a more sophisticated version of this to make custom gates. For example, here is a circuit that implements a `cx` between qubits 0 and 2, using qubit 1 to mediate the process.

```
sub_circuit = QuantumCircuit(3, name='toggle_cx')
sub_circuit.cx(0,1)
sub_circuit.cx(1,2)
sub_circuit.cx(0,1)
sub_circuit.cx(1,2)

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



We can now turn this into a gate

```
toggle_cx = sub_circuit.to_instruction()
```

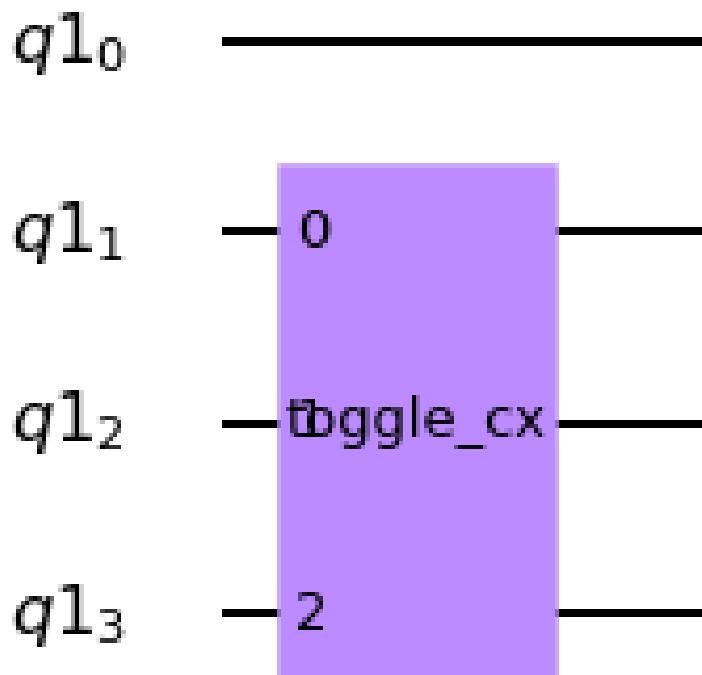


and then insert it into other circuits using any set of qubits we choose

```
qr = QuantumRegister(4)
new_qc = QuantumCircuit(qr)

new_qc.append(toggle_cx, [qr[1],qr[2],qr[3]])

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



Accessing on real quantum hardware

Backend objects can also be set up using the `IBMQ` package. The use of these requires us to [sign with an IBMQ account](#). Assuming the credentials are already loaded onto your computer, you sign in with

```
IBMQ.load_account()
```



```
<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>
```

Now let's see what additional backends we have available.

```
provider = IBMQ.get_provider(hub='ibm-q')
provider.backends()
```



```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmqx4') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmqx2') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_16_melbourne') from IBMQ(hub='ibm-q', group='open', project='main')>,
 <IBMQBackend('ibmq_ourense') from IBMQ(hub='ibm-q', group='open', project='main')>]
```



Here there is one simulator, but the rest are prototype quantum devices.

We can see what they are up to with the `status()` method.

```
for backend in provider.backends():
    print(backend.status())
```



```
BackendStatus(backend_name='ibmq_qasm_simulator', backend_version='0.0.0', operational=True)
BackendStatus(backend_name='ibmqx4', backend_version='1.0.0', operational=False, pending=False)
BackendStatus(backend_name='ibmqx2', backend_version='1.3.0', operational=False, pending=False)
BackendStatus(backend_name='ibmq_16_melbourne', backend_version='1.0.0', operational=False, pending=False)
BackendStatus(backend_name='ibmq_ourense', backend_version='1.0.0', operational=True, pending=False)
```

Let's get the backend object for the largest public device.

```
real_device = provider.get_backend('ibmq_16_melbourne')
```



We can use this to run a job on the device in exactly the same way as for the emulator.

We can also extract some of its properties.

```
properties = real_device.properties()
coupling_map = real_device.configuration().coupling_map
```



From this we can construct a noise model to mimic the noise on the device.

```
from qiskit.providers.aer import noise  
  
noise_model = noise.device.basic_device_noise_model(properties)
```

And then run the job on the emulator, with it reproducing all these features of the real device. Here's an example with a circuit that should output '10' in the noiseless case.

```
qc = QuantumCircuit(2,2)  
qc.x(1)  
qc.measure(0,0)  
qc.measure(1,1)  
  
job = execute(qc, emulator, shots=1024, noise_model=noise_model,  
             coupling_map=coupling_map,  
             basis_gates=noise_model.basis_gates)  
  
job.result().get_counts()
```

```
{'00': 40, '01': 1, '10': 967, '11': 16}
```

Now the very basics have been covered, let's learn more about what qubits and quantum circuits are all about.

An Introduction to Linear Algebra for Quantum Computing

```
from matplotlib import pyplot as plt
import numpy as np
from qiskit import *
from qiskit.visualization import plot_bloch_vector
```



Introduction

Linear algebra is the language of quantum computing. For this reason, it is crucial to develop a good understanding of the basic mathematical concepts that are built upon in order to arrive at many of the amazing and interesting constructions seen in quantum computation! The goal of this section of the textbook is to start at the absolute basics of linear algebra: vectors and vector spaces, and build a foundation that the reader can then leverage in their study of quantum computing.

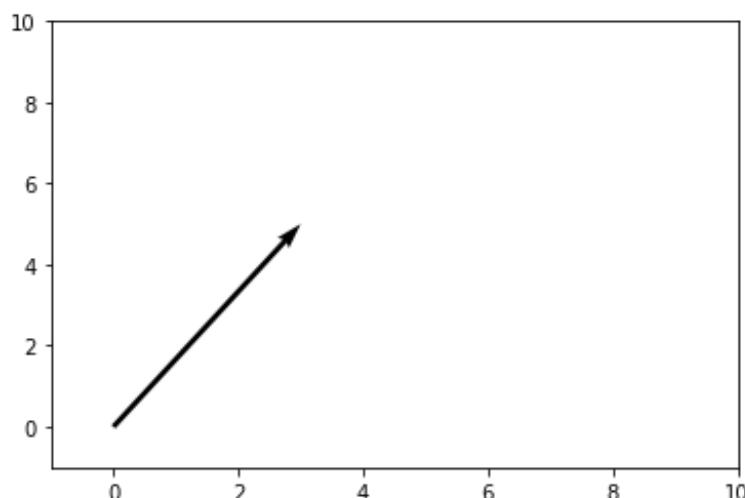
Vectors and Vector Spaces

We will start our investigation into introductory linear algebra by first discussing one of the most important mathematical quantities in quantum computation, the vector!

Formally, a **vector** $|v\rangle$ is defined as elements of a set known as a vector space. A more intuitive and geometric definition is that a vector "is a mathematical quantity with both direction and magnitude". For instance, consider a vector with x and y components

of the form $\begin{pmatrix} 3 \\ 5 \end{pmatrix}$. This vector can be visualized as an arrow pointing in the direction of 3 units down the x axis and 5 units up the y axis:

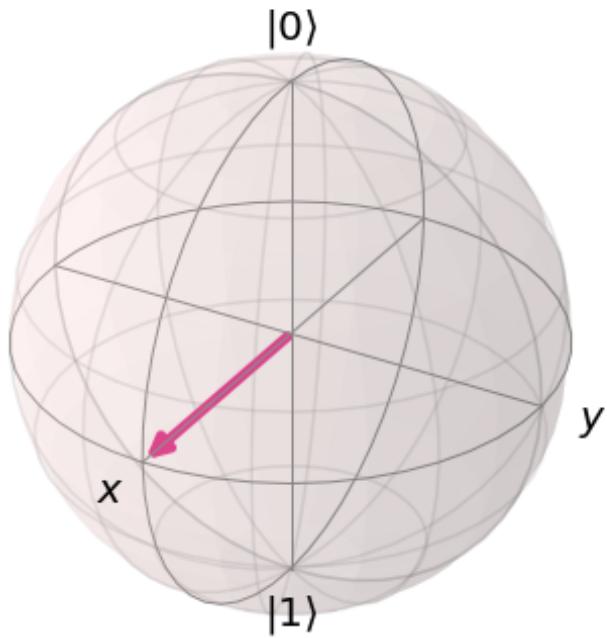
```
plt.figure()
ax = plt.gca()
ax.quiver([3], [5], angles='xy', scale_units='xy', scale=1)
ax.set_xlim([-1, 10])
ax.set_ylim([-1, 10])
plt.draw()
plt.show()
```



Note that "tail" of the vector doesn't have to be positioned at the origin, all that matters is that it is pointing in the correct direction.

Now, let's talk about a concrete example of vectors in quantum computing. Oftentimes, in quantum computing, we deal with **state vectors**, which are simply vectors, no different than the one just presented that point to a specific point in space that corresponds to a particular quantum state. Oftentimes, this is visualized using a Bloch sphere. For instance, a vector, representing the state of a quantum system could look something like this arrow, enclosed inside the Bloch sphere, which is the so-called "state" space of all possible points to which our state vectors can "point":

```
plot_bloch_vector([1, 0, 0])
```



Hopefully this help to illuminate how vectors can be used to store information about quantum states. This particular state corresponds to an even superposition between $|0\rangle$ and $|1\rangle$ (you can see that the arrow is halfway between $|0\rangle$ at the top and $|1\rangle$ at the bottom of the sphere!). Our vectors are allowed to rotate anywhere on the surface of the sphere, and each of these points represents a different quantum state!

Now that we have talked about vectors a bit more intuitively, in the context of quantum computing, we can go back to our more formal definition of a vector, which is that a vector is an element of a vector space. From this, it follows that we must define a vector space. A **vector space** V over a **field** F is a set of objects (vectors), where two conditions hold. Firstly, **vector addition** of two vectors $|a\rangle, |b\rangle \in V$ will yield a third vector $|a\rangle + |b\rangle = |c\rangle$, also contained in V . The second condition is that **scalar multiplication** between some $|a\rangle \in V$ and some $n \in F$, denoted by $n|a\rangle$ is also contained within V .

We will now clarify this previous definition by working through a basic example. Let us demonstrate that the set \mathbb{R}^2 over the field \mathbb{R} is a vector space. We assert that

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \\ y_1 + y_2 \end{pmatrix}$$

is contained within \mathbb{R}^2 . Well, this is evidently the case, as the sum of two real numbers is a real number, making both components of the newly-formed vector real numbers, and in turn making the vector be contained in \mathbb{R}^2 by definition. We also assert that:

$$n|v\rangle = \begin{pmatrix} nx \\ ny \end{pmatrix} \in V \quad \forall n \in \mathbb{R}$$

This is true as well, as the product of a real number and a real number is in turn a real number, making the entires of the new vector real, and thus proving this statement.

Matrices and Matrix Operations

Now that we have introduced vectors, we can turn our attention to another fundamental concept: a **matrix**. The best way to think of matrices from a quantum computational/quantum mechanical perspective is the fact that matrices are mathematical objects that transform vectors to other vectors:

$$|v\rangle \rightarrow |v'\rangle = M|v\rangle$$

Generally, matrices are written as "arrays" of numbers, looking something like this:

$$M = \begin{pmatrix} 1 & -2 & 31 & 5i & 01 & + i & 7 & -4 \end{pmatrix}$$

We can actually "apply" a matrix to a vector by performing matrix multiplication. In general, matrix multiplication between two matrices involves taking the first row of the first matrix, and multiplying each element by its "partner" in the first column of the second matrix (the first number of the row is multiplied by the first number of the column, second number of the row and second number of column, etc.) These new numbers are then added up, and we have the first element of the first row of the new matrix! To fill in the rest of the first row, we repeat this process for the second, third, etc. columns of the second matrix. Then we take the second row of the first matrix, and repeat the process for each column of the second matrix, getting the second row. We perform this process until we have used all rows of the first matrix. The resulting matrix is our new matrix! That may have been confusing, so here is an example:

$$\begin{pmatrix} 2 & 05 & -1 \end{pmatrix} \begin{pmatrix} -3 & 12 & 1 \end{pmatrix} = \begin{pmatrix} (2)(-3) + (0)(2) & (2)(1) + (0)(1)(5) & (-1)(2) + (5)(1) + (-1)(1) \end{pmatrix} = \begin{pmatrix} -6 & 2-17 & 4 \end{pmatrix}$$

As you can see, we simply used the previously outlined process to arrive at the final matrix! Going back to "applying" matrices to vectors, all we have to realize is that a vector is simply a matrix with 1 column, so matrix multiplication behaves the exact same way! As we previously discussed, in quantum computing we have some quantum state vector that we are manipulating in order to perform quantum computation. Well, as you can probably guess, the way that we represent the manipulation of that vector mathematically is through the application of matrices. We manipulate qubits in our quantum computer by applying sequences of

quantum gates. As it turns out, we can express each of these quantum gates as a different matrix that can be "applied" to a start vectors, thus changing the state. For instance, one of the most commonly seen quantum gates is the Pauli-X gate, which is represented by the following matrix:

$$\sigma_x = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

This gate acts kind of similarly to the classical NOT logic gate. It maps the computational basis state $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$ (it "flips" the state). As actual column vectors, we write the two basis states:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$$

So when we apply this matrix to each of the vectors:

$$\sigma_x |0\rangle = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} (0)(1) + (1)(0)(1)(1) + (0)(0) \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = |1\rangle$$

$$\sigma_x |1\rangle = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} (0)(0) + (1)(1)(1)(0) + (0)(1) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = |0\rangle$$

And so the matrix acts on the state vectors exactly as expected!

Within quantum computation, we often deal with two very important types of matrices: **Hermitian** and **Unitary** matrices. The former is more important in the study of quantum mechanics, but is still definitely worth talking about in a study of quantum computation. The latter is of unparalleled importance in both quantum mechanics and quantum computation, if there is one concept that the reader should take away from this entire linear algebra section, it should be the idea of a unitary matrix.

Firstly, a Hermitian matrix is simply a matrix that is equal to its **conjugate transpose** (denoted with a $+$ symbol). This essentially means taking a matrix, flipping the sign in its imaginary components, and then reflecting the entries of the matrix across its main diagonal (the diagonal that goes from the top left corner to the bottom right corner). For instance, a matrix that we commonly use in quantum computation, the Pauli-Y matrix is Hermitian:

$$\sigma_y = \begin{pmatrix} 0 & -i & 0 \\ i & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \Rightarrow \sigma_{ty} = \begin{pmatrix} 0 & -(i) & (-i) \\ i & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & -i & 0 \\ i & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \sigma_y$$

Notice how we switched the places of the i and the $-i$ (as we are reflecting across the main diagonal, the zeroes remain unchanged), and then flip the sign. A unitary matrix is very similar. Specifically, a unitary matrix is a matrix such that the **inverse matrix** is equal to the conjugate transpose of the original matrix.

Let's go on a quick tangent and talk a bit about what the **inverse matrix** is. The inverse of some matrix A , denoted as A^{-1} is simply a matrix such that:

$$A^{-1}A = AA^{-1} = I$$

Where I is the identity matrix. The identity matrix is just a matrix that has zeroes everywhere, except along the main diagonal (top left to bottom right), which is all ones. The reason why it is called the identity matrix is because it acts trivially on any other matrix (it has no effect). If you don't believe me, check it in a couple cases and convince yourself!

I don't want to go too deep into the inverse matrix, and frankly, when matrices get larger than 2×2 , calculating the inverse becomes a huge pain and is left to computers for the most part. For a 2×2 matrix, the inverse is defined as:

$$A = \begin{pmatrix} a & bc \\ d & \end{pmatrix} \Rightarrow A^{-1} = \frac{1}{\det A} \begin{pmatrix} d & -b-c \\ a & \end{pmatrix}$$

Where $\det A$ is the **determinant** of the matrix. Calculating the matrix for anything larger than 2×2 is really annoying, so I won't go into it. In the 2×2 case, $\det A = ad - bc$.

To be completely honest, the calculating of inverse matrices is **rarely** important in quantum computing. Since most of the matrices we deal with are unitary, we already know that the inverse is simply given by taking the conjugate transpose, so we don't have to go through this annoying process of rigorously calculating the inverse.

Anyways, going back to unitary matrices, we can now look at a basic example. As it turns out, the Pauli-Y matrix, in addition to being Hermitian, is also unitary (it is equal to its conjugate transpose, which is also equal to its inverse, therefore, the Pauli-Y matrix is its own inverse, that's pretty cool!). We can verify that this matrix is in fact unitary:

$$\sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad \sigma_{ty} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \Rightarrow \sigma_{ty}\sigma_y = \begin{pmatrix} (0)(0) + (-i)(i) & (0)(-i) + (-i)(0)(i)(0) \\ (0)(i) + (i)(-i) & (0)(0) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$$

The reason why unitary matrices are so important in the context of quantum computation will become more apparent in the section on Hilbert spaces, but the basic idea is that evolution of a quantum state by application of a unitary matrix "preserve" the quantum state in a sense (we will discuss this more rigorously in the Hilbert spaces section, and even more so in the quantum mechanics subtopic of the textbook).

Spanning Sets, Linear Dependence and Bases

We are now in a position to discuss the construction of vector spaces. Consider some vector space V . We say that some set of vectors S spans a subspace $V_S \subset V$ (subset closed under vector space operations) of the vector space if we can write any vector in the subspace as a **linear combination** of vectors contained within the spanning set.

A linear combination of some collection vectors $|v_1\rangle, \dots, |v_n\rangle$ in some vector space over a field F is defined as an arbitrary sum of these vectors (which of course will be another vector, which we will call $|v\rangle$):

$$|v\rangle = f_1|v_1\rangle + f_2|v_2\rangle + \dots + f_n|v_n\rangle = \sum i f_i |v_i\rangle$$

Where each f_i is some element of F. Now, if we have a set of vectors that spans a space, we are simply saying that **any other vector** in the vector space can be written as a linear combination of these vectors.

Now, we are in a position to define a **basis**, which is a specific case of a spanning set, but first, we must talk about **linear dependence**. This is a fairly straightforward idea as well. A set of vectors $|v_1\rangle, \dots, |v_n\rangle$ is said to be linearly dependent if there exist corresponding, coefficients for each vector, $b_i \in F$, such that:

$$b_1|v_1\rangle + b_2|v_2\rangle + \dots + b_n|v_n\rangle = \sum i b_i |v_i\rangle = 0$$

Where at least one of the b_i coefficients is non-zero. This is equivalent to the more intuitive statement that "the set of vectors can be expressed as linear combinations of each other". This can be proven fairly simply. Let us have the set $\{|v_1\rangle, \dots, |v_n\rangle\}$ along with the corresponding coefficients $\{b_1, \dots, b_n\}$, such that the linear combination is equal to 0. Since there is at least one vector with a non-zero coefficient, we choose a term in the linear combination $b_a|v_a\rangle$:

$$\sum i b_i |v_i\rangle = b_a |v_a\rangle + \sum i, i \neq a b_i |v_i\rangle = 0 \Rightarrow |v_a\rangle = - \sum i, i \neq a b_i b_a |v_i\rangle = \sum i, i \neq a c_i |v_i\rangle$$

In the case that b_a is the only non-zero coefficient, it is necessarily true that $|v_a\rangle$ is the null vector, automatically making the set linearly dependent. If this is not the case, $|v_a\rangle$ has been written as a linear combination of non-zero vectors, as was shown above. To prove the converse, we assume that there exists some vector $|v_a\rangle$ in the subspace $|v_1\rangle, \dots, |v_n\rangle$ that can be written as a linear combination of other vectors in the subspace. This means that:

$$|v_a\rangle = \sum s b_s |v_s\rangle$$

Where s is an index that runs over a subset of the subspace. It follows that:

$$|v_a\rangle - \sum s b_s |v_s\rangle = |v_a\rangle - (|v_{s_1}\rangle + \dots + |v_{s_r}\rangle) = 0$$

For all vectors in the subspace that are not included in the subset indexed by s, we set their coefficients, indexed by q equal to 0. Thus,

$$|v_a\rangle - (|v_{s_1}\rangle + \dots + |v_{s_r}\rangle) + (0)(|v_{q_1}\rangle + \dots + |v_{q_t}\rangle) = 0$$

Which is a linear combination of all elements in the subspace $|v_1\rangle, \dots, |v_n\rangle$ which is equal to 0, thus completing the proof that the two definitions of linear dependence imply each other.

Let's now consider a basic example. For instance, consider the set of two vectors in \mathbb{R}^2 , consisting of $|a\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and

$|b\rangle = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$. Well, for example, if we choose the field over our vector space to be \mathbb{R} , then we can create a linear combination of these vectors that equates to 0. For example:

$$2|a\rangle - |b\rangle = 0$$

A set of vectors is said to be **linearly independent** if a linear combination of vectors is 0 only if every coefficient is equal to 0.

The notion of a basis is simply a **linearly independent spanning set**. In this sense, the basis of a vector space is the minimal possible set that spans the entire space. We call the size of the basis set the **dimension** of the vector space.

The reason why bases and spanning sets are so important is because they allow us to "shrink down" vector spaces and express them in terms of only a few vectors rather than a ton! Oftentimes, we can come to certain conclusions about our basis set that we can generalize to the entire vector space, simply because we know every vector in the space is just a linear combination of the basis vectors. Consider in quantum computation, how one of the bases that we often deal with is $|0\rangle, |1\rangle$. We can write any other qubit state as a linear combination of these basis vectors. For instance, the linear combination

$$|0\rangle + |1\rangle\sqrt{2}$$

represents a superposition between the $|0\rangle$ and $|1\rangle$ basis state, with equal probability of measuring the state to be in either one of the basis vector states (this is kind of intuitive, as the "weight" or the "amount of each basis vector" in the linear combination is equal, both being scaled by $1/\sqrt{2}$).

Hilbert Spaces, Orthonormality, and the Inner Product

Hilbert Spaces are one of the most important mathematical constructs in quantum mechanics and quantum computation. Less rigorously, a Hilbert space can be thought of as the space state in which all quantum state vectors "live". The main fact that differentiates a Hilbert space from any random vector space is that a Hilbert space is equipped with an **inner product**, which is an operation that can be performed between two vectors, returning a scalar.

In the context of quantum mechanics and quantum computation, the inner product between two state vectors returns a scalar quantity representing the amount to which the first vector lies along the second vector. From this, the probabilities of measurement in different quantum states, among other things can be calculated (this will be discussed more in the quantum mechanics subtopic).

For two vectors $|a\rangle$ and $|b\rangle$ in a Hilbert space, we denote the inner product as $\langle a | b \rangle$, where $\langle a |$ is equal to the conjugate transpose of $|a\rangle$, denoted $|a\rangle^\dagger$. Thus, the inner product between two vectors of the Hilbert space looks something like:

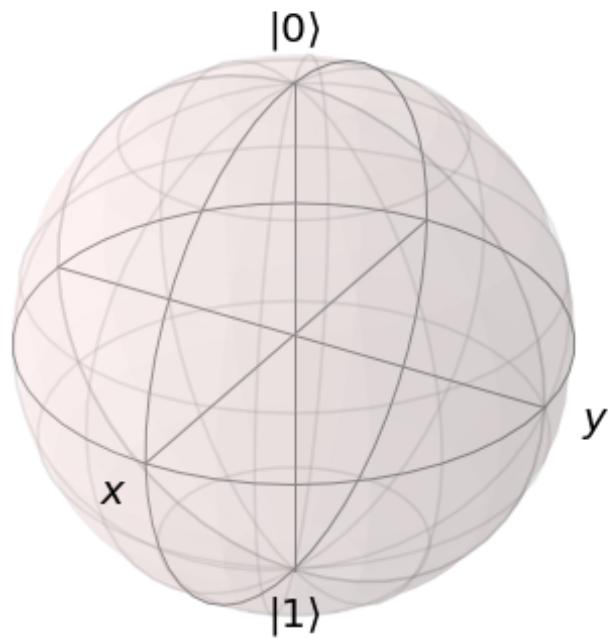
$$\langle a | b \rangle = \begin{pmatrix} a_{*1} & a_{*2} & \dots & a_{*n} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = a_{*1}b_1 + a_{*2}b_2 + \dots + a_{*n}b_n$$

Where $*$ denotes the complex conjugate of the vector.

One of the most important conditions for a Hilbert space representing a quantum system is that the inner product of a vector with itself is equal to one: $\langle \psi | \psi \rangle = 1$. This is the so-called normalization condition, and essentially just states that the length of the vector squared (each component of the vector is squared and summed together, by definition of the inner product) must be equal to one. The physical significance of this is that the length of a vector in a particular direction is representative of the "probability amplitude" of the quantum system with regards to being measured in that particular state. Obviously, the probability of the quantum system being measured in the state that it is actually in must be 1, after all, the sum of the probabilities of finding the quantum system in any particular state has to equal one (I mean, it has to be in **some** state!).

Let's consider the Bloch sphere:

```
plot_bloch_vector([0, 0, 0])
```



The surface of this sphere, along with the inner product between qubit state vectors, is a valid Hilbert space! In addition to this, it can be seen that the normalization condition holds true, as the radius of the Bloch sphere is 1, therefore the length squared of each vector must also be equal to one!

The last thing that is worth noting about Hilbert spaces and the inner product is their relationship to **unitary matrices**. The reason why unitary matrices are so important in quantum computation is because they **preserve the inner product**, meaning that no matter how you transform a vector under a sequence of unitary matrices, the normalization condition still holds true. This can be demonstrated in the following short proof:

$$\langle \psi | \psi \rangle = 1 \Rightarrow |\psi\rangle \rightarrow U|\psi\rangle = |\psi'\rangle \Rightarrow \langle \psi' | \psi' \rangle = (U|\psi\rangle)^\dagger U|\psi\rangle = \langle \psi | U^\dagger U|\psi\rangle = \langle \psi | \psi \rangle = 1$$

This essentially means that unitary evolution sends quantum states to other valid quantum states. for a single qubit Hilbert space, represented by the Bloch sphere, unitary transformations correspond to rotations of state vectors to different points on the sphere, not changing the length of the state vector in any way.

Eigenvectors and Eigenvalues

Consider the relationship of the form:

$$A|v\rangle = \lambda|v\rangle$$

Where A is a matrix, and λ is some number. If we are given some matrix A, and need to find the vectors $|v\rangle$ and numbers λ that satisfy this relationship, we call these vectors **eigenvectors** and their corresponding number multipliers **eigenvalues**.

Eigenvectors and eigenvalues have very important physical significance in the context of quantum mechanics, and therefore quantum computation. Given some A, we exploit an interesting trick in order to find the set of eigenvectors and corresponding eigenvalues. Let us re-arrange our equation as:

$$A|v\rangle - \lambda|v\rangle = 0 \Rightarrow (A - \lambda I)|v\rangle = 0$$

Now, if we multiply both sides of this equation by the inverse matrix $(A - \lambda I)^{-1}$, we get $|v\rangle = 0$. This is an extraneous solution (we don't allow eigenvectors to be the null vector, or else any eigenvalue/matrix combination would satisfy the eigenvector-eigenvalue relationship). Thus, in order to find the allowed eigenvectors and eigenvalues, we have to assume that the matrix $(A - \lambda I)$ is **non-invertible**. Recall from earlier that the inverse of a matrix is of the form:

$$M^{-1} = \frac{1}{\det(M)} F(M)$$

Where $F(M)$ is some new matrix (don't worry about what that matrix actually is, it doesn't matter in this context) that depends on M. The part of this equation we are interested in is the inverse of the determinant. If the determinant of the matrix M is 0, it follows that the inverse is undefined, and thus so is the inverse, making the matrix M non-invertible! Thus, we require that:

$$\det(A - \lambda I) = 0$$

From this, we can determine λ , then we plug each value of lambda back into the original equation to get the eigenvalues! Let's do an example, and find the eigenvectors/eigenvalues of the Pauli-Z matrix, σ_z . We start with:

$$\det(\sigma_z - \lambda I) = \det \begin{pmatrix} 1 - \lambda & 0 & 0 \\ 0 & 1 - \lambda & 0 \\ 0 & 0 & -1 - \lambda \end{pmatrix} = (-1 - \lambda)(1 - \lambda) = 1 - \lambda^2 = 0 \Rightarrow \lambda = \pm 1$$

The equation, in terms of lambda that is obtained when solving the determinant is called the **characteristic polynomial**. We can then plug each of these values back into the original equation. We'll start with $\lambda = 1$:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} |v\rangle = |v\rangle \Rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} ab \\ ab \\ ab \end{pmatrix} = \begin{pmatrix} ab \\ ab \\ ab \end{pmatrix} \Rightarrow \begin{pmatrix} a-b \\ a-b \\ a-b \end{pmatrix} = \begin{pmatrix} ab \\ ab \\ ab \end{pmatrix}$$

So this means that a can be any number and b is 0. Thus, the vector $\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$ forms a basis for all vectors that satisfy our relationship, and is thus the eigenvector that corresponds to the eigenvalue of 1! We do the same thing for $\lambda = -1$:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} |v\rangle = -|v\rangle \Rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \begin{pmatrix} ab \\ ab \\ ab \end{pmatrix} = \begin{pmatrix} -a-b \\ -a-b \\ -a-b \end{pmatrix} \Rightarrow \begin{pmatrix} a-b \\ a-b \\ a-b \end{pmatrix} = \begin{pmatrix} -a-b \\ -a-b \\ -a-b \end{pmatrix}$$

This time, b can be any number and a is 0, thus our basis vector (and thus our eigenvector corresponding to -1) is $\begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$. Notice

how the eigenvectors of the Pauli-Z matrix are the quantum computational basis states $|0\rangle$ and $|1\rangle$! This is no coincidence! For instance, when we measure a qubit in the Z-basis, we are referring to performing a measurement that collapses the qubit's state into one of the eigenvectors of the Z matrix, either $|0\rangle$ or $|1\rangle$!

Matrix Exponentials

The notion of a matrix exponential is a very specific idea, but one that is so important that it warrants its own section in this part of the textbook. Oftentimes in quantum computing (and when I say oftentimes, I mean **all the time**, especially during the creation of variational or parametrized quantum circuits), we will see unitary transformations in the form:

$$U = e^{i\gamma H}$$

Where H is some Hermitian matrix and γ is some real number. It is fairly simple to prove that all matrices of this form are unitary. Taking the conjugate transpose of U , we get:

$$U^\dagger = (e^{i\gamma H})^\dagger = e^{-i\gamma H^\dagger}$$

But since H is Hermitian, we know that $H^\dagger = H$, thus:

$$e^{-iyH^\dagger} = e^{-iyH} \Rightarrow U^\dagger U = e^{-iyH} e^{iyH} = I$$

See, fairly straightforward! But wait, a matrix inside of an exponential seems super weird, how is it even still a matrix? Well, this actually becomes much more apparent when we expand our exponential function as a Taylor series. If you recall from calculus, a Taylor series is essentially a way to write any function as an infinite-degree polynomial. I won't go too far into this idea, but the main idea is that we choose the terms of the polynomial and centre it at some point x_0 lying on the function we are trying to transform into the polynomial, such that the zeroth, first, second, third, etc. derivative at this point is the same for both the original function and the polynomial. Thus, we write our Taylor series in the form:

$$g(x) = \sum_{n=0}^{\infty} f^{(n)}(x_0) (x - x_0)^n n!$$

Where $g(x)$ is the polynomial, $f(x)$ is the original function, $f^{(n)}$ is the n -th derivative of f , and x_0 is the point at which we centre the function. Since we are not approximating, x_0 doesn't actually matter, so for simplicity, we choose $x_0 = 0$, and the Taylor series becomes a Maclaurin series:

$$g(x) = \sum_{n=0}^{\infty} f^{(n)}(0) x^n n!$$

And, so if we choose $f(x) = e^x$, we can create an equivalent polynomial using the Maclaurin series. Since the derivative of e^x is simply e^x , and evidently, $e^0 = 1$, we get:

$$g(x) = \sum_{n=0}^{\infty} x^n n! = e^x$$

And so for some matrix, iyH , we get:

$$e^{iyH} = \sum_{n=0}^{\infty} (iyH)^n n!$$

This makes much more sense, the exponential of a matrix is a matrix. It is an infinite sum of powers of matrices, which looks kind of intimidating, but at least we have concluded that the matrix exponential is in fact a matrix! We are now in a position to demonstrate a very important fact, if we have some matrix B such that $B^2 = I$ (this is called an **involutory matrix**), then:

$$e^{iyB} = \cos(y)I + i\sin(y)B$$

We start with the Maclaurin series:

$$e^{iyB} = \sum_{n=0}^{\infty} (iyB)^n n!$$

Notice that we can split the summation into an imaginary part and a real part, based on whether n is even or odd in each term of the sum:

$$\sum_{n=0}^{\infty} (iyB)^n n! = \sum_{n=0}^{\infty} (-1)^n y^{2n} B^{2n} (2n)! + i \sum_{n=0}^{\infty} (-1)^n y^{2n+1} B^{2n+1} (2n+1)!$$

Now, let us find the Maclaurin series for both $\sin x$ and $\cos x$. We'll start with $f(x) = \sin x$:

$$\sin x = \sum_{n=0}^{\infty} f^n(0)x^n n!$$

Well, the derivative of $\sin x$ is **cyclical** in a sense (each arrow represents taking the derivative of the previous function):

$$\sin x \rightarrow \cos x \rightarrow -\sin x \rightarrow -\cos x \rightarrow \sin x$$

Since $\sin(0) = 0$ and $\cos(0) = 1$, all of the terms with even n become 0 and we get:

$$\sum_{n=0}^{\infty} f^n(0)x^n n! = \sum_{n=0}^{\infty} (-1)^n x^{2n+1} (2n+1)!$$

This looks awfully similar to the odd term of our original equation, in fact, if we let $x = yB$, they are exactly the same. We follow a process that is almost identical to show that the even terms are identical to the Maclaurin series for $f(x) = \cos x$:

$$\cos x = \sum_{n=0}^{\infty} f^n(0)x^n n!$$

$$\Rightarrow \cos x \rightarrow -\sin x \rightarrow -\cos x \rightarrow \sin x \rightarrow \cos x$$

$$\Rightarrow \sum_{n=0}^{\infty} f^n(0)x^n n! = \sum_{n=0}^{\infty} (-1)^n x^{2n} (2n)!$$

Now, let us go back to the original equation. Recall that $B^2 = I$ For any n , we have:

$$B^{2n} = (B^2)^n = I^n = I$$

$$B^{2n+1} = B(B^2)^n = B I^n = B I = B$$

Substituting in all of this new information, we get:

$$\sum_{n=0}^{\infty} (-1)^n \gamma^{2n} B^{2n} (2n)! + i \sum_{n=0}^{\infty} (-1)^n \gamma^{2n+1} B^{2n+1} (2n+1)! = \sum_{n=0}^{\infty} (-1)^n \gamma^{2n} (2n)! + iB \sum_{n=0}^{\infty} (-1)^n \gamma^{2n+1} B^{2n+1} (2n+1)! = \cos(\gamma)I + i\sin(\gamma)B$$

We did it! This fact is **super** useful in quantum computation! Consider the Pauli matrices:

$$\sigma_x = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

$$\sigma_y = \begin{pmatrix} 0 & 0 & -i \\ 0 & 0 & i \\ i & -i & 0 \end{pmatrix}$$

$$\sigma_z = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

These matrices are incredibly important to quantum computation, they are some of the most fundamental "quantum gates" used to manipulate qubits. As it turns out, these operations are not only unitary, they are also **Hermitian** and **Involutory**. This means that a matrix of the form $e^{i\gamma\sigma_k}$ $k \in \{x, y, z\}$ is not only a valid unitary matrix that can act upon a quantum state vector (a qubit), but it can be expressed using the sine-cosine relationship that we just proved! This is fact very powerful, and is seen throughout quantum computational theory, as gates of this type are used all the time (as you will see in future sections of this textbook)!

Before we end this section, there is one other useful fact about matrix exponentials that is worth discussing: if we have some matrix M , with eigenvectors $|v\rangle$ and corresponding eigenvalues v , then:

$$e^M |v\rangle = e^v |v\rangle$$

This one is much more straightforward to prove:

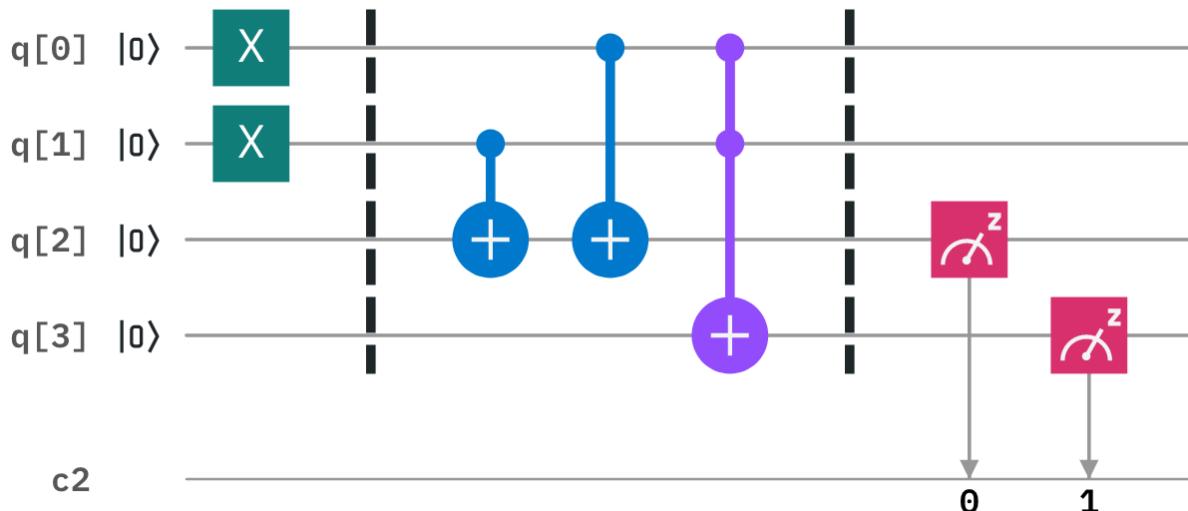
$$e^M |v\rangle = \sum_{n=0}^{\infty} B^n |v\rangle n! = \sum_{n=0}^{\infty} v^n |v\rangle n! = e^v |v\rangle$$

This fact is super useful as well. Often when creating quantum circuits that simulate a certain Hamiltonian (especially for variational circuits), gates of the form $e^{i\gamma\sigma_z}$ will be used. Well, since $|0\rangle$ and $|1\rangle$ are eigenvalues of σ_z , we can easily determine mathematically that $e^{i\gamma\sigma_z}$ will add a phase of $e^{i\gamma}$ to $|0\rangle$ and will add a phase of $e^{-i\gamma}$ to $|1\rangle$. This then allows us to construct this gate in terms of CNOT and phase/rotation gates fairly easily, as we know mathematically the outcome of the gate on each of the computational basis states.

This fact doesn't only apply to exponentials of the σ_z gate. For example, we can determine the outcome of a gate of the form $e^{i\gamma\sigma_x}$ on the eigenvectors of σ_x , $(|0\rangle + |1\rangle)/\sqrt{2}$ and $(|0\rangle - |1\rangle)/\sqrt{2}$. The same applies for exponentials of the σ_y matrix.

Introduction

If you think quantum mechanics sounds challenging, you are not alone. All of our intuitions are based on day-to-day experiences, and so are better at understanding the behavior of balls and bananas than atoms or electrons. Though quantum objects can seem random and chaotic at first, they just follow a different set of rules. Once we know what those rules are, we can use them to create new and powerful technology. Quantum computing will be the most revolutionary example of this.



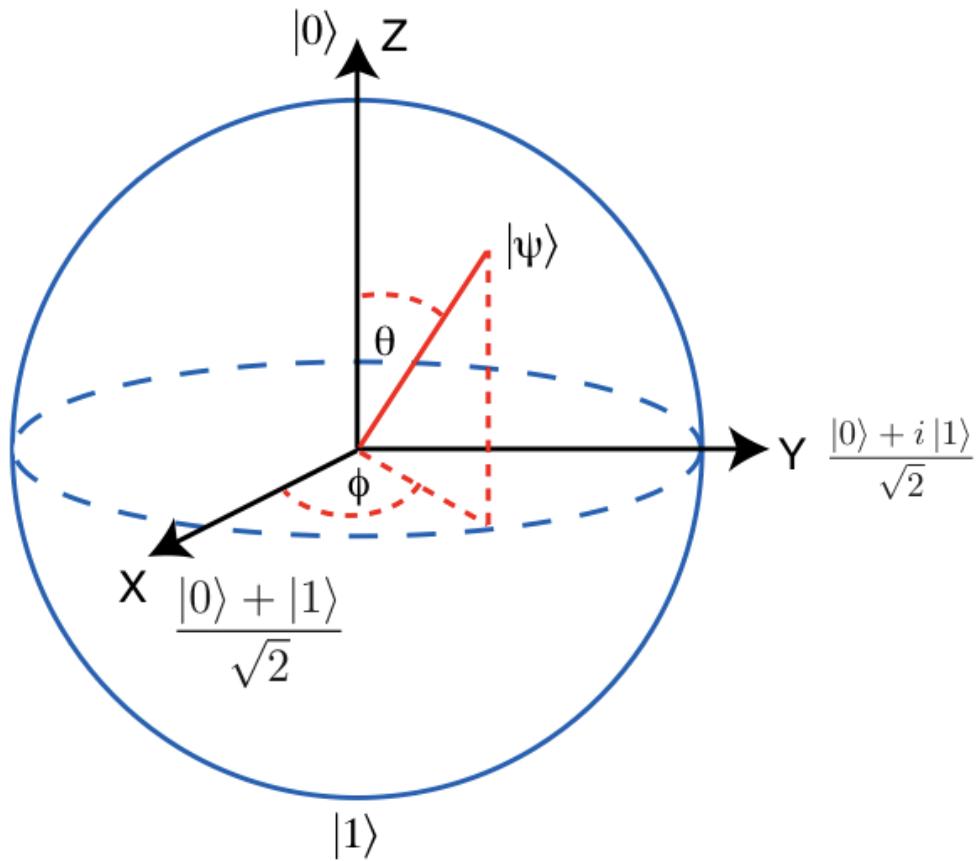
To get you started on your journey towards quantum computing, let's test what you already know. Which of the following is the correct description of a *bit*?

- A blade used by a carpenter.
- The smallest unit of information: either a `0` or a `1`.
- Something you put in a horse's mouth.

Actually, they are all correct: it's a very multi-purpose word! But if you chose the second one, it shows that you are already thinking along the right lines. The idea that information can be stored and processed as a series of `0`s and `1`s is quite a big conceptual hurdle, but it's something most people today know without even thinking about it. Taking this as a starting point, we can start to imagine bits that obey the rules of quantum mechanics. These quantum bits, or *qubits*, will then allow us to process information in new and different ways.

The first few sections in this chapter are intended for the broadest possible audience. You won't see any math that you didn't learn before you were age 10. We'll look at how bits work in standard computers, and then start to explore how qubits can allow us to do things in a different way. After reading this, you should already be able to start thinking about interesting things to try out with qubits.

We'll start diving deeper into the world of qubits. For this, we'll need some way of keeping track of what they are doing when we apply gates. The most powerful way to do this is to use the mathematical language of vectors and matrices.



This chapter will be most effective for readers who are already familiar with vectors and matrices. Those who aren't familiar will likely be fine too, though it might be useful to consult our [Introduction to Linear Algebra for Quantum Computing](#) from time to time.

Since we will be using Qiskit, our Python-based framework for quantum computing, it would also be useful to know the basics of Python. Those who need a primer can consult the [Introduction to Python and Jupyter notebooks](#).

The Atoms of Computation

Programming a quantum computer is now something that anyone can do in the comfort of their own home.

But what to create? What is a quantum program anyway? In fact, what is a quantum computer?

These questions can be answered by making comparisons to standard digital computers. Unfortunately, most people don't actually understand how digital computers work either. So in this article we'll look at the basics principles behind these devices. To help us transition over to quantum computing later on, we'll do it using the same tools as we'll use for quantum.

```
from qiskit import QuantumCircuit, execute, Aer  
from qiskit.visualization import plot_histogram
```



Splitting information into bits

The first thing we need to know about is the idea of bits. These are designed to be the world's simplest alphabet. With only two characters, 0 and 1, we can represent any piece of information.

One example is numbers. In European languages, numbers are usually represented using a string of the ten digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. In this string of digits, each represents how many times the number contains a certain powers of ten. For example, when we write 9213, we mean

$$9000 + 200 + 10 + 3$$

or, expressed in a way that emphasizes the powers of ten

$$(9 \times 10^3) + (2 \times 10^2) + (1 \times 10^1) + (3 \times 10^0)$$

Though we usually use this system based on the number 10, we can just as easily use one based on any other number. The binary number system, for example, is based on the number two. This means using the two characters 0 and 1 to express numbers as multiples of powers of two. For example, 9213 becomes 1000111111101, since

$$\begin{aligned}
 9213 = & (1 \times 2^{13}) + (0 \times 2^{12}) + (0 \times 2^{11}) + (0 \times 2^{10}) + (1 \times 2^9) + (1 \times 2^8) \\
 & + (1 \times 2^7) \\
 & + (1 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) \\
 & + (1 \times 2^0)
 \end{aligned}$$

In this we are expressing numbers as multiples of 2, 4, 8, 16, 32, etc instead of 10, 100, 1000, etc.

These strings of bits, known as binary strings, can be used to represent more than just numbers. For example, there is a way to represent any text using bits. For any letter, number or punctuation mark you want to use, you can find a corresponding string of at most eight bits using [this table](#). Though these are quite arbitrary, this is a widely agreed upon standard. In fact, it's what was used to transmit this article to you through the internet.

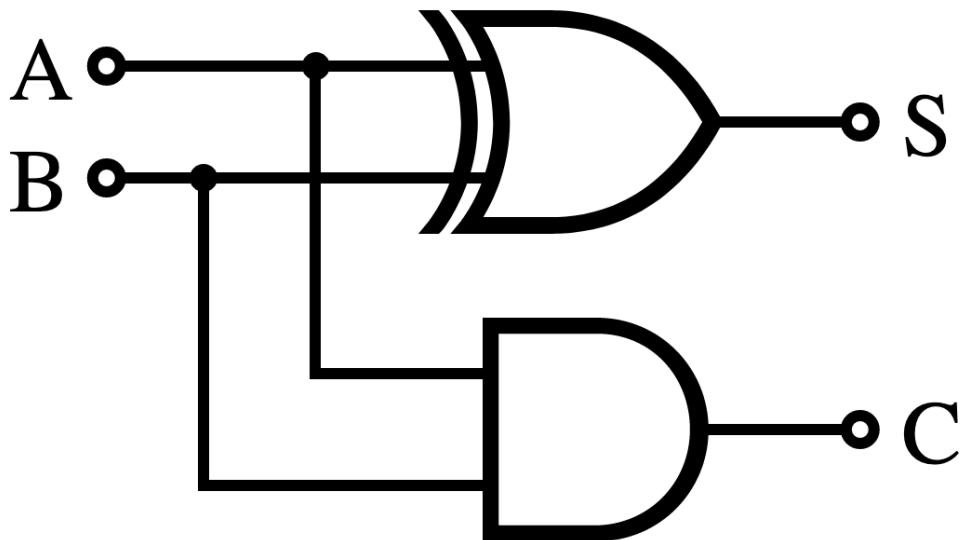
This is how all information is represented in computers. Whether numbers, letters, images or sound, it all exists in the form of binary strings.

Like our standard digital computers, quantum computers are based on this same basic idea. The main difference is that they use *qubits*, a variant of the bit that can be manipulated in quantum ways. In the rest of this textbook, we will explore what qubits are, what they can do and how they do it. In this section, however, we are not talking about quantum at all. So we just use qubits as if they were bits.

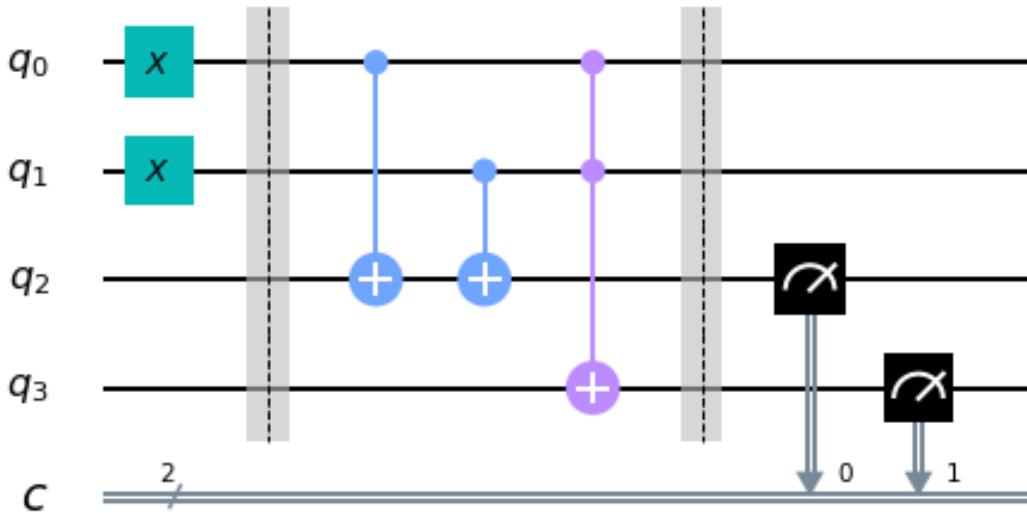
Computation as a diagram

Whether we are using qubits or bits, we need to manipulate them in order to turn the inputs we have into the outputs we need. For the very simplest programs for small numbers of bits, it is useful to represent this process in a diagram known as a *circuit diagram*. These have inputs on the left, outputs on the right, and operations represented by arcane symbols in between. These operations are called 'gates', mostly for historical reasons.

Here's an example of what a circuit looks like for standard, bit-based computers. You aren't expected to understand what it does. It is simply to give you an idea of what these circuits look like.



For quantum computers we use the same basic idea, but we have different conventions for how to represent inputs, outputs, and the symbols used for operations. Here is the quantum circuit that represents the same process as above.



In the rest of this section, we will explain how to build circuits. At the end you'll know how to create the circuit above, what it does and why it is useful.

Your first quantum circuit

In a circuit we typically need to do three jobs: First encode the input, then do some actual computation and finally extract an output. For your first quantum circuit, we'll focus on the last of these jobs. We start by creating a circuit with eight qubits and eight outputs.

```
n = 8
n_q = 8
n_b = 8
qc_output = QuantumCircuit(n_q,n_b)
```



This circuit, which we have called `qc_output`, is created by Qiskit using `QuantumCircuit`. The number `n_q` defines the number of qubits in the circuit. With `n_b` we define the number of output bits we will extract from the circuit at the end.

The extraction of outputs in a quantum circuit is done using an operation called `measure`. Each measurement tells a specific qubit to give an output to a specific output bit. The following code adds a `measure` operation to each of our eight qubits. The qubits and bits are both labelled by the numbers from 0 to 7 (because that's how programmers like to do things). The command `qc.measure(j,j)` adds a measurement to our circuit `qc` that tells qubit `j` to write an output to bit `j`.

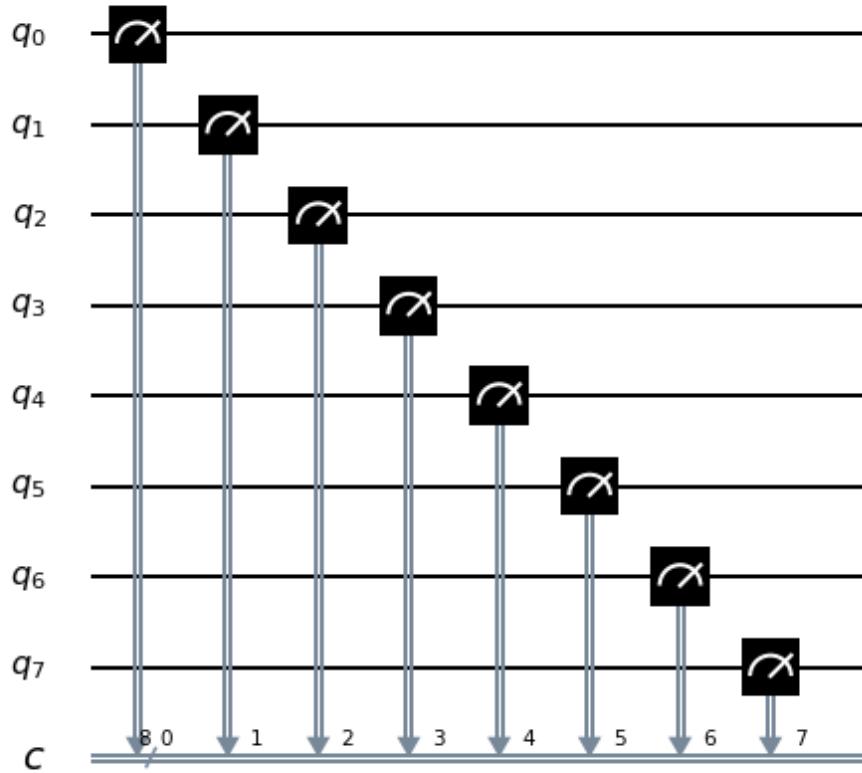
```
for j in range(n):
    qc_output.measure(j,j)
```



Now our circuit has something in, let's take a look at it.

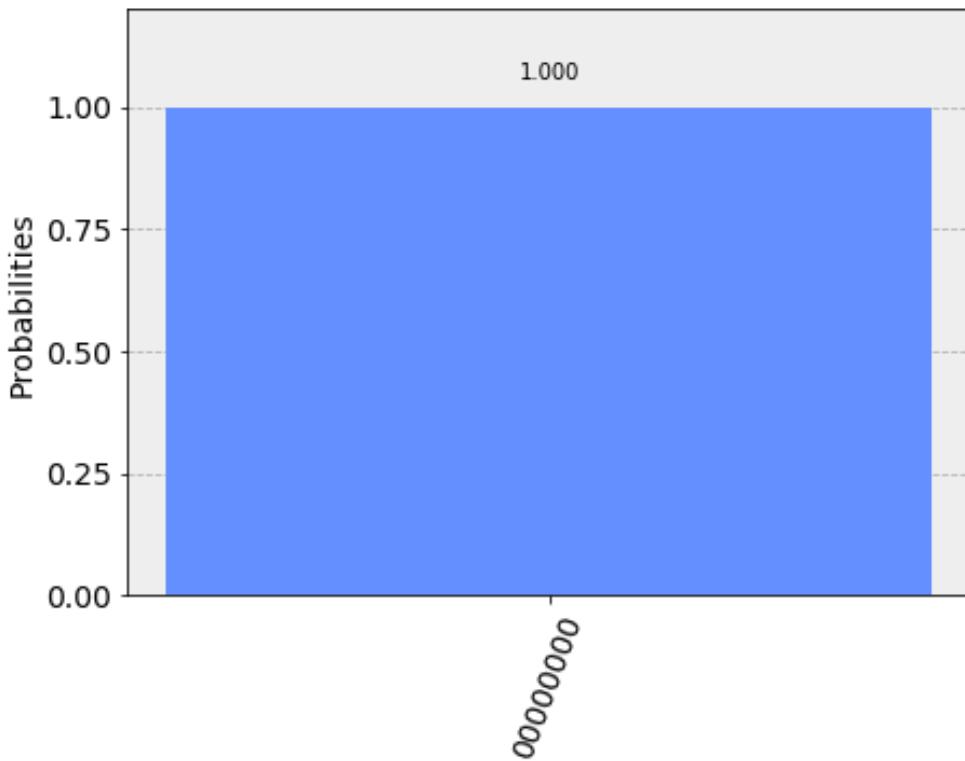
```
qc_output.draw(output='mpl')
```





Qubits are always initialized to give the output $|0\rangle$. Since we don't do anything to our qubits in the circuit above, this is exactly the result we'll get when we measure them. We can see this by running the circuit many times and plotting the results in a histogram. We will find that the result is always 00000000 : a $|0\rangle$ from each qubit.

```
counts = execute(qc_output,Aer.get_backend('qasm_simulator')).result().get_counts()
plot_histogram(counts)
```



The reason for running many times and showing the result as a histogram is because sometimes quantum computers have some randomness in their results. In this case, since we aren't doing anything quantum, we get just the `00000000` result with certainty.

Note that this result comes from a quantum simulator, which is a standard computer calculating what a quantum computer would do. Simulations are only possible for small numbers of qubits, but they are nevertheless a very useful tool when designing your first quantum circuits. To run on a real device you simply need to replace `Aer.get_backend('qasm_simulator')` with the backend object of the device you want to use.

Encoding an input

Now let's look at how to encode a different binary string as an input. For this we need what is known as a NOT gate. This is the most basic operation that you can do in a computer. It simply flips the bit value: `0` becomes `1` and `1` becomes `0`. For qubits, it is an operation called `x` that does the job of the NOT.

Below we create a new circuit dedicated to the job of encoding and call it `qc_encode`. For this we only specify the number of qubits for now.

```
qc_encode = QuantumCircuit(n)
qc_encode.x(7)
```



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

q_0 —————

q_1 —————

q_2 —————

q_3 —————

q_4 —————

q_5 —————

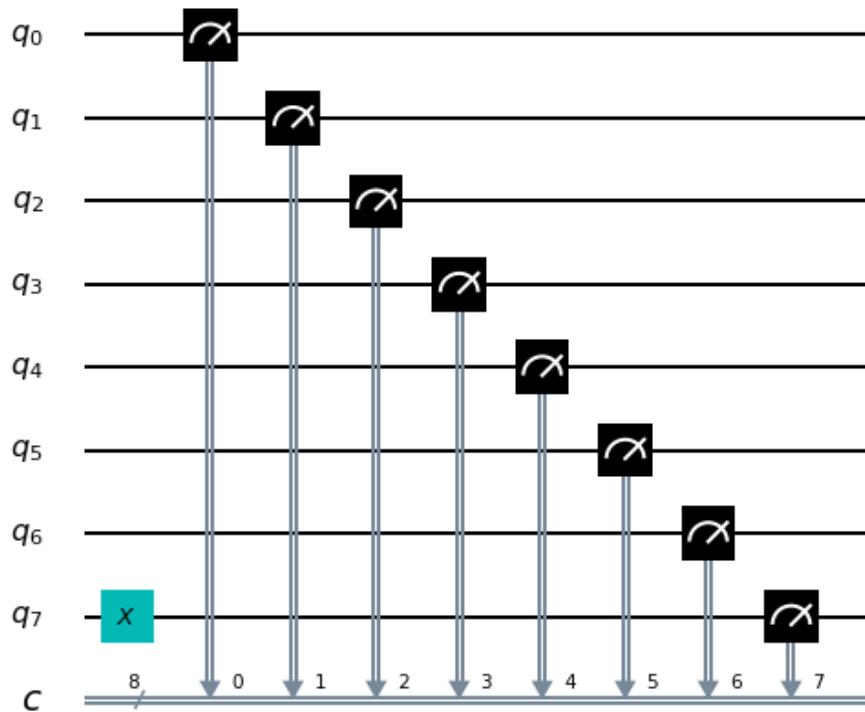
q_6 —————

α_-  ω



Extracting results can be done using the circuit we have from before: `qc_output`. Adding the two circuits using `qc_encode + qc_output` creates a new circuit with everything needed to extract an output added at the end.

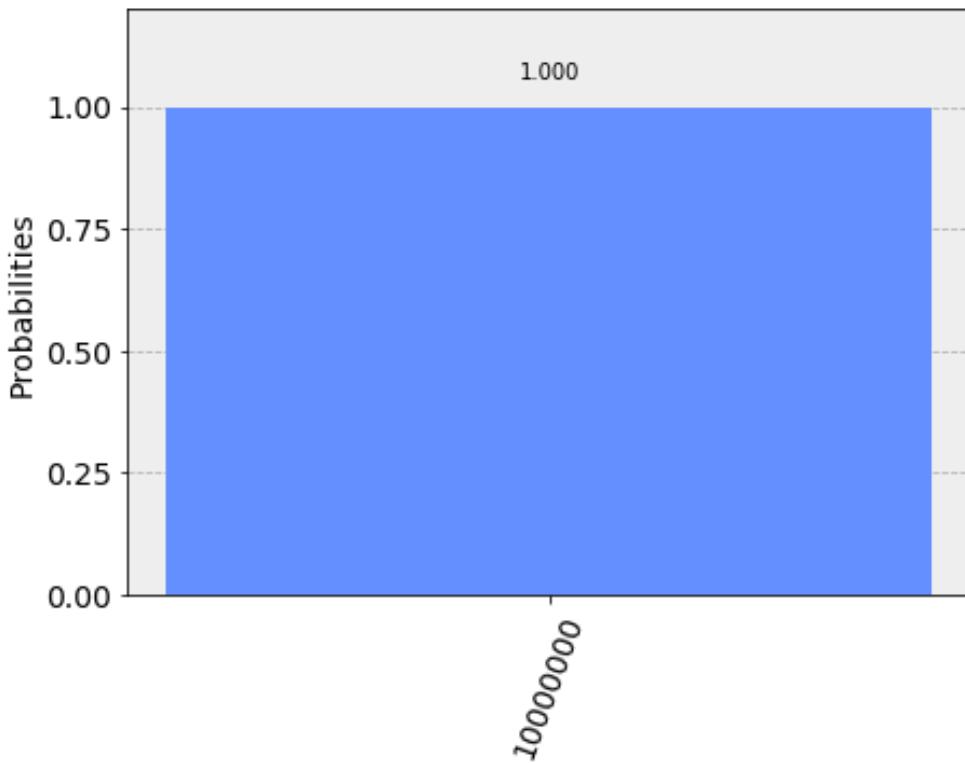
```
qc = qc_encode + qc_output
qc.draw(output='mpl',justify='none')
```



Now we can run the combined circuit and look at the results.

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





Now our computer outputs the string `10000000` instead.

The bit we flipped, which comes from qubit 7, lives on the far left of the string. This is because Qiskit numbers the bits in a string from left to right. If this convention seems odd to you, don't worry. It seems odd to lots of other people too, and some prefer to number their bits the other way around. But this system certainly has its advantages when we are using the bits to represent numbers. Specifically, it means that qubit 7 is telling us about how many 2^7 s we have in our number. So by flipping this bit, we've now written the number 128 in our simple 8-bit computer.

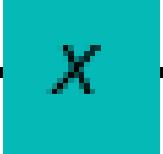
Now try out writing another number for yourself. You could do your age, for example. Just use a search engine to find out what the number looks like in binary (if it includes a '0b', just ignore it), and then add some 0s to the left side if you are younger than 64.

```
qc_encode = QuantumCircuit(n)
qc_encode.x(1)
qc_encode.x(5)

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



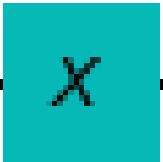
q_0 —————

q_1 —  —

q_2 —————

q_3 —————

q_4 —————

q_5 —  —

q_6 —————

σ_-

Now we know how to encode information in a computer. The next step is to process it: To take an input that we have encoded, and turn it into an output that we need.

Remembering how to add

To look at turning inputs into outputs, we need a problem to solve. Let's do some basic maths. In primary school you will have learned how to take large mathematical problems and break them down into manageable pieces. For example, how would you go about solving the following?

$$\begin{array}{r} 9213 \\ + 1854 \\ = \text{????} \end{array}$$

One way is to do it digit by digit, from right to left. So we start with $3+4$

$$\begin{array}{r} 9213 \\ + 1854 \\ = \text{??7} \end{array}$$

And then $1+5$

$$\begin{array}{r} 9213 \\ + 1854 \\ = \text{??67} \end{array}$$

Then we have $2+8=10$. Since this is a two digit answer, we need to carry the one over to the next column.

$$\begin{array}{r} 9213 \\ + 1854 \\ = ?067 \\ 1 \end{array}$$

Finally we have $9+1+1=11$, and get our answer

$$\begin{array}{r} 9213 \\ + 1854 \\ = 11067 \\ \hline 1 \end{array}$$

This may just be simple addition, but it demonstrates the principles behind all algorithms. Whether the algorithm is designed to solve mathematical problems or process text or images, we always break big tasks down into small and simple steps.

To run on a computer, algorithms need to be compiled down to the smallest and simplest steps possible. To see what these look like, let's do the above addition problem again, but in binary.

$$\begin{array}{r} 1000111111101 \\ + 00011100111110 \\ \hline = ??????????????? \end{array}$$

Note that the second number has a bunch of extra 0s on the left. This just serves to make the two strings the same length.

Our first task is to do the 1+0 for the column on the right. In binary, as in any number system, the answer is 1. We get the same result for the 0+1 of the second column.

$$\begin{array}{r} 1000111111101 \\ + 00011100111110 \\ \hline = ??????????????11 \end{array}$$

Next we have 1+1. As you'll surely be aware, $1+1=2$. In binary, the number 2 is written `10`, and so requires two bits. This means that we need to carry the 1, just as we would for the number 10 in decimal.

$$\begin{array}{r} 1000111111101 \\ + 00011100111110 \\ \hline = ?????????????011 \\ \hline 1 \end{array}$$

The next column now requires us to calculate `1+1+1`. This means adding three numbers together, so things are getting complicated for our computer. But we can still compile it down to simpler

operations, and do it in a way that only ever requires us to add two bits together. For this we can start with just the first two 1s.

$$\begin{array}{r} 1 \\ + 1 \\ = 10 \end{array}$$

Now we need to add this `10` to the final `1`, which can be done using our usual method of going through the columns.

$$\begin{array}{r} 10 \\ + 01 \\ = 11 \end{array}$$

The final answer is `11` (also known as 3).

Now we can get back to the rest of the problem. With the answer of `11`, we have another carry bit.

$$\begin{array}{r} 1000111111101 \\ + 0001110011110 \\ = ??????????1011 \\ \quad \quad \quad \text{11} \end{array}$$

So now we have another $1+1+1$ to do. But we already know how to do that, so it's not a big deal.

In fact, everything left so far is something we already know how to do. This is because, if you break everything down into adding just two bits, there's only four possible things you'll ever need to calculate. Here are the four basic sums (we'll write all the answers with two bits to be consistent).

`0+0 = 00` (in decimal, this is $0+0=0$)

`0+1 = 01` (in decimal, this is $0+1=1$)

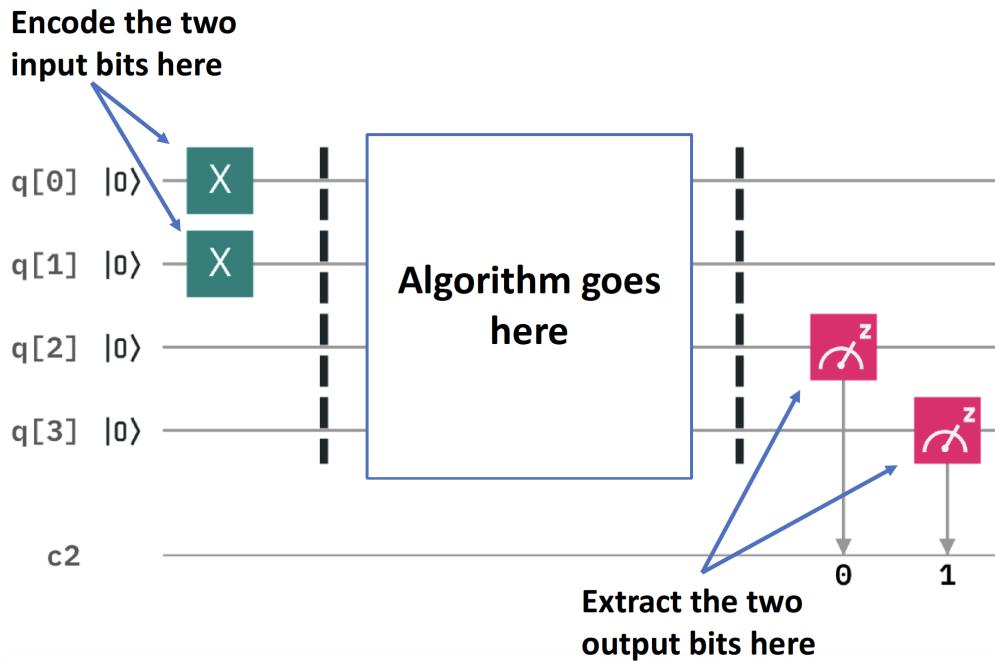
`1+0 = 01` (in decimal, this is $1+0=1$)

`1+1 = 10` (in decimal, this is $1+1=2$)

This is called a *half adder*. If our computer can implement this, and if it can chain many of them together, it can add anything.

Adding with Qiskit

Let's make our own half adder using Qiskit. This will include a part of the circuit that encodes the input, a part that executes the algorithm, and a part that extracts the result. The first part will need to be changed whenever we want to use a new input, but the rest will always remain the same.



The two bits we want to add are encoded in the qubits 0 and 1. The above example encodes a 1 in both these qubits, and so it seeks to find the solution of $1+1$. The result will be a string of two bits, which we will read out from the qubits 2 and 3. All that remains is to fill in the actual program, which lives in the blank space in the middle.

The dashed lines in the image are just to distinguish the different parts of the circuit (although they can have more interesting uses too). They are made by using the `barrier` command.

The basic operations of computing are known as logic gates. We've already used the NOT gate, but this is not enough to make our half adder. We could only use it to manually write out the answers. But since we want the computer to do the actual computing for us, we'll need some more powerful gates.

To see what we need, let's take another look at what our half adder needs to do.

$$0+0 = 00$$

$$0+1 = 01$$

$$1+0 = 01$$

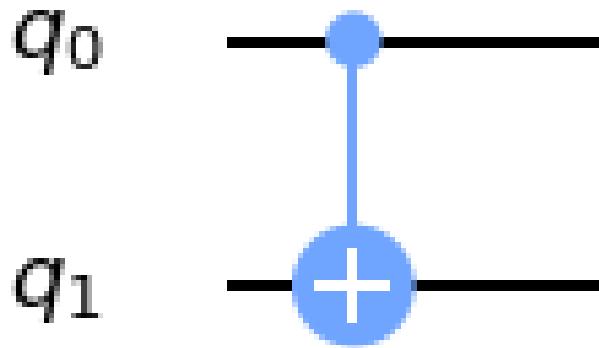
$$1+1 = 10$$

The rightmost bit in all four of these answers is completely determined by whether the two bits we are adding are the same or different. So for $0+0$ and $1+1$, where the two bits are equal, the rightmost bit of the answer comes out 0 . For $0+1$ and $1+0$, where we are adding different bit values, the rightmost bit is 1 .

To get this part of our solution correct, we need something that can figure out whether two bits are different or not. Traditionally, in the study of digital computation, this is called an XOR gate.

In quantum computers, the job of the XOR gate is done by the controlled-NOT gate. Since that's quite a long name, we usually just call it the CNOT. In Qiskit its name is `cx`, which is even shorter. In circuit diagrams it is drawn as in the image below.

```
qc_cnot = QuantumCircuit(2)
qc_cnot.cx(0,1)
qc_cnot.draw(output='mpl')
```



This is applied to a pair of qubits. One acts as the control qubit (this is the one with the little dot). The other acts as the *target qubit* (with the big circle).

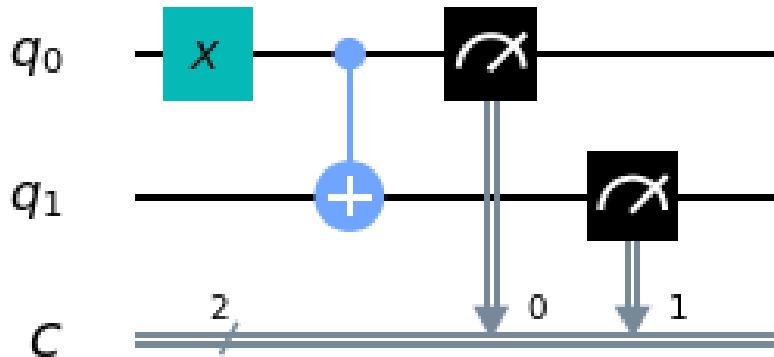
There are multiple ways to explain the effect of the CNOT. One is to say that it looks at its two input bits to see whether they are the same or different. Then it writes over the target qubit with the answer. The target becomes 0 if they are the same, and 1 if they are different.

Another way of explaining the CNOT is to say that it does a NOT on the target if the control is 1 , and does nothing otherwise. This explanation is just as valid as the previous one (in fact, it's the one

that gives the gate its name).

Try the CNOT out for yourself by trying each of the possible inputs. For example, here's a circuit that tests the CNOT with the input `01`.

```
qc = QuantumCircuit(2,2)
qc.x(0)
qc.cx(0,1)
qc.measure(0,0)
qc.measure(1,1)
qc.draw(output='mpl')
```



If you execute this circuit, you'll find that the output is `11`. We can think of this happening because of either of the following reasons.

- The CNOT calculates whether the input values are different and finds that they are, which means that it wants to output `1`. It does this by writing over the state of qubit 1 (which, remember, is on the left of the bit string), turning `01` into `11`.
- The CNOT sees that qubit 0 is in state `1`, and so applies a NOT to qubit 1. This flips the `0` of qubit 1 into a `1`, and so turns `01` into `11`.

For our half adder, we don't want to overwrite one of our inputs. Instead, we want to write the result on a different pair of qubits. For this we can use two CNOTs.

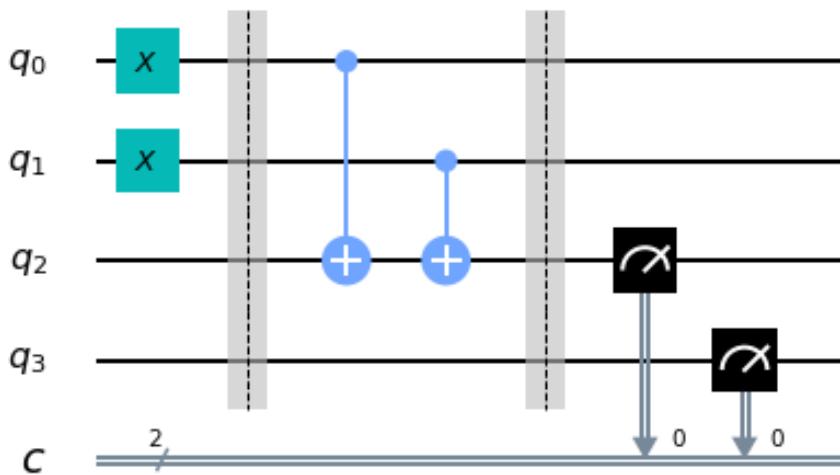
```
qc_ha = QuantumCircuit(4,2)
# encode inputs in qubits 0 and 1
qc_ha.x(0) # For a=0, remove this line. For a=1, leave it.
qc_ha.x(1) # For b=0, remove this line. For b=1, leave it.
```

```

qc_ha.barrier()
# use cnots to write the XOR of the inputs on qubit 2
qc_ha.cx(0,2)
qc_ha.cx(1,2)
qc_ha.barrier()
# extract outputs
qc_ha.measure(2,0) # extract XOR value
qc_ha.measure(3,0)

qc_ha.draw(output='mpl')

```



We are now halfway to a fully working half adder. We just have the other bit of the output left to do: the one that will live on qubit 4.

If you look again at the four possible sums, you'll notice that there is only one case for which this is 1 instead of 0: $1+1 = 10$. It happens only when both the bits we are adding are 1.

To calculate this part of the output, we could just get our computer to look at whether both of the inputs are 1. If they are—and only if they are—we need to do a NOT gate on qubit 4. That will flip it to the required value of 1 for this case only, giving us the output we need.

For this we need a new gate: like a CNOT, but controlled on two qubits instead of just one. This will perform a NOT on the target qubit only when both controls are in state 1. This new gate is called the *Toffoli*. For those of you who are familiar with Boolean logic gates, it is basically an AND gate.

In Qiskit, the Toffoli is represented with the `ccx` command.

```

qc_ha = QuantumCircuit(4,2)
# encode inputs in qubits 0 and 1

```

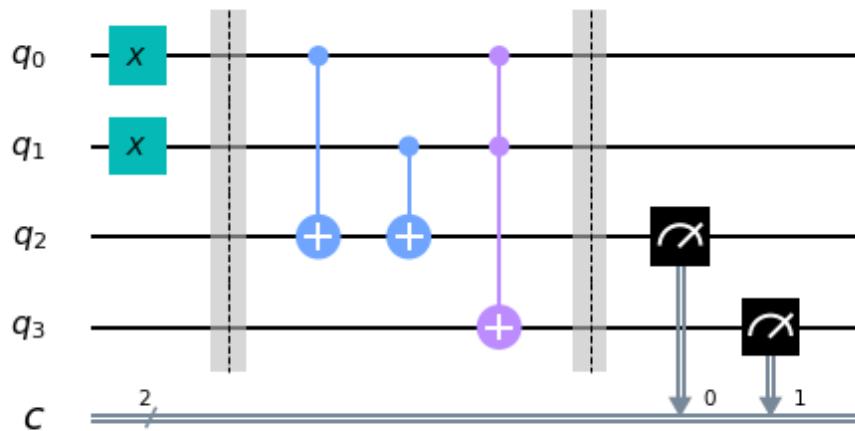


```

qc_ha.x(0) # For a=0, remove this line. For a=1, leave it.
qc_ha.x(1) # For b=0, remove this line. For b=1, leave it.
qc_ha.barrier()
# use cnots to write the XOR of the inputs on qubit 2
qc_ha.cx(0,2)
qc_ha.cx(1,2)
# use ccx to write the AND of the inputs on qubit 3
qc_ha.ccx(0,1,3)
qc_ha.barrier()
# extract outputs
qc_ha.measure(2,0) # extract XOR value
qc_ha.measure(3,1) # extract AND value

qc_ha.draw(output='mpl')

```

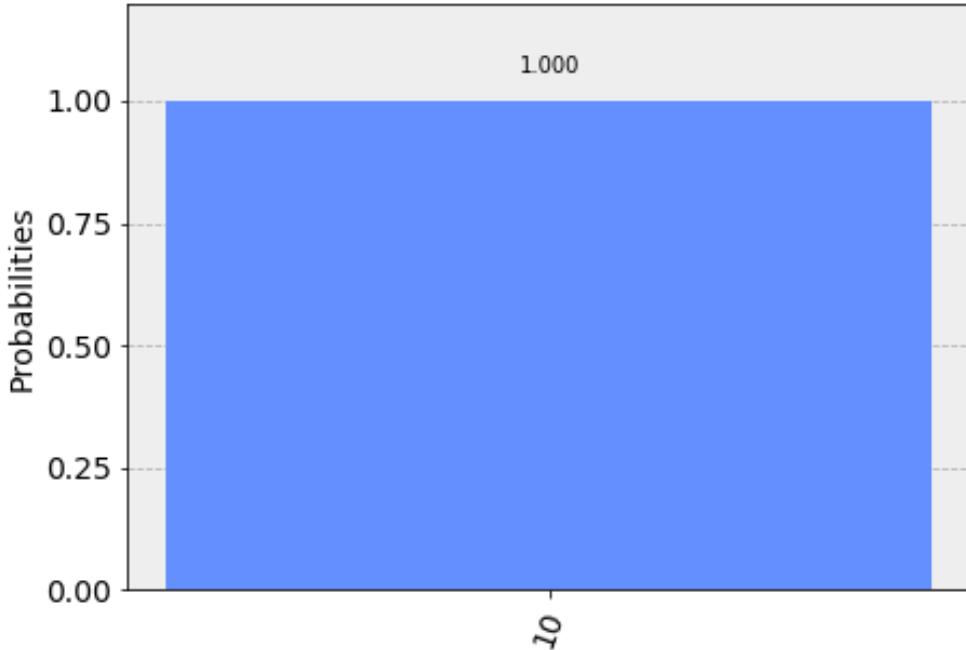


In this example we are calculating $1+1$, because the two input bits are both 1 . Let's see what we get.

```

counts = execute(qc_ha,Aer.get_backend('qasm_simulator')).result().get_counts()
plot_histogram(counts)

```



The result is `10`, which is the binary representation of the number 2. We have built a computer that can solve the famous mathematical problem of $1+1$!

Now you can try it out with the other three possible inputs, and show that our algorithm gives the right results for those too.

The half adder contains everything you need for addition. With the NOT, CNOT and Toffoli gates, we can create programs that add any set of numbers of any size.

These three gates are enough to do everything else in computing too. In fact, we can even do without the CNOT, and the NOT gate is only really needed to create bits with value `1`. The Toffoli gate is essentially the atom of mathematics. It is simplest element into which every other problem-solving technique can be compiled.

As we'll see, in quantum computing we split the atom.

The Unique Properties of Qubits

```
from qiskit import *
from qiskit.visualization import plot_histogram
```



You now know something about bits, and about how our familiar digital computers work. All the complex variables, objects and data structures used in modern software are basically all just big piles of bits. Those of us who work on quantum computing call these *classical variables*. The computers that use them, like the one you are using to read this article, we call *classical computers*.

In quantum computers, our basic variable is the *qubit*: a quantum variant of the bit. These are quantum objects, obeying the laws of quantum mechanics. Unlike any classical variable, these cannot be represented by some number of classical bits. They are fundamentally different.

The purpose of this section is to give you your first taste of what a qubit is, and how they are unique. We'll do this in a way that requires essentially no math. This means leaving terms like 'superposition' and 'entanglement' until future sections, since it is difficult to properly convey their meaning without pointing at an equation.

Instead, we will use another well-known feature of quantum mechanics: the uncertainty principle.

Heisenberg's uncertainty principle

The most common formulation of the uncertainty principle refers to the position and momentum of a particle: the more precisely its position is defined, the more uncertainty there is in its momentum, and vice-versa.



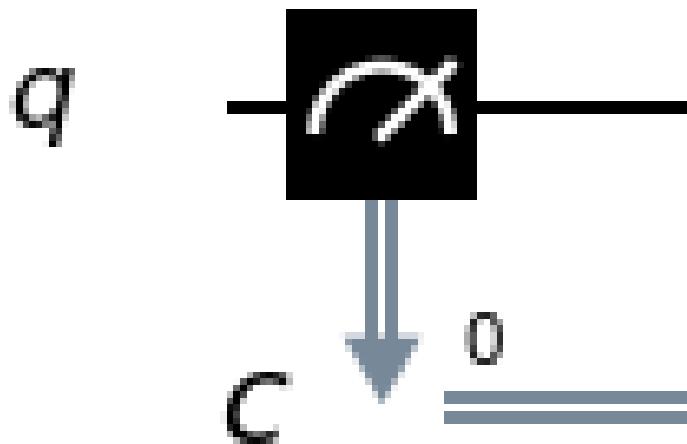
PERMANENT LINK TO THIS COMIC: [HTTPS://XKCD.COM/824/](https://xkcd.com/824/)

This is a common feature of quantum objects, though it need not always refer to position and momentum. There are many possible sets of parameters for different quantum objects, where certain knowledge of one means that our observations of the others will be completely random.

To see how the uncertainty principle affects qubits, we need to look at measurement. As we saw in the last section, this is the method by which we extract a bit from a qubit.

```
measure_z = QuantumCircuit(1,1)
measure_z.measure(0,0)

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



On the [Circuit Composer](#), the same operation looks like this.



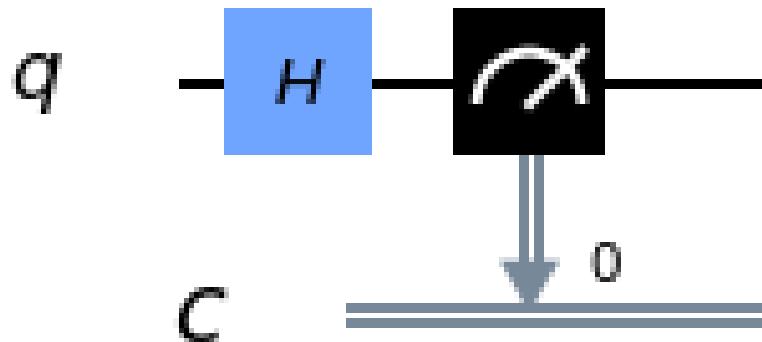
This version has a small 'z' written in the box that represents the operation. This hints at the fact that this kind of measurement is not the only one. In fact, it is only one of an infinite number of

possible ways to extract a bit from a qubit. Specifically, it is known as a *z measurement*.

Another commonly used measurement is the *x measurement*. It can be performed using the following sequence of gates.

```
measure_x = QuantumCircuit(1,1)
measure_x.h(0)
measure_x.measure(0,0)

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



Later chapters will explain why this sequence of operations performs a new kind of measurement. For now, you'll need to trust us.

Like the position and momentum of a quantum particle, the z and x measurements of a qubit are governed by the uncertainty principle. Below we'll look at results from a few different circuits to see this effect in action.

Results for an empty circuit

The easiest way to see an example is to take a freshly initialized qubit.

```
qc_0 = QuantumCircuit(1)
qc_0.draw(output='mpl')
```



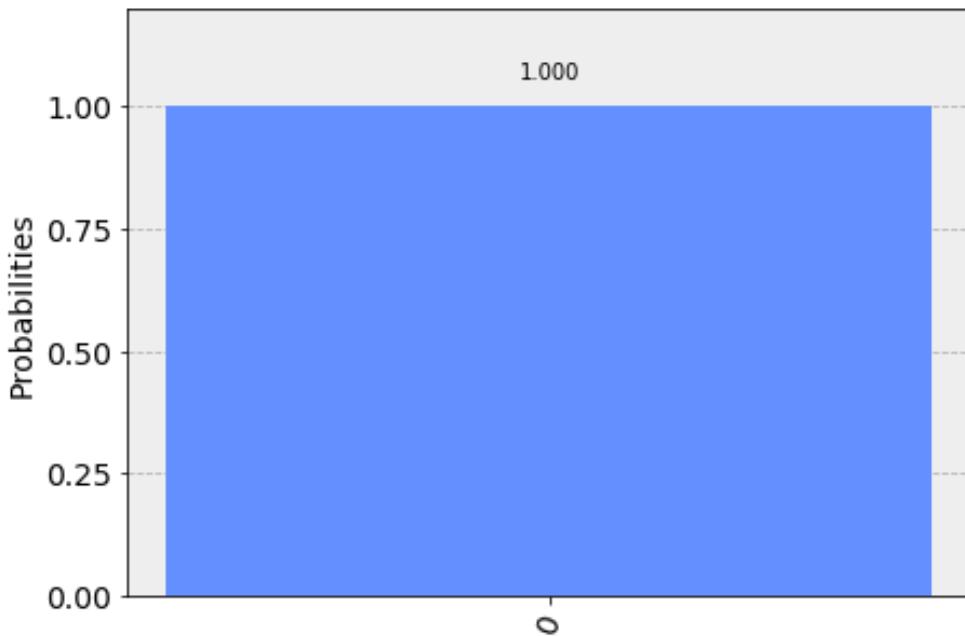
Qubits are always initialized such that they are certain to give the result $|0\rangle$ for a z measurement. The resulting histogram will therefore simply have a single column, showing the 100% probability of getting a $|0\rangle$.

```
qc = qc_0 + measure_z

print('Results for z measurement:')
counts = execute(qc,Aer.get_backend('qasm_simulator')).result().get_counts()
plot_histogram(counts)
```



Results for z measurement:



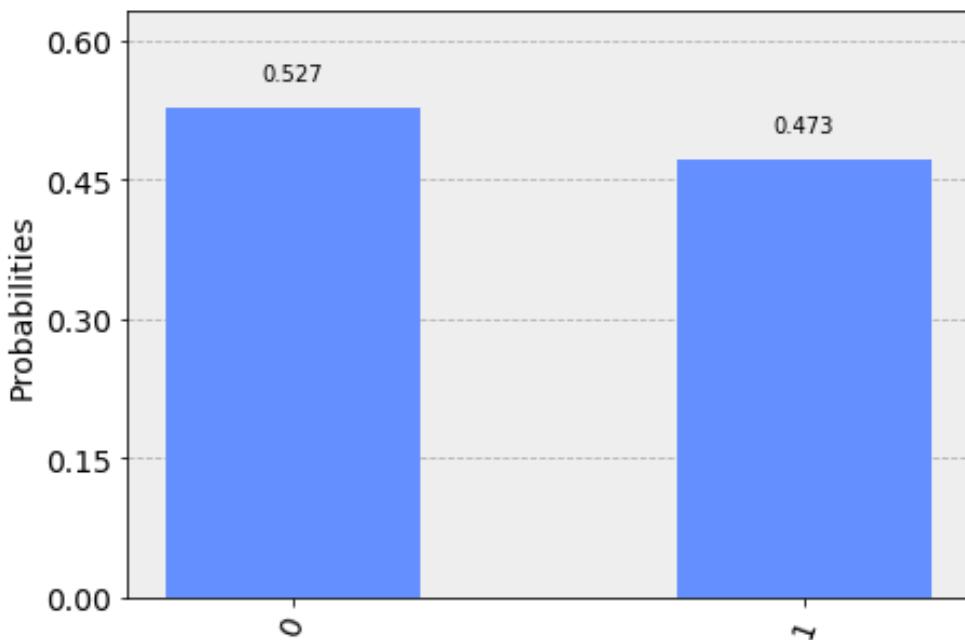
If we instead do an x measurement, the results will be completely random.

```
qc = qc_0 + measure_x

print('Results for x measurement:')
counts = execute(qc,Aer.get_backend('qasm_simulator')).result().get_counts()
plot_histogram(counts)
```



Results for x measurement:



Note that the reason why the results are not split exactly 50/50 here is because we take samples by repeating the circuit a finite number of times, and so there will always be statistical noise. In this

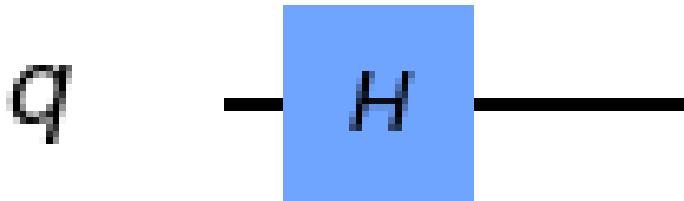
case, the default of `shots=1024` was used.

Results for a single Hadamard

Now we'll try a different circuit. This has a single gate called a Hadamard, which we will learn more about in future sections.

```
qc_plus = QuantumCircuit(1)
qc_plus.h(0)

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



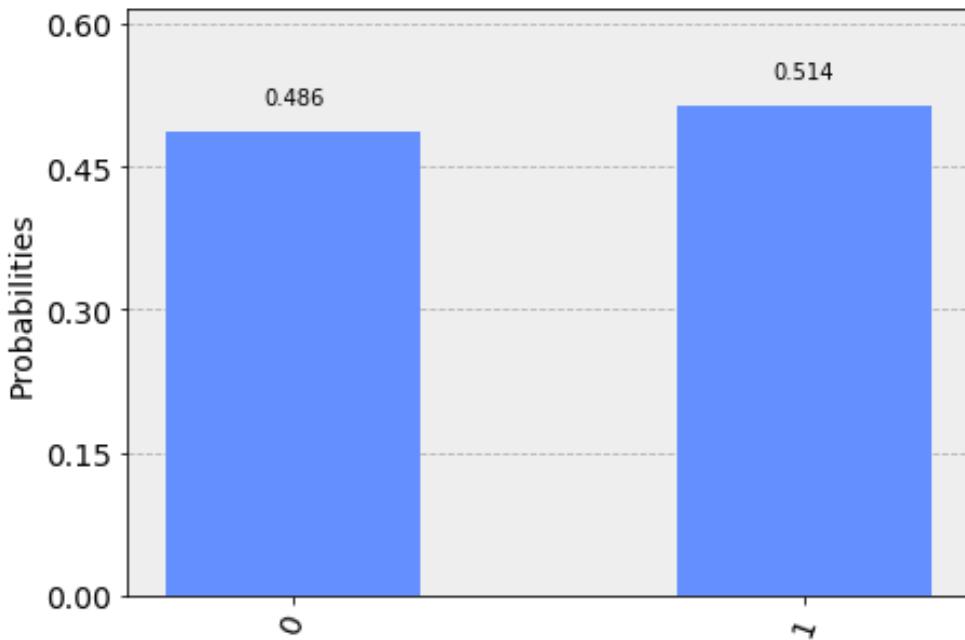
To see what effect it has, let's first try the z measurement.

```
qc = qc_plus + measure_z

qc.draw()

print('Results for z measurement:')
counts = execute(qc,Aer.get_backend('qasm_simulator')).result().get_counts()
plot_histogram(counts)
```

```
Results for z measurement:
```



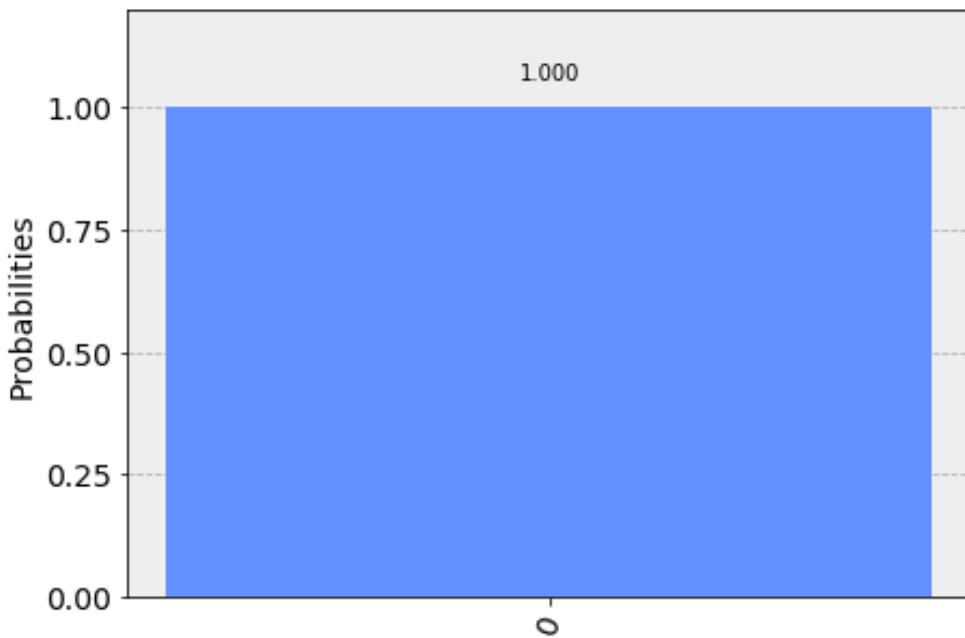
Here we see that it is the results of the z measurement that are random for this circuit.

Now let's see what happens for an x measurement.

```
qc = qc_plus + measure_x

print('Results for x measurement:')
counts = execute(qc,Aer.get_backend('qasm_simulator')).result().get_counts()
plot_histogram(counts)
```

Results for x measurement:



For the x measurement, it is certain that the output for this circuit is θ . The results here are therefore very different to what we saw for the empty circuit. The Hadamard has lead to an entirely opposite set of outcomes.

Results for a y rotation

Using other circuits we can manipulate the results in different ways. Here is an example with an `ry` gate.

```
qc_y = QuantumCircuit(1)
qc_y.ry(-3.14159/4, 0)

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

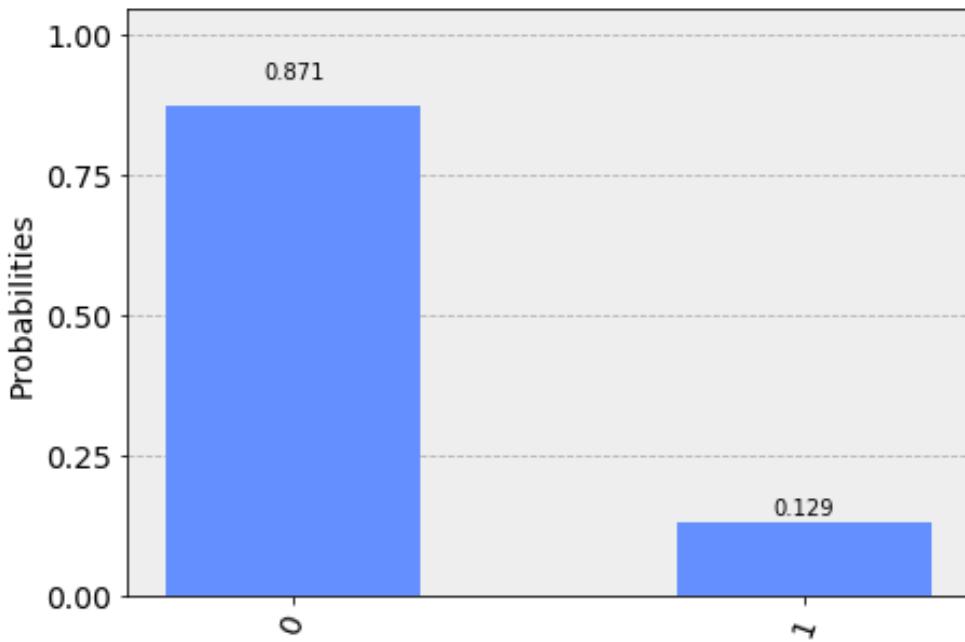


We will learn more about `ry` in future sections. For now, just notice the effect it has for the z and x measurements.

```
qc = qc_y + measure_z

print('Results for z measurement:')
counts = execute(qc,Aer.get_backend('qasm_simulator')).result().get_counts()
plot_histogram(counts)
```

```
Results for z measurement:
```

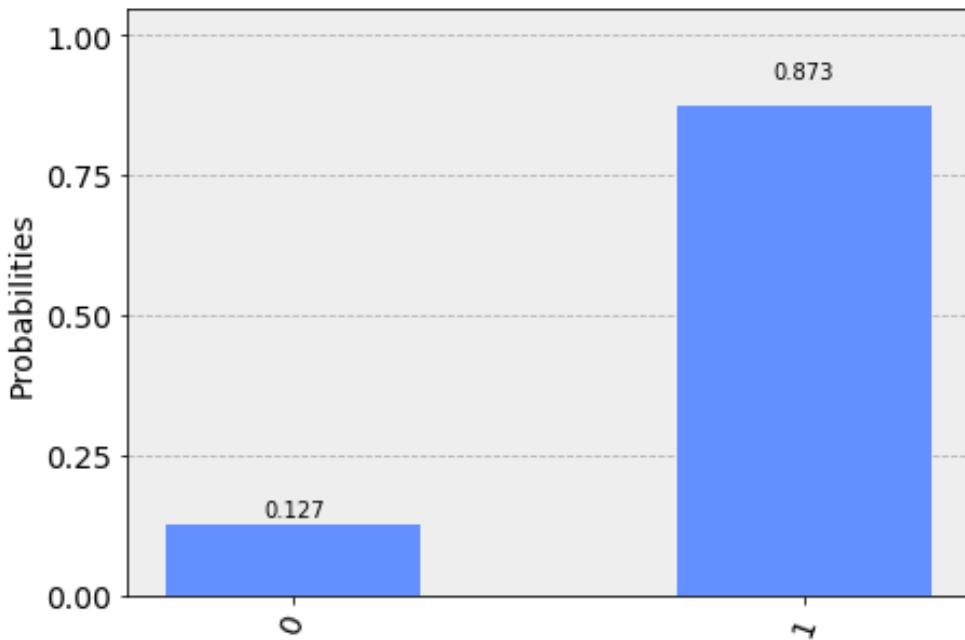


Here we have a case that we have not seen before. The z measurement is most likely to output 0, but it is not completely certain. A similar effect is seen below for the x measurement: it is most likely, but not certain, to output 1.

```
qc = qc_y + measure_x

print('\nResults for x measurement:')
counts = execute(qc,Aer.get_backend('qasm_simulator')).result().get_counts()
plot_histogram(counts)
```

Results for x measurement:



These results hint at an important principle: Qubits have a limited amount of certainty that they can hold. This ensures that, despite the different ways we can extract outputs from a qubit, it can only be used to store a single bit of information. In the case of the blank circuit, this certainty was dedicated entirely to the outcomes of z measurements. For the circuit with a single Hadamard, it was dedicated entirely to x measurements. In this case, it is shared between the two.

Einstein vs. Bell

We have now played with some of the features of qubits, but we haven't done anything that couldn't be reproduced by a few bits and a random number generator. You can therefore be forgiven for thinking that quantum variables are just classical variables with some randomness bundled in.

This is essentially the claim made by Einstein, Podolsky and Rosen back in 1935. They objected to the uncertainty seen in quantum mechanics, and thought it meant that the theory was incomplete. They thought that a qubit should always know what output it would give for both kinds of measurement, and that it only seems random because some information is hidden from us. As Einstein said: God does not play dice with the universe.

No one spoke of qubits back then, and people hardly spoke of computers. But if we translate their arguments into modern language, they essentially claimed that qubits can indeed be described by some form of classical variable. They didn't know how to do it, but they were sure it could be done. Then quantum mechanics could be replaced by a much nicer and more sensible theory.

It took until 1964 to show that they were wrong. J. S. Bell proved that quantum variables behaved in a way that was fundamentally unique. Since then, many new ways have been found to prove this, and extensive experiments have been done to show that this is exactly the way the universe works. We'll now consider a simple demonstration, using a variant of *Hardy's paradox*.

For this we need two qubits, set up in such a way that their results are correlated. Specifically, we want to set them up such that we see the following properties.

1. If z measurements are made on both qubits, they never both output 0.
2. If an x measurement of one qubit outputs 1, a z measurement of the other will output 0.

If we have qubits that satisfy these properties, what can we infer about the remaining case: an x measurement of both?

For example, let's think about the case where both qubits output 1 for an x measurement. By applying property 2 we can deduce what the result would have been if we had made z

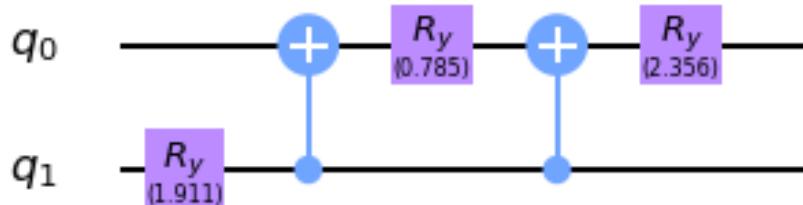
measurements instead: We would have gotten an output of 0 for both. However, this result is impossible according to property 1. We can therefore conclude that an output of 1 for x measurements of both qubits must also be impossible.

The paragraph you just read contains all the math in this section. Don't feel bad if you need to read it a couple more times!

Now let's see what actually happens. Here is a circuit, composed of gates you will learn about in later sections. It prepares a pair of qubits that will satisfy the above properties.

```
qc_hardy = QuantumCircuit(2)
qc_hardy.ry(1.911,1)
qc_hardy.cx(1,0)
qc_hardy.ry(0.785,0)
qc_hardy.cx(1,0)
qc_hardy.ry(2.356,0)

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



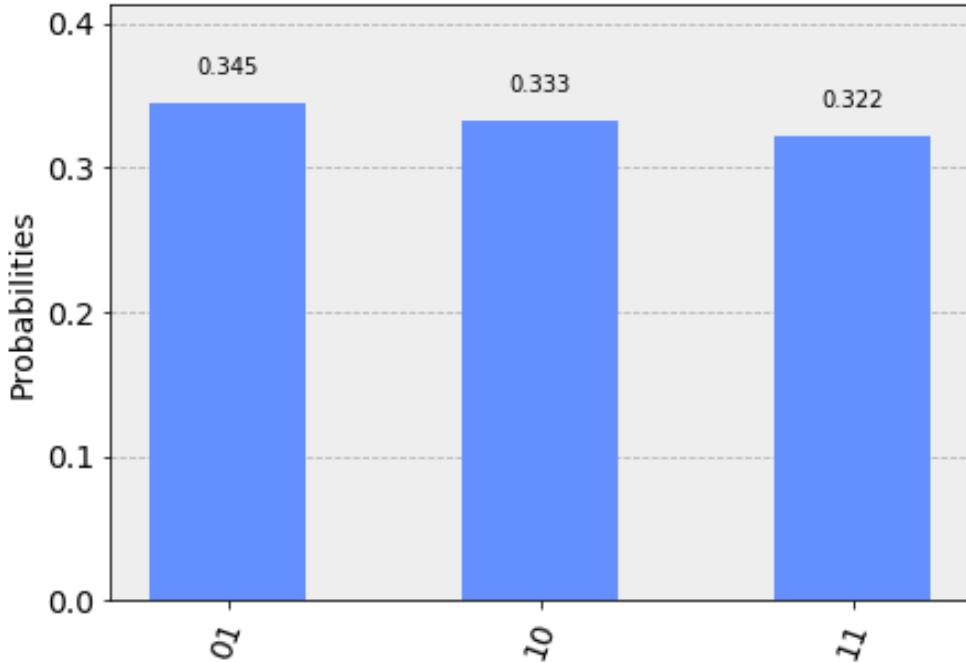
Let's see it in action. First a z measurement of both qubits.

```
measurements = QuantumCircuit(2,2)
# z measurement on both qubits
measurements.measure(0,0)
measurements.measure(1,1)

qc = qc_hardy + measurements

print('\nResults for two z measurements:')
counts = execute(qc,Aer.get_backend('qasm_simulator')).result().get_counts()
plot_histogram(counts)
```

Results for two z measurements:



The probability of $|00\rangle$ is zero, and so these qubits do indeed satisfy property 1.

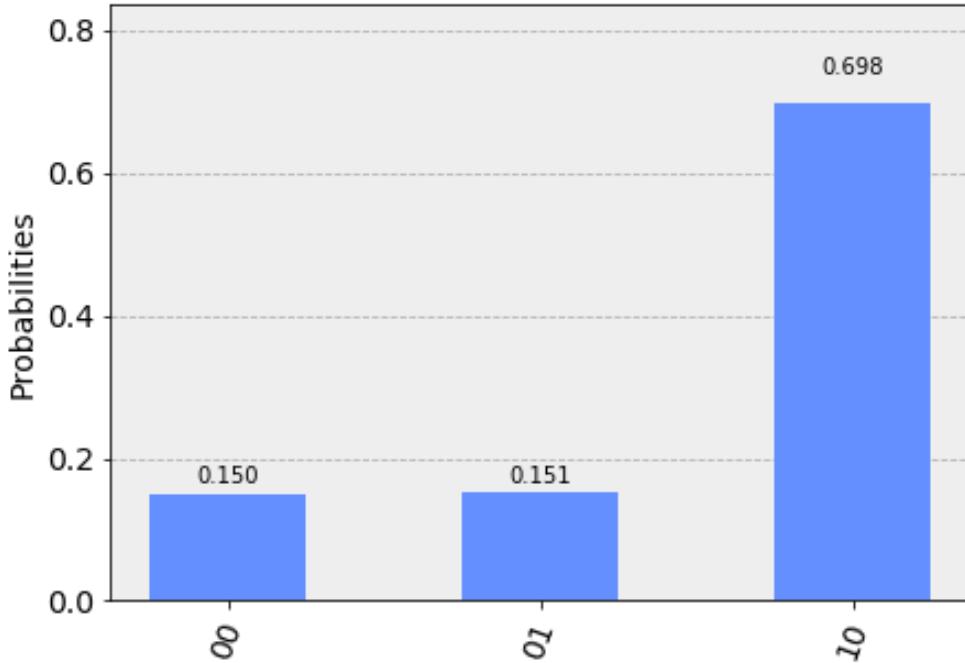
Next, let's see the results of an x measurement of one and a z measurement of the other.

```
measurements = QuantumCircuit(2,2)
# x measurement on qubit 0
measurements.h(0)
measurements.measure(0,0)
# z measurement on qubit 1
measurements.measure(1,1)

qc = qc_hardy + measurements

print('\nResults for two x measurement on qubit 0 and z measurement on qubit 1:')
counts = execute(qc,Aer.get_backend('qasm_simulator')).result().get_counts()
plot_histogram(counts)
```

Results for two x measurement on qubit 0 and z measurement on qubit 1:



The probability of $|11\rangle$ is zero. You'll see the same if you swap around the measurements. These qubits therefore also satisfy property 2.

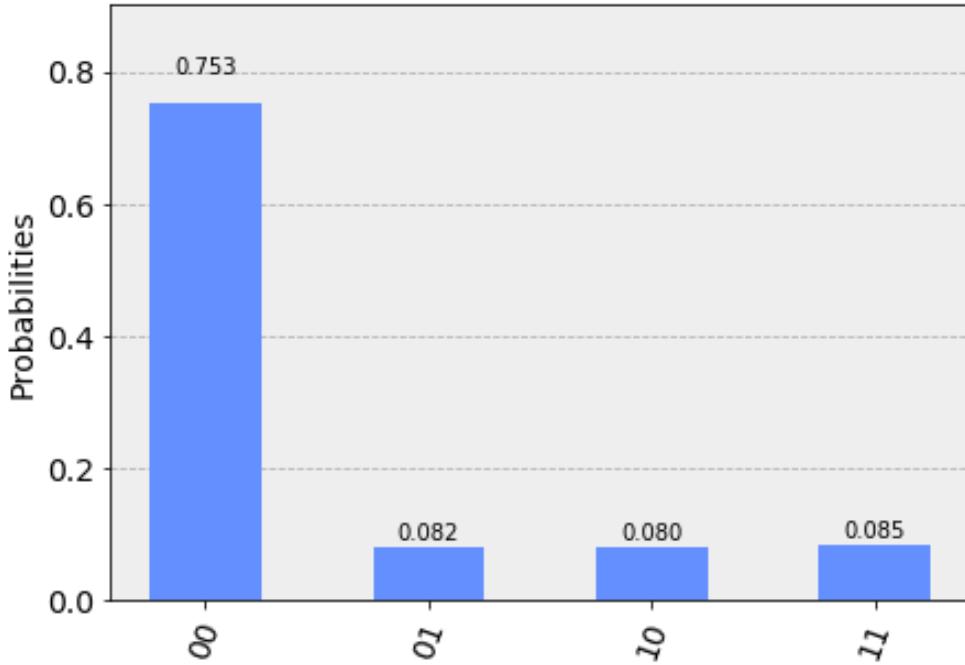
Finally, let's look at an x measurement of both.

```
measurements = QuantumCircuit(2,2)
measurements.h(0)
measurements.measure(0,0)
measurements.h(1)
measurements.measure(1,1)

qc = qc_hardy + measurements

print('\nResults for two x measurement on both qubits:')
counts = execute(qc,Aer.get_backend('qasm_simulator')).result().get_counts()
plot_histogram(counts)
```

Results for two x measurement on both qubits:



We reasoned that, given properties 1 and 2, it would be impossible to get the output `11`. From the results above, we see that our reasoning was not correct: one in every dozen results will have this 'impossible' result.

So where did we go wrong? Our mistake was in the following piece of reasoning.

By applying property 2 we can deduce what the result would have been if we had made z measurements instead

We used our knowledge of the x outputs to work out what the z outputs were. Once we'd done that, we assumed that we were certain about the value of both. More certain than the uncertainty principle allows us to be. And so we were wrong.

Our logic would be completely valid if we weren't reasoning about quantum objects. If it was some non-quantum variable, that we initialized by some random process, the x and z outputs would indeed both be well defined. They would just be based on some pre-determined list of random numbers in our computer, or generated by some deterministic process. Then there would be no reason why we shouldn't use one to deduce the value of the other, and our reasoning would be perfectly valid. The restriction it predicts would apply, and it would be impossible for both x measurements to output `1`.

But our qubits behave differently. The uncertainty of quantum mechanics allows qubits to dodge restrictions placed on classical variables. It allows them to do things that would otherwise be impossible. Indeed, this is the main thing to take away from this section:

A physical system in a definite state can still behave randomly.

This is the first of the key principles of the quantum world. It needs to become your new intuition, as it is what makes quantum systems different to classical systems. It's what makes quantum computers able to outperform classical computers. It leads to effects that allow programs made with quantum variables to solve problems in ways that those with normal variables cannot. But just because qubits don't follow the same logic as normal computers, it doesn't mean they defy logic entirely. They obey the definite rules laid out by quantum mechanics.

If you'd like to learn these rules, we'll use the remainder of this chapter to guide you through them. We'll also show you how to express them using math. This will provide a foundation for later chapters, in which we'll explain various quantum algorithms and techniques.

Writing Down Qubit States

```
from qiskit import *
```



In the previous chapter we saw that there are multiple ways to extract an output from a qubit. The two methods we've used so far are the z and x measurements.

```
# z measurement of qubit 0
measure_z = QuantumCircuit(1,1)
measure_z.measure(0,0);

# x measurement of qubit 0
measure_x = QuantumCircuit(1,1)
measure_x.h(0)
measure_x.measure(0,0);
```



Sometimes these measurements give results with certainty. Sometimes their outputs are random. This all depends on which of the infinitely many possible states our qubit is in. We therefore need a way to write down these states and figure out what outputs they'll give. For this we need some notation, and we need some math.

The z basis

If you do nothing in a circuit but a measurement, you are certain to get the outcome $|0\rangle$. This is because the qubits always start in a particular state, whose defining property is that it is certain to output a $|0\rangle$ for a z measurement.

We need a name for this state. Let's be unimaginative and call it $|0\rangle$. Similarly, there exists a qubit state that is certain to output a $|1\rangle$. We'll call this $|1\rangle$.

These two states are completely mutually exclusive. Either the qubit definitely outputs a $|0\rangle$, or it definitely outputs a $|1\rangle$. There is no overlap.

One way to represent this with mathematics is to use two orthogonal vectors.

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

This is a lot of notation to take in all at once. First let's unpack the weird $|$ and \rangle . Their job is essentially just to remind us that we are talking about the vectors that represent qubit states labelled 0 and $|1\rangle$. This helps us distinguish them from things like the bit values 0 and 1 or the numbers 0 and 1. It is part of the bra-ket notation, introduced by Dirac.

If you are not familiar with vectors, you can essentially just think of them as lists of numbers which we manipulate using certain rules. If you are familiar with vectors from your high school physics classes, you'll know that these rules make vectors well-suited for describing quantities with a magnitude and a direction. For example, velocity of an object is described perfectly with a vector. However, the way we use vectors for quantum states is slightly different to this. So don't hold on too hard to your previous intuition. It's time to do something new!

In the example above, we wrote the vector as a vertical list of numbers. We call these *column vectors*. In Dirac notation, they are also called *kets*.

Horizontal lists are called *row vectors*. In Dirac notation they are *bras*. They are represented with a $($ and a $|$.

$$\langle 0 | = (1 \ 0) \langle 1 | = (0 \ 1).$$

The rules on how to manipulate vectors define what it means to add or multiply them. For example, to add two vectors we need them to be the same type (either both column vectors, or both row vectors) and the same length. Then we add each element in one list to the corresponding element in the other. For a couple of arbitrary vectors that we'll call a and b, this works as follows.

$$\begin{pmatrix} a_0 \\ a_1 \end{pmatrix} + \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = \begin{pmatrix} a_0 + b_0 \\ a_1 + b_1 \end{pmatrix}.$$

To multiple a vector by a number, we simply multiply every element in the list by that number:

$$x \times \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} = \begin{pmatrix} x \times a_0 \\ x \times a_1 \end{pmatrix}$$

Multiplying a vector with another vector is a bit more tricky, since there are multiple ways we can do it. One is called the 'inner product', and works as follows.

$$\begin{pmatrix} a_0 & a_1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} = a_0 b_0 + a_1 b_1.$$

Note that the right hand side of this equation contains only normal numbers being multiplied and added in a normal way. The inner product of two vectors therefore yields just a number. As we'll see, we can interpret this as a measure of how similar the vectors are.

The inner product requires the first vector to be a bra and the second to be a ket. In fact, this is where their names come from. Dirac wanted to write the inner product as something like $\langle a | b \rangle$, which looks like the names of the vectors enclosed in brackets. Then he worked backwards to split the bracket into a *bra* and a *ket*.

If you try out the inner product on the vectors we already know, you'll find

$$\langle 0 | 0 \rangle = \langle 1 | 1 \rangle = 1, \langle 0 | 1 \rangle = \langle 1 | 0 \rangle = 0.$$

Here we are using a concise way of writing the inner products where, for example, $\langle 0 | 1 \rangle$ is the inner product of $\langle 0 |$ with $| 1 \rangle$. The top line shows us that the inner product of these states with themselves always gives a 1. When done with two orthogonal states, as on the bottom line, we get the outcome 0. These two properties will come in handy later on.

The x basis - part 1

So far we've looked at states for which the z measurement has a certain outcome. But there are also states for which the outcome of a z measurement is equally likely to be 0 or 1. What might these look like in the language of vectors?

A good place to start would be something like $| 0 \rangle + | 1 \rangle$, since this includes both $| 0 \rangle$ and $| 1 \rangle$ with no particular bias towards either. But let's hedge our bets a little and multiply it by some number x.

$$x (| 0 \rangle + | 1 \rangle) = \begin{pmatrix} xx \\ xx \end{pmatrix}$$

We can choose the value of x to make sure that the state plays nicely in our calculations. For example, think about the inner product,

$$(x \ x) \times \begin{pmatrix} xx \end{pmatrix} = 2x^2.$$

We can get any value for the inner product that we want, just by choosing the appropriate value of x .

As mentioned earlier, we are going to use the inner product as a measure of how similar two vectors are. With this interpretation in mind, it is natural to require that the inner product of any state with itself gives the value 1. This is already achieved for the inner products of $|0\rangle$ and $|1\rangle$ with themselves, so let's make it true for all other states too.

This condition is known as the normalization condition. In this case, it means that $x = 1/\sqrt{2}$. Now we know what our new state is, so here's a few ways of writing it down.

$$\begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \end{pmatrix} = 1/\sqrt{2} \begin{pmatrix} 1 & 1 \end{pmatrix} = |0\rangle + |1\rangle/\sqrt{2}$$

This state is essentially just $|0\rangle$ and $|1\rangle$ added together and then normalized, so we will give it a name to reflect that origin. We call it $|+\rangle$.

The Born rule

Now we've got three states that we can write down as vectors. We can also calculate inner products for them. For example, the inner product of each with $\langle 0|$ is

$$\langle 0|0\rangle = 1\langle 0|1\rangle = 0\langle 0|+ \rangle = 1/\sqrt{2}.$$

We also know the probabilities of getting various outcomes from a z measurement for these states. For example, let's use p_{z0} to denote the probability of the result 0 for a z measurement. The values this has for our three states are

$$p_{z0}(|0\rangle) = 1, p_{z0}(|1\rangle) = 0, p_{z0}(|+\rangle) = 1/2.$$

As you might have noticed, there's a lot of similarity between the numbers we get from the inner products and those we get for the probabilities. Specifically, the three probabilities can all be

written as the square of the inner products:

$$p_{z0}(|a\rangle) = (\langle 0|a\rangle)^2.$$

Here $|a\rangle$ represents any generic qubit state.

This property doesn't just hold for the $|0\rangle$ outcome. If we compare the inner products with $\langle 1|$ with the probabilities of the $|1\rangle$ outcome, we find a similar relation.

$$p_{z1}(|a\rangle) = (\langle 1|a\rangle)^2.$$

The same also holds true for other types of measurement. All probabilities in quantum mechanics can be expressed in this way. It is known as the *Born rule*.

Global and relative phases

Vectors are how we use math to represent the state of a qubit. With them we can calculate the probabilities of all the possible things that could ever be measured. These probabilities are essentially all that is physically relevant about a qubit. It is by measuring them that we can determine or verify what state our qubits are in. Any aspect of the state that doesn't affect the probabilities is therefore just a mathematical curiosity.

Let's find an example. Consider a state that looks like this:

$$|\tilde{0}\rangle = \begin{pmatrix} -1 \\ 0 \end{pmatrix} = -|0\rangle.$$

This is equivalent to multiplying the state $|0\rangle$ by -1 . It means that every inner product we could calculate with $|\tilde{0}\rangle$ is the same as for $|0\rangle$, but multiplied by -1 .

$$\langle a|\tilde{0}\rangle = -\langle a|0\rangle$$

As you probably know, any negative number squares to the same value as its positive counterpart:

$$(-x)^2 = x^2.$$

Since we square inner products to get probabilities, this means that any probability we could ever calculate for $|\tilde{0}\rangle$ will give us the same value as for $|0\rangle$. If the probabilities of everything are the same, there is no observable difference between $|\tilde{0}\rangle$ and $|0\rangle$; they are just different ways of representing the same state.

This is known as the irrelevance of the global phase. Quite simply, this means that multiplying the whole of a quantum state by -1 gives us a state that will look different mathematically, but which is actually completely equivalent physically.

The same is not true if the phase is *relative* rather than *global*. This would mean multiplying only part of the state by -1 , for example:

$$\begin{pmatrix} a_0 \\ a_1 \end{pmatrix} - \begin{pmatrix} a_0 \\ -a_1 \end{pmatrix}.$$

Doing this with the $|+\rangle$ state gives us a new state. We'll call it $|-\rangle$.

$$|-\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix} = |0\rangle - |1\rangle \frac{1}{\sqrt{2}}$$

The values p_{z0} and p_{z1} for $|-\rangle$ are the same as for $|+\rangle$. These two states are thus indistinguishable when we make only z measurements. But there are other ways to distinguish them. To see how, consider the inner product of $|+\rangle$ and $|-\rangle$.

$$\langle - | + \rangle = \langle + | - \rangle = 0$$

The inner product is 0, just as it is for $|0\rangle$ and $|1\rangle$. This means that the $|+\rangle$ and $|-\rangle$ states are orthogonal: they represent a pair of mutually exclusive possible ways for a qubit to be a qubit.

The x basis - part 2

Whenever we find a pair of orthogonal qubit states, we can use it to define a new kind of measurement.

First, let's apply this to the case we know well: the z measurement. This asks a qubit whether it is $|0\rangle$ or $|1\rangle$. If it is $|0\rangle$, we get the result 0. For $|1\rangle$ we get 1. Anything else, such as $|+\rangle$, is treated as a superposition of the two.

$$|+\rangle = |0\rangle + |1\rangle \frac{1}{\sqrt{2}}$$

For a superposition, the qubit needs to randomly choose between the two possibilities according to the Born rule.

We can similarly define a measurement based on $| + \rangle$ and $| - \rangle$. This asks a qubit whether it is $| + \rangle$ or $| - \rangle$. If it is $| + \rangle$, we get the result `0`. For $| - \rangle$ we get `1`. Anything else is treated as a superposition of the two. This includes the states $|0\rangle$ and $|1\rangle$, which we can write as

$$|0\rangle = |+\rangle + |-\rangle\sqrt{2}, |1\rangle = |+\rangle - |-\rangle\sqrt{2}.$$

For these, and any other superpositions of $| + \rangle$ and $| - \rangle$, the qubit chooses its outcome randomly with probabilities

$$p_{x0}(|a\rangle) = (\langle + |a\rangle)^2, p_{x1}(|a\rangle) = (\langle - |a\rangle)^2.$$

This is the x measurement.

The conservation of certainty

Qubits in quantum circuits always start out in the state $|0\rangle$. By applying different operations, we can make them explore other states.

Try this out yourself using a single qubit, creating circuits using operations from the following list, and then doing the x and z measurements in the way described at the top of the page.

```
qc = QuantumCircuit(1)

qc.h(0) # the hadamard

qc.x(0) # x gate

qc.y(0) # y gate

qc.z(0) # z gate

# for the following, replace theta by any number
theta = 3.14159/4
qc.ry(theta,0); # y axis rotation
```



You'll find examples where the z measurement gives a certain result, but the x is completely random. You'll also find examples where the opposite is true. Furthermore, there are many examples where both are partially random. With enough experimentation, you might even uncover the rule that underlies this behavior:

$$(p_{z0} - p_{z1})^2 + (p_{x0} - p_{x1})^2 = 1.$$

This is a version of Heisenberg's famous uncertainty principle. The $(p_{z0} - p_{z1})^2$ term measures how certain the qubit is about the outcome of a z measurement. The $(p_{x0} - p_{x1})^2$ term measures the same for the x measurement. Their sum is the total certainty of the two combined. Given that this total always takes the same value, we find that the amount of information a qubit can be certain about is a limited and conserved resource.

Here is a program to calculate this total certainty. As you should see, whatever gates from the above list you choose to put in `qc`, the total certainty comes out as 1 (or as near as possible given statistical noise).

```
shots = 2**14 # number of samples used for statistics
uncertainty = 0
for measure_circuit in [measure_z, measure_x]:
    # run the circuit with a the selected measurement and get the number of samples tha
    counts = execute(qc+measure_circuit,Aer.get_backend('qasm_simulator'),shots=shots).

    # calculate the probabilities for each bit value
    probs = {}
    for output in ['0','1']:
        if output in counts:
            probs[output] = counts[output]/shots
        else:
            probs[output] = 0

    uncertainty += (probs['0'] - probs['1'])**2

# print the total uncertainty
print('The total uncertainty is',uncertainty )
```

The total uncertainty is 0.9984774887561798

Now we have found this rule, let's try to break it! Then we can hope to get a deeper understanding of what is going on. We can do this by simply implementing the operation below, and then recalculating the total uncertainty.

```
# for the following, replace theta by any number
theta = 3.14159/2
```

```
qc.rx(theta,0); # x axis rotation
```

For a circuit with a single `rx` with $\theta = \pi/2$, we will find that $(p_{z0} - p_{z1})^2 + (p_{x0} - p_{x1})^2 = 0$. This operation seems to have reduced our total certainty to zero.

All is not lost, though. We simply need to perform another identical `rx` gate to our circuit to go back to obeying $(p_{z0} - p_{z1})^2 + (p_{x0} - p_{x1})^2 = 1$. This shows that the operation does not destroy our certainty; it simply moves it somewhere else and then back again. So let's find that somewhere else.

The y basis - part 1

There are infinitely many ways to measure a qubit, but the z and x measurements have a special relationship with each other. We say that they are *mutually unbiased*. This simply means that certainty for one implies complete randomness for the other.

At the end of the last section, it seemed that we were missing a piece of the puzzle. We need another type of measurement to plug the gap in our total certainty, and it makes sense to look for one that is also mutually unbiased with x and z.

The first step is to find a state that seems random to both x and z measurements. Let's call it $| \circlearrowleft \rangle$, for no apparent reason.

$$| \circlearrowleft \rangle = c_0 | 0 \rangle + c_1 | 1 \rangle$$

Now the job is to find the right values for c_0 and c_1 . You could try to do this with standard positive and negative numbers, but you'll never be able to find a state that is completely random for both x and z measurements. To achieve this, we need to use complex numbers.

Complex numbers

Hopefully you've come across complex numbers before, but here is a quick reminder.

Normal numbers, such as the ones we use for counting bananas, are known as *real numbers*. We cannot solve all possible equations using only real numbers. For example, there is no real number that serves as the square root of -1 . To deal with this issue, we need more numbers, which we call *complex numbers*.

To define complex numbers we start by accepting the fact that -1 has a square root, and that its name is i . Any complex number can then be written

$$x = x_r + i x_i.$$

Here x_r and x_i are both normal numbers (positive or negative), where x_r is known as the real part and x_i as the imaginary part.

For every complex number x there is a corresponding complex conjugate x^*

$$x^* = x_r - i x_i.$$

Multiplying x by x^* gives us a real number. It's most useful to write this as

$$|x| = \sqrt{x x^*}.$$

Here $|x|$ is known as the magnitude of x (or, equivalently, of x^*).

If we are going to allow the numbers in our quantum states to be complex, we'll need to upgrade some of our equations.

First, we need to ensure that the inner product of a state with itself is always 1. To do this, the bra and ket versions of the same state must be defined as follows:

$$|a\rangle = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}, \quad \langle a| = \begin{pmatrix} a_{*0} & a_{*1} \end{pmatrix}.$$

Then we just need a small change to the Born rule, where we square the magnitudes of inner products, rather than just the inner products themselves.

$$p_{z0}(|a\rangle) = |\langle 0 | a \rangle|^2, p_{z1}(|a\rangle) = |\langle 1 | a \rangle|^2, p_{x0}(|a\rangle) = |\langle + | a \rangle|^2, p_{x1}(|a\rangle) = |\langle - | a \rangle|^2.$$

The irrelevance of the global phase also needs an upgrade. Previously, we only talked about multiplying by -1 . In fact, we can multiply a state by any complex number whose magnitude is 1. This will give us a state that will look different, but which is actually completely equivalent. This includes multiplying by i , $-i$ or infinitely many other possibilities.

The y basis - part 2

Now that we have complex numbers, we can define the following pair of states.

$$|\psi\rangle = |0\rangle + i|1\rangle\sqrt{2}, \quad |\phi\rangle = |0\rangle - i|1\rangle\sqrt{2}$$

You can verify yourself that they both give random outputs for x and z measurements. They are also orthogonal to each other. They therefore define a new measurement, and that basis is mutually unbiased with x and z. This is the third and final fundamental measurement for a single qubit. We call it the y measurement, and can implement it with

```
# y measurement of qubit 0
measure_y = QuantumCircuit(1,1)
measure_y.sdg(0)
measure_y.h(0)
measure_y.measure(0,0);
```

With the x, y and z measurements, we now have everything covered. Whatever operations we apply, a single isolated qubit will always obey

$$(p_{z0} - p_{z1})^2 + (p_{y0} - p_{y1})^2 + (p_{x0} - p_{x1})^2 = 1.$$

To see this, we can incorporate the y measurement into our measure of total certainty.

```
shots = 2**14 # number of samples used for statistics

uncertainty = 0
for measure_circuit in [measure_z, measure_x, measure_y]:

    # run the circuit with a the selected measurement and get the number of samples that
    counts = execute(qc+measure_circuit,Aer.get_backend('qasm_simulator'),shots=shots).

    # calculate the probabilities for each bit value
    probs = {}
    for output in ['0','1']:
        if output in counts:
            probs[output] = counts[output]/shots
        else:
            probs[output] = 0

    uncertainty += (probs['0'] - probs['1'])**2

# print the total uncertainty
print('The total uncertainty is',uncertainty )
```

The total uncertainty is 0.986548513174057

For more than one qubit, this relation will need another upgrade. This is because the qubits can spend their limited certainty on creating correlations that can only be detected when multiple qubits are measured. The fact that certainty is conserved remains true, but it can only be seen when looking at all the qubits together.

Before we move on to entanglement, there is more to explore about just a single qubit. As we'll see in the next section, the conservation of certainty leads to a particularly useful way of visualizing single-qubit states and gates.

Pauli Matrices and the Bloch Sphere

```
from qiskit import *
from qiskit.visualization import plot_bloch_vector
```



In this section we'll further develop the topics introduced in the last, and introduce a useful visualization of single-qubit states.

Pauli matrices

Wherever there are vectors, matrices are not far behind. The three important matrices for qubits are known as the Pauli matrices.

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$
$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$
$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

These have many useful properties, as well as a deep connection to the x, y and z measurements. Specifically, we can use them to calculate the three quantities used in the last section:

$$\langle a|X|a\rangle = p_0^x(|a\rangle) - p_1^x(|a\rangle),$$
$$\langle a|Y|a\rangle = p_0^y(|a\rangle) - p_1^y(|a\rangle),$$
$$\langle a|Z|a\rangle = p_0^z(|a\rangle) - p_1^z(|a\rangle).$$

These quantities are known as the expectation values of the three matrices. In calculating them, we make use of standard matrix multiplication.

Typically, we prefer to use a more compact notation for the quantities above. Since we usually know what state we are talking about in any given situation, we don't explicitly write it in. This allows us to write $\langle X \rangle = \langle a|X|a\rangle$ $\langle X \rangle = \langle a|X|a\rangle$, etc. Our statement from the last section, regarding the conservation of certainty for an isolated qubit, can then be written

$$\langle X \rangle^2 + \langle Y \rangle^2 + \langle Z \rangle^2 = 1.$$

To calculate these values in Qiskit, we first need a single qubit circuit to analyze.

```
qc = QuantumCircuit(1)
```



Then we need to define the x, y and z measurements.

```
# z measurement of qubit 0
measure_z = QuantumCircuit(1,1)
measure_z.measure(0,0);

# x measurement of qubit 0
measure_x = QuantumCircuit(1,1)
measure_x.h(0)
measure_x.measure(0,0)

# y measurement of qubit 0
measure_y = QuantumCircuit(1,1)
measure_y.sdg(0)
measure_y.h(0)
measure_y.measure(0,0);
```



Finally we can run the circuit with each kind of measurement, calculate the probabilities and use them to determine $\langle X \rangle \langle X \rangle$, $\langle Y \rangle \langle Y \rangle$ and $\langle Z \rangle \langle Z \rangle$. This requires a process largely similar to the one used in the last section to calculate total certainty.

Here we place the results in a list called `bloch_vector`, for which `bloch_vector[0]` is $\langle X \rangle \langle X \rangle$, `bloch_vector[1]` is $\langle Y \rangle \langle Y \rangle$ and `bloch_vector[2]` is $\langle Z \rangle \langle Z \rangle$

```
shots = 2**14 # number of samples used for statistics
```



```
bloch_vector = []
for measure_circuit in [measure_x, measure_y, measure_z]:

    # run the circuit with a the selected measurement and get the number of samples tha
    counts = execute(qc+measure_circuit,Aer.get_backend('qasm_simulator'),shots=shots).

    # calculate the probabilities for each bit value
    probs = {}
    for output in ['0','1']:
        if output in counts:
            probs[output] = counts[output]/shots
```



```
else:  
    probs[output] = 0  
  
bloch_vector.append( probs['0'] - probs['1'] )
```

The Bloch sphere

Let's take a moment to think a little about the numbers $\langle X \rangle \langle X \rangle$, $\langle Y \rangle \langle Y \rangle$ and $\langle Z \rangle \langle Z \rangle$. Though their values depend on what state our qubit is in, they are always constrained to be no larger than 1, and no smaller than -1. They also collectively obey the condition $\langle X \rangle^2 + \langle Y \rangle^2 + \langle Z \rangle^2 = 1$ $\langle X \rangle^2 + \langle Y \rangle^2 + \langle Z \rangle^2 = 1$.

The same properties are also shared by another set of three numbers that we know from a completely different context. To see what they are, first consider a sphere. For this, we can describe every point on the surface in terms of its x, y and z coordinates. We'll place the origin of our coordinate system at the center of the sphere. The coordinates are then constrained by the radius in both directions: they can be no greater than r , and no less than $-r$. For simplicity, let's set the radius to be $r = 1$.

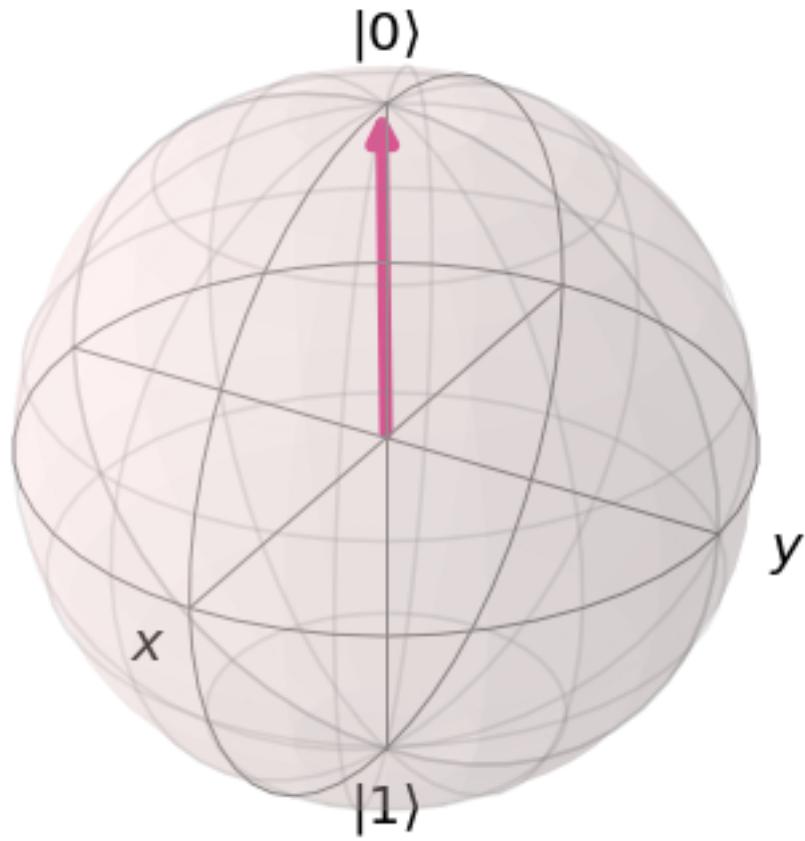
For any point, the distance from the center of the sphere can be determined by the 3D version of Pythagoras' theorem. Specifically, $x^2 + y^2 + z^2 = r^2$. For points on the surface, this distance is always 1.

So now we have three numbers that can each be no greater than 1, no less than -1, and for which the sum of the squares is always 1. All exactly the same as $\langle X \rangle \langle X \rangle$, $\langle Y \rangle \langle Y \rangle$ and $\langle Z \rangle \langle Z \rangle$. They even have pretty much the same names as these values.

Because of this correspondence, we can apply all our existing knowledge and intuition about balls to our understanding of qubits. Specifically, we can visualize any single-qubit state as a point on the surface of a sphere. We call this the Bloch sphere.

```
plot_bloch_vector( bloch_vector )
```





We usually associate $|0\rangle|0\rangle$ with the north pole, $|1\rangle|1\rangle$ with the south, and the states for the x and y measurements around the equator. Any pair of orthogonal states correspond to diametrically opposite points on this sphere.

As we'll see in future sections, the Bloch sphere makes it easier to understand single-qubit operations. Each moves points around on the surface of the sphere, and so can be interpreted as a simple rotation.

States for Many Qubits

```
from qiskit import *
```



Introduction

We've already seen how to write down the state of a single qubit. Now we can look at how to do it when we have more than just one.

Let's start by looking at bits. The state of a single bit is expressed as `0` or `1`. For two bits we can have `00`, `01`, `10` or `11`, where each digit tells us the state of one of the bits. For more bits, we just use longer strings of bit values, known as 'bit strings'.

The conversion to qubits is quite straightforward: we simply put a `|` and around bit strings. For example, to describe two qubits, both of which are in state $|0\rangle$, we write $|00\rangle$. The four possible bit strings for two bits are then converted into four orthogonal states, which together completely specify the state of two qubits:

$$|a\rangle = a_{00}|00\rangle + a_{01}|01\rangle + a_{10}|10\rangle + a_{11}|11\rangle = \begin{pmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{pmatrix}.$$

As in the single-qubit case, the elements of this vector are complex numbers. We require the state to be normalized so that $\langle a|a\rangle = 1$, and probabilities are given by the Born rule ($p_{00}^{zz} = |\langle 00|a\rangle|^2$, etc).

When designing quantum software, there are times when we will want to look at the state of our qubits. This can be done in Qiskit using the 'statevector simulator'.

For example, here is the state vector for a simple circuit on two qubits.

```

# set up circuit (no measurements required)
qc = QuantumCircuit(2)
qc.h(0)
qc.h(1)
qc.rz(3.14/4,1)

# set up simulator that returns statevectors
backend = Aer.get_backend('statevector_simulator')

# run the circuit to get the state vector
state = execute(qc,backend).result().get_statevector()

# now we use some fanciness to display it in Latex
from IPython.display import display, Markdown, Latex
def state2latex(state):
    state_latex = '\\begin{pmatrix}'
    for amplitude in state:
        state_latex += str(amplitude) + '\\\\'
    state_latex = state_latex[0:-4]
    state_latex += '\\end{pmatrix}'
    display(Markdown(state_latex))

state2latex(state)

```

$$\begin{pmatrix} 0.5000000000000001 + 0j \\ 0.5 + 0j \\ 0.3536941345835999 + 0.353412590552683j \\ 0.35369413458359983 + 0.3534125905526829 \end{pmatrix}$$

Note that Python uses j to denote $\sqrt{-1}$, rather than i as we use.

The tensor product

Suppose we have two qubits, with one in state $|a\rangle = a_0|0\rangle + a_1|1\rangle$ and the other in state $|b\rangle = b_0|0\rangle + b_1|1\rangle$, and we want to create the two-qubit state that describes them both.

To see how to do this, we can use the Born rule as a guide. We know that the probability of getting a 0 is $|a_0|^2$ for one qubit and $|b_0|^2$ for the other. The probability of getting 00 is therefore $|a_0|^2|b_0|^2 = |a_0 b_0|^2$. Working backwards from this probability, it makes sense for the $|00\rangle$ state to have the amplitude $a_0 b_0$. Repeating this principle, the whole state becomes.

$$a_0 b_0 |00\rangle + a_0 b_1 |01\rangle + a_1 b_0 |10\rangle + a_1 b_1 |11\rangle.$$

This is exactly the result we would get when using the 'tensor product' [1], which is a standard method for combining vectors and matrices in a way that preserves all the information they contain. Using the notation of the tensor product, we can write this state as $|a\rangle \otimes |b\rangle$.

We also make use of the tensor product to represent the action of single-qubit matrices on these multiqubit vectors. For example, here's an X that acts only on the qubit on the right:

$$I \otimes X = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

This was made by combining the X matrix for the qubit on the right with the single-qubit identity operator, I , for the qubit on the left. The identity operator is the unique operator that does absolutely nothing to a vector. The two-qubit operation resulting from the tensor product allows us to calculate expectation values for x measurements of the qubit on the left, in exactly the same way as before.

Entangled states

Using the tensor product we can construct matrices such as $X \otimes X$, $Z \otimes Z$, $Z \otimes X$, and so on. The expectation values of these also represent probabilities. For example, for a general two qubit state $|a\rangle$,

$$\langle a|Z \otimes Z|a\rangle = P_0^{zz} - P_1^{zz}.$$

The zz in P_0^{zz} and P_1^{zz} refers to the fact that these probabilities describe the outcomes when a z measurement is made on both qubits. A quantity such as $\langle a|Z \otimes X|a\rangle$ will reflect similar probabilities for different choices of measurements on the qubits.

The 0 and 1 of P_0^{zz} and P_1^{zz} refer to whether there are an even (for 0) or odd (for 1) number of 1 outcomes in the output. So P_0^{zz} is the probability that the result is either 00 or 11 , and P_{01}^{zz} is the probability that the result is either 01 or 10 .

These multiqubit Pauli operators can be used to analyze a new kind of state, that cannot be described as a simple tensor product of two independent qubit states. For example,

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

This represents a quantum form of correlated state, known as an entangled state. The correlations can be easily seen from the fact that only the `00` and `11` outcomes are possible when making z measurements of both qubits, and so the outcomes of these measurements will always agree. This can also be seen from the fact that

$$\langle \Phi^+ | Z \otimes Z | \Phi^+ \rangle = 1, \quad \therefore P_0^{zz} = 1.$$

These aren't the only correlations in this state. If you use x measurements, you'd find that the results still always agree. For y measurements, they always disagree. So we find that $\langle \Phi^+ | X \otimes X | \Phi^+ \rangle = 1$ and $\langle \Phi^+ | Y \otimes Y | \Phi^+ \rangle = -1$. There are a lot of correlations in this little state!

For more qubits, we can get ever larger multiqubit Pauli operators. In this case, the probabilities such as $P_0^{zz...zz}$ and $P_1^{zz...zz}$ are understood in the same way as for two qubits: they reflect the cases where the total output bit string consists of an even or odd number of `1`s, respectively. We can use these to quantify even more complex correlations.

The generation of complex entangled states is a necessary part of gaining a quantum advantage. The use of large vectors and multiqubit correlation functions is therefore important if we want to mathematically analyze what our qubits are doing.

References

- [1] For more on tensor products, see: Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum Computation and Quantum Information: 10th Anniversary Edition (10th ed.)*. Cambridge University Press: New York, NY, USA.

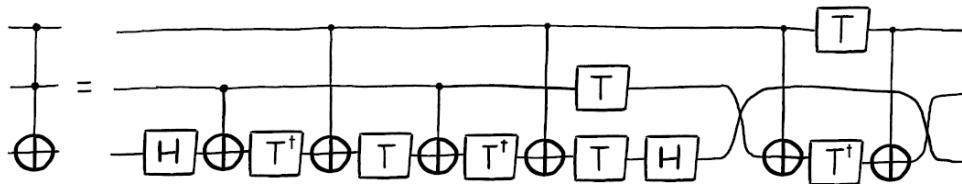
Introduction

Just having some qubits is not enough: We also need to manipulate them. All possible ways of doing this can be compiled down to a basic set of operations, known as quantum gates.

Typically, the gates that can be directly implemented in hardware will act only on one or two qubits. In our circuits, we may want to use complex gates that act on a great number of qubits. Fortunately, this will not be a problem. With the one and two qubit gates given to us by the hardware, it is possible to build any other gate.

In this chapter we will first introduce the most basic gates, as well as the mathematics used to describe and analyze them. Then we'll show how to prove that these gates can be used to create any possible quantum algorithm.

The chapter then concludes by looking at small-scale uses of quantum gates. For example, we see how to build three-qubit gates like the Toffoli from single- and two-qubit operations.



Quantum Gates

```
from qiskit import *
```



To manipulate an input state we need to apply the basic operations of quantum computing. These are known as quantum gates. Here we'll give an introduction to some of the most fundamental gates in quantum computing. Most of those we'll be looking at act only on a single qubit. This means that their actions can be understood in terms of the Bloch sphere.

The Pauli operators

The simplest quantum gates are the Paulis: X, Y and Z. Their action is to perform a half rotation of the Bloch sphere around the x, y and z axes. They therefore have effects similar to the classical NOT gate or bit-flip. Specifically, the action of the X gate on the states $|0\rangle$ and $|1\rangle$ is

$$X|0\rangle = |1\rangle, X|1\rangle = |0\rangle.$$

The Z gate has a similar effect on the states $|+\rangle$ and $|-\rangle$:

$$Z|+\rangle = |-\rangle, Z|-\rangle = |+\rangle.$$

These gates are implemented in Qiskit as follows (assuming a circuit named `qc`).

```
qc.x(0) # x on qubit 0  
qc.y(0) # y on qubit 0  
qc.z(0) # z on qubit 0
```



The matrix representations of these gates have already been shown in a previous section.

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

There, their job was to help us make calculations regarding measurements. But since these matrices are unitary, and therefore define a reversible quantum operation, this additional interpretation of them as gates is also possible.

Note that here we referred to these gates as X, Y and Z and `x`, `y` and `z`, depending on whether we were talking about their matrix representation or the way they are written in Qiskit. Typically we

will use the style of X, Y and Z when referring to gates in text or equations, and x, y and z when writing Qiskit code.

Hadamard and S

The Hadamard gate is one that we've already used. It's a key component in performing an x measurement:

```
measure_x = QuantumCircuit(1,1)
measure_x.h(0);
measure_x.measure(0,0);
```



Like the Paulis, the Hadamard is also a half rotation of the Bloch sphere. The difference is that it rotates around an axis located halfway between x and z. This gives it the effect of rotating states that point along the z axis to those pointing along x, and vice versa.

$$H|0\rangle = |+\rangle, H|1\rangle = |-\rangle, H|+\rangle = |0\rangle, H|-\rangle = |1\rangle.$$

This effect makes it an essential part of making x measurements, since the hardware behind quantum computing typically only allows the z measurement to be performed directly. By moving x basis information to the z basis, it allows an indirect measurement of x.

The property that $H|0\rangle = |+\rangle$ also makes the Hadamard our primary means of generating superposition states. Its matrix form is

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

The S and S^\dagger gates have a similar role to play in quantum computation.

```
qc.s(0) # s gate on qubit 0
qc.sdg(0) # s† on qubit 0
```



They are quarter turns of the Bloch sphere around the z axis, and so can be regarded as the two possible square roots of the Z gate,

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, S^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}.$$

The effect of these gates is to rotate between the states of the x and y bases.

$$S|+\rangle = |\psi\rangle, S|-\rangle = |\phi\rangle, S^\dagger|\psi\rangle = |+\rangle, S^\dagger|\phi\rangle = |-\rangle.$$

They are therefore used as part of y measurements.

```
measure_y = QuantumCircuit(1,1)
measure_y.sdg(0)
measure_y.h(0)
measure_y.measure(0,0);
```



The H, S and S^\dagger gates, along with the Paulis, form the so-called 'Clifford group' for a single qubit, which will be discussed more in later sections. These gates are extremely useful for many tasks in making and manipulating superpositions, as well as facilitating different kinds of measurements. But to unlock the full potential of qubits, we need the next set of gates.

Other single-qubit gates

We've already seen the X, Y and Z gates, which are rotations around the x , y and z axes by a specific angle. More generally we can extend this concept to rotations by an arbitrary angle θ . This gives us the gates $R_x(\theta)$, $R_y(\theta)$ and $R_z(\theta)$. The angle is expressed in radians, so the Pauli gates correspond to $\theta = \pi$. Their square roots require half this angle, $\theta = \pm \pi/2$, and so on.

In Qasm, these rotations can be implemented with `rx` , `ry` , and `rz` as follows.

```
qc.rx(theta,0) # rx rotation on qubit 0
qc.ry(theta,0) # ry rotation on qubit 0
qc.rz(theta,0) # rz rotation on qubit 0
```



Two specific examples of $R_z(\theta)$ have their own names: those for $\theta = \pm \pi/4$. These are the square roots of S, and are known as T and T^\dagger .

```
qc.t(0) # t gate on qubit 0
qc.tdg(0) # t† on qubit 1
```



Their matrix form is

$$T = \begin{pmatrix} 1 & 00 & e^{i\pi/4} \end{pmatrix}, T^\dagger = \begin{pmatrix} 1 & 00 & e^{-i\pi/4} \end{pmatrix}.$$

All single-qubit operations are compiled down to gates known as U_1 , U_2 and U_3 before running on real IBM quantum hardware. For that reason they are sometimes called the *physical gates*. Let's have a more detailed look at them. The most general is

$$U_3(\theta, \phi, \lambda) = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2)e^{i\phi}\sin(\theta/2) & e^{i\lambda+i\phi}\cos(\theta/2) \end{pmatrix}.$$

This has the effect of rotating a qubit in the initial $|0\rangle$ state to one with an arbitrary superposition and relative phase:

$$U_3|0\rangle = \cos(\theta/2)|0\rangle + \sin(\theta/2)e^{i\phi}|1\rangle.$$

The U_1 gate is known as the phase gate and is essentially the same as $R_z(\lambda)$. Its relationship with U_3 and its matrix form are,

$$U_1(\lambda) = U_3(0, 0, \lambda) = \begin{pmatrix} 1 & 00 & e^{i\lambda} \end{pmatrix}.$$

In IBM Q hardware, this gate is implemented as a frame change and takes zero time.

The second gate is U_2 , and has the form

$$U_2(\phi, \lambda) = U_3(\pi/2, \phi, \lambda) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -e^{i\lambda}e^{i\phi} & e^{i\lambda+i\phi} \end{pmatrix}.$$

From this gate, the Hadamard is done by $H = U_2(0, \pi)$. In IBM Q hardware, this is implemented by a pre- and post-frame change and an $X_{\pi/2}$ pulse.

Multiqubit gates

To create quantum algorithms that beat their classical counterparts, we need more than isolated qubits. We need ways for them to interact. This is done by multiqubit gates.

The most prominent multiqubit gates are the two-qubit CNOT and the three-qubit Toffoli. These have already been introduced in 'The atoms of computation'. They essentially perform reversible versions of the classical XOR and AND gates, respectively.

```
qc.cx(0,1) # CNOT controlled on qubit 0 with qubit 1 as target
qc.ccx(0,1,2) # Toffoli controlled on qubits 0 and 1 with qubit 2 as target
```



Note that the CNOT is referred to as `cx` in Qiskit.

We can also interpret the CNOT as performing an X on its target qubit, but only when its control qubit is in state $|1\rangle$, and doing nothing when the control is in state $|0\rangle$. With this interpretation in mind, we can similarly define gates that work in the same way, but instead perform a Y or Z on the target qubit depending on the $|0\rangle$ and $|1\rangle$ states of the control.

```
qc.cy(0,1) # controlled-Y, controlled on qubit 0 with qubit 1 as target
qc.cz(0,1) # controlled-Z, controlled on qubit 0 with qubit 1 as target
```



The Toffoli gate can be interpreted in a similar manner, except that it has a pair of control qubits. Only if both are in state $|1\rangle$ is the X applied to the target.

Composite gates

When we combine gates, we make new gates. If we want to see the matrix representation of these, we can use the 'unitary simulator' of Qiskit.

For example, let's try something simple: a two qubit circuit with an `x` applied to one and a `z` to the other. Using tensor products, we can expect the result to be,

$$Z \otimes X = \begin{pmatrix} 1 & 00 & -1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 11 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 01 & 0 & 0 & 00 & 0 & 0 & -10 & 0 & -1 \end{pmatrix}.$$

This is exactly what we find when we analyze the circuit with this tool.



```
# set up circuit (no measurements required)
qc = QuantumCircuit(2)
qc.x(0) # qubits numbered from the right, so qubit 0 is the qubit on the right
qc.z(1) # and qubit 1 is on the left

# set up simulator that returns unitary matrix
backend = Aer.get_backend('unitary_simulator')

# run the circuit to get the matrix
gate = execute(qc,backend).result().get_unitary()

# now we use some fanciness to display it in Latex
from IPython.display import display, Markdown, Latex
gate_latex = '\\begin{pmatrix}'
for line in gate:
    for element in line:
        gate_latex += str(element) + '&'
    gate_latex = gate_latex[0:-1]
    gate_latex += '\\\\'
gate_latex = gate_latex[0:-2]
gate_latex += '\\end{pmatrix}'
display(Markdown(gate_latex))
```

$$\begin{pmatrix} 0j & (1 + 0j) & 0j & 0j(1 + 0j) & 0j & 0j & 0j0j & 0j & 0j & (-1 + 0j)0j & 0j & (-1 + 0j) \end{pmatrix}$$

Fun with Matrices

Manipulating matrices is the heart of how we analyze quantum programs. In this section we'll look at some of the most common tools that can be used for this.

Unitary and Hermitian matrices

Studying universality is inherently an endeavour that needs math -- so we'll need to get ourselves some mathematical tools.

Firstly, we need the concept of the Hermitian conjugate. For this we take a matrix M , then we replace every element with its complex conjugate, and finally we transpose them (replace the top left element with the bottom right, and so on). This gives us a new matrix that we call M^\dagger .

Two families of matrices that are very important to quantum computing are defined by their relationship with the Hermitian conjugate. One is the family of unitary matrices, for which

$$UU^\dagger = U^\dagger U = 1.$$

$$UU^\dagger = U^\dagger U = 1.$$

This means that the Hermitian conjugate of a unitary is its inverse. This is another unitary U^\dagger with the power to undo the effects of U . All gates in quantum computing, with the exception of measurement, can be represented by unitary matrices.

The other important family of matrices are the Hermitian matrices. These are the matrices that are unaffected by the Hermitian conjugate

$$H = H^\dagger.$$

The matrices X , Y , Z and H are examples of Hermitian matrices that you've already seen (coincidentally, they are also all unitary since they are their own inverses).

Matrices as outer products

In a previous section, we calculated many inner products, such as $\langle 0 | 0 \rangle = 1$. These combine a bra and a ket to give us a single number. We can also combine them in a way that gives us a matrix,

simply by putting them in the opposite order. This is called an outer product, and works by standard matrix multiplication. For example

$$|0\rangle\langle 0| = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} (1 \ 0) = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}, |0\rangle\langle 1| = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} (0 \ 1) = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, |1\rangle\langle 0| = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} (1 \ 0) =$$

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, |1\rangle\langle 1| = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} (0 \ 1) = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

This also means that we can write any matrix purely in terms of outer products. In the examples above, we constructed the four matrices that cover each of the single elements in a single-qubit matrix, so we can write any other single-qubit matrix in terms of them.

$$|1\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix},$$

$$M_0 = \begin{pmatrix} 0 & 0 \\ m_{0,0} & m_{0,1} \\ m_{1,0} & m_{1,1} \end{pmatrix} = m_{0,0}|0\rangle\langle 0| + m_{0,1}|0\rangle\langle 1| + m_{1,0}|1\rangle\langle 0| + m_{1,1}|1\rangle\langle 1|$$

$$M = \begin{pmatrix} 0 & 0 \\ m_{0,0} & m_{0,1} \\ 0 & 1 \\ m_{1,0} & m_{1,1} \end{pmatrix}_n = m_{0,0}|0\rangle\langle 0| + m_{0,1}|0\rangle\langle 1| + m_{1,0}|1\rangle\langle 0| + m_{1,1}|1\rangle\langle 1|$$

Spectral form

Outer products provide a very useful way of writing matrices. They are especially useful for unitaries, since we can simply capture the way that states are transformed.

To do this, we first pick a set of orthogonal states that describe our qubits. For example, for two qubits we could simply choose $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$. Then we determine the state to which the unitary rotates each of these states. Let's call these $\{|u_{00}\rangle, |u_{01}\rangle, |u_{10}\rangle, |u_{11}\rangle\}$. The unitary that performs this can then be written

$$U = |u_{00}\rangle\langle 00| + |u_{01}\rangle\langle 01| + |u_{10}\rangle\langle 10| + |u_{11}\rangle\langle 11|$$

If we know $|u_{00}\rangle\langle 00|$, this tells us directly what $|u_{01}\rangle\langle 01|$, $|u_{10}\rangle\langle 10|$ and $|u_{11}\rangle\langle 11|$ are. From this, we can directly read out the effect of the unitary on the basis states we have chosen.

This way of writing a unitary is not unique for each unitary. You can do it for every possible set of orthogonal input states. However, for at least one possible set of states it will take an especially simple form. These are the eigenstates of the matrix, for which

$$U = \sum_j e^{ih_j} |h_j\rangle\langle h_j|$$

$U = \sum_j e^{ih_j} |h_j\rangle\langle h_j|$
 Here the unitary takes each state of this basis, which we've called $|h_j\rangle$, and replaces it with $e^{ih_j} |h_j\rangle$.
 Since the $e^{ih_j} |h_j\rangle$ must themselves be valid quantum states, the e^{ih_j} must be complex numbers of magnitude 1. In fact, this is exactly why we wrote them as a complex exponential; to ensure that they have magnitude 1, we simply need to ensure that the h_j are real numbers.

For these states, the unitary simply induces a global phase. The non-trivial effects of this unitary will come for superpositions of these states, for which a relative phase may be induced.

Hermitian matrices also have well-defined eigenstates and eigenvalues, and can be written in the same form as the unitary matrix above.

$$H = \sum_j h_j |h_j\rangle\langle h_j|.$$

$H = \sum_j h_j |h_j\rangle\langle h_j|$
 In order to satisfy the constraint that $H = H^\dagger$, we must determine what properties are required for the eigenstates and eigenvalues. $H = H^\dagger$

For the eigenstates, we can see what happens when we take the outer product of a state with itself. For this we use the fact that the Hermitian conjugate of a product can be evaluated by taking the Hermitian conjugate of each factor, and then reversing the order of the factors. If we also note that the Hermitian conjugate of a ket is the corresponding bra, and vice versa, we find

$$(|h_j\rangle\langle h_j|)^\dagger = ((h_j)^\dagger) (|h_j\rangle^\dagger) = |h_j\rangle\langle h_j|.$$

The outer product of a state with itself is therefore inherently Hermitian. To ensure that H is Hermitian as a whole, we only need to require the eigenvalues h_j to be real. H

If you were wondering about the coincidence of notation used above for U and H in spectral form, this should hopefully begin to explain it. Essentially, these two types of matrices differ only in that one must have real numbers for eigenvalues, and the other must have complex exponentials of real numbers. This means that, for every unitary, we can define a corresponding Hermitian matrix. For this

we simply reuse the same eigenstates, and use the h_j from each e^{ih_j} as the corresponding eigenvalue.

Similarly, for each Hermitian matrix there exists a unitary. We simply reuse the same eigenstates, and exponentiate the h_j to create the eigenvalues e^{ih_j} . This can be expressed as

$$U = e^{iH}$$

Here we have used the standard definition of how to exponentiate a matrix. This has exactly the properties we require: preserving the eigenstates and exponentiating the eigenvalues.

We can also build a whole family of unitaries for each given Hermitian, using

$$U(\theta) = e^{i\theta H},$$

Where θ is an arbitrary real number. This allows us to interpolate between $\theta = 0$, which will be the identity matrix, to $\theta = 1$, which is U . It also allows us to define a notion of a gate that is the square root of U : one that must be done twice to get the full effect of U . This would simply have $\theta = 1/2$.

$$U$$

$$U$$

$$\theta = 1/2$$

Pauli decomposition

As we saw above, it is possible to write matrices entirely in terms of outer products.

$$M = \begin{pmatrix} m_{0,0} & m_{0,1} & m_{1,0} \\ m_{0,1} & m_{1,1} & m_{1,0} \\ m_{1,0} & m_{1,1} & m_{0,0} \end{pmatrix} = m_{0,0}|0\rangle\langle 0| + m_{0,1}|0\rangle\langle 1| + m_{1,0}|1\rangle\langle 0| + m_{1,1}|1\rangle\langle 1|$$

Now we will see that it is also possible to write them completely in terms of Pauli operators. For this, the key thing to note is that

$$M = \begin{pmatrix} m_{0,0} & m_{0,1} \\ m_{0,1} & m_{1,1} \\ m_{1,0} & m_{1,1} \end{pmatrix} = m_{0,0}|0\rangle\langle 0| + m_{0,1}|0\rangle\langle 1| + m_{1,0}|1\rangle\langle 0| + m_{1,1}|1\rangle\langle 1|$$

$$1 + Z2 = \frac{1}{2} \left[\begin{pmatrix} 1 & 00 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 00 & -1 \end{pmatrix} \right] = |0\rangle\langle 0|, 1 - Z2 = \frac{1}{2} \left[\begin{pmatrix} 1 & 00 & 1 \end{pmatrix} - \begin{pmatrix} 1 & 00 & -1 \end{pmatrix} \right] = |1\rangle\langle 1|$$

$$\langle 1 | \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} | 1 \rangle$$

This shows that $|0\rangle\langle 0|$ and $|1\rangle\langle 1|$ can be expressed using the identity matrix and Z . Now, using the property that $X|0\rangle = |1\rangle\langle 0|$ we can also produce

$$Z$$

$$|\psi\rangle\langle\psi| = 0|0\rangle\langle 0|X = 12(1 + Z) \stackrel{X}{=} X + iY, |\psi\rangle\langle\psi| = X|0\rangle\langle 0| = X 12(1 + Z) = X - iY.$$

Since we have all the outer products, we can now use this to write the matrix in terms of Pauli matrices: $\frac{X - iY}{2}$.
 $(1 + Z) = \frac{X + iY}{2}$.

$$M = m_{0,0} + m_{1,1}^2 I + m_{0,1} + m_{1,0}^2 X + im_{0,1} - m_{1,0}^2 Y + m_{0,0} - m_{1,1}^2 Z.$$

This example was for a general single-qubit matrix, but the corresponding result is true also for matrices for any number of qubits. We simply start from the observation that $\frac{m_{1,1}}{2} Z$.

$$(1 + Z^2) \otimes (1 + Z^2) \otimes \dots \otimes (1 + Z^2) = |00\dots 0\rangle\langle 00\dots 0|,$$

and can then proceed in the same manner as above. In the end it can be shown that any matrix can be expressed in terms of tensor products of Pauli matrices:

$$M = \sum P_{n-1\dots n} C_{P_{n-1\dots n}} P_{n-1} \otimes P_{n-2} \otimes \dots \otimes P_0.$$

For Hermitian matrices, note that the coefficients $C_{P_{n-1\dots n}}$ here will all be real.

Now we have some powerful tools to analyze quantum operations, let's look at the operations we will need to analyze for our study of universality.

The Standard Gate Set

For every possible realization of fault-tolerant quantum computing, there is a set of quantum operations that are most straightforward to realize. Often these consist of multiple so-called Clifford gates, combined with a few single-qubit gates that do not belong to the Clifford group. In this section we'll introduce these concepts, in preparation for showing that they are universal.

Clifford gates

Some of the most important quantum operations are the so-called Clifford operations. A prominent example is the Hadamard gate:

$$H = |+\rangle\langle 0| + |-\rangle\langle 1| = |0\rangle\langle +| + |1\rangle\langle -|.$$

This gate is expressed above using outer products, as described in the last section. When expressed in this form, its famous effect becomes obvious: it takes $|0\rangle|0\rangle$, and rotates it to $|+\rangle|+\rangle$. More generally, we can say it rotates the basis states of the z measurement, $\{|0\rangle, |1\rangle\}$ $\{|0\rangle, |1\rangle\}$, to the basis states of the x measurement, $\{|+\rangle, |-\rangle\}$ $\{|+\rangle, |-\rangle\}$, and vice versa.

This effect of the Hadamard is to move information around a qubit. It swaps any information that would previously be accessed by an x measurement with that accessed by a z measurement. Indeed, one of the most important jobs of the Hadamard is to do exactly this. We use it when wanting to make an x measurement, given that we can only physically make z measurements.

```
// x measurement of qubit 0
h q[0];
measure q[0] -> c[0];
```



The Hadamard can also be used to change how other operations function. For example,

$$\begin{aligned} HXH &= Z, \\ HZH &= X. \end{aligned}$$

By doing a Hadamard before and after an XX, we cause the action it previously applied to the z basis states to be transferred to the x basis states instead. The combined effect is then identical to that of a ZZ. Similarly, the Hadamards cause a ZZ to behave as an XX.

Similar behavior can be seen for the SS gate and its Hermitian conjugate,

$$\begin{aligned} SXS^\dagger &= Y, \\ SYS^\dagger &= -X, \\ SZS^\dagger &= Z. \end{aligned}$$

This has a similar effect to the Hadamard, except that it swaps XX and YY instead of XX and ZZ . In combination with the Hadamard, we could then make a composite gate that shifts information between y and z . This therefore gives us full control over single-qubit Paulis.

The property of transforming Paulis into other Paulis is the defining feature of Clifford gates. Stated explicitly for the single-qubit case: if UU is a Clifford and $P P$ is a Pauli, $UPU^\dagger UPU^\dagger$ will also be a Pauli. For Hermitian gates, like the Hadamard, we can simply use $UPU^\dagger UPU$.

Further examples of single-qubit Clifford gates are the Paulis themselves. These do not transform the Pauli they act on. Instead, they simply assign a phase of -1 to the two that they anticommute with. For example,

$$\begin{aligned} ZXZ &= -Y, \\ ZYZ &= -X, \\ ZZZ &= Z. \end{aligned}$$

You may have noticed that a similar phase also arose in the effect of the $S S$ gate. By combining this with a Pauli, we could make a composite gate that would cancel this phase, and swap XX and YY in a way more similar to the Hadamard's swap of XX and ZZ .

For multiple-qubit Clifford gates, the defining property is that they transform tensor products of Paulis to other tensor products of Paulis. For example, the most prominent two-qubit Clifford gate is the CNOT. The property of this that we will make use of in this chapter is

$$CX_{j,k} (X \otimes 1) CX_{j,k} = X \otimes X.$$

This effectively 'copies' an XX from the control qubit over to the target.

The process of sandwiching a matrix between a unitary and its Hermitian conjugate is known as conjugation by that unitary. This process transforms the eigenstates of the matrix, but leaves the eigenvalues unchanged. The reason why conjugation by Cliffords can transform between Paulis is because all Paulis share the same set of eigenvalues.

Non-Clifford gates

The Clifford gates are very important, but they are not powerful on their own. In order to do any quantum computation, we need gates that are not Cliffords. Three important examples are arbitrary rotations around the three axes of the qubit, $R_x(\theta)$, $R_y(\theta)$ and $R_z(\theta)$.

Let's focus on $R_x(\theta) R_x(\theta)$. As we saw in the last section, any unitary can be expressed in an exponential form using a Hermitian matrix. For this gate, we find

$$R_x(\theta) = e^{i\frac{\theta}{2}X}.$$

The last section also showed us that the unitary and its corresponding Hermitian matrix have the same eigenstates. In this section, we've seen that conjugation by a unitary transforms eigenstates and leaves eigenvalues unchanged. With this in mind, it can be shown that

$$UR_x(\theta)U^\dagger = e^{i\frac{\theta}{2}UXU^\dagger}.$$

By conjugating this rotation by a Clifford, we can therefore transform it to the same rotation around another axis. So even if we didn't have a direct way to perform $R_y(\theta) Ry(\theta)$ and $R_z(\theta) Rz(\theta)$, we could do it with $R_x(\theta) Rx(\theta)$ combined with Clifford gates. This technique of boosting the power of non-Clifford gates by combining them with Clifford gates is one that we make great use of in quantum computing.

Certain examples of these rotations have specific names. Rotations by $\theta = \pi$ $\theta=\pi$ around the x, y and z axes are X, Y and Z, respectively. Rotations by $\theta = \pm\pi/2$ $\theta=\pm\pi/2$ around the z axis are S S and $S^\dagger S^\dagger$, and rotations by $\theta = \pm\pi/4$ $\theta=\pm\pi/4$ around the z axis are T T and $T^\dagger T^\dagger$.

Composite gates

As another example of combining $R_x(\theta) Rx(\theta)$ with Cliffords, let's conjugate it with a CNOT.

$$\begin{aligned} CX_{j,k} (R_x(\theta) \otimes 1) CX_{j,k} &= CX_{j,k} e^{i\frac{\theta}{2}(X \otimes 1)} CX_{j,k} = e^{i\frac{\theta}{2}CX_{j,k}(X \otimes 1)CX_{j,k}} \\ &= e^{i\frac{\theta}{2}X \otimes X} \end{aligned}$$

This transforms our simple, single-qubit rotation into a much more powerful two-qubit gate. This is not just equivalent to performing the same rotation independently on both qubits. Instead, it is a gate capable of generating and manipulating entangled states.

We needn't stop there. We can use the same trick to extend the operation to any number of qubits. All that's needed is more conjugates by the CNOT to keep copying the XX over to new qubits.

Furthermore, we can use single-qubit Cliffords to transform the Pauli on different qubits. For example, in our two-qubit example we could conjugate by S S on the qubit on the left to turn the XX there into a YY:

$$S e^{i\frac{\theta}{2}X \otimes X} S^\dagger = e^{i\frac{\theta}{2}X \otimes Y}.$$

With these techniques, we can make complex entangling operations that act on any arbitrary number of qubits, of the form

$$U = e^{i \frac{\theta}{2} P_{n-1} \otimes P_{n-2} \otimes \dots \otimes P_0}, \quad P_j \in \{I, X, Y, Z\}.$$

This all goes to show that combining the single- and two-qubit Clifford gates with rotations around the x axis gives us a powerful set of possibilities. What's left to demonstrate is that we can use them to do anything.

Proving Universality

What does it mean for a computer to do everything that it could possibly do? This was a question tackled by Alan Turing before we even had a good idea of what a computer was.

To ask this question for our classical computers, and specifically for our standard digital computers, we need to strip away all the screens, speakers and fancy input devices. What we are left with is simply a machine that converts input bit strings into output bit strings. If a device can perform any such conversion, taking any arbitrary set of inputs and converting them to an arbitrarily chosen set of outputs, we call it *universal*.

It turns out that the requirements for universality on these devices are quite reasonable. The gates we needed to perform addition in 'The atoms of computation' are also sufficient to implement any possible computation. In fact, just the classical NAND gate is enough, when combined together in sufficient quantities.

Though our current computers can do everything in theory, some tasks are too resource-intensive in practice. In our study of how to add, we saw that the required resources scaled linearly with the problem size. For example, if we double the number of digits in the numbers, we double the number of small scale additions we need to make.

For many other problems, the required resources scale exponentially with the input size. Factorization is a prominent example. In a recent study [1], a 320-digit number took CPU years to factorize. For numbers that are not much larger, there aren't enough computing resources in the world to tackle them -- even though those same numbers could be added or multiplied on just a smartphone in a much more reasonable time.

Quantum computers will alleviate these problems by achieving universality in a fundamentally different way. As we saw in 'The unique properties of qubits', the variables of quantum computing are not equivalent to those of standard computers. The gates that we use, such as those in the last section, go beyond what is possible for the gates of standard computers. Because of this, we can find ways to achieve results that are otherwise impossible.

So how to define what universality is for a quantum computer? We can do this in a way that mirrors the definition discussed above. Just as digital computers convert sets of input bit strings to sets of output bit strings, unitary operations convert sets of orthogonal input states into orthogonal output states.

As a special case, these states could describe bit strings expressed in quantum form. If we can achieve any unitary, we can therefore achieve universality in the same way as for digital computers.

Another special case is that the input and output states could describe real physical systems. The unitary would then correspond to a time evolution. When expressed in an exponential form using a suitable Hermitian matrix, that matrix would correspond to the Hamiltonian. Achieving any unitary would therefore correspond to simulating any time evolution, and engineering the effects of any Hamiltonian. This is also an important problem that is impractical for classical computers, but is a natural application of quantum computers.

Universality for quantum computers is then simply this: the ability to achieve any desired unitary on any arbitrary number of qubits.

As for classical computers, we will need to split this big job up into manageable chunks. We'll need to find a basic set of gates that will allow us to achieve this. As we'll see, the single- and two-qubit gates of the last section are sufficient for the task.

Suppose we wish to implement the unitary

$$U = e^{i(aX+bZ)},$$

but the only gates we have are $R_x(\theta) = e^{i\frac{\theta}{2}X}$ $Rx(\theta)=ei\theta 2X$ and $R_z(\theta) = e^{i\frac{\theta}{2}Z}$ $Rz(\theta)=ei\theta 2Z$. The best way to solve this problem would be to use Euler angles. But let's instead consider a different method.

The Hermitian matrix in the exponential for UU is simply the sum of those for the $R_x(\theta) Rx(\theta)$ and $R_z(\theta) Rz(\theta)$ rotations. This suggests a naive approach to solving our problem: we could apply $R_z(a) = e^{ibZ}$ $Rz(a)=eibZ$ followed by $R_x(b) = e^{iaX}$ $Rx(b)=eiaX$. Unfortunately, because we are exponentiating matrices that do not commute, this approach will not work.

$$e^{iaX} e^{ibX} \neq e^{i(aX+bZ)}$$

However, we could use the following modified version:

$$U = \lim_{n \rightarrow \infty} (e^{iaX/n} e^{ibZ/n})^n.$$

Here we split UU up into n small slices. For each slice, it is a good approximation to say that

$$e^{iaX/n} e^{ibZ/n} = e^{i(aX+bZ)/n}$$

The error in this approximation scales as $1/n^2$ $1/n^2$. When we combine the n slices, we get an approximation of our target unitary whose error scales as $1/n$ $1/n$. So by simply increasing the number of slices, we can get as close to UU as we need. Other methods of creating the sequence are also possible to get even more accurate versions of our target unitary.

The power of this method is that it can be used in complex cases than just a single qubit. For example, consider the unitary

$$U = e^{i(aX \otimes X \otimes X + bZ \otimes Z \otimes Z)}.$$

We know how to create the unitary $e^{i\frac{\theta}{2}X \otimes X \otimes X} e^{i\theta/2Z \otimes Z \otimes Z}$ from a single qubit $R_x(\theta) R_z(\theta)$ and two controlled-NOTs.

```
qc.cx(0,2)
qc.cx(0,1)
qc.rx(theta,0)
qc.cx(0,1)
qc.cx(0,1)
```



With a few Hadamards, we can do the same for $e^{i\frac{\theta}{2}Z \otimes Z \otimes Z} e^{i\theta/2Z \otimes Z \otimes Z}$.

```
qc.h(0)
qc.h(1)
qc.h(2)
qc.cx(0,2)
qc.cx(0,1)
qc.rx(theta,0)
qc.cx(0,1)
qc.cx(0,1)
qc.h(2)
qc.h(1)
qc.h(0)
```



This gives us the ability to reproduce a small slice of our new, three-qubit UU :

$$e^{iaX \otimes X \otimes X/n} e^{ibZ \otimes Z \otimes Z/n} = e^{i(aX \otimes X \otimes X + bZ \otimes Z \otimes Z)/n}.$$

As before, we can then combine the slices together to get an arbitrarily accurate approximation of UU .

This method continues to work as we increase the number of qubits, and also the number of terms that need simulating. Care must be taken to ensure that the approximation remains accurate, but this can be done in ways that require reasonable resources. Adding extra terms to simulate, or increasing the desired accuracy, only require the complexity of the method to increase polynomially.

Methods of this form can reproduce any unitary $U = e^{iH}$ $U = e^{iH}$ for which H can be expressed as a sum of tensor products of Paulis. Since we have shown previously that all matrices can be

expressed in this way, this is sufficient to show that we can reproduce all unitaries. Though other methods may be better in practice, the main concept to take away from this chapter is that there is certainly a way to reproduce all multi-qubit unitaries using only the basic operations found in Qiskit. Quantum universality can be achieved.

References

- [1] "[Factorization of a 1061-bit number by the Special Number Field Sieve](#)" by Greg Childers.

Basic Circuit Identities

```
from qiskit import *
from qiskit.circuit import Gate
```



When we program quantum computers, our aim is always to build useful quantum circuits from the basic building blocks. But sometimes, we might not have all the basic building blocks we want. In this section, we'll look at how we can transform basic gates into each other, and how to use them to build some gates that are slightly more complex (but still pretty basic).

Many of the techniques discussed in this chapter were first proposed in a paper by Barenco and coauthors in 1995 [1].

Making a controlled-Z from a CNOT

The controlled-Z or `cz` gate is another well-used two-qubit gate. Just as the CNOT applies an `X` to its target qubit whenever its control is in state $|1\rangle$, the controlled-Z applies a `Z` in the same case. In Qasm it can be invoked directly with

```
# a controlled-Z
qc.cz(c,t)
```



where `c` and `t` are the control and target qubits. In IBM Q devices, however, the only kind of two-qubit gate that can be directly applied is the CNOT. We therefore need a way to transform one to the other.

The process for this is quite simple. We know that the Hadamard transforms the states $|0\rangle$ and $|1\rangle$ to the states $|+\rangle$ and $|-\rangle$. We also know that the effect of the `Z` gate on the states $|+\rangle$ and $|-\rangle$ is the same as that for `X` on the state $|0\rangle$ and $|1\rangle$. From this reasoning, or from simply multiplying matrices, we find that

$$\begin{aligned} HXH &= Z, \\ HZH &= X. \end{aligned}$$

The same trick can be used to transform a CNOT into a controlled-Z. All we need to do is precede and follow the CNOT with a Hadamard on the target qubit. This will transform any `X` applied to that qubit into a `Z`.

```
# also a controlled-Z
qc.h(t)
qc.cx(c,t)
qc.h(t)
```



More generally, we can transform a single CNOT into a controlled version of any rotation around the Bloch sphere by an angle π , by simply preceding and following it with the correct rotations. For example, a controlled-Y:

```
# a controlled-Y
qc.sdg(t)
qc.cx(c,t)
qc.s(t)
```



and a controlled-H:

```
# a controlled-H
qc.ry(-pi/4,t)
qc.cx(c,t)
qc.ry(pi/4,t)
```



Swapping qubits

Sometimes we need to move information around in a quantum computer. For some qubit implementations, this could be done by physically moving them. Another option is simply to move the state between two qubits. This is done by the SWAP gate.

```
# swaps states of qubits a and b
qc.swap(a,b)
```



The command above directly invokes this gate, but let's see how we might make it using our standard gate set. For this, we'll need to consider a few examples.

First, we'll look at the case that qubit a is in state $|1\rangle$ and qubit b is in state $|0\rangle$. For this we'll apply the following gates:

```
# swap a 1 from a to b
qc.cx(a,b) # copies 1 from a to b
qc.cx(b,a) # uses the 1 on b to rotate the state of a to 0
```



This has the effect of putting qubit b in state $|1\rangle$ and qubit a in state $|0\rangle$. In this case at least, we have done a SWAP.

Now let's take this state and SWAP back to the original one. As you may have guessed, we can do this with the reverse of the above process:

```
# swap a q from b to a
qc.cx(b,a) # copies 1 from b to a
qc.cx(a,b) # uses the 1 on a to rotate the state of b to 0
```



Note that in these two processes, the first gate of one would have no effect on the initial state of the other. For example, when we swap the $|1\rangle$ b to a, the first gate is `cx q[b], q[a]`. If this were instead applied to a state where no $|1\rangle$ was initially on b, it would have no effect.

Note also that for these two processes, the final gate of one would have no effect on the final state of the other. For example, the final `cx q[b], q[a]` that is required when we swap the $|1\rangle$ from a to b has no effect on the state where the $|1\rangle$ is not on b.

With these observations, we can combine the two processes by adding an ineffective gate from one onto the other. For example,

```
qc.cx(b,a)
qc.cx(a,b)
qc.cx(b,a)
```



We can think of this as a process that swaps a $|1\rangle$ from a to b, but with a useless `qc.cx(b,a)` at the beginning. We can also think of it as a process that swaps a $|1\rangle$ from b to a, but with a useless `qc.cx(b,a)` at the end. Either way, the result is a process that can do the swap both ways around.

It also has the correct effect on the $|00\rangle$ state. This is symmetric, and so swapping the states should have no effect. Since the CNOT gates have no effect when their control qubits are $|0\rangle$, the process correctly does nothing.

The $|11\rangle$ state is also symmetric, and so needs a trivial effect from the swap. In this case, the first CNOT gate in the process above will cause the second to have no effect, and the third undoes the first. Therefore, the whole effect is indeed trivial.

We have thus found a way to decompose SWAP gates into our standard gate set of single-qubit rotations and CNOT gates.

```
# swaps states of qubits a and b  
qc.cx(b,a)  
qc.cx(a,b)  
qc.cx(b,a)
```



It works for the states $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$, as well as for all superpositions of them. It therefore swaps all possible two-qubit states.

The same effect would also result if we changed the order of the CNOT gates:

```
# swaps states of qubits a and b  
qc.cx(a,b)  
qc.cx(b,a)  
qc.cx(a,b)
```



This is an equally valid way to get the SWAP gate.

The derivation used here was very much based on the z basis states, but it could also be done by thinking about what is required to swap qubits in states $|+\rangle$ and $|-\rangle$. The resulting ways of implementing the SWAP gate will be completely equivalent to the ones here.

Making the CNOTs we need from the CNOTs we have

The gates in any quantum computer are driven by the physics of the underlying system. In IBM Q devices, the physics behind CNOTs means that they cannot be directly applied to all possible pairs of qubits. For those pairs for which a CNOT can be applied, it typically has a particular orientation. One specific qubit must act as control, and the other must act as the target, without allowing us to choose.

Changing the direction of a CNOT

Let's deal with the second problem described above: If we have a CNOT with control qubit c and target qubit t, how can we make one for which qubit t acts as the control and qubit c is the target?

This question would be very simple to answer for the controlled-Z. For this gate, it doesn't matter which way around the control and target qubits are.

```
qc.cz(c,t)
```



has exactly the same effect as

```
qc.cz(t,c)
```



This means that we can think of either one as the control, and the other as the target.

To see why this is true, let's remind ourselves of what the Z gate is:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

We can think of this as multiplying the state by -1 , but only when it is $|1\rangle$.

For a controlled-Z gate, the control qubit must be in state $|1\rangle$ for a Z to be applied to the target qubit. Given the above property of Z, this only has an effect when the target is in state $|1\rangle$. We can therefore think of the controlled-Z gate as one that multiplies the state of two qubits by -1 , but only when the state is $|11\rangle$.

This new interpretation is phrased in a perfectly symmetric way, and demonstrates that the labels of 'control' and 'target' are not necessary for this gate.

This property gives us a way to reverse the orientation of a CNOT. We can first turn the CNOT into a controlled-Z by using the method described earlier: placing a Hadamard both before and after on the target qubit.

```
# a cz  
qc.h(t)  
qc.cx(c,t)  
qc.h(t)
```



Then, since we are free to choose which way around to think about a controlled-Z's action, we can choose to think of t as the control and c as the target. We can then transform this controlled-Z into a corresponding CNOT. We just need to place a Hadamard both before and after on the target qubit (which is now qubit c).

```
# a cx with control qubit t and target qubit c  
qc.h(c)  
qc.h(t)  
qc.cx(c,t)  
qc.h(t)  
qc.h(c)
```



And there we have it: we've turned around the CNOT. All that is needed is a Hadamard on both qubits before and after.

The rest of this subsection is dedicated to another explanation of how to turn around a CNOT, with a bit of math (introduced in the 'States for Many Qubits' article of the previous chapter, and the 'Fun with Matrices' article of this chapter), and some different insight. Feel free to skip over it.

Here is another way to write the CNOT gate:

$$CX_{c,t} = |0\rangle\langle 0| \otimes I + |1\rangle\langle 1| \otimes X.$$

Here the $|1\rangle\langle 1|$ ensures that the second term only affects those parts of a superposition for which the control qubit c is in state $|1\rangle$. For those, the effect on the target qubit t is X . The first terms similarly address those parts of the superposition for which the control qubit is in state $|0\rangle$, in which case it leaves the target qubit unaffected.

Now let's do a little math. The X gate has eigenvalues ± 1 for the states $|+\rangle$ and $|-\rangle$. The I gate has an eigenvalue of 1 for all states including $|+\rangle$ and $|-\rangle$. We can thus write them in spectral form as

$$X = |+\rangle\langle +| - |-\rangle\langle -|, \quad I = |+\rangle\langle +| + |-\rangle\langle -|$$

Substituting these into the expression above gives us

$$CX_{c,t} = |0\rangle\langle 0| \otimes |+\rangle\langle +| + |0\rangle\langle 0| \otimes |-\rangle\langle -| + |1\rangle\langle 1| \otimes |+\rangle\langle +| - |1\rangle\langle 1| \otimes |-\rangle\langle -|$$

Using the states $|0\rangle$ and $|1\rangle$, we can write the Z gate in spectral form, and also use an alternative (but completely equivalent) spectral form for I :

$$Z = |0\rangle\langle 0| - |1\rangle\langle 1|, \quad I = |0\rangle\langle 0| + |1\rangle\langle 1|.$$

With these, we can factorize the parts of the CNOT expressed with the $|0\rangle$ and $|1\rangle$ state:

$$CX_{c,t} = I \otimes |+\rangle\langle +| + Z \otimes |-\rangle\langle -|$$

This gives us a whole new way to interpret the effect of the CNOT. The $Z \otimes |-\rangle\langle -|$ term addresses the parts of a superposition for which qubit t is in state $|-\rangle$ and then applies a Z gate to qubit c . The other term similarly does nothing to qubit c when qubit t is in state $|+\rangle$.

In this new interpretation, it is qubit t that acts as the control. It is the $|+\rangle$ and $|-\rangle$ states that decide whether an action is performed, and that action is the gate Z . This sounds like a very different gate to our familiar CNOT, and yet it is the CNOT. These are two equally true descriptions of its effects.

Among the many uses of this property is the method to turn around a CNOT. For example, consider applying a Hadamard to qubit c both before and after this CNOT:

```
h(c)
cx(c,t)
h(c)
```



This transforms the Z in the $Z \otimes |-\rangle\langle -|$ term into an X , and leaves the other term unchanged. The combined effect is then a gate that applies an X to qubit c when qubit t is in state $|-\rangle$. This is halfway to what we are wanting to build.

To complete the process, we can apply a Hadamard both before and after on qubit t. This transforms the $|+\rangle$ and $|-\rangle$ states in each term into $|0\rangle$ and $|1\rangle$. Now we have something that applies an X to qubit c when qubit t is in state $|1\rangle$. This is exactly what we want: a CNOT in reverse, with qubit t as the control and c as the target.

CNOT between distant qubits

Suppose we have a control qubit c and a target qubit t, and we want to do a CNOT gate between them. If this gate is directly possible on a device, we can just do it. If it's only possible to do the CNOT in the wrong direction, we can use the method explained above. But what if qubits c and t are not connected at all?

If qubits c and t are on completely different devices in completely different labs in completely different countries, you may be out of luck. But consider the case where it is possible to do a CNOT between qubit c and an additional qubit a, and it is also possible to do one between qubits a and t. The new qubit can then be used to mediate the interaction between c and t.

One way to do this is with the SWAP gate. We can simply SWAP a and t, do the CNOT between c and a, and then swap a and t back again. The end result is that we have effectively done a CNOT between c and t. The drawback of this method is that it costs a lot of CNOT gates, with six needed to implement the two SWAPs.

Another method is to use the following sequence of gates.

```
# a CNOT between qubits c and t, with no end effect on qubit a
qc.cx(a,t)
qc.cx(c,a)
qc.cx(a,t)
qc.cx(c,a)
```



To see how this works, first consider the case where qubit c is in state $|0\rangle$. The effect of the `cx(c,a)` gates in this case are trivial. This leaves only the two `cx q[a], q[t]` gates, which cancel each other out. The net effect is therefore that nothing happens.

If qubit c is in state $|1\rangle$, things are not quite so simple. The effect of the `cx q(c,a)` gates is to toggle the value of qubit a; it turns any $|0\rangle$ in the state of qubit a into $|1\rangle$ and back again, and vice versa.

This toggle effect affects the action of the two `cx(a,t)` gates. It ensures that whenever one is controlled on a $|0\rangle$ and has trivial effect, the other is controlled on a $|1\rangle$ and applies an X to qubit t. The end effect is that qubit a is left unchanged, but qubit t will always have had an X applied to it.

Putting everything together, this means that an X is applied to qubit t only when qubit c is in state $|1\rangle$. Qubit a is left unaffected. We have therefore engineered a CNOT between qubits c and t. Unlike when using SWAP gates, this required only four CNOT gates to implement.

It is similarly possible to engineer CNOT gates when there is a longer chain of qubits required to connect our desired control and target. The methods described above simply need to be scaled up.

Controlled rotations

We have already seen how to build controlled π rotations from a single CNOT gate. Now we'll look at how to build any controlled rotation.

First, let's consider arbitrary rotations around the y axis. Specifically, consider the following sequence of gates.

```
qc.ry(theta/2,t)
qc.cx(c,t)
qc.ry(-theta/2,t)
qc.cx(c,t)
```



If the control qubit is in state $|0\rangle$, all we have here is a $R_y(\theta/2)$ immediately followed by its inverse, $R_y(-\theta/2)$. The end effect is trivial. If the control qubit is in state $|1\rangle$, however, the `ry(-theta/2)` is effectively preceded and followed by an X gate. This has the effect of flipping the direction of the y rotation and making a second $R_y(\theta/2)$. The net effect in this case is therefore to make a controlled version of the rotation $R_y(\theta)$.

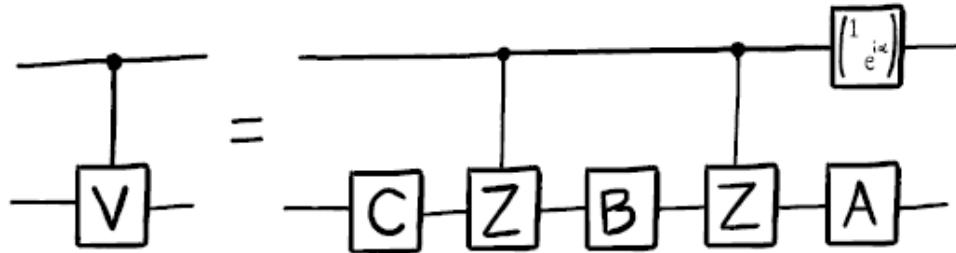
This method works because the x and y axis are orthogonal, which causes the x gates to flip the direction of the rotation. It therefore similarly works to make a controlled $R_z(\theta)$. A controlled $R_x(\theta)$ could similarly be made using CNOT gates.

We can also make a controlled version of any single-qubit rotation, U . For this we simply need to find three rotations A, B and C, and a phase α such that

$$ABC = I, \quad e^{i\alpha} AZBZC = U$$

We then use controlled-Z gates to cause the first of these relations to happen whenever the control is in state $|0\rangle$, and the second to happen when the control is state $|1\rangle$. An $R_z(2\alpha)$ rotation is also used on the control to get the right phase, which will be important whenever there are superposition states.

```
qc.append(a, [t])
qc.cz(c,t)
qc.append(b, [t])
qc.cz(c,t)
qc.append(c, [t])
qc.u1(alpha,c)
```



Here A , B and C are gates that implement A , B and C , respectively, and must be defined as custom gates. For example, if we wanted A to be $R_x(\pi/4)$, the custom would be defined as

```
qc_a = QuantumCircuit(1, name='A')
qc_a.rx(np.pi/4, 0)
A = qc_a.to_instruction()
```



The Toffoli

The Toffoli gate is a three-qubit gate with two controls and one target. It performs an X on the target only if both controls are in the state $|1\rangle$. The final state of the target is then equal to either the AND or the NAND of the two controls, depending on whether the initial state of the target was

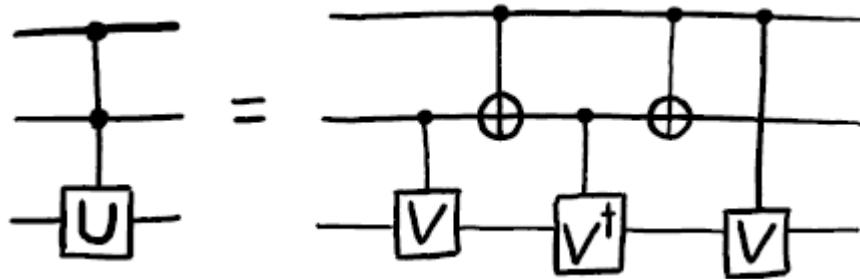
$|0\rangle$ or $|1\rangle$. A Toffoli can also be thought of as a controlled-controlled-NOT, and is also called the CCX gate.

```
# Toffoli with control qubits a and b and target t
qc.ccx(a,b,t)
```

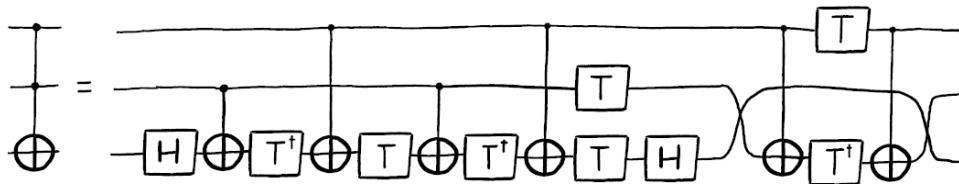


To see how to build it from single- and two-qubit gates, it is helpful to first show how to build something even more general: an arbitrary controlled-controlled-U for any single-qubit rotation U. For this we need to define controlled versions of $V = \sqrt{U}$ and V^\dagger . In the Qasm code below, we assume that subroutines `cv` and `cvdg` have been defined for these, respectively. The controls are qubits a and b, and the target is qubit t.

```
qc.cv(b,t)
qc.cx(a,b)
qc.cvdg(b,t)
qc.cx(a,b)
qc.cv(a,t)
```



By tracing through each value of the two control qubits, you can convince yourself that a U gate is applied to the target qubit if and only if both controls are 1. Using ideas we have already described, you could now implement each controlled-V gate to arrive at some circuit for the doubly-controlled-U gate. It turns out that the minimum number of CNOT gates required to implement the Toffoli gate is six [2].



The Toffoli is not the unique way to implement an AND gate in quantum computing. We could also define other gates that have the same effect, but which also introduce relative phases. In these cases, we can implement the gate with fewer CNOTs.

For example, suppose we use both the controlled-Hadamard and controlled-Z gates, which can both be implemented with a single CNOT. With these we can make the following circuit:

```
qc.ch(a,t)
qc.cz(b,t)
qc.ch(a,t)
```



For the state $|00\rangle$ on the two controls, this does nothing to the target. For $|11\rangle$, the target experiences a Z gate that is both preceded and followed by an H. The net effect is an X on the target. For the states $|01\rangle$ and $|10\rangle$, the target experiences either just the two Hadamards (which cancel each other out) or just the Z (which only induces a relative phase). This therefore also reproduces the effect of an AND, because the value of the target is only changed for the $|11\rangle$ state on the controls -- but it does it with the equivalent of just three CNOT gates.

Arbitrary rotations from H and T

The qubits in current devices are subject to noise, which basically consists of gates that are done by mistake. Simple things like temperature, stray magnetic fields or activity on neighboring qubits can make things happen that we didn't intend.

For large applications of quantum computers, it will be necessary to encode our qubits in a way that protects them from this noise. This is done by making gates much harder to do by mistake, or to implement in a manner that is slightly wrong.

This is unfortunate for the single-qubit rotations $R_x(\theta)$, $R_y(\theta)$ and $R_z(\theta)$. It is impossible to implement an angle θ with perfect accuracy, such that you are sure that you are not accidentally implementing something like $\theta + 0.0000001$. There will always be a limit to the accuracy we can achieve, and it will always be larger than is tolerable when we account for the build-up of imperfections over large circuits. We will therefore not be able to implement these rotations directly in fault-tolerant quantum computers, but will instead need to build them in a much more deliberate manner.

Fault-tolerant schemes typically perform these rotations using multiple applications of just two gates: H and T.

The T gate is expressed in Qasm as

```
qc.t(0) # T gate on qubit 0
```



It is a rotation around the z axis by $\theta = \pi/4$, and so is expressed mathematically as
 $R_z(\pi/4) = e^{i\pi/8 Z}$.

In the following we assume that the H and T gates are effectively perfect. This can be engineered by suitable methods for error correction and fault-tolerance.

Using the Hadamard and the methods discussed in the last chapter, we can use the T gate to create a similar rotation around the x axis.

```
qc.h(0)
qc.t(0)
qc.h(0)
```



Now let's put the two together. Let's make the gate $R_z(\pi/4) R_x(\pi/4)$.

```
qc.h(0)
qc.t(0)
qc.h(0)
qc.t(0)
```



Since this is a single-qubit gate, we can think of it as a rotation around the Bloch sphere. That means that it is a rotation around some axis by some angle. We don't need to think about the axis too much here, but it clearly won't be simply x, y or z. More important is the angle.

The crucial property of the angle for this rotation is that it is irrational. You can prove this yourself with a bunch of math, but you can also see the irrationality in action by applying the gate.

Repeating it n times results in a rotation around the same axis by a different angle. Due to the irrationality, the angles that result from different repetitions will never be the same.

We can use this to our advantage. Each angle will be somewhere between 0 and 2π . Let's split this interval up into n slices of width $2\pi/n$. For each repetition, the resulting angle will fall in one of these slices. If we look at the angles for the first $n + 1$ repetitions, it must be true that at least one slice contains two of these angles. Let's use n_1 to denote the number of repetitions required for the first, and n_2 for the second.

With this, we can prove something about the angle for $n_2 - n_1$ repetitions. This is effectively the same as doing n_2 repetitions, followed by the inverse of n_1 repetitions. Since the angles for these are not equal (because of the irrationality) but also differ by no greater than $2\pi/n$ (because they correspond to the same slice), the angle for $n_2 - n_1$ repetitions satisfies

$$\theta_{n_2-n_1} \neq 0, \quad -\frac{2\pi}{n} \leq \theta_{n_2-n_1} \leq \frac{2\pi}{n}.$$

We therefore have the ability to do rotations around small angles. We can use this to rotate around angles that are as small as we like, just by increasing the number of times we repeat this gate.

By using many small-angle rotations, we can also rotate by any angle we like. This won't always be exact, but it is guaranteed to be accurate up to $2\pi/n$, which can be made as small as we like. We now have power over the inaccuracies in our rotations.

So far, we only have the power to do these arbitrary rotations around one axis. For a second axis, we simply do the $R_z(\pi/4)$ and $R_x(\pi/4)$ rotations in the opposite order.

```
qc.h(θ)
qc.t(θ)
qc.h(θ)
qc.t(θ)
```



The axis that corresponds to this rotation is not the same as that for the gate considered previously. We therefore now have arbitrary rotation around two axes, which can be used to generate any arbitrary rotation around the Bloch sphere. We are back to being able to do everything, though it costs quite a lot of T gates.

It is because of this kind of application that T gates are so prominent in quantum computation. In fact, the complexity of algorithms for fault-tolerant quantum computers is often quoted in terms of how many T gates they'll need. This motivates the quest to achieve things with as few T gates as possible. Note that the discussion above was simply intended to prove that T gates can be used in this way, and does not represent the most efficient method we know.

References

[1] Barenco, et al. 1995

[2] Shende and Markov, 2009

Classical Logic Gates with Quantum Circuits

```
from qiskit import *
from qiskit.tools.visualization import plot_histogram
import numpy as np
```



Using the NOT gate (expressed as `x` in Qiskit), the CNOT gate (expressed as `cx` in Qiskit) and the Toffoli gate (expressed as `ccx` in Qiskit) create functions to implement the XOR, AND, NAND and OR gates.

An implementation of the NOT gate is provided as an example.

NOT gate

This function takes a binary string input ('0' or '1') and returns the opposite binary output'.

```
def NOT(input):
    q = QuantumRegister(1) # a qubit in which to encode and manipulate the input
    c = ClassicalRegister(1) # a bit to store the output
    qc = QuantumCircuit(q, c) # this is where the quantum program goes

    # We encode '0' as the qubit state |0>, and '1' as |1>
    # Since the qubit is initially |0>, we don't need to do anything for an input of '0'
    # For an input of '1', we do an x to rotate the |0> to |1>
    if input=='1':
        qc.x( q[0] )

    # Now we've encoded the input, we can do a NOT on it using x
    qc.x( q[0] )

    # Finally, we extract the |0>/|1> output of the qubit and encode it in the bit c[0]
    qc.measure( q[0], c[0] )

    # We'll run the program on a simulator
    backend = Aer.get_backend('qasm_simulator')
    # Since the output will be deterministic, we can use just a single shot to get it
```



```
job = execute(qc,backend,shots=1)
output = next(iter(job.result().get_counts()))

return output
```

XOR gate

Takes two binary strings as input and gives one as output.

The output is '0' when the inputs are equal and '1' otherwise.

```
def XOR(input1,input2):

    q = QuantumRegister(2) # two qubits in which to encode and manipulate the input
    c = ClassicalRegister(1) # a bit to store the output
    qc = QuantumCircuit(q, c) # this is where the quantum program goes

    # YOUR QUANTUM PROGRAM GOES HERE
    qc.measure(q[1],c[0]) # YOU CAN CHANGE THIS IF YOU WANT TO

    # We'll run the program on a simulator
    backend = Aer.get_backend('qasm_simulator')
    # Since the output will be deterministic, we can use just a single shot to get it
    job = execute(qc,backend,shots=1,memory=True)
    output = job.result().get_memory()[0]

return output
```

AND gate

Takes two binary strings as input and gives one as output.

The output is '1' only when both the inputs are '1' .

```
def AND(input1,input2):

    q = QuantumRegister(3) # two qubits in which to encode the input, and one for the o
    c = ClassicalRegister(1) # a bit to store the output
    qc = QuantumCircuit(q, c) # this is where the quantum program goes
```

```

# YOUR QUANTUM PROGRAM GOES HERE
qc.measure(q[2],c[0]) # YOU CAN CHANGE THIS IF YOU WANT TO

# We'll run the program on a simulator
backend = Aer.get_backend('qasm_simulator')
# Since the output will be deterministic, we can use just a single shot to get it
job = execute(qc,backend,shots=1,memory=True)
output = job.result().get_memory()[0]

return output

```

NAND gate

Takes two binary strings as input and gives one as output.

The output is '0' only when both the inputs are '1' .

```

def NAND(input1,input2):
    q = QuantumRegister(3) # two qubits in which to encode the input, and one for the output
    c = ClassicalRegister(1) # a bit to store the output
    qc = QuantumCircuit(q, c) # this is where the quantum program goes

    # YOUR QUANTUM PROGRAM GOES HERE
    qc.measure(q[2],c[0]) # YOU CAN CHANGE THIS IF YOU WANT TO

    # We'll run the program on a simulator
    backend = Aer.get_backend('qasm_simulator')
    # Since the output will be deterministic, we can use just a single shot to get it
    job = execute(qc,backend,shots=1,memory=True)
    output = job.result().get_memory()[0]

return output

```

OR gate

Takes two binary strings as input and gives one as output.

The output is '1' if either input is '1'.

```
def OR(input1,input2):  
  
    q = QuantumRegister(3) # two qubits in which to encode the input, and one for the o  
    c = ClassicalRegister(1) # a bit to store the output  
    qc = QuantumCircuit(q, c) # this is where the quantum program goes  
  
    # YOUR QUANTUM PROGRAM GOES HERE  
    qc.measure(q[2],c[0]) # YOU CAN CHANGE THIS IF YOU WANT TO  
  
    # We'll run the program on a simulator  
    backend = Aer.get_backend('qasm_simulator')  
    # Since the output will be deterministic, we can use just a single shot to get it  
    job = execute(qc,backend,shots=1,memory=True)  
    output = job.result().get_memory()[0]  
  
    return output
```

Tests

The following code runs the functions above for all possible inputs, so that you can check whether they work.

```
print('\nResults for the NOT gate')  
for input in ['0','1']:  
    print('    Input',input,'gives output',NOT(input))  
  
print('\nResults for the XOR gate')  
for input1 in ['0','1']:  
    for input2 in ['0','1']:  
        print('    Inputs',input1,input2,'give output',XOR(input1,input2))  
  
print('\nResults for the AND gate')  
for input1 in ['0','1']:  
    for input2 in ['0','1']:  
        print('    Inputs',input1,input2,'give output',AND(input1,input2))  
  
print('\nResults for the NAND gate')  
for input1 in ['0','1']:  
    for input2 in ['0','1']:  
        print('    Inputs',input1,input2,'give output',NAND(input1,input2))
```

```
print('\nResults for the OR gate')
for input1 in ['0','1']:
    for input2 in ['0','1']:
        print('    Inputs',input1,input2,'give output',OR(input1,input2))
```

Results for the NOT gate

Input 0 gives output 1
Input 1 gives output 0

Results for the XOR gate

Inputs 0 0 give output 0
Inputs 0 1 give output 0
Inputs 1 0 give output 0
Inputs 1 1 give output 0

Results for the AND gate

Inputs 0 0 give output 0
Inputs 0 1 give output 0
Inputs 1 0 give output 0
Inputs 1 1 give output 0

Results for the NAND gate

Inputs 0 0 give output 0
Inputs 0 1 give output 0
Inputs 1 0 give output 0
Inputs 1 1 give output 0

Results for the OR gate

Inputs 0 0 give output 0
Inputs 0 1 give output 0
Inputs 1 0 give output 0
Inputs 1 1 give output 0

Basic Synthesis of Single-Qubit Gates

```
from qiskit import *
from qiskit.tools.visualization import plot_histogram
import numpy as np
```



1

Show that the Hadamard gate can be written in the following two forms

$$H = \frac{X+Z}{\sqrt{2}} \equiv \exp(i\frac{\pi}{2} \frac{X+Z}{\sqrt{2}}).$$

Here \equiv is used to denote that the equality is valid up to a global phase, and hence that the resulting gates are physically equivalent.

Hint: it might even be easiest to prove that $e^{i\frac{\pi}{2}M} \equiv M e^{i\pi/2} M^{-1}$ for any matrix whose eigenvalues are all ± 1 , and that such matrices uniquely satisfy $M^2 = I$.

2

The Hadamard can be constructed from `rx` and `rz` operations as

$$R_x(\theta) = e^{i\frac{\theta}{2}X}, \quad R_z(\theta) = e^{i\frac{\theta}{2}Z},$$
$$H \equiv \lim_{n \rightarrow \infty} \left(R_x\left(\frac{\theta}{n}\right) R_z\left(\frac{\theta}{n}\right) \right)^n.$$

For some suitably chosen θ . When implemented for finite n , the resulting gate will be an approximation to the Hadamard whose error decreases with n .

The following shows an example of this implemented with Qiskit with an incorrectly chosen value of θ (and with the global phase ignored).

- Determine the correct value of θ .
- Show that the error (when using the correct value of θ) decreases quadratically with n .



```
q = QuantumRegister(1)
c = ClassicalRegister(1)

error = {}
for n in range(1,11):

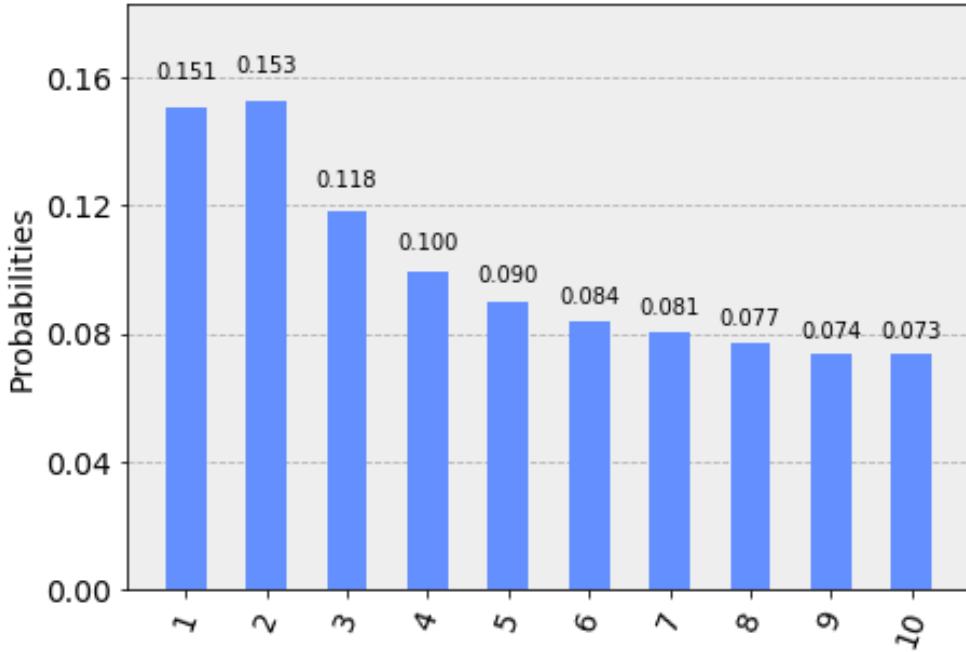
    # Create a blank circuit
    qc = QuantumCircuit(q,c)

    # Implement an approximate Hadamard
    theta = np.pi # here we incorrectly choose theta=pi
    for j in range(n):
        qc.rx(theta/n,q[0])
        qc.rz(theta/n,q[0])

    # We need to measure how good the above approximation is. Here's a simple way to do
    # Step 1: Use a real hadamard to cancel the above approximation.
    # For a good approximatuon, the qubit will return to state 0. For a bad one, it wil
    qc.h(q[0])

    # Step 2: Run the circuit, and see how many times we get the outcome 1.
    # Since it should return 0 with certainty, the fraction of 1s is a measure of the e
    qc.measure(q,c)
    shots = 2000
    job = execute(qc, Aer.get_backend('qasm_simulator'),shots=shots)
    try:
        error[n] = (job.result().get_counts())['1']/shots
    except:
        pass

plot_histogram(error)
```



3

An improved version of the approximation can be found from,

$$H \equiv \lim_{n \rightarrow \infty} \left(R_z \left(\frac{\theta}{2n} \right) R_x \left(\frac{\theta}{n} \right) R_z \left(\frac{\theta}{2n} \right) \right)^n.$$

Implement this, and investigate the scaling of the error.

Building the Best AND Gate

```
from qiskit import *
from qiskit.tools.visualization import plot_histogram
from qiskit.providers.aer import noise
import numpy as np
```



In Problem Set 1, you made an AND gate with quantum gates. This time you'll do the same again, but for a real device. Using real devices gives you two major constraints to deal with. One is the connectivity, and the other is noise.

The connectivity tells you what `cx` gates it is possible to perform directly. For example, the device `ibmq_5_tenerife` has five qubits numbered from 0 to 4. It has a connectivity defined by

```
coupling_map = [[1, 0], [2, 0], [2, 1], [3, 2], [3, 4], [4, 2]]
```



Here the `[1,0]` tells us that we can implement a `cx` with qubit 1 as control and qubit 0 as target, the `[2,0]` tells us we can have qubit 2 as control and 0 as target, and so on. These are the `cx` gates that the device can implement directly.

The 'noise' of a device is the collective effects of all the things that shouldn't happen, but nevertheless do happen. Noise results in the output not always having the result we expect. There is noise associated with all processes in a quantum circuit: preparing the initial states, applying gates and measuring the output. For the gates, noise levels can vary between different gates and between different qubits. The `cx` gates are typically more noisy than any single qubit gate.

We can also simulate noise using a noise model. And we can set the noise model based on measurements of the noise for a real device. The following noise model is based on `ibmq_5_tenerife`.

```
noise_dict = {'errors': [{'type': 'qerror', 'operations': ['u2'], 'instructions': [[...]]}, ...]}
noise_model = noise.noise_model.NoiseModel.from_dict(noise_dict)
```



Running directly on the device requires you to have an IBMQ account, and for you to sign in to it within your program. In order to not worry about all this, we'll instead use a simulation of the 5

qubit device defined by the constraints set above.

```
qr = QuantumRegister(5, 'qr')
cr = ClassicalRegister(1, 'cr')
backend = Aer.get_backend('qasm_simulator')
```

We now define the `AND` function. This has a few differences to the version in Exercise 1. Firstly, it is defined on a 5 qubit circuit, so you'll need to decide which of the 5 qubits are used to encode `input1`, `input2` and the output. Secondly, the output is a histogram of the number of times that each output is found when the process is repeated over 10000 samples.

```
def AND (input1,input2, q_1=0,q_2=1,q_out=2):
    # The keyword q_1 specifies the qubit used to encode input1
    # The keyword q_2 specifies     qubit used to encode input2
    # The keyword q_out specifies     qubit to be as output

    qc = QuantumCircuit(qr, cr)

    # prepare input on qubits q1 and q2
    if input1=='1':
        qc.x( qr[ q_1 ] )
    if input2=='1':
        qc.x( qr[ q_2 ] )

    qc.ccx(qr[ q_1 ],qr[ q_2 ],qr[ q_out ]) # the AND just needs a c
    qc.measure(qr[ q_out ],cr[0]) # output from qubit 1 is measured

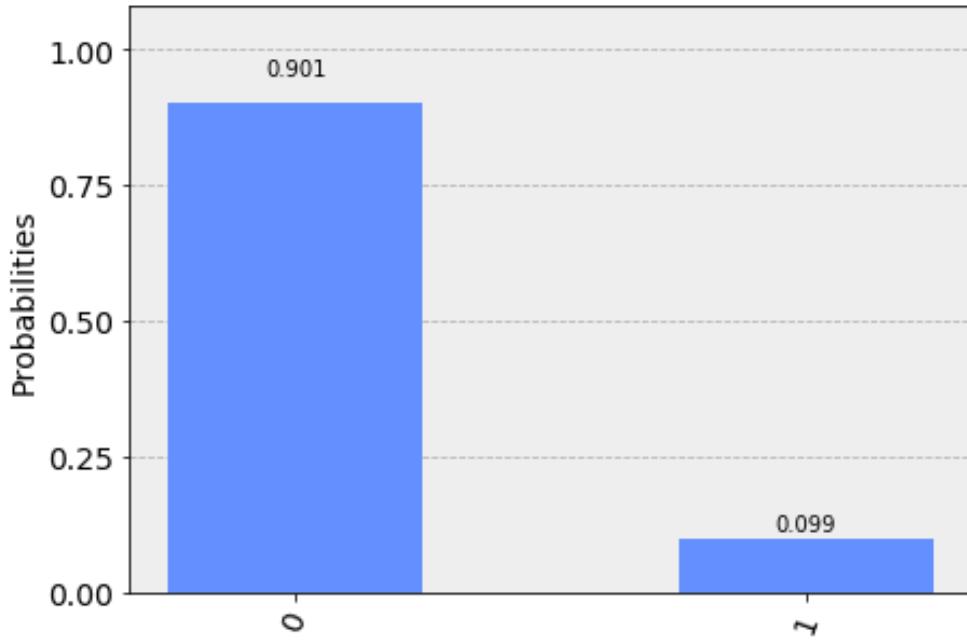
    # the circuit is run on a simulator, but we do it so that the noise and connectivit
    job = execute(qc, backend, shots=10000, noise_model=noise_model,
                  coupling_map=coupling_map,
                  basis_gates=noise_model.basis_gates)
    output = job.result().get_counts()

    return output
```

For example, here are the results when both inputs are `0`.

```
result = AND('0','0')
print( result )
plot_histogram( result )
```

```
{'1': 991, '0': 9009}
```



We'll compare across all results to find the most unreliable.

```
worst = 1
for input1 in ['0','1']:
    for input2 in ['0','1']:
        print('\nProbability of correct answer for inputs',input1,input2)
        prob = AND(input1,input2, q_1=0,q_2=1,q_out=2)[str(int( input1=='1' and input2==1))]
        print( prob )
        worst = min(worst,prob)
print('\nThe lowest of these probabilities was',worst)
```



Probability of correct answer for inputs 0 0
0.9035

Probability of correct answer for inputs 0 1
0.8964

Probability of correct answer for inputs 1 0
0.8972

Probability of correct answer for inputs 1 1
0.9019

The lowest of these probabilities was 0.8964

The AND function above uses the `ccx` gate to implement the required operation. But you now know how to make your own. Find a way to implement an AND for which the lowest of the above probabilities is better than for a simple `ccx`.

Quantum Teleportation

This notebook demonstrates quantum teleportation. We first use Qiskit's built-in simulator to test our quantum circuit, and then try it out on a real quantum computer.

The concept

Alice wants to send quantum information to Bob. Specifically, suppose she wants to send the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ to Bob. This entails passing on information about $\alpha\alpha$ and $\beta\beta$ to Bob.

There exists a theorem in quantum mechanics which states that you cannot simply make an exact copy of an unknown quantum state. This is known as the no-cloning theorem. As a result of this we can see that Alice can't simply generate a copy of $|\psi\rangle|\psi\rangle$ and give the copy to Bob. Copying a state is only possible with a classical computation.

However, by taking advantage of two classical bits and entanglement, Alice can transfer the state $|\psi\rangle|\psi\rangle$ to Bob. We call this teleportation as at the end Bob will have $|\psi\rangle|\psi\rangle$ and Alice won't anymore. Let's see how this works in some detail.

How does quantum teleportation work?

Step 1: Alice and Bob create an entangled pair of qubits and each one of them holds on to one of the two qubits in the pair.

The pair they create is a special pair called a Bell pair. In quantum circuit language, the way to create a Bell pair between two qubits is to first transfer one of them to the Bell basis ($|+\rangle|+\rangle$ and $|-\rangle|-\rangle$) by using a Hadamard gate, and then to apply a CNOT gate onto the other qubit controlled by the one in the Bell basis.

Let's say Alice owns q_1 and Bob owns q_2 after they part ways.

Step 2: Alice applies a CNOT gate on q_1 , controlled by $|\psi\rangle|\psi\rangle$ (the qubit she is trying to send Bob).

Step 3: Next, Alice applies a Hadamard gate to $|\psi\rangle|\psi\rangle$, and applies a measurement to both qubits that she owns - q_1 q_1 and $|\psi\rangle|\psi\rangle$.

Step 4: Then, it's time for a phone call to Bob. She tells Bob the outcome of her two qubit measurement. Depending on what she says, Bob applies some gates to his qubit, q_2 q_2 . The gates to be applied, based on what Alice says, are as follows :

$00 \rightarrow$ Do nothing

$01 \rightarrow$ Apply XX gate

$10 \rightarrow$ Apply ZZ gate

$11 \rightarrow$ Apply $ZX ZX$ gate

Note that this transfer of information is classical.

And voila! At the end of this protocol, Alice's qubit has now teleported to Bob.

How will we test this result on a real quantum computer?

In this notebook, we will give Alice a secret state $|\psi\rangle|\psi\rangle$. This state will be generated by applying a series of unitary gates on a qubit that is initialized to the ground state, $|0\rangle|0\rangle$. Go ahead and fill in the secret unitary that will be applied to $|0\rangle|0\rangle$ before passing on the qubit to Alice.

```
secret_unitary = 'hz'
```



If the quantum teleportation circuit works, then at the output of the protocol discussed above will be the same state passed on to Alice. Then, we can undo the applied `secret_unitary` (by applying its conjugate transpose), to yield the $|0\rangle|0\rangle$ that we started with.

We will then do repeated measurements of Bob's qubit to see how many times it gives 0 and how many times it gives 1.

What do we expect?

In the ideal case, and assuming our teleportation protocol works, we will always measure 0 from Bob's qubit because we started off with $|0\rangle|0\rangle$.

In a real quantum computer, errors in the gates will cause a small fraction of the results to be 1. We'll see how it looks.

1. Simulating the teleportation protocol

```
# make the imports that are necessary for our work
import qiskit as qk
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit
from qiskit import execute, Aer
from qiskit import IBMQ
from qiskit.tools.visualization import plot_histogram
```

```
# simple function that applies a series of unitary gates from a given string
def apply_secret_unitary(secret_unitary, qubit, quantum_circuit, dagger):
    functionmap = {
        'x':quantum_circuit.x,
        'y':quantum_circuit.y,
        'z':quantum_circuit.z,
        'h':quantum_circuit.h,
        't':quantum_circuit.t,
    }
    if dagger: functionmap['t'] = quantum_circuit.tdg

    if dagger:
        [functionmap[unitary](qubit) for unitary in secret_unitary]
    else:
        [functionmap[unitary](qubit) for unitary in secret_unitary[::-1]]
```

```
# Create the quantum circuit
q = QuantumRegister(3)
c = ClassicalRegister(3)
qc = QuantumCircuit(q, c)

''' Qubit ordering as follows (classical registers will just contain measured values of
q[0]: qubit to be teleported (Alice's first qubit. It was given to her after the applic
which she doesn't know)
q[1]: Alice's second qubit
q[2]: Bob's qubit, which will be the destination for the teleportation
'''

# Apply the secret unitary that we are using to generate the state to teleport. You can
apply_secret_unitary(secret_unitary, q[0], qc, dagger = 0)
```

```

qc.barrier()
# Next, generate the entangled pair between Alice and Bob (Remember: Hadamard followed
qc.h(q[1])
qc.cx(q[1], q[2])
qc.barrier()
# Next, apply the teleportation protocol.
qc.cx(q[0], q[1])
qc.h(q[0])
qc.measure(q[0], c[0])
qc.measure(q[1], c[1])
qc.cx(q[1], q[2])
qc.cz(q[0], q[2])
qc.barrier()

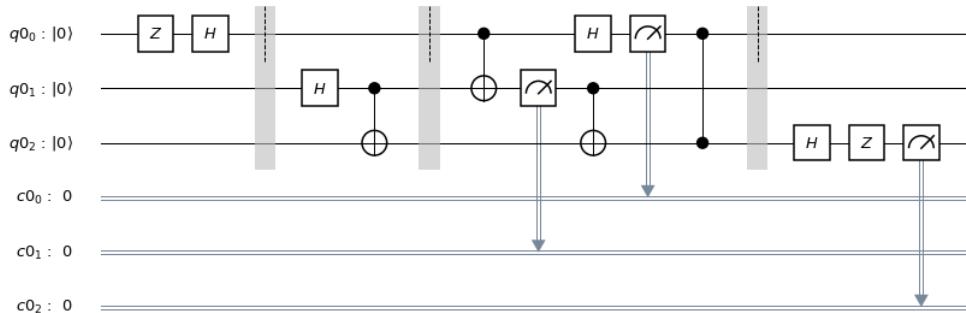
...
In principle, if the teleportation protocol worked, we have q[2] = secret_unitary|0>
As a result, we should be able to recover q[2] = |0> by applying the reverse of secret_
since for a unitary u, u^dagger u = I.
...
apply_secret_unitary(secret_unitary, q[2], qc, dagger=1)
qc.measure(q[2], c[2])

```

```
<qiskit.circuit.measure.Measure at 0x117a3a668>
```

It's always a good idea to draw the circuit that we have generated in code. Let's draw it below.

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



```

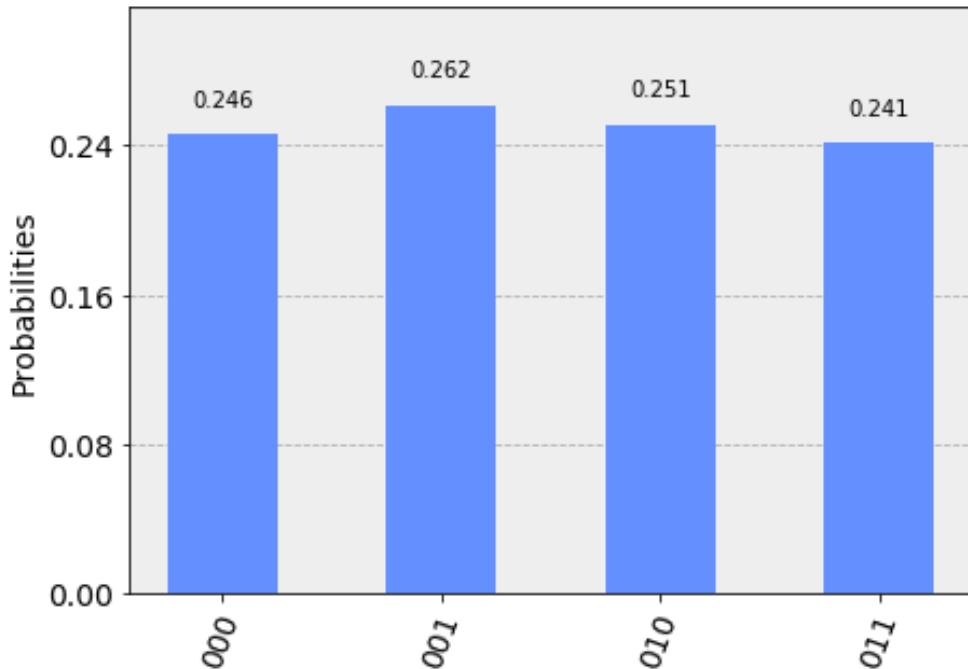
backend = Aer.get_backend('qasm_simulator')
job_sim = execute(qc, backend, shots=1024)
sim_result = job_sim.result()

measurement_result = sim_result.get_counts(qc)

```

```
print(measurement_result)
plot_histogram(measurement_result)
```

```
{'000': 252, '001': 268, '011': 247, '010': 257}
```



Note that the results on the x-axis in the histogram above are ordered as $c_2 c_1 c_0 c_2 c_1 c_0$. We can see that only results where $c_2 = 0$ appear, indicating that the teleportation protocol has worked.

2. Teleportation on a real quantum computer

You will now see how the teleportation algorithm works on a real quantum computer. Recall that we need one qubit for $|\psi\rangle |\psi\rangle$, one qubit for Alice, and one qubit for Bob, for a total of three qubits.

```
# First, see what devices we are allowed to use by loading our saved accounts
IBMQ.load_account()
provider = IBMQ.get_provider(hub='ibm-q')
provider.backends()
```



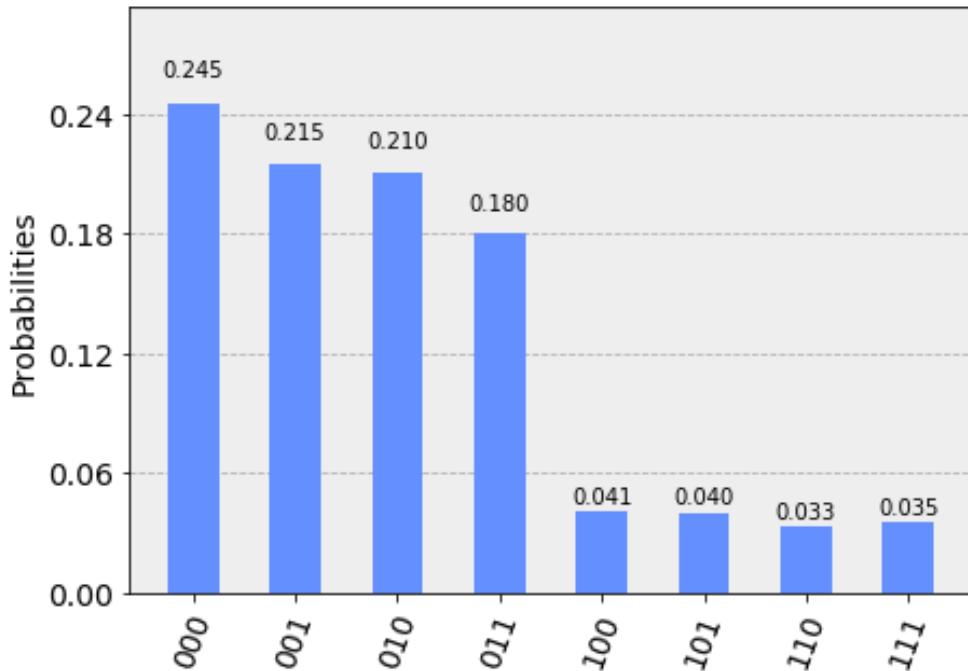
```
[<IBMQBackend('ibmqx4') from IBMQ(),>
 <IBMQBackend('ibmqx2') from IBMQ(),>
 <IBMQBackend('ibmq_16_melbourne') from IBMQ(),>
 <IBMQBackend('ibmq_qasm_simulator') from IBMQ(),>
 <IBMQBackend('ibmq_20_tokyo') from IBMQ(ibm-q-internal, yrk, main),>,
```

```
<IBMQBackend('ibmq_poughkeepsie') from IBMQ(ibm-q-internal, yrk, main)>,
<IBMQBackend('ibmq_qasm_simulator') from IBMQ(ibm-q-internal, yrk, main)>]
```

```
# get the Least-busy backend at IBM and run the quantum circuit there
from qiskit.providers.ibmq import least_busy
backend = least_busy(provider.backends(simulator=False))
job_exp = execute(qc, backend=backend, shots=8192)
exp_result = job_exp.result()

exp_measurement_result = exp_result.get_counts(qc)
print(exp_measurement_result)
plot_histogram(exp_measurement_result)
```

```
{'000': 2005, '001': 1761, '011': 1476, '101': 330, '111': 290, '010': 1724, '110': 211, '100': 1500}
```



As we see here, there are a few results that contain the case when $c_2 = 1$ in a real quantum computer. These arise due to errors in the gates that were applied. Another source of error is the way we're checking for teleportation - we need the series of operators on $q_2 q_2$ to be exactly the inverse unitary of those that we applied to $q_0 q_0$ at the beginning.

In contrast, our simulator in the earlier part of the notebook had zero errors in its gates, and allowed error-free teleportation.

```
error_rate_percent = sum([exp_measurement_result[result] for result in exp_measurement_result]) * 100./ sum(list(exp_measurement_result.values()))
```

```
print("The experimental error rate : ", error_rate_percent, "%")
```

```
14.9658203125
```

Deutsch-Josza Algorithm

In this section, we first introduce the Deutsch-Josza problem, and classical and quantum algorithms to solve it. We then implement the quantum algorithm using Qiskit, and run on a simulator and device.

Contents

1. Introduction

- o [Deutsch-Josza Problem](#)
- o [Deutsch-Josza Algorithm](#)

2. Example

3. Qiskit Implementation

- o [Simulation](#)
- o [Device](#)

4. Problems

5. References

1. Introduction

The Deutsch-Josza algorithm, first introduced in Reference [1], was the first example of a quantum algorithm that performs better than the best classical algorithm. It showed that there can be advantages in using a quantum computer as a computational tool for a specific problem.

1a. Deutsch-Josza Problem

We are given a hidden Boolean function f , which takes as input a string of bits, and returns either 0 or 1, that is

$$f \quad 0 \quad 1$$

$$f(\{x_0, x_1, x_2, \dots\}) \rightarrow 0 \text{ or } 1, \text{ where } x_n \text{ is } 0 \text{ or } 1.$$

$$f(\{x_0, x_1, x_2, \dots\}) \rightarrow 0 \text{ or } 1, \text{ where } x_n$$

The property of the given Boolean function is that it is guaranteed to either be balanced or constant. A constant function returns all 0's or all 1's for any input, while a balanced function returns 0's for exactly half of all inputs and 1's for the other half. Our task is to determine whether the given function is balanced or constant.

1

Note that the Deutsch-Josza problem is an n -bit extension of the single bit Deutsch problem.

n

1b. Deutsch-Josza Algorithm

Classical Solution

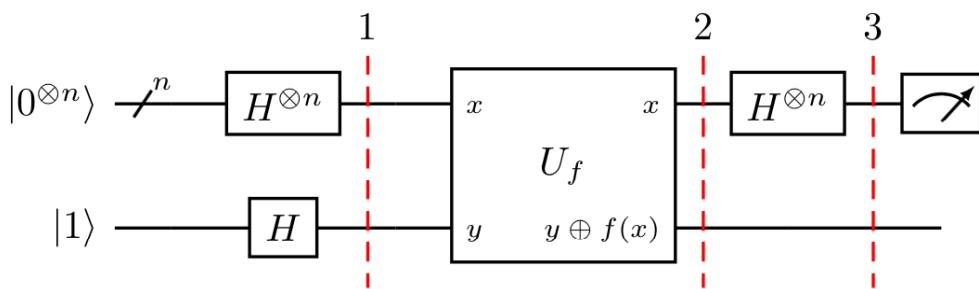
Classically, in the best case, two queries to the oracle can determine if the hidden Boolean function, $f(x)$, is balanced: e.g. if we get both $f(0, 0, 0, \dots \rightarrow 0)$ and $f(1, 0, 0, \dots \rightarrow 1)$ we know the function is balanced as we have obtained the two different outputs.

In the worst case, if we continue to see the same output for each input we try, we will have to check exactly $2^{n-1} + 1$ inputs to be certain that $f(x)$ is constant: e.g. for a 4-bit string, if we checked 8 out of the 16 possible combinations, getting all 0's, it is still possible that the 9th input returns a 1 and $f(x)$ is balanced. Probabilistically, this is a very unlikely event. In fact, if we get the same result continually in k successive calls to $f(x)$, we can express the probability that the function is constant as a function of k inputs as: $P_{\text{constant}}(k) = 1 - \frac{1}{2^{k-1}}$ for $k \leq 2^{n-1}$

Realistically, we could opt to terminate our classical algorithm early, say if we were over $x\%$ confident. But if we want to be 100% confident, we would need to check $2^{n-1} + 1$ inputs.

Quantum Solution

Using a quantum computer, we can solve this problem with 100% confidence after only one call to the function $f(x)$, provided we have the function f implemented as a quantum oracle, which maps the state $|x\rangle|y\rangle$ to $|x\rangle|y \oplus f(x)\rangle$, where \oplus is addition modulo 2. Below is the generic circuit for the Deutsh-Josza algorithm.



Now, let's go through the steps of the algorithm:

1. Prepare two quantum registers. The first is an n -qubit register initialised to $|0\rangle$, and the second is a one-qubit register initialised to $|1\rangle$: $|\psi_0\rangle = |0\rangle^{\otimes n}|1\rangle^n|0\rangle$

$$|\psi_0\rangle = |0\rangle^{\otimes n}|1\rangle$$

$$2. \text{ Apply a Hadamard gate to each qubit: } |\psi_1\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} |x\rangle |0\rangle - |1\rangle$$

$$3. \text{ Apply the quantum oracle } |x\rangle|y\rangle \text{ to } |x\rangle|y \oplus f(x)\rangle: |\psi_2\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} |x\rangle(|f(x)\rangle$$

$$- |1 \oplus f(x)\rangle) = \frac{1}{\sqrt{2^{n+1}}} \sum_{x=0}^{2^n-1} (-1)^{f(x)} |x\rangle(|0\rangle - |1\rangle)$$

since each $x, f(x)$ is either 0 or 1.

4. At this point the second single qubit register may be ignored. Apply a Hadamard gate to each qubit in the first

$$\text{register: } |\psi_3\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} (-1)^{f(x)} \left[\sum_{y=0}^{2^n-1} (-1)^{x \cdot y} |y\rangle \right] = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} (-1)^{f(x)}$$

$$(-1)^{x \cdot y} |y\rangle$$

where $x \cdot y = x_0 y_0 \oplus x_1 y_1 \oplus \dots \oplus x_{n-1} y_{n-1}$ is the sum of the bitwise product.

5. Measure the first register. Notice that the probability of measuring $|0\rangle^{\otimes n} = |\sum_{x=0}^{2^n-1} (-1)^{f(x)}|^2$, which evaluates to 1 if $f(x)$ is constant and 0 if $f(x)$ is balanced.

Why does this work?

When the hidden Boolean function is *constant*, the quantum states before and after querying the oracle are the same. The inverse of the Hadamard gate is the Hadamard gate itself. Thus, by Step 4, we essentially reverse Step 2 to obtain the initial quantum state of all-zero at the first register.

When the hidden Boolean function is *balanced*, the quantum state after querying the oracle is orthogonal to the quantum state before querying the oracle. Thus, by Step 4, when reverting the operation, we must end up with a quantum state that is orthogonal to the initial quantum state of all-zero at the first register. This means we should never obtain the all-zero state.

Quantum Oracle

The key to the Deutsch-Josza Algorithm is the implementation of the quantum oracle.

For a constant function, it is simple:

1. if $f(x) = 0$, then apply the I gate to the qubit in register 2.
2. if $f(x) = 1$, then apply the X gate to the qubit in register 2.

For a balanced function, it is more complicated:

There are $C(2^n, 2^{n-1})$ different n -bit balanced function possibilities. A subset of these can be defined by one of the bitstrings from 1 to $2^n - 1$ inclusive, where for a given particular hidden bitstring, a , the oracle is the bitwise product of x and a , which is implemented as a multi-qubit f-controlled-NOT gate with the second register, as per Reference [2].

2. Example

Let's go through a specific example for a two bit balanced function with hidden bitstring $a = 3$.

1. The first register of two qubits is initialized to zero and the second register qubit to one $|\psi_0\rangle = |00\rangle_1 |1\rangle_2$
2. Apply Hadamard on all qubits $|\psi_1\rangle = \frac{1}{2}(|00\rangle_1 + |01\rangle_1 + |10\rangle_1 + |11\rangle_1) \frac{1}{\sqrt{2}}(|0\rangle_2 - |1\rangle_2)$

3. For $a = 3$, (11 in binary) the oracle function can be implemented as $Q_f = CX_{1a}CX_{2a'} |\psi_2\rangle = \frac{1}{2}\sqrt{2} [|00\rangle$

$$_1(|0\oplus 0\oplus 0\rangle_2 - |1\oplus 0\oplus 0\rangle_2) + |01\rangle_1(|0\oplus 0\oplus 1\rangle_2 - |1\oplus 0\oplus 1\rangle_2) + |10\rangle_1(|0\oplus 1\oplus 0\rangle_2 - |1\oplus 1\oplus 0\rangle_2) + |11\rangle_1(|0\oplus 1\oplus 1\rangle_2 - |1\oplus 1\oplus 1\rangle_2)]$$

$$\text{Thus } |\psi_2\rangle = \frac{1}{2}\sqrt{2} [|00\rangle_1(|0\rangle_2 - |1\rangle_2) - |01\rangle_1(|0\rangle_2 - |1\rangle_2) - |10\rangle_1(|0\rangle_2 - |1\rangle_2) + |11\rangle_1(|0\rangle_2 - |1\rangle_2)] = \frac{1}{2}(|00\rangle_1 - |01\rangle_1 - |10\rangle_1 + |11\rangle_1) \frac{1}{2}(\sqrt{2}(|0\rangle_2 - |1\rangle_2)) = \frac{1}{2}(|0\rangle_{10} - |1\rangle_{10}) \frac{1}{2}(\sqrt{2}(|0\rangle_{11} - |1\rangle_{11})) \frac{1}{2}(\sqrt{2}(|0\rangle_2 - |1\rangle_2))$$

4. Apply Hadamard on the first register $|\psi_3\rangle = |1\rangle_{10}|1\rangle_{11}(|0\rangle_2 - |1\rangle_2)$

5. Measuring the first two qubits will give the non-zero 11, indicating a balanced function.

3. Qiskit Implementation

We now implement the Deutsch-Josza algorithm for the example of a two bit balanced function with hidden bitstring $a = 3$.

```
# initialization
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np

# importing Qiskit
from qiskit import IBMQ, BasicAer
from qiskit.providers.ibmq import least_busy
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, execute

# import basic plot tools
from qiskit.tools.visualization import plot_histogram
```

```
# set the Length of the $n$-bit string.
n = 2

# set the oracle, b for balanced, c for constant
oracle = "b"

# if the oracle is balanced, set the hidden bitstring, b
if oracle == "b":
    b = 3 # np.random.randint(1,2**n) uncomment for a random value

# if the oracle is constant, set c = 0 or 1 randomly.
if oracle == "c":
    c = np.random.randint(2)
```

```

# Creating registers
# n qubits for querying the oracle and one qubit for storing the answer
qr = QuantumRegister(n+1)
cr = ClassicalRegister(n)

djCircuit = QuantumCircuit(qr, cr)
barriers = True

# Since all qubits are initialized to |0>, we need to flip the second register qubit to the |1> state
djCircuit.x(qr[n])

# Apply barrier
if barriers:
    djCircuit.barrier()

# Apply Hadamard gates to all qubits
djCircuit.h(qr)

# Apply barrier
if barriers:
    djCircuit.barrier()

# Query the oracle
if oracle == "c": # if the oracle is constant, return c
    if c == 1:
        djCircuit.x(qr[n])
    else:
        djCircuit.iden(qr[n])
else: # otherwise, the oracle is balanced and it returns the inner product of the input with b (non-zero)
    for i in range(n):
        if (b & (1 << i)):
            djCircuit.cx(qr[i], qr[n])

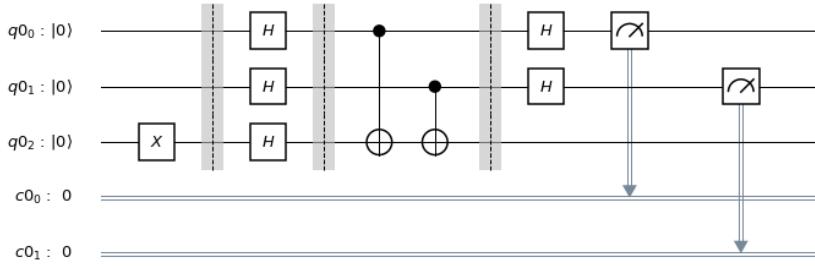
# Apply barrier
if barriers:
    djCircuit.barrier()

# Apply Hadamard gates to the first register after querying the oracle
for i in range(n):
    djCircuit.h(qr[i])

# Measure the first register
for i in range(n):
    djCircuit.measure(qr[i], cr[i])

```

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

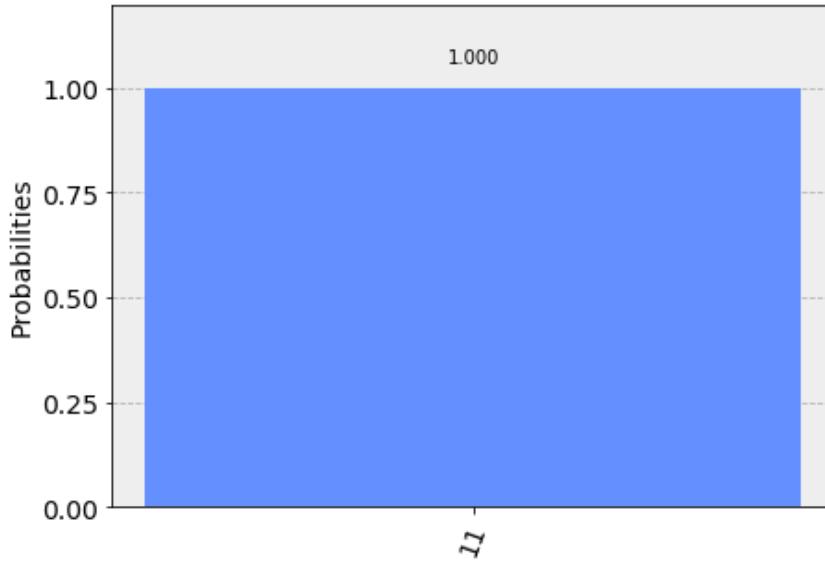


3b. Experiment with Simulators

We can run the above circuit on the simulator.

```
# use Local simulator
backend = BasicAer.get_backend('qasm_simulator')
shots = 1024
results = execute(djCircuit, backend=backend, shots=shots).result()
answer = results.get_counts()

plot_histogram(answer)
```



We can see that the result of the measurement is 11 as expected.

3a. Experiment with Real Devices

We can run the circuit on the real device as shown below.

```
# Load our saved IBMQ accounts and get the least busy backend device with less than or equal to 5 qubit
IBMQ.load_account()
provider = IBMQ.get_provider(hub='ibm-q')
provider.backends()
backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits <= 5 and
                                         not x.configuration().simulator and x.status().operational==True))
print("least busy backend: ", backend)
```

```
least busy backend: ibmqx4
```

```
# Run our circuit on the least busy backend. Monitor the execution of the job in the queue
from qiskit.tools.monitor import job_monitor

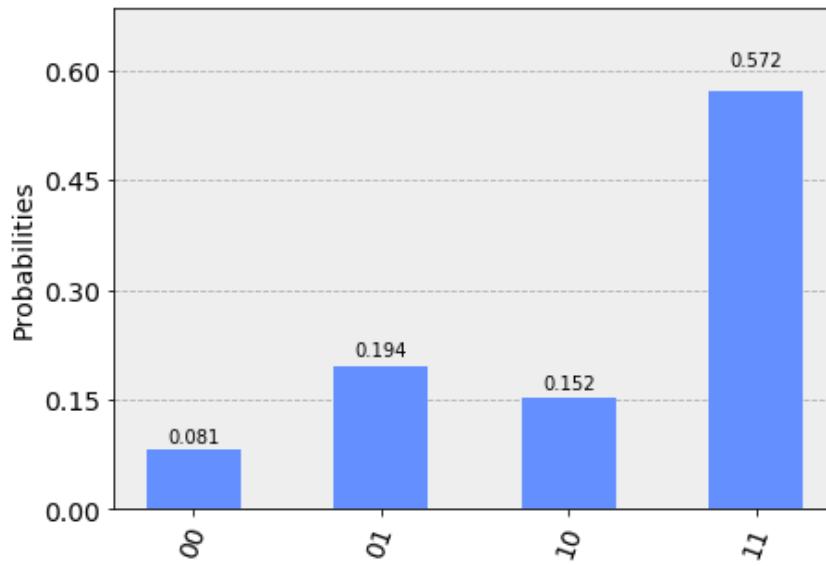
shots = 1024
job = execute(djCircuit, backend=backend, shots=shots)

job_monitor(job, interval = 2)
```

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

```
# Get the results of the computation
results = job.result()
answer = results.get_counts()

plot_histogram(answer)
```



As we can see, most of the results are 11. The other results are due to errors in the quantum computation.

4. Problems

1. The above [implementation](#) of Deutsch-Josza is for a balanced function with a two bit input of 3. Modify the implementation for a constant function. Are the results what you expect? Explain.
2. The above [implementation](#) of Deutsch-Josza is for a balanced function with a two bit random input. Modify the implementation for a balanced function with a 4 bit input of 13. Are the results what you expect? Explain.

5. References

1. David Deutsch and Richard Jozsa (1992). "Rapid solutions of problems by quantum computation". Proceedings of the Royal Society of London A. 439: 553–558. doi:[10.1098/rspa.1992.0167](https://doi.org/10.1098/rspa.1992.0167).
2. R. Cleve; A. Ekert; C. Macchiavello; M. Mosca (1998). "Quantum algorithms revisited". Proceedings of the Royal Society of London A. 454: 339–354. doi:[10.1098/rspa.1998.0164](https://doi.org/10.1098/rspa.1998.0164).

```
qiskit.__qiskit_version__
```



```
{'qiskit': '0.10.4',
 'qiskit-terra': '0.8.2',
 'qiskit-ignis': '0.1.1',
 'qiskit-aer': '0.2.1',
 'qiskit-ibmq-provider': '0.2.2',
 'qiskit-aqua': '0.5.1'}
```

Bernstein-Vazirani Algorithm

In this section, we first introduce the Bernstein-Vazirani problem, and classical and quantum algorithms to solve it. We then implement the quantum algorithm using Qiskit, and run on a simulator and device.

Contents

1. Introduction

- o [Bernstein-Vazirani Problem](#)
- o [Bernstein-Vazirani Algorithm](#)

2. Example

3. Qiskit Implementation

- o [Simulation](#)
- o [Device](#)

4. Problems

5. References

1. Introduction

The Bernstein-Vazirani algorithm, first introduced in Reference [1], can be seen as an extension of the Deutsch-Josza algorithm covered in the last section. It showed that there can be advantages in using a quantum computer as a computational tool for more complex problems compared to the Deutsch-Josza problem.

1a. Bernstein-Vazirani Problem

We are again given a hidden function Boolean f , which takes as input a string of bits, and returns either 0 or 1, that is:

0

$f(\{x_0, x_1, x_2, \dots\}) \rightarrow 0 \text{ or } 1 \text{ where } x_n \text{ is } 0 \text{ or } 1.$

Instead of the function being balanced or constant as in the Deutsch-Josza problem, now the function is guaranteed to return the bitwise product of the input with some string, s . In other words, given an input x , $f(x) = s \cdot x \pmod{2}$. We are expected to find s .

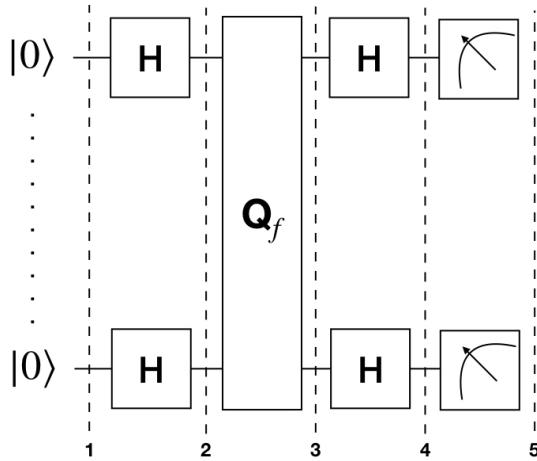
1b. Bernstein-Vazirani Algorithm

Classical Solution

Classically, the oracle returns $f_s(x) = s \cdot x \pmod{2}$ given an input x . Thus, the hidden bit string s can be revealed by querying the oracle with $x = 1, 2, \dots, 2^i, \dots, 2^{n-1}$, where each query reveals the i -th bit of s (or, s_i). For example, with $x = 1$ one can obtain the least significant bit of s , and so on. This means we would need to call the function $f_s(x)$ n times.

Quantum Solution

Using a quantum computer, we can solve this problem with 100% confidence after only one call to the function $f(x)$. The quantum Bernstein-Vazirani algorithm to find the hidden integer is very simple: (1) start from a $|0\rangle^{\otimes n}$ state, (2) apply Hadamard gates, (3) query the oracle, (4) apply Hadamard gates, and (5) measure, generically illustrated below:



The correctness of the algorithm is best explained by looking at the transformation of a quantum register $|a\rangle$ by n Hadamard gates, each applied to the qubit of the register. It can be shown that:

$$|a\rangle H^{\otimes n} \rightarrow \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} (-1)^{a \cdot x} |x\rangle.$$

In particular, when we start with a quantum register $|0\rangle$ and apply n Hadamard gates to it, we have the familiar quantum superposition:

$$|0\rangle H^{\otimes n} \rightarrow \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle,$$

which is slightly different from the Hadamard transform of the register $|a\rangle$ by the phase $(-1)^{a \cdot x}$.

Now, the quantum oracle f_a returns 1 on input x such that $a \cdot x \equiv 1 \pmod{2}$, and returns 0 otherwise. This means we have the following transformation:

$$|x\rangle f_a \rightarrow |x\rangle = (-1)^{a \cdot x} |x\rangle.$$

The algorithm to reveal the hidden integer follows naturally by querying the quantum oracle f_a with the quantum superposition obtained from the Hadamard transformation of $|0\rangle$. Namely,

$$|0\rangle H^{\otimes n} \rightarrow \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle f_a \rightarrow \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} (-1)^{a \cdot x} |x\rangle.$$

Because the inverse of the n Hadamard gates is again the n Hadamard gates, we can obtain $|a\rangle$ by

$$\frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} (-1)^{a \cdot x} |x\rangle H^{\otimes n} \rightarrow |a\rangle.$$

2. Example

Let's go through a specific example for $n = 2$ qubits and a secret string $s = 11$. Note that we are following the formulation in Reference [2] that generates a circuit for the Bernstein-Vazirani quantum oracle using only one register.

1. The register of two qubits is initialized to zero: $|\psi_0\rangle = |00\rangle$
2. Apply a Hadamard gate to both qubits: $|\psi_1\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$
3. For the string $s = 11$, the quantum oracle can be implemented as $Q_f = Z_1 Z_2$: $|\psi_2\rangle = \frac{1}{2}(|00\rangle - |01\rangle - |10\rangle + |11\rangle)$
4. Apply a Hadamard gate to both qubits: $|\psi_3\rangle = |11\rangle$
5. Measure to find the secret string $s = 11$

3. Qiskit Implementation

We now implement the Bernstein-Vazirani algorithm with Qiskit for a two bit function with $s = 11$.

```
# initialization
import matplotlib.pyplot as plt
```



```
%matplotlib inline
import numpy as np

# importing Qiskit
from qiskit import IBMQ, BasicAer
from qiskit.providers.ibmq import least_busy
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, execute

# import basic plot tools
from qiskit.tools.visualization import plot_histogram
```

We first set the number of qubits used in the experiment, and the hidden integer s to be found by the algorithm. The hidden integer s determines the circuit for the quantum oracle.

```
nQubits = 2 # number of physical qubits used to represent s
s = 3       # the hidden integer

# make sure that a can be represented with nqubits
s = s % 2**(nQubits)
```

We then use Qiskit to program the Bernstein-Vazirani algorithm.

```
# Creating registers
# qubits for querying the oracle and finding the hidden integer
qr = QuantumRegister(nQubits)
# bits for recording the measurement on qr
cr = ClassicalRegister(nQubits)

bvCircuit = QuantumCircuit(qr, cr)
barriers = True

# Apply Hadamard gates before querying the oracle
for i in range(nQubits):
    bvCircuit.h(qr[i])

# Apply barrier
if barriers:
    bvCircuit.barrier()

# Apply the inner-product oracle
for i in range(nQubits):
    if (s & (1 << i)):
        bvCircuit.z(qr[i])
    else:
```

```

bvCircuit.iden(qr[i])

# Apply barrier
if barriers:
    bvCircuit.barrier()

#Apply Hadamard gates after querying the oracle
for i in range(nQubits):
    bvCircuit.h(qr[i])

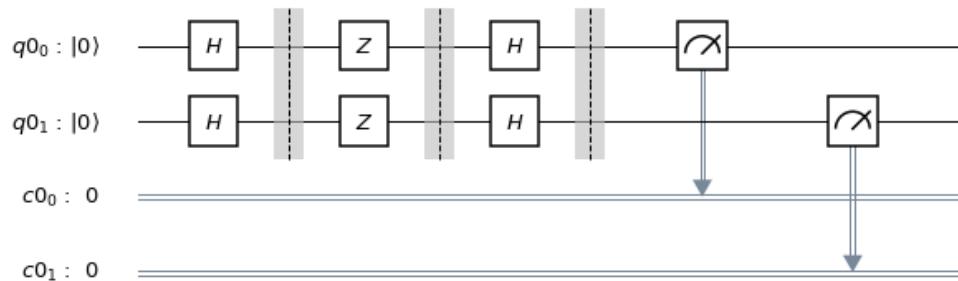
# Apply barrier
if barriers:
    bvCircuit.barrier()

# Measurement
bvCircuit.measure(qr, cr)

```

<qiskit.circuit.instructionset.InstructionSet at 0x1276416d8>

bvCircuit.draw(output='mpl')



3a. Experiment with Simulators

We can run the above circuit on the simulator.

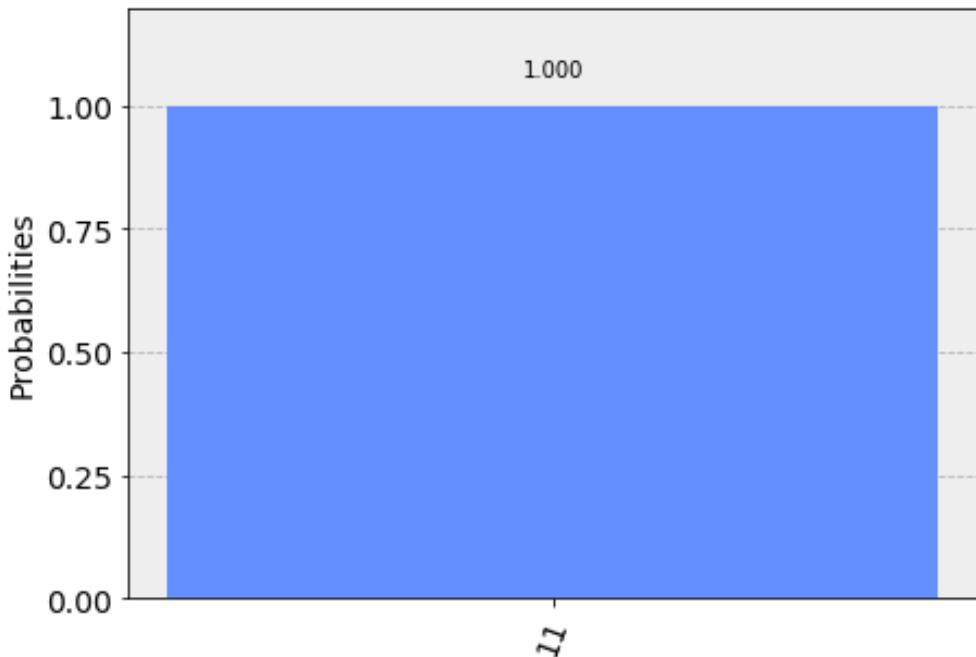
```

# use local simulator
backend = BasicAer.get_backend('qasm_simulator')
shots = 1024
results = execute(bvCircuit, backend=backend, shots=shots).result()
answer = results.get_counts()

plot_histogram(answer)

```





We can see that the result of the measurement is the binary representation of the hidden integer 3 (11).

3b. Experiment with Real Devices

We can run the circuit on the real device as below.

```
# Load our saved IBMQ accounts and get the Least busy backend device with Less than 11 qubits
IBMQ.load_account()
provider = IBMQ.get_provider(hub='ibm-q')
provider.backends()
backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits <= 11
                                         and not x.configuration().simulator and x.status().operational()))
print("least busy backend: ", backend)
```

least busy backend: ibmqx2

```
# Run our circuit on the Least busy backend. Monitor the execution of the job in the background
from qiskit.tools.monitor import job_monitor

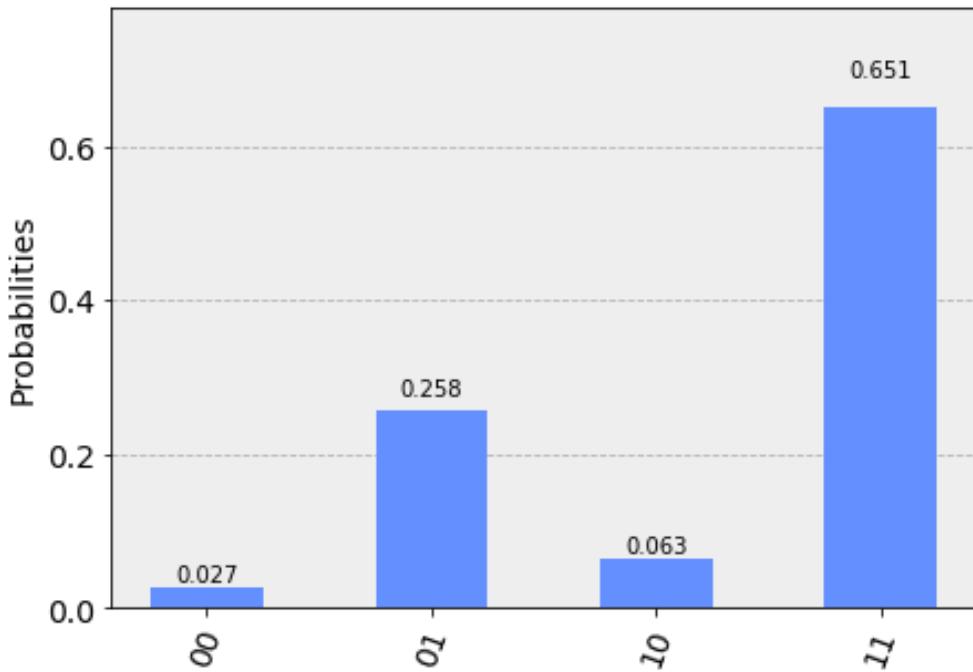
shots = 1024
job = execute(bvCircuit, backend=backend, shots=shots)

job_monitor(job, interval = 2)
```

Job Status: job has successfully run

```
# Get the results from the computation
results = job.result()
answer = results.get_counts()

plot_histogram(answer)
```



As we can see, most of the results are 11. The other results are due to errors in the quantum computation.

4. Problems

1. The above [implementation](#) of Bernstein-Vazirani is for a secret bit string of $s = 11$. Modify the implementation for a secret string os $s = 1011$. Are the results what you expect? Explain.
2. The above [implementation](#) of Bernstein-Vazirani is for a secret bit string of $s = 11$. Modify the implementation for a secret string os $s = 1110110101$. Are the results what you expect? Explain.

5. References

1. Ethan Bernstein and Umesh Vazirani (1997) "Quantum Complexity Theory" SIAM Journal on Computing, Vol. 26, No. 5: 1411-1473, [doi:10.1137/S0097539796300921](https://doi.org/10.1137/S0097539796300921).

2. Jiangfeng Du, Mingjun Shi, Jihui Wu, Xianyi Zhou, Yangmei Fan, BangJiao Ye, Rongdian Han
(2001) "Implementation of a quantum algorithm to solve the Bernstein-Vazirani parity problem
without entanglement on an ensemble quantum computer", Phys. Rev. A 64, 042306,
[10.1103/PhysRevA.64.042306](https://doi.org/10.1103/PhysRevA.64.042306), arXiv:quant-ph/0012114.

```
import qiskit
qiskit.__qiskit_version__
```



```
{'qiskit': '0.10.4',
 'qiskit-terra': '0.8.2',
 'qiskit-ignis': '0.1.1',
 'qiskit-aer': '0.2.1',
 'qiskit-ibmq-provider': '0.2.2',
 'qiskit-aqua': '0.5.1'}
```

Simon's Algorithm

In this section, we first introduce the Simon problem, and classical and quantum algorithms to solve it. We then implement the quantum algorithm using Qiskit, and run on a simulator and device.

Contents

1. Introduction

- o [Simon's Problem](#)
- o [Simon's Algorithm](#)

2. Example

3. Qiskit Implementation

- o [Simulation](#)
- o [Device](#)

4. Oracle

5. Problems

6. References

1. Introduction

Simon's algorithm, first introduced in Reference [1], was the first quantum algorithm to show an exponential speed-up versus the best classical algorithm in solving a specific problem. This inspired the quantum algorithm for the discrete Fourier transform, also known as quantum Fourier transform, which is used in the most famous quantum algorithm: Shor's factoring algorithm.

1a. Simon's Problem

We are given an unknown blackbox function f , which is guaranteed to be either one-to-one or two-to-one, where one-to-one and two-to-one functions have the following properties:

- *one-to-one*: maps exactly one unique output for every input, eg. $f(1) \rightarrow 1, f(2) \rightarrow 2, f(3) \rightarrow 3, f(4) \rightarrow 4$.
- *two-to-one*: maps exactly two inputs to every unique output, eg. $f(1) \rightarrow 1, f(2) \rightarrow 1, f(3) \rightarrow 2, f(4) \rightarrow 2$ according to a hidden bitstring, s where: given $x_1, x_2: f(x_1) = f(x_2)$ it is guaranteed: $x_1 \oplus x_2 = s$ if $f(2) \rightarrow 1, f(3) \rightarrow 2, f(4) \rightarrow 2$

$$x_2 : f(x_1) = f(x_2)$$

Given this blackbox f , how quickly can we determine if f is one-to-one or two-to-one? Then, if f turns out to be two-to-one, how quickly can we determine s ? As it turns out, both cases boil down to the same problem of finding s , where a bitstring of $s = 000\dots$ represents the one-to-one f .

$$s = 000\dots \quad f$$

1b. Simon's Algorithm

Classical Solution

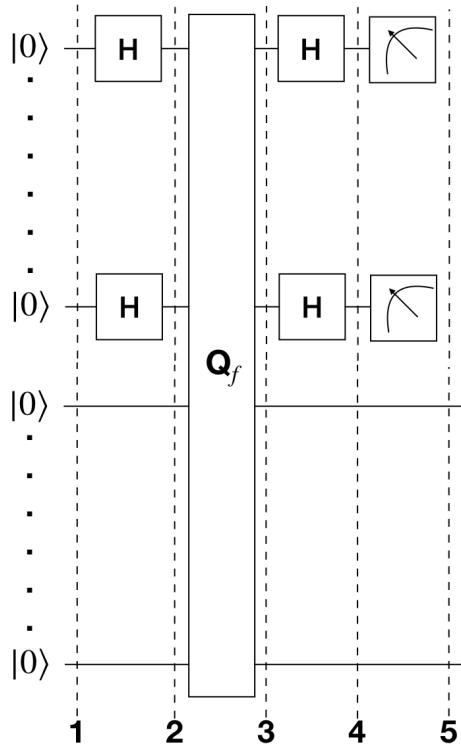
Classically, if we want to know what s is for a given f , with 100% certainty, we have to check up to $2^{N-1} + 1$ inputs, where N is the number of bits in the input. This means checking just over half of all the possible inputs until we find 1 two cases of the same output. Although, probabilistically the average number of inputs will be closer to the order of $(\log 2)^2$. Much like the Deutsch-Jozsa problem, if we get lucky, we could solve the problem with our first two tries. But if we happen to get an f that is one-to-one, or get *really* unlucky with an f that's two-to-one, then we're stuck with the full $2^{N-1} + 1$.

$$\frac{f}{2^{N-1} + 1}$$

$$f$$

Quantum Solution

The quantum circuit that implements Simon's algorithm is shown below.



Where the query function, Q_f acts on two quantum registers as: $|x\rangle|0\rangle \rightarrow |x\rangle|f(x)\rangle$

$$|x\rangle|0\rangle \rightarrow |x\rangle|f(x)\rangle$$

The algorithm involves the following steps.

1. Two n -qubit input registers are initialized to the zero state: $|\psi_1\rangle = |0\rangle^{\otimes n}|0\rangle^{\otimes n}$

$$|\psi_1\rangle = |0\rangle^{\otimes n}|0\rangle^{\otimes n}$$

2. Apply a Hadamard transform to the first register: $|\psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle|0\rangle^{\otimes n}$

3. Apply the query function Q_f : $|\psi_3\rangle = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle|f(x)\rangle$

$$|\psi_3\rangle = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} |x\rangle|f(x)\rangle$$

4. Measure the second register. A certain value of $f(x)$ will be observed. Because of the setting of the problem, the observed value $f(x)$ could correspond to two possible inputs: x and $y = x \oplus s$. Therefore the first register becomes: $|\psi_3\rangle = \sum_{x \in \{0,1\}^n} |x\rangle |f(x)\rangle$

$$\psi_4 = \frac{1}{\sqrt{2}}(|x\rangle + |y\rangle) \quad x \quad y = x \oplus s$$

Where we omitted the second register since it has been measured.

5. Apply Hadamard on the first register: $|\psi_5\rangle = \frac{1}{\sqrt{2^{n+1}}} \sum_{z \in \{0,1\}^n} [(-1)^{x \cdot z} + (-1)^{y \cdot z}] |z\rangle$

6. $|\psi_5\rangle = \frac{1}{\sqrt{2^n}} \sum_{z \in \{0,1\}^n} [(-1)^{x \cdot z} + (-1)^{y \cdot z}] |z\rangle$ Measuring the first register will give an output of: $(-1)^{x \cdot z} = (-1)^{y \cdot z}$

(which means $x \cdot z = y \cdot z = (x \oplus s) \cdot zx \cdot z = x \cdot z \oplus s \cdot z = 0 \pmod{2}$)

z = A string z whose inner product with s will be measured. Thus, repeating the algorithm $\approx n$ times, we will be able to
 $\vdash (x \oplus s) \cdot z \pmod{2}$ $\vdash s \cdot z \pmod{2}$
 $\vdash x \cdot z \oplus s \cdot z \pmod{2}$
 $\vdash 0 \pmod{2}$

obtain n different values of z and the following system of equation can be written: $s \cdot z_1 = 0$, $s \cdot z_2 = 0$, ..., $s \cdot z_n = 0$

}

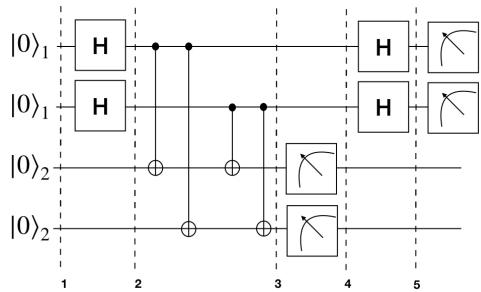
From which s can be determined, for example by Gaussian elimination.

$$\begin{cases} s \cdot z_1 = 0 \\ s \cdot z_2 = 0 \\ \dots \\ s \cdot z_n = 0 \end{cases}$$

So, in this particular problem the quantum algorithm performs exponentially fewer steps than the classical one. Once again, it might be difficult to envision an application of this algorithm (although it inspired the most famous algorithm created by Shor) but it represents the first proof that there can be an exponential speed-up in solving a specific problem by using a quantum computer rather than a classical one.

2. Example

Let's see the example of Simon's algorithm for 2 qubits with the secret string $s = 11$, so that $f(x) = f(y)$ if $y = x \oplus s$. The quantum circuit to solve the problem is:



1. Two 2-qubit input registers are initialized to the zero state: $|\psi_1\rangle = |00\rangle_1 |00\rangle_2$

$|\psi_1\rangle = |00\rangle_1 |00\rangle_2$
2. Apply Hadamard gates to the qubits in the first register: $|\psi_2\rangle = \frac{1}{2}(|00\rangle_1 + |01\rangle_1 + |10\rangle_1 + |11\rangle_1) |00\rangle_2$

$|\psi_2\rangle = \frac{1}{2}(|00\rangle_1 + |01\rangle_1 + |10\rangle_1 + |11\rangle_1)|00\rangle_2$

3. For the string $s = 11$, the query function can be implemented as $Q_f = CX_{13}CX_{14}CX_{23}CX_{24}$: $|\psi_3\rangle = {}^{12}\sqrt{(|00\rangle_1|0\oplus 0\oplus 0, 0\oplus 0\oplus 0\rangle_2 + |01\rangle_1|0\oplus 0\oplus 1, 0\oplus 0\oplus 1\rangle_2 + |10\rangle_1|0\oplus 1\oplus 0, 0\oplus 1\oplus 1\rangle_2 + |11\rangle_1|1\oplus 0, 1\oplus 1\rangle_2)}$

$$|\psi_3\rangle = \frac{1}{2}(|00\rangle_1|0\oplus 0\oplus 0, 0\oplus 0\oplus 0\rangle_2 + |01\rangle_1|0\oplus 0\oplus 1, 0\oplus 0\oplus 1\rangle_2 + |10\rangle_1|0\oplus 1\oplus 0, 0\oplus 1\oplus 1\rangle_2 + |11\rangle_1|1\oplus 0, 1\oplus 1\rangle_2)$$

$$\text{Thus } |\psi_3\rangle = {}^{12}\sqrt{(|00\rangle_1|0\oplus 0\oplus 0, 0\oplus 0\oplus 0\rangle_2 + |01\rangle_1|0\oplus 0\oplus 1, 0\oplus 0\oplus 1\rangle_2 + |10\rangle_1|1\oplus 0, 1\oplus 0\rangle_2 + |11\rangle_1|1\oplus 1, 1\oplus 1\rangle_2)}$$

4. We measure the first register. With 50% probability we will see either $|00\rangle_2$ or $|11\rangle_2$. For the sake of the example,

$$\text{let us assume that we see } |11\rangle_2. \text{ The state of the system is then } |\psi_4\rangle = {}^{12}\sqrt{(|01\rangle_1 + |10\rangle_1)}$$

where we omitted the second register since it has been measured.

5. Apply Hadamard on the first register $|\psi_5\rangle = {}^{12}\sqrt{[(|0\rangle + |1\rangle) \otimes (|0\rangle - |1\rangle) + (|0\rangle - |1\rangle) \otimes (|0\rangle + |1\rangle)]} = {}^{12}\sqrt{(|00\rangle - |01\rangle + |10\rangle - |11\rangle + |00\rangle + |01\rangle - |10\rangle - |11\rangle)} = {}^{12}\sqrt{(|00\rangle - |11\rangle)}$

6. Measuring the first register will give either $|0, 0\rangle$ or $|1, 1\rangle$ with equal probability. If we see $|1, 1\rangle$, then: $s \cdot 11 = 0$

This is one equation, but s has two variables. Therefore, we need to repeat the algorithm at least another time to have enough equations that will allow us to determine s .

3. Qiskit Implementation

We now implement Simon's algorithm for the above [example](#) for 2-qubits with a $s = 11$.

```
#initialization
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np

# importing Qiskit
from qiskit import IBMQ, BasicAer
from qiskit.providers.ibmq import least_busy
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, execute

# import basic plot tools
from qiskit.tools.visualization import plot_histogram
```

`s = '11'`

```
# Creating registers
# qubits for querying the oracle and finding the hidden period s
qr = QuantumRegister(2*len(str(s)))
# classical registers for recording the measurement from qr
cr = ClassicalRegister(2*len(str(s)))

simonCircuit = QuantumCircuit(qr, cr)
barriers = True
```

```

# Apply Hadamard gates before querying the oracle
for i in range(len(str(s))):
    simonCircuit.h(qr[i])

# Apply barrier
if barriers:
    simonCircuit.barrier()

# Apply the query function
## 2-qubit oracle for s = 11
simonCircuit.cx(qr[0], qr[len(str(s)) + 0])
simonCircuit.cx(qr[0], qr[len(str(s)) + 1])
simonCircuit.cx(qr[1], qr[len(str(s)) + 0])
simonCircuit.cx(qr[1], qr[len(str(s)) + 1])

# Apply barrier
if barriers:
    simonCircuit.barrier()

# Measure ancilla qubits
for i in range(len(str(s)), 2*len(str(s))):
    simonCircuit.measure(qr[i], cr[i])

# Apply barrier
if barriers:
    simonCircuit.barrier()

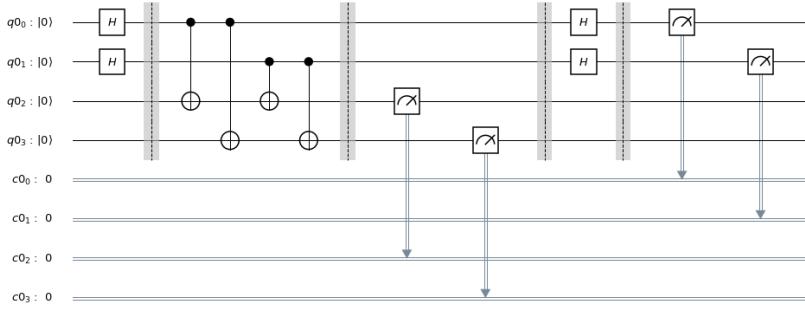
# Apply Hadamard gates to the input register
for i in range(len(str(s))):
    simonCircuit.h(qr[i])

# Apply barrier
if barriers:
    simonCircuit.barrier()

# Measure input register
for i in range(len(str(s))):
    simonCircuit.measure(qr[i], cr[i])

```

simonCircuit.draw(output='mpl')



3a. Experiment with Simulators

We can run the above circuit on the simulator.

```

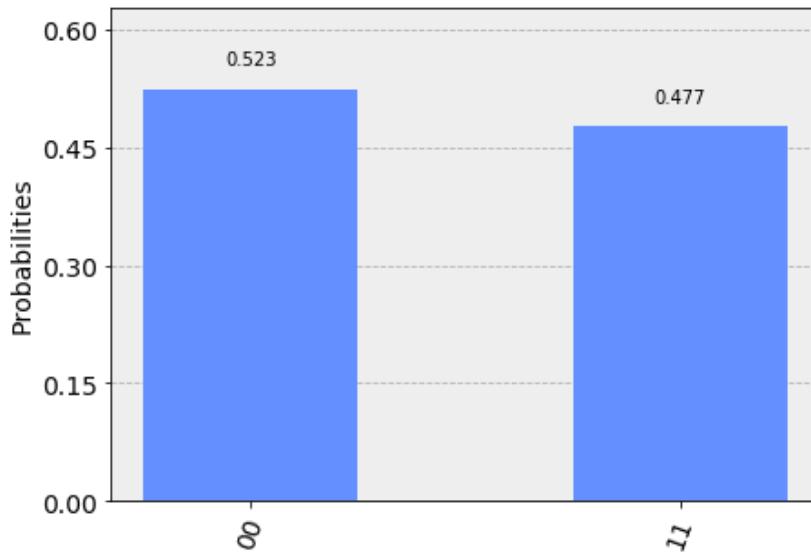
# use Local simulator
backend = BasicAer.get_backend('qasm_simulator')
shots = 1024
results = execute(simonCircuit, backend=backend, shots=shots).result()
answer = results.get_counts()

# Categorize measurements by input register values
answer_plot = {}
for measresult in answer.keys():
    measresult_input = measresult[len(str(s)):]
    if measresult_input in answer_plot:
        answer_plot[measresult_input] += answer[measresult]
    else:
        answer_plot[measresult_input] = answer[measresult]

# Plot the categorized results
print( answer_plot )
plot_histogram(answer_plot)

```

```
{'11': 488, '00': 536}
```



```

# Calculate the dot product of the results
def sdotz(a, b):
    accum = 0
    for i in range(len(a)):
        accum += int(a[i]) * int(b[i])
    return (accum % 2)

print('s, z, s.z (mod 2)')
for z_rev in answer_plot:
    z = z_rev[::-1]
    print( '{}, {}, {}'.format(s, z, sdotz(s,z)) )

```

```
s, z, s.z (mod 2)
11, 11, 11.11=0
11, 00, 11.00=0
```

Using these results, we can recover the value of $s = 11$.

3b. Experiment with Real Devices

We can run the circuit on the real device as below.

```
# Load our saved IBMQ accounts and get the Least busy backend device with Less than or equal to 5 qubits
IBMQ.load_account()
provider = IBMQ.get_provider(hub='ibm-q')
provider.backends()
backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits <= 5 and
                                         not x.configuration().simulator and x.status().operational==True))
print("least busy backend: ", backend)
```

least busy backend: ibmqx2

```
# Run our circuit on the Least busy backend. Monitor the execution of the job in the queue
from qiskit.tools.monitor import job_monitor

shots = 1024
job = execute(simonCircuit, backend=backend, shots=shots)

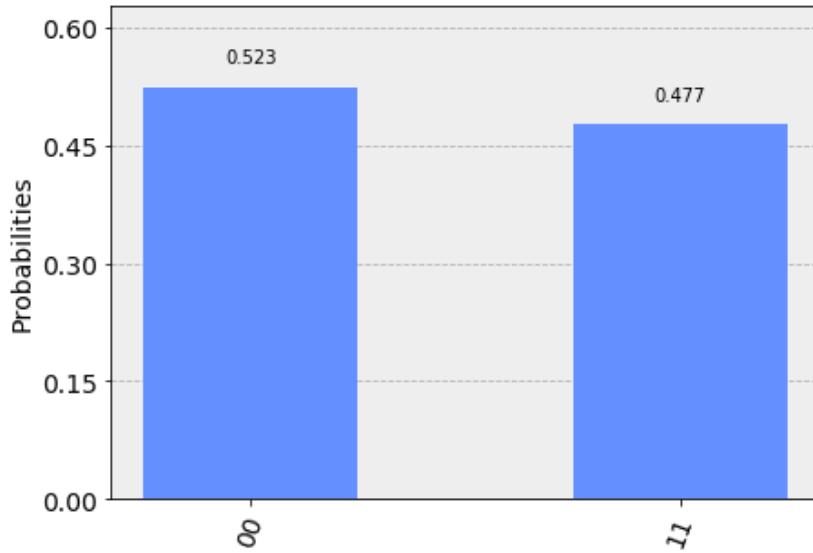
job_monitor(job, interval = 2)
```

Job Status: job has successfully run

```
# Categorize measurements by input register values
answer_plot = {}
for measresult in answer.keys():
    measresult_input = measresult[len(str(s)):]
    if measresult_input in answer_plot:
        answer_plot[measresult_input] += answer[measresult]
    else:
        answer_plot[measresult_input] = answer[measresult]

# Plot the categorized results
print( answer_plot )
plot_histogram(answer_plot)
```

{'11': 488, '00': 536}



```
# Calculate the dot product of the most significant results
print('s, z, s.z (mod 2)')
for z_rev in answer_plot:
    if answer_plot[z_rev] >= 0.1*shots:
        z = z_rev[::-1]
        print( '{}, {}, {}.{.}={}'.format(s, z, s,z,sdotz(s,z)) )
```

```
s, z, s.z (mod 2)
11, 11, 11.11=0
11, 00, 11.00=0
```

As we can see, the most significant results are those for which $s \cdot z = 0 \pmod{2}$. Using a classical computer, we can then recover the value of s by solving the linear system of equations. For this $n = 2$ case, $s = 11$.

4. Oracle

The above [example](#) and [implementation](#) of Simon's algorithm are specifically for $s = 11$. To extend the problem to other secret bit strings, we need to discuss the Simon query function or oracle in more detail.

The Simon algorithm deals with finding a hidden bitstring $s \in \{0, 1\}^n$ from an oracle f_s that satisfies $f_s(x) = f_s(y)$ if and only if $y = x \oplus s$ for all $x \in \{0, 1\}^n$. Here, the \oplus is the bitwise XOR operation. Thus, if $s = 0\dots0$, i.e., the all-zero bitstring, then f_s is a 1-to-1 (or, permutation) function. Otherwise, if $s \neq 0\dots0$, then f_s is a 2-to-1 function.

In the algorithm, the oracle receives $|x\rangle|0\rangle$ as input. With regards to a predetermined s , the oracle writes its output to the second register so that it transforms the input to $|x\rangle|f_s(x)\rangle$ such that $f(x) = f(x \oplus s)$ for all $x \in \{0, 1\}^n$.

Such a blackbox function can be realized by the following procedures.

- Copy the content of the first register to the second register. $|x\rangle|0\rangle \rightarrow |x\rangle|x\rangle$

- **(Creating 1-to-1 or 2-to-1 mapping)** If s is not all-zero, then there is the least index j so that $s_j = 1$. If $x_j = 0$, then XOR the second register with s . Otherwise, do not change the second register. $|x\rangle|x\rangle \rightarrow |x\rangle|x\oplus s\rangle$ if $x_j = 0$ for the least index j
- **(Creating random permutation)** Randomly permute and flip the qubits of the second register. $|x\rangle|y\rangle \rightarrow |x\rangle|f_s(y)\rangle$

5. Problems

1. Implement a general Simon oracle.
2. Test your general Simon oracle with the secret bitstring $s = 1001$, on a simulator and device. Are the results what you expect? Explain.

6. References

1. Daniel R. Simon (1997) "On the Power of Quantum Computation" SIAM Journal on Computing, 26(5), 1474–1483, doi:[10.1137/S0097539796298637](https://doi.org/10.1137/S0097539796298637)

```
import qiskit
qiskit.__qiskit_version__
```

```
{'qiskit': '0.10.4',
 'qiskit-terra': '0.8.2',
 'qiskit-ignis': '0.1.1',
 'qiskit-aer': '0.2.1',
 'qiskit-ibmq-provider': '0.2.2',
 'qiskit-aqua': '0.5.1'}
```

Quantum Fourier Transform

In this tutorial, we introduce the quantum fourier transform (QFT), derive the circuit, and implement it using Qiskit. We show how to run QFT on a simulator and a five qubit device.

Contents

1. Introduction
2. Example 1: 1-qubit QFT
3. The Quantum Fourier transform
4. The circuit that implements QFT
5. Example 2: 3-qubit QFT
6. A note about the form of the QFT circuit
7. Qiskit Implementation
 - o Running QFT on a simulator
 - o Running QFT on a real quantum device
8. Problems
9. References

1. Introduction

The Fourier transform occurs in many different versions throughout classical computing, in areas ranging from signal processing to data compression to complexity theory. The quantum Fourier transform (QFT) is the quantum implementation of the discrete Fourier transform over the amplitudes of a wavefunction. It is part of many quantum algorithms, most notably Shor's factoring algorithm and quantum phase estimation.

The discrete Fourier transform acts on a vector (x_0, \dots, x_{N-1}) and maps it to the vector (y_0, \dots, y_{N-1}) according to the formula $y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \omega_{jkN}$ where $\omega_{jkN} = e^{2\pi i j k N / N}$.

Similarly, the quantum Fourier transform acts on a quantum state $\sum_{i=0}^{N-1} x_i |i\rangle$ and maps it to the quantum state $\sum_{i=0}^{N-1} y_i |i\rangle$ according to the formula $y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \omega_{jkN}$ with ω_{jkN} defined as above. Note that only the amplitudes of the state were affected by this transformation.

This can also be expressed as the map: $|x\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} \omega_{xyN}^y |y\rangle$

Or the unitary matrix: $U_{QFT} = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \omega_{xyN}^y |y\rangle\langle x|$

2. Example 1: 1-qubit QFT

Consider how the QFT operator as defined above acts on a single qubit state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. In this case, $x_0 = \alpha$, $x_1 = \beta$, and $N = 2$. Then,

$$y_0 = \frac{1}{\sqrt{2}} \left(\alpha \exp\left(2\pi i 0 \times 02\right) + \beta \exp\left(2\pi i 1 \times 02\right) \right) = \frac{1}{\sqrt{2}}(\alpha + \beta)$$

and

$$y_1 = \frac{1}{\sqrt{2}} \left(\alpha \exp\left(2\pi i 0 \times 12\right) + \beta \exp\left(2\pi i 1 \times 12\right) \right) = \frac{1}{\sqrt{2}}(\alpha - \beta)$$

such that the final result is the state

$$U_{QFT}|\psi\rangle = \frac{1}{\sqrt{2}}(\alpha + \beta)|0\rangle + \frac{1}{\sqrt{2}}(\alpha - \beta)|1\rangle$$

This operation is exactly the result of applying the Hadamard operator (H) on the qubit:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

If we apply the H operator to the state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, we obtain the new state:

$$\frac{1}{\sqrt{2}}(\alpha + \beta)|0\rangle + \frac{1}{\sqrt{2}}(\alpha - \beta)|1\rangle \equiv |\tilde{\alpha}\rangle + |\tilde{\beta}\rangle$$

Notice how the Hadamard gate performs the discrete Fourier transform for $N = 2$ on the amplitudes of the state.

3. The Quantum Fourier transform

So what does the quantum Fourier transform look like for larger N ? Let's derive a circuit for $N = 2^n$, QFT_N acting on the state $|x\rangle = |x_1...x_n\rangle$ where x_1 is the most significant bit.

$$\text{QFT}_N |x\rangle = \sqrt{N} \sum_{y=0}^{N-1} \omega_{xyN} |y\rangle$$

$$= \sqrt{N} \sum_{y=0}^{N-1} e^{2\pi i xy/N} |y\rangle \text{ since } \omega_{xyN} = e^{2\pi i xy/N} \text{ and } N = 2^n = \sqrt{N} \sum_{y=0}^{N-1} e^{2\pi i (\sum_{k=1}^n y_k/2^k) x} |y_1 \dots y_n\rangle$$

rewriting in fractional binary notation $y = y_1 \dots y_n$

$y_n/2^n = \sum_{k=1}^n y_k/2^k = \sqrt{N} \sum_{y=0}^{N-1} e^{2\pi i xy_k/2^k} |y_1 \dots y_n\rangle$ after expanding the exponential of a sum to a product of exponentials $= \sqrt{N} \prod_{k=1}^n (|0\rangle + e^{2\pi i x_k/2^k} |1\rangle)$ after rearranging the sum and products, and expanding $\sum_{y=0}^{N-1} = \sum_{y_1=0}^1 \sum_{y_2=0}^1 \dots$

$$\begin{aligned} \sum_{y=0}^{N-1} &= \sqrt{N} \left(|0\rangle + e^{2\pi i x_1/2^1} |1\rangle \right) \otimes \left(|0\rangle + e^{2\pi i x_2/2^2} |1\rangle \right) \otimes \dots \\ &\otimes \left(|0\rangle + e^{2\pi i x_{n-1}/2^{n-1}} |1\rangle \right) \otimes \left(|0\rangle + e^{2\pi i x_n/2^n} |1\rangle \right) \end{aligned}$$

4. The circuit that implements QFT

The circuit that implements QFT makes use of two gates. The first one is a single-qubit Hadamard gate, H , that you already know. From the discussion in [Example 1](#) above, you have already seen that the action of H on the single-qubit state $|x_k\rangle$ is

$$H|x_k\rangle = |0\rangle + \exp\left(\frac{2\pi i x_k}{2}\right) |1\rangle$$

The second is a two-qubit controlled rotation CROT_k given in block-diagonal form as

$$\text{CROT}_k = \begin{bmatrix} I & 00 & \text{UROT}_k \end{bmatrix}$$

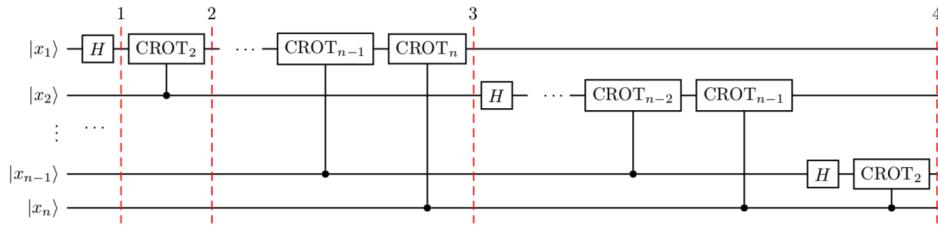
$$\text{where } \text{UROT}_k = \begin{bmatrix} 1 & 00 & \exp\left(\frac{2\pi i x_k}{2}\right) \end{bmatrix}$$

The action of CROT_k on the two-qubit state $|x_j x_k\rangle$ where the first qubit is the control and the second is the target is given by

$$\text{CROT}_k |x_j 0\rangle = |x_j 0\rangle$$

$$\text{and } \text{CROT}_k |x_j 1\rangle = \exp\left(2\pi i 2^k x_j\right) |x_j 1\rangle$$

Given these two gates, a circuit that implements [an n-qubit QFT](#) is shown below.



The circuit operates as follows. We start with an n-qubit input state $|x_1 x_2 \dots x_n\rangle$.

1. After the first Hadamard gate on qubit 1, the state is transformed from the input state to $H_1 |x_1 x_2 \dots x_n\rangle$

$$|x_n\rangle = \frac{1}{\sqrt{2}} \left[|0\rangle + \exp\left(2\pi i 2^{n-1} x_n\right) |1\rangle \right] \otimes |x_1 x_2 \dots x_{n-1}\rangle$$

2. After the CROT_2 gate on qubit 1 controlled by qubit 2, the state is transformed to

$$\frac{1}{\sqrt{2}} \left[|0\rangle + \exp\left(2\pi i 2^{n-2} x_2 + 2\pi i 2^{n-1} x_1\right) |1\rangle \right] \otimes |x_1 x_2 \dots x_{n-1}\rangle$$

3. After the application of the last CROT_n gate on qubit 1 controlled by qubit n, the state becomes

$$\frac{1}{\sqrt{2}} \left[|0\rangle + \exp\left(2\pi i 2^n x_n + 2\pi i 2^{n-1} x_{n-1} + \dots + 2\pi i 2^2 x_2 + 2\pi i 2 x_1\right) |1\rangle \right] \otimes |x_1 x_2 \dots x_{n-1}\rangle$$

Noting that

$x = 2^{n-1} x_1 + 2^{n-2} x_2 + \dots + 2^1 x_{n-1} + 2^0 x_n$ we can write the above state as

$$\frac{1}{\sqrt{2}} \left[|0\rangle + \exp\left(2\pi i 2^n x\right) |1\rangle \right] \otimes |x_1 x_2 \dots x_{n-1}\rangle$$

4. After the application of a similar sequence of gates for qubits 2...n, we find the final state to be

$$1/\sqrt{2} \left[|0\rangle + \exp\left(\frac{2\pi i 2^n}{x}\right) |1\rangle \right] \otimes 1/\sqrt{2} \left[|0\rangle + \exp\left(\frac{2\pi i 2^{n-1}}{x}\right) |1\rangle \right] \otimes \dots$$

$$\otimes 1/\sqrt{2} \left[|0\rangle + \exp\left(\frac{2\pi i 2^2}{x}\right) |1\rangle \right] \otimes 1/\sqrt{2} \left[|0\rangle + \exp\left(\frac{2\pi i 2^1}{x}\right) |1\rangle \right]$$

which is exactly the QFT of the input state as derived [above](#) with the caveat that the order of the qubits is reversed in the output state.

5. Example 2: 3-qubit QFT

The steps to creating the circuit for $|y_1 y_2 y_3\rangle = \text{QFT}_8 |x_1 x_2 x_3\rangle$ would be:

1. Apply a Hadamard gate to $|x_3\rangle$

$$\Psi_1 = |x_1\rangle \otimes |x_2\rangle \otimes 1/\sqrt{2} \left[|0\rangle + \exp\left(\frac{2\pi i 2 x_3}{x}\right) |1\rangle \right]$$

2. Apply a CROT₂ gate to $|x_3\rangle$ depending on $|x_2\rangle$

$$\Psi_2 = |x_1\rangle \otimes |x_2\rangle \otimes 1/\sqrt{2} \left[|0\rangle + \exp\left(\frac{2\pi i 2^2 x_2 + 2\pi i 2 x_3}{x}\right) |1\rangle \right]$$

3. Apply a CROT₃ gate to $|x_3\rangle$ depending on $|x_1\rangle$

$$\Psi_3 = |x_1\rangle \otimes |x_2\rangle \otimes 1/\sqrt{2} \left[|0\rangle + \exp\left(\frac{2\pi i 2^3 x_1 + 2\pi i 2^2 x_2 + 2\pi i 2 x_3}{x}\right) |1\rangle \right]$$

4. Apply a Hadamard gate to $|x_2\rangle$

$$\psi_4 = |x_1\rangle \otimes \frac{1}{\sqrt{2}} \left[|0\rangle + \exp\left(2\pi i 2^{x_2}\right) |1\rangle \right] \otimes \frac{1}{\sqrt{2}} \left[|0\rangle + \exp\left(2\pi i 2^3 x_1 + 2\pi i 2^2 x_2 + 2\pi i 2 x_3\right) |1\rangle \right]$$

5. Apply a CROT_2 gate to $\langle x_2 |$ depending on $\langle x_1 |$

$$\begin{aligned} |\psi_5\rangle = & \langle x_1 | \otimes \frac{1}{\sqrt{2}} \left[|0\rangle + \exp\left(\frac{2\pi i}{2} x_1 + \frac{2\pi i}{2} x_2\right) |1\rangle \right] \otimes \frac{1}{\sqrt{2}} \left[|0\rangle + \exp\left(2\pi i 2^3 x_1 + 2\pi i 2^2 x_2 + 2\pi i 2 x_3\right) |1\rangle \right] \\ & + \frac{1}{\sqrt{2}} \left[|0\rangle + \exp\left(\frac{2\pi i}{2} x_1 + \frac{2\pi i}{2} x_2 + \frac{2\pi i}{2} x_3\right) |1\rangle \right] \otimes \frac{1}{\sqrt{2}} \left[|0\rangle + \exp\left(2\pi i 2^3 x_1 + 2\pi i 2^2 x_2 + 2\pi i 2 x_3\right) |1\rangle \right] \end{aligned}$$

6. Apply a Hadamard gate to $\langle x_1 |$

$$\begin{aligned} |\psi_6\rangle = & \frac{1}{\sqrt{2}} \left[\langle 0 | + \exp\left(\frac{2\pi i}{2} x_1\right) \langle 1 | \right] \otimes \frac{1}{\sqrt{2}} \left[|0\rangle + \exp\left(2\pi i 2^3 x_1 + 2\pi i 2^2 x_2 + 2\pi i 2 x_3\right) |1\rangle \right] \\ & + \frac{1}{\sqrt{2}} \left[\langle 0 | + \exp\left(\frac{2\pi i}{2} x_1 + \frac{2\pi i}{2} x_2 + \frac{2\pi i}{2} x_3\right) \langle 1 | \right] \otimes \frac{1}{\sqrt{2}} \left[|0\rangle + \exp\left(2\pi i 2^3 x_1 + 2\pi i 2^2 x_2 + 2\pi i 2 x_3\right) |1\rangle \right] \end{aligned}$$

7. Keep in mind the reverse order of the output state relative to the desired QFT. Therefore, measure the bits in reverse order, that is $y_3 = x_1$, $y_2 = x_2$, $y_1 = x_3$.

6. A note about the form of the QFT circuit

The example above demonstrates a very useful form of the QFT for $N=2^n$. Note that only the last qubit depends on the values of all the other input qubits and each further bit depends less and less on the input qubits. This becomes important in physical implementations of the QFT, where nearest-neighbor couplings are easier to achieve than distant couplings between qubits.

7. Qiskit Implementation

In Qiskit, the implementation of the CROT gate used in the discussion above is a controlled phase rotation gate. This gate is defined in [OpenQASM](#) as

$$\text{CU_1}(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{bmatrix}$$

Hence, the mapping from the CROT_k gate in the discussion above into the CU_1 gate is found from the equation

$$\theta = 2\pi/2^k = \pi/2^{k-1}$$

It is instructive to write out the relevant code for the 3-qubit case before generalizing to the n-qubit case. In Qiskit, it is:

```
q = QuantumRegister(3)
c = ClassicalRegister(3)

qft3 = QuantumCircuit(q, c)
qft3.h(q[0])
qft3.cu1(math.pi/2.0, q[1], q[0]) # CROT_2 from q[1] to q[0]
qft3.cu1(math.pi/4.0, q[2], q[0]) # CROT_3 from q[2] to q[0]
qft3.h(q[1])
qft3.cu1(math.pi/2.0, q[2], q[1]) # CROT_2 from q[2] to q[1]
qft3.h(q[2])
```

Following the above example, the case for n qubits can be generalized as:

```
def qft(circ, q, n):
    """n-qubit QFT on q in circ."""
    for j in range(n):
        circ.h(q[j])
        for k in range(j+1,n):
            circ.cu1(math.pi/float(2***(k-j)), q[k], q[j])
```

We will now implement the three-qubit QFT as discussed above. We first create a state whose QFT is known. The output after a QFT is applied to this special state is $\left| \text{001} \right\rangle$.

```
import math

# importing Qiskit
from qiskit import Aer, IBMQ
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit, execute

from qiskit.providers.ibmq import least_busy
from qiskit.tools.monitor import job_monitor
from qiskit.tools.visualization import plot_histogram
```

```
IBMQ.load_account()
```

First let's define the QFT function, as well as a function that creates a state from which a QFT will return 001:

```
def input_state(circ, q, n):
    """n-qubit input state for QFT that produces output 1."""
    for j in range(n):
        circ.h(q[j])
        circ.u1(-math.pi/float(2**(j))), q[j])

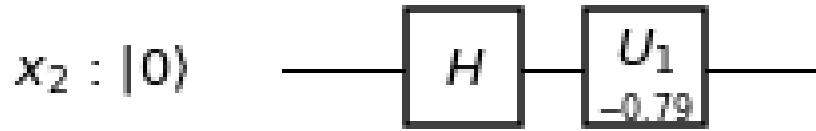
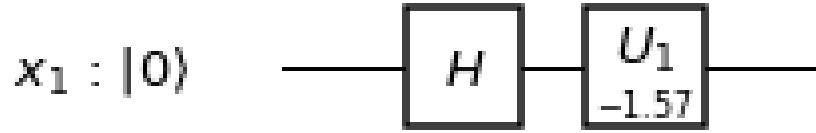
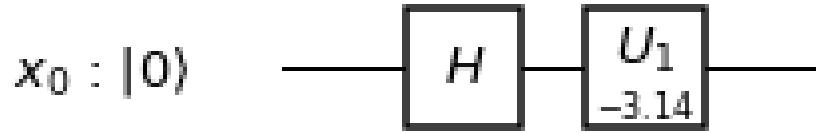
def qft(circ, q, n):
    """n-qubit QFT on q in circ."""
    for j in range(n):
        circ.h(q[j])
        for k in range(j+1,n):
            circ.cu1(math.pi/float(2**((k-j))), q[k], q[j])
        circ.barrier()
```

Let's now implement a QFT on a prepared three qubit input state that should return 001:

```
q = QuantumRegister(3, 'x')
c = ClassicalRegister(3, 'c')
qft3 = QuantumCircuit(q, c)

# first, prepare the state that should return 001 and draw that circuit
input_state(qft3, q, 3)

qft3.draw(output='mpl')
```



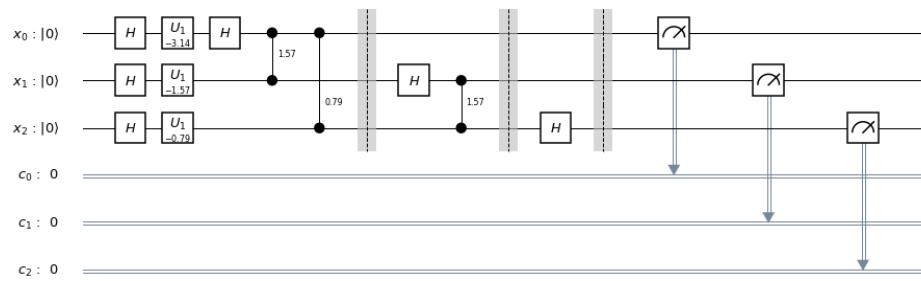
$c_0 : 0$

$c_1 : 0$

$c_2 : 0$

```
# next, do a qft on the prepared state and draw the entire circuit
qft(qft3, q, 3)
for i in range(3):
    qft3.measure(q[i], c[i])

qft3.draw(output='mpl')
```



7a. Running QFT on a simulator

```
# run on Local simulator
backend = Aer.get_backend("qasm_simulator")

simulate = execute(qft3, backend=backend, shots=1024).result()
simulate.get_counts()
```



```
{'001': 1024}
```

We indeed see that the outcome is always 001 when we execute the code on the simulator.

7b. Running QFT on a real quantum device

We then see how the same circuit can be executed on real-device backends.

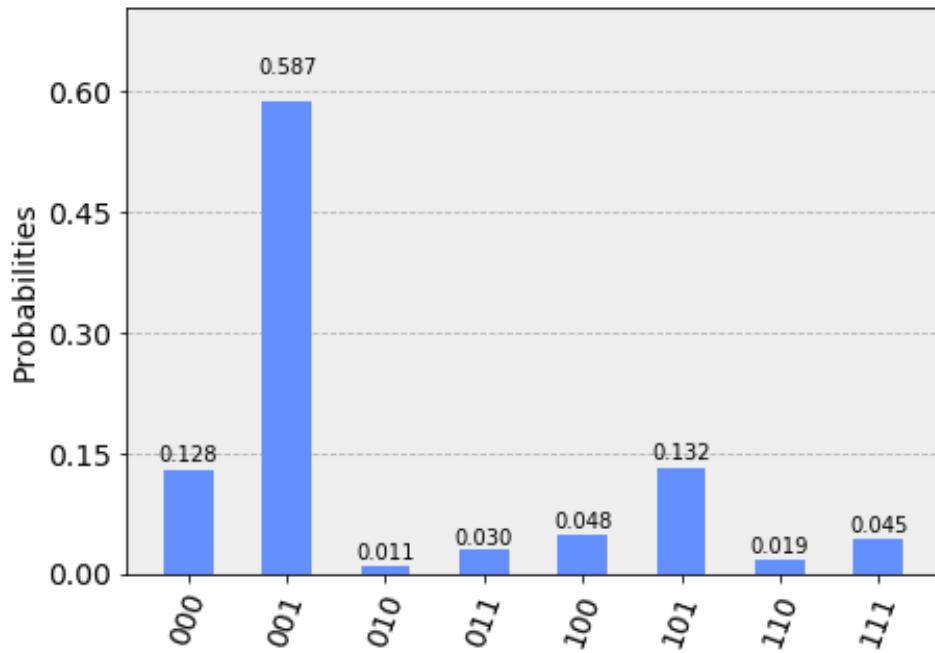
```
# Use the IBMQ Vigo device with 5 qubits
provider = IBMQ.get_provider(hub='ibm-q')
backend = provider.get_backend('ibmq_vigo')
shots = 2048
job_exp = execute(qft3, backend=backend, shots=shots)
job_monitor(job_exp)
```



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

```
results = job_exp.result()
plot_histogram(results.get_counts())
```





We see that the highest probability outcome is still 001 when we execute the code on a real device.

8. Problems

1. The [above implementation](#) of QFT was tested by using a special input state for which $\text{QFT}(\text{input state}) = 001$. Implement an input state for which $\text{QFT}(\text{input state}) = 100$.
2. The [above implementation](#) of QFT was tested by using a special input state for which $\text{QFT}(\text{input state}) = 001$. Implement an input state for which $\text{QFT}(\text{input state}) = 101$.

9. References

1. M. Nielsen and I. Chuang, Quantum Computation and Quantum Information, Cambridge Series on Information and the Natural Sciences (Cambridge University Press, Cambridge, 2000).

```
import qiskit
qiskit.__qiskit_version__
```



```
{'qiskit': '0.10.3',
 'qiskit-terra': '0.8.1',
 'qiskit-ignis': '0.1.1',
 'qiskit-aer': '0.2.1',
 'qiskit-ibmq-provider': '0.2.2',
 'qiskit-aqua': '0.5.1'}
```

Grover's Algorithm

In this section, we introduce Grover's algorithm and how it can be used to solve unstructured search problems. We then implement the quantum algorithm using Qiskit, and run on a simulator and device.

Contents

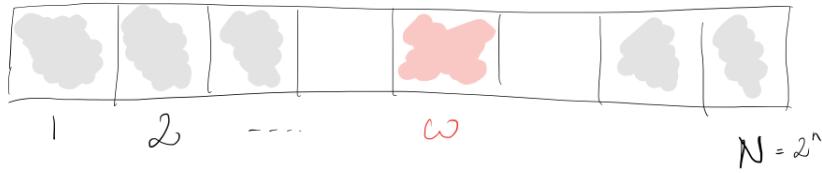
1. [Introduction](#)
2. [Example: 2 Qubits](#)
 - o [Simulation](#)
 - o [Device](#)
3. [Example: 3 Qubits](#)
 - o [Simulation](#)
 - o [Device](#)
4. [Problems](#)
5. [References](#)

1. Introduction

You have likely heard that one of the many advantages a quantum computer has over a classical computer is its superior speed searching databases. Grover's algorithm demonstrates this capability. This algorithm can speed up an unstructured search problem quadratically, but its uses extend beyond that; it can serve as a general trick or subroutine to obtain quadratic run time improvements for a variety of other algorithms. This is called the amplitude amplification trick.

Unstructured Search

Suppose you are given a large list of N items. Among these items there is one item with a unique property that we wish to locate; we will call this one the winner w . Think of each item in the list as a box of a particular color. Say all items in the list are gray except the winner w , which is pink.



To find the pink box -- the *marked item* -- using classical computation, one would have to check on average $N/2$ of these boxes, and in the worst case, all N of them. On a quantum computer, however, we can find the marked item in roughly \sqrt{N} steps with Grover's amplitude amplification trick. A quadratic speedup is indeed a substantial time-saver for finding marked items in long lists. Additionally, the algorithm does not use the list's internal structure, which makes it *generic*; this is why it immediately provides a quadratic quantum speed-up for many classical problems.

Oracle

How will the list items be provided to the quantum computer? A common way to encode such a list is in terms of a function f which returns $f(x) = 0$ for all unmarked items x and $f(w) = 1$ for the winner. To use a quantum computer for this problem, we must provide the items in superposition to this function, so we encode the function into a unitary matrix called an *oracle*. First we choose a binary encoding of the items $x, w \in \{0,1\}^n$ so that $N = 2^n$; now we can represent it in terms of qubits on a quantum computer. Then we define the oracle matrix U_f to act on any of the simple, standard basis states $|x\rangle$ by $U_f |x\rangle = (-1)^{f(x)} |x\rangle$.

We see that if x is an unmarked item, the oracle does nothing to the state. However, when we apply the oracle to the basis state $|w\rangle$, it maps $U_f |w\rangle = -|w\rangle$. Geometrically, this unitary matrix corresponds to a reflection about the origin for the marked item in an $N = 2^n$ dimensional vector space.

Amplitude Amplification

So how does the algorithm work? Before looking at the list of items, we have no idea where the marked item is. Therefore, any guess of its location is as good as any other, which can be expressed in terms of a uniform superposition: $|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$.

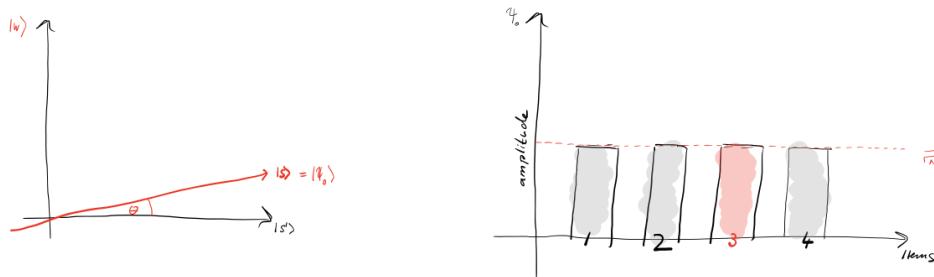
If at this point we were to measure in the standard basis $\{|x\rangle\}$, this superposition would collapse, according to the fifth quantum law, to any one of the basis states with the same probability of $\frac{1}{N} = \frac{1}{2^n}$. Our chances of guessing the right value w is therefore

$\$1\$$ in $\$2^n\$$, as could be expected. Hence, on average we would need to try about $N = 2^n$ times to guess the correct item.

Enter the procedure called amplitude amplification, which is how a quantum computer significantly enhances this probability. This procedure stretches out (amplifies) the amplitude of the marked item, which shrinks the other items' amplitude, so that measuring the final state will return the right item with near-certainty.

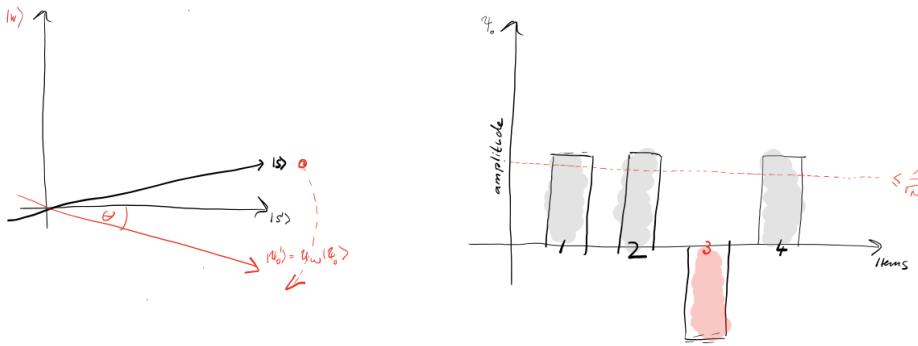
This algorithm has a nice geometrical interpretation in terms of two reflections, which generate a rotation in a two-dimensional plane. The only two special states we need to consider are the winner $|w\rangle$ and the uniform superposition $|s\rangle$. These two vectors span a two-dimensional plane in the vector space \mathbb{C}^N . They are not quite perpendicular because $|w\rangle$ occurs in the superposition with amplitude $N^{-1/2}$ as well. We can, however, introduce an additional state $|s'\rangle$ that is in the span of these two vectors, which is perpendicular to $|w\rangle$ and is obtained from $|s\rangle$ by removing $|w\rangle$ and rescaling.

Step 1: The amplitude amplification procedure starts out in the uniform superposition $|s\rangle$, which is easily constructed from $|s\rangle = H^{\otimes n} |0\rangle^N$.



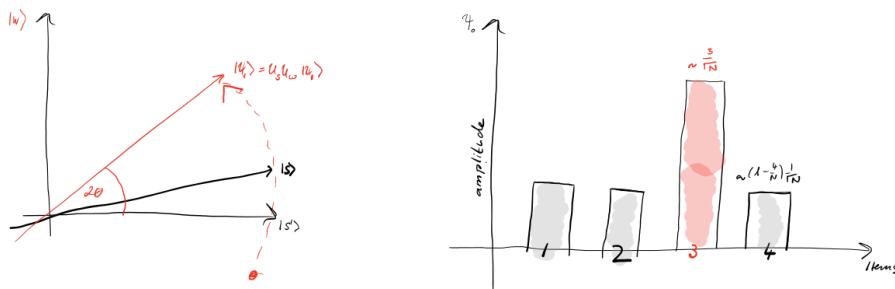
The left graphic corresponds to the two-dimensional plane spanned by perpendicular vectors $|w\rangle$ and $|s'\rangle$ which allows to express the initial state as $|s\rangle = \sin \theta |w\rangle + \cos \theta |s'\rangle$, where $\theta = \arcsin \langle s | w \rangle = \arcsin \frac{1}{\sqrt{N}}$. The right graphic is a bar graph of the amplitudes of the state $|\psi_t\rangle$ for the case $N = 2^2 = 4$. The average amplitude is indicated by a dashed line.

Step 2: We apply the oracle reflection U_f to the state $U_f |\psi_t\rangle = |\psi_{t'}\rangle$.



Geometrically this corresponds to a reflection of the state $|\psi_t\rangle$ about $-|w\rangle$. This transformation means that the amplitude in front of the $|w\rangle$ state becomes negative, which in turn means that the average amplitude has been lowered.

Step 3: We now apply an additional reflection U_s about the state $|s\rangle$: $U_s = 2|s\rangle\langle s| - \mathbb{1}$. This transformation maps the state to $U_s |\psi_{t'}\rangle$ and completes the transformation $|\psi_{t+1}\rangle = U_s U_f |\psi_t\rangle$.



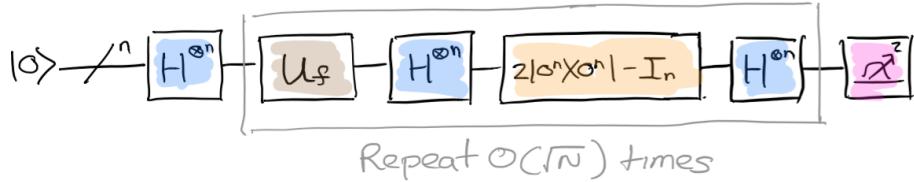
Two reflections always correspond to a rotation. The transformation $U_s U_f$ rotates the initial state $|s\rangle$ closer towards the winner $|w\rangle$. The action of the reflection U_s in the amplitude bar diagram can be understood as a reflection about the average amplitude. Since the average amplitude has been lowered by the first reflection, this transformation boosts the negative amplitude of $|w\rangle$ to roughly three times its original value, while it decreases the other amplitudes. We then go to **step 2** to repeat the application. This procedure will be repeated several times to zero in on the winner.

After t steps the state will have transformed to $|\psi_t\rangle = (U_s U_f)^t |\psi_0\rangle$.

How many times do we need to apply the rotation? It turns out that roughly \sqrt{N} rotations suffice. This becomes clear when looking at the amplitudes of the state $|\psi_t\rangle$. We can see that the amplitude of $|w\rangle$ grows linearly with the number of applications $\sim t$

$N^{-1/2}$. However, since we are dealing with amplitudes and not probabilities, the vector space's dimension enters as a square root. Therefore it is the amplitude, and not just the probability, that is being amplified in this procedure.

In the case that there are multiple solutions, M , it can be shown that roughly $\sqrt{N/M}$ rotations will suffice.



2. Example: 2 Qubits

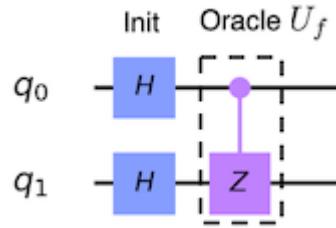
Let's first have a look at the case of Grover's algorithm for $N=4$ which is realized with 2 qubits. In this particular case, contrary to intuition, only **one rotation** is required which will rotate the initial state $|s\rangle$ to the winner $|w\rangle$ which can easily be shown [3]:

1. Following the above introduction, in the case $N=4$ we have $\theta = \arcsin \frac{1}{2} = \frac{\pi}{6}$
2. After t steps, we have $(U_s U_f)^t |\psi_s\rangle = \sin \theta_t |\psi_w\rangle + \cos \theta_t |\psi_s'\rangle$ where $\theta_t = (2t+1)\theta$.
3. In order to obtain $|\psi_w\rangle$ we need $\theta_t = \frac{\pi}{2}$, which with $\theta = \frac{\pi}{6}$ inserted above results to $t=1$. This implies that after $t=1$ rotation the searched element is found.

Now let us look into the possible oracles. We have $N=4$ possible elements, i.e. $|\psi_{00}\rangle, |\psi_{01}\rangle, |\psi_{10}\rangle, |\psi_{11}\rangle$ and hence require in total 4 oracles.

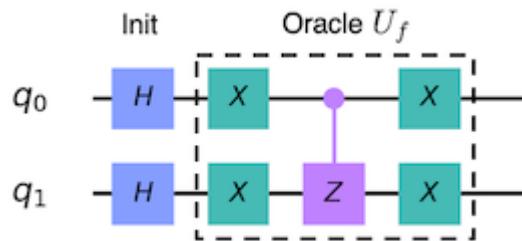
Oracle for $|\psi_w\rangle = |\psi_{11}\rangle$

Let us start with the case $|\psi_w\rangle = |\psi_{11}\rangle$. The oracle U_f in this case acts as follows: $U_f |\psi_s\rangle = U_f \frac{1}{2}(|\psi_{00}\rangle + |\psi_{01}\rangle + |\psi_{10}\rangle + |\psi_{11}\rangle) = \frac{1}{2}(|\psi_{00}\rangle + |\psi_{01}\rangle + |\psi_{10}\rangle - |\psi_{11}\rangle)$. In order to realize the sign flip for $|\psi_{11}\rangle$ we simply need to apply a controlled Z gate to the initial state. This leads to the following circuit:



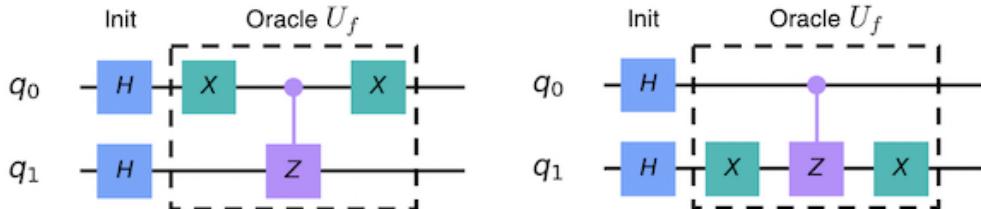
Oracle for $\lvert w \rangle = \lvert 00 \rangle$

In the case of $\lvert w \rangle = \lvert 00 \rangle$ the oracle U_f acts as follows: $U_f \lvert s \rangle = U_f \frac{1}{2}(\lvert 00 \rangle + \lvert 01 \rangle + \lvert 10 \rangle + \lvert 11 \rangle) = \frac{1}{2}(-\lvert 00 \rangle + \lvert 01 \rangle + \lvert 10 \rangle + \lvert 11 \rangle)$. In order to realize the sign flip for $\lvert 00 \rangle$ we need to apply an "inverted" controlled Z gate to the initial state leading to the following circuit:



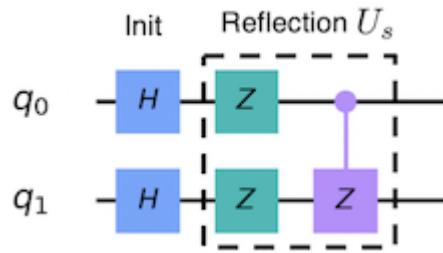
Oracles for $\lvert w \rangle = \lvert 01 \rangle$ and $\lvert w \rangle = \lvert 10 \rangle$

Following the above logic one can straight forwardly construct the oracles for $\lvert w \rangle = \lvert 01 \rangle$ (left circuit) and $\lvert w \rangle = \lvert 10 \rangle$ (right circuit):



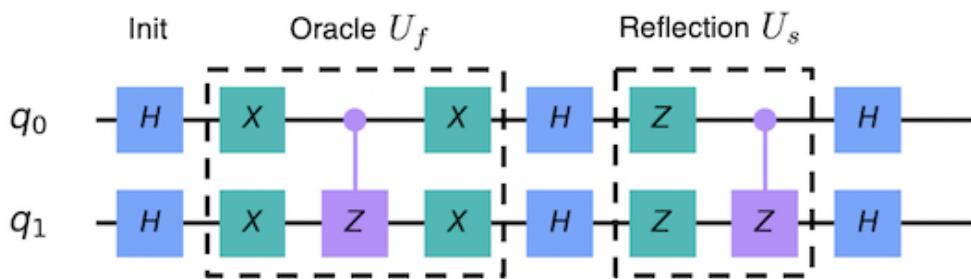
Reflection U_s

In order to complete the circuit we need to implement the additional reflection $U_s = 2\lvert s \rangle \langle s \rvert - \mathbb{1}$ which acts as follows $U_s = \frac{1}{2}(\lvert 00 \rangle + \lvert 01 \rangle + \lvert 10 \rangle + \lvert 11 \rangle) - \frac{1}{2}(-\lvert 00 \rangle + \lvert 01 \rangle + \lvert 10 \rangle - \lvert 11 \rangle)$, i.e. the signs of each state are flipped except for $\lvert 00 \rangle$. As can easily be verified, one way of implementing U_s is the following circuit:



Full Circuit for $|\psi_w\rangle = |\psi_{00}\rangle$

Since in the particular case of $N=4$ only one rotation is required we can combine the above components to build the full circuit for Grover's algorithm for the case $|\psi_w\rangle = |\psi_{00}\rangle$:



The other three circuits can be constructed in the same way and will not be depicted here.

2.1 Qiskit Implementation

We now implement Grover's algorithm for the above case of 2 qubits for $|\psi_w\rangle = |\psi_{00}\rangle$.

```
#initialization
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np

# importing Qiskit
from qiskit import IBMQ, BasicAer, Aer
from qiskit.providers.ibmq import least_busy
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister, execute

# import basic plot tools
from qiskit.visualization import plot_histogram
```

We start by preparing a quantum circuit for two qubits and a classical register with two bits.

```
qr = QuantumRegister(2)
cr = ClassicalRegister(2)

groverCircuit = QuantumCircuit(qr,cr)
```



Then we simply need to write out the commands for the circuit depicted above. First, Initialize the state $|s\rangle$:

```
groverCircuit.h(qr)
```



```
<qiskit.circuit.instructionset.InstructionSet at 0x1a2608d4d0>
```

Apply the Oracle for $|w\rangle$ = $|00\rangle$:

```
groverCircuit.x(qr)
groverCircuit.cz(qr[0],qr[1])
groverCircuit.x(qr)
```



```
<qiskit.circuit.instructionset.InstructionSet at 0x1a2608ddd0>
```

Apply a Hadamard operation to both qubits:

```
groverCircuit.h(qr)
```



```
<qiskit.circuit.instructionset.InstructionSet at 0x1a2613a3d0>
```

Apply the reflection U_s :

```
groverCircuit.z(qr)
groverCircuit.cz(qr[0],qr[1])
```



```
<qiskit.circuit.instructionset.InstructionSet at 0x1a2613f090>
```

Apply the final Hadamard to both qubits:

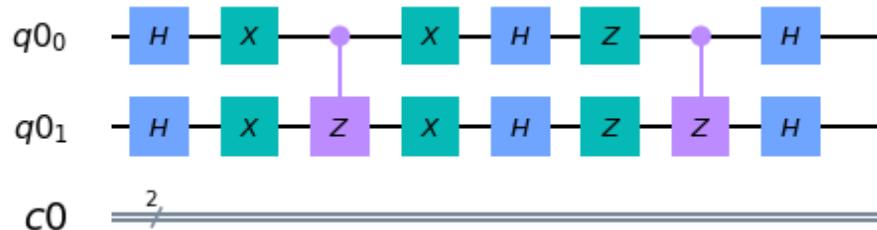
```
groverCircuit.h(qr)
```



```
<qiskit.circuit.instructionset.InstructionSet at 0x1a2613f6d0>
```

Drawing the circuit confirms that we have assembled it correctly:

```
groverCircuit.draw(output="mpl")
```



2.1.1 Experiment with Simulators

Let's run the circuit in simulation. First, we can verify that we have the correct statevector:

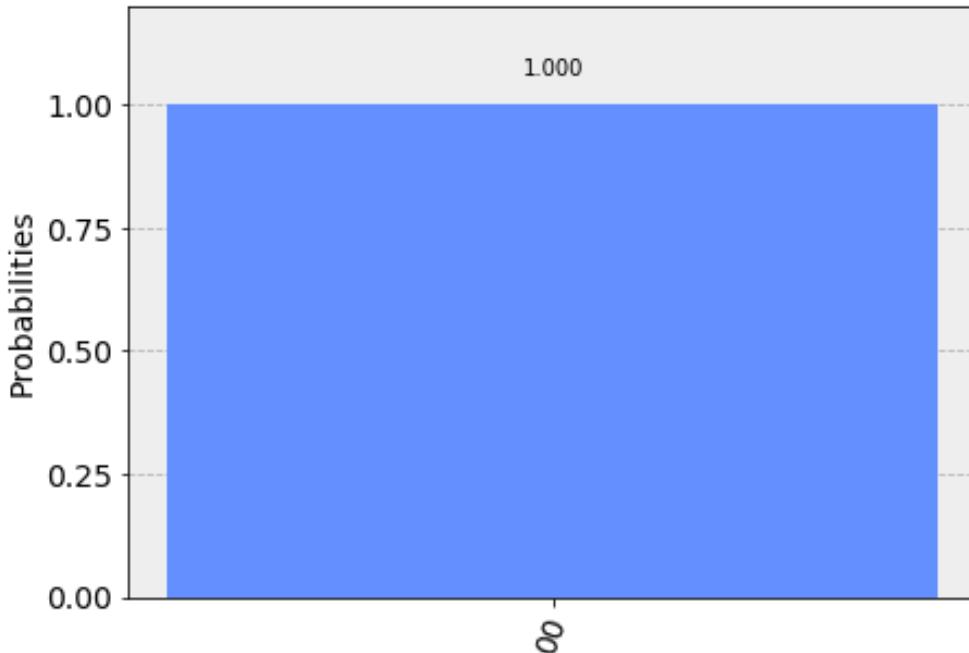
```
backend_sim = Aer.get_backend('statevector_simulator')
job_sim = execute(groverCircuit, backend_sim)
statevec = job_sim.result().get_statevector()
print(statevec)
```

```
[1.+0.j 0.+0.j 0.+0.j 0.+0.j]
```

Now let us measure the state and create the corresponding histogram experiments:

```
groverCircuit.measure(qr,cr)

backend = BasicAer.get_backend('qasm_simulator')
shots = 1024
results = execute(groverCircuit, backend=backend, shots=shots).result()
answer = results.get_counts()
plot_histogram(answer)
```



We confirm that in 100% of the cases the element $|00\rangle$ is found.

2.1.2 Experiment with Real Devices

We can run the circuit on the real device as below.

```
# Load IBM Q account and get the Least busy backend device
provider = IBMQ.load_account()
device = least_busy(provider.backends(simulator=False))
print("Running on current least busy device: ", device)
```



Running on current least busy device: ibmqx2

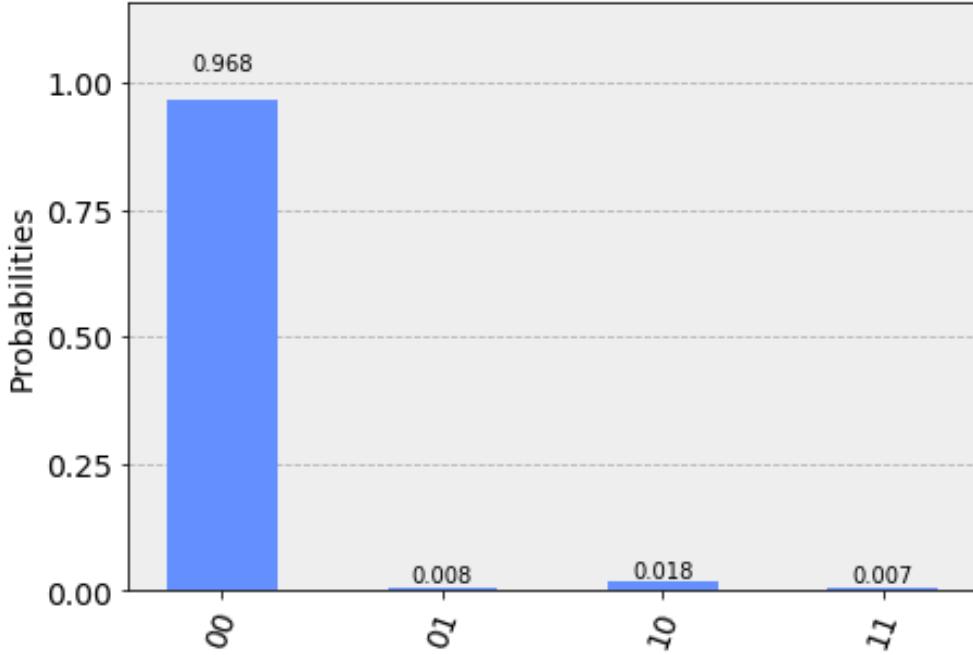
```
# Run our circuit on the Least busy backend. Monitor the execution of the job in the
from qiskit.tools.monitor import job_monitor
job = execute(groverCircuit, backend=device, shots=1024, max_credits=10)
job_monitor(job, interval = 2)
```



Job Status: job has successfully run

```
# Get the results from the computation
results = job.result()
answer = results.get_counts(groverCircuit)
plot_histogram(answer)
```

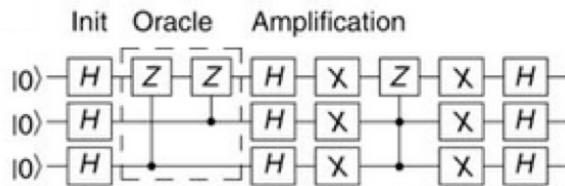




We confirm that in the majority of the cases the element $|00\rangle$ is found. The other results are due to errors in the quantum computation.

3. Example: 3 Qubits

We now go through the example of Grover's algorithm for 3 qubits with two marked states $|101\rangle$ and $|110\rangle$, following the implementation found in Reference [2]. The quantum circuit to solve the problem using a phase oracle is:



1. Apply Hadamard gates to 3 qubits initialised to $|\psi_1\rangle = \frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle)$
2. Mark states $|\psi_2\rangle = \frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle - |101\rangle - |110\rangle + |111\rangle)$ using a phase oracle
3. Perform the reflection around the average amplitude:

1. Apply Hadamard gates to the qubits $\lvert \psi_{3a} \rangle = \frac{1}{2} (\lvert 000 \rangle + \lvert 011 \rangle + \lvert 100 \rangle - \lvert 111 \rangle)$
2. Apply X gates to the qubits $\lvert \psi_{3b} \rangle = \frac{1}{2} (\lvert 000 \rangle + \lvert 011 \rangle + \lvert 100 \rangle + \lvert 111 \rangle)$
3. Apply a doubly controlled Z gate between the 1, 2 (controls) and 3 (target) qubits $\lvert \psi_{3c} \rangle = \frac{1}{2} (\lvert 000 \rangle + \lvert 011 \rangle + \lvert 100 \rangle - \lvert 111 \rangle)$
4. Apply X gates to the qubits $\lvert \psi_{3d} \rangle = \frac{1}{2} (\lvert 000 \rangle + \lvert 011 \rangle + \lvert 100 \rangle - \lvert 111 \rangle)$
5. Apply Hadamard gates to the qubits $\lvert \psi_{3e} \rangle = \frac{1}{\sqrt{2}} (\lvert 101 \rangle - \lvert 110 \rangle)$
4. Measure the 3\$ qubits to retrieve states $\lvert 101 \rangle$ and $\lvert 110 \rangle$

Note that since there are 2 solutions and 8 possibilities, we will only need to run one iteration (steps 2 & 3).

3.1 Qiskit Implementation

We now implement Grover's algorithm for the above [example](#) for \$3\$-qubits and searching for two marked states $\lvert 101 \rangle$ and $\lvert 110 \rangle$.

We create a phase oracle that will mark states $\lvert 101 \rangle$ and $\lvert 110 \rangle$ as the results (step 1).

```
def phase_oracle(circuit, register):
    circuit.cz(qr[2],qr[0])
    circuit.cz(qr[2],qr[1])
```

Next we set up the circuit for inversion about the average (step 2), where we will first need to define a function that creates a multiple-controlled Z gate.

```
def n_controlled_Z(circuit, controls, target):
    """Implement a Z gate with multiple controls"""
    if (len(controls) > 2):
        raise ValueError('The controlled Z with more than 2 controls is not implemented')
    elif (len(controls) == 1):
        circuit.h(target)
        circuit.cx(controls[0], target)
        circuit.h(target)
    elif (len(controls) == 2):
```

```
circuit.h(target)
circuit.ccx(controls[0], controls[1], target)
circuit.h(target)
```

```
def inversion_about_average(circuit, register, n, barriers):
    """Apply inversion about the average step of Grover's algorithm."""
    circuit.h(register)
    circuit.x(register)

    if barriers:
        circuit.barrier()

    n_controlled_Z(circuit, [register[j] for j in range(n-1)], register[n-1])

    if barriers:
        circuit.barrier()

    circuit.x(register)
    circuit.h(register)
```

Now we put the pieces together, with the creation of a uniform superposition at the start of the circuit and a measurement at the end. Note that since there are 2 solutions and 8 possibilities, we will only need to run one iteration.

```
barriers = True

qr = QuantumRegister(3)
cr = ClassicalRegister(3)

groverCircuit = QuantumCircuit(qr,cr)
groverCircuit.h(qr)

if barriers:
    groverCircuit.barrier()

phase_oracle(groverCircuit, qr)

if barriers:
    groverCircuit.barrier()

inversion_about_average(groverCircuit, qr, 3, barriers)

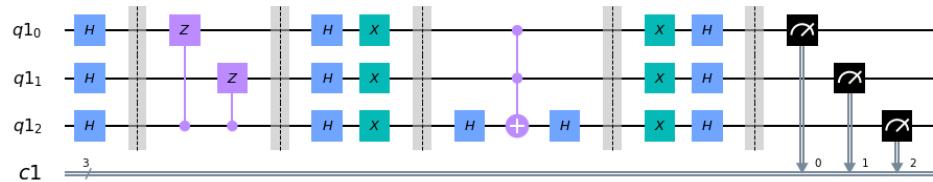
if barriers:
```

```
groverCircuit.barrier()
```

```
groverCircuit.measure(qr,cr)
```

```
<qiskit.circuit.instructionset.InstructionSet at 0x1a26e6e890>
```

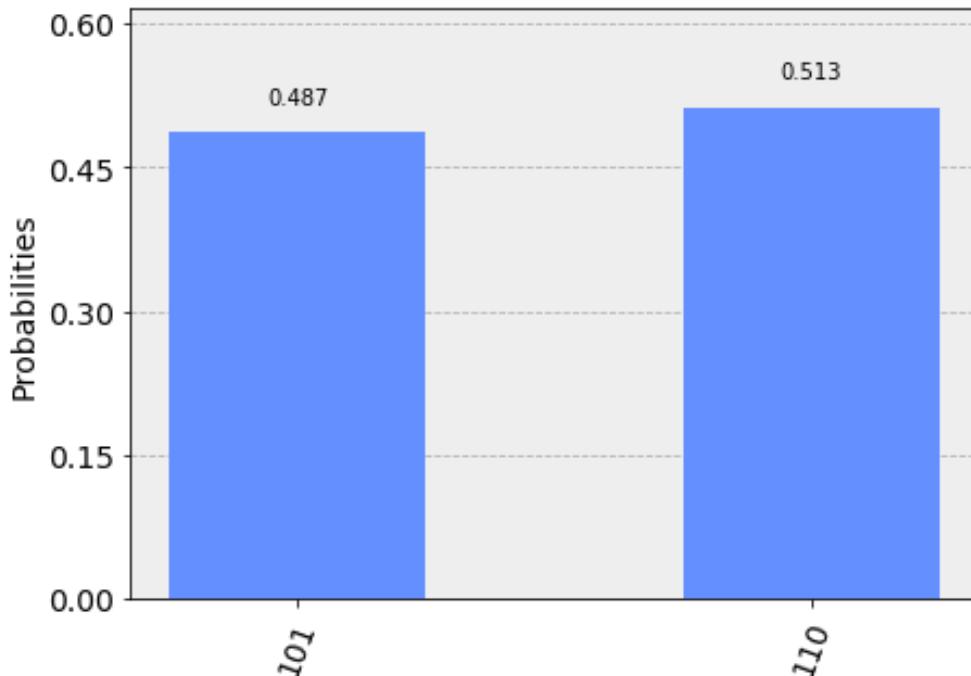
```
groverCircuit.draw(output="mpl")
```



3.1.1 Experiment with Simulators

We can run the above circuit on the simulator.

```
backend = BasicAer.get_backend('qasm_simulator')
shots = 1024
results = execute(groverCircuit, backend=backend, shots=shots).result()
answer = results.get_counts()
plot_histogram(answer)
```



As we can see, the algorithm discovers our marked states $|\psi_{101}\rangle$ and $|\psi_{110}\rangle$.

3.1.2 Experiment with Real Devices

We can run the circuit on the real device as below.

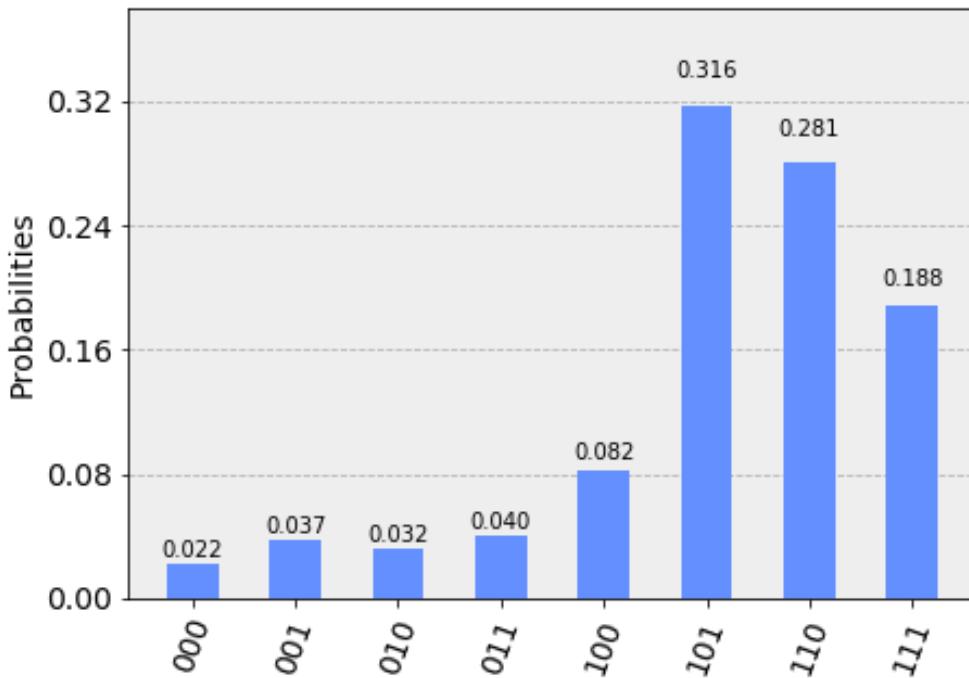
```
backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits <= 3  
                                         and not x.configuration().simulator and x.status().operating)  
print("least busy backend: ", backend)
```

least busy backend: ibmqx2

```
# Run our circuit on the Least busy backend. Monitor the execution of the job in the job monitor.  
from qiskit.tools.monitor import job_monitor  
  
shots = 1024  
job = execute(groverCircuit, backend=backend, shots=shots)  
  
job_monitor(job, interval = 2)
```

Job Status: job has successfully run

```
# Get the results from the computation  
results = job.result()  
answer = results.get_counts(groverCircuit)  
plot_histogram(answer)
```



As we can see, the algorithm discovers our marked states $|\psi_{101}\rangle$ and $|\psi_{110}\rangle$. The other results are due to errors in the quantum computation.

4. Problems

1. The above [example](#) and [implementation](#) of Grover is to find the two marked \$3\$-qubit states $|\psi_{101}\rangle$ and $|\psi_{110}\rangle$. Modify the implementation to find one marked \$2\$-qubit state $|\psi_{01}\rangle$. Are the results what you expect? Explain.
2. The above [example](#) and [implementation](#) of Grover is to find the two marked \$3\$-qubit states $|\psi_{101}\rangle$ and $|\psi_{110}\rangle$. Modify the implementation to find one marked \$4\$-qubit state $|\psi_{0101}\rangle$. Are the results what you expect? Explain.

5. References

1. L. K. Grover (1996), "A fast quantum mechanical algorithm for database search", Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC 1996), [doi:10.1145/237814.237866](https://doi.org/10.1145/237814.237866), [arXiv:quant-ph/9605043](https://arxiv.org/abs/quant-ph/9605043)
2. C. Figgatt, D. Maslov, K. A. Landsman, N. M. Linke, S. Debnath & C. Monroe (2017), "Complete 3-Qubit Grover search on a programmable quantum computer", Nature Communications, Vol 8, Art 1918, [doi:10.1038/s41467-017-01904-7](https://doi.org/10.1038/s41467-017-01904-7), [arXiv:1703.10535](https://arxiv.org/abs/1703.10535)
3. I. Chuang & M. Nielsen, "Quantum Computation and Quantum Information", Cambridge: Cambridge University Press, 2000.

```
import qiskit
qiskit.__qiskit_version__
```

```
{'qiskit-terra': '0.10.0',
 'qiskit-aer': '0.3.2',
 'qiskit-ignis': '0.2.0',
 'qiskit-ibmq-provider': '0.3.3',
 'qiskit-aqua': '0.6.1',
 'qiskit': '0.13.0'}
```

Simulating Molecules using VQE

In this tutorial, we introduce the Variational Quantum Eigensolver (VQE), motivate its use, explain the necessary theory, and demonstrate its implementation in finding the ground state energy of molecules.

Contents

- 1. Introduction
- 2. The Variational Method of Quantum Mechanics
 - 1. Mathematical Background
 - 2. Bounding the Ground State
- 3. The Variational Quantum Eigensolver
 - 1. Variational Forms
 - 2. Simple Variational Forms
 - 3. Parameter Optimization
 - 4. Example with a Single Qubit Variational Form
 - 5. Structure of Common Variational Forms
- 4. VQE Implementation in Qiskit
 - 1. Running VQE on a Statevector Simulator
 - 2. Running VQE on a Noisy Simulator
- 5. Problems
- 6. References

Introduction

In many applications it is important to find the minimum eigenvalue of a matrix. For example, in chemistry, the minimum eigenvalue of a Hermitian matrix characterizing the molecule is the ground state energy of that system. In the future, the quantum phase estimation algorithm may be used to find the minimum eigenvalue. However, its implementation on useful problems requires circuit depths exceeding the limits of hardware available in the NISQ era. Thus, in 2014, Peruzzo *et al.* proposed VQE to estimate the ground state energy of a molecule using much shallower circuits [1].

Formally stated, given a Hermitian matrix H with an unknown minimum eigenvalue $\lambda_{\min'}$ associated with the eigenstate $|\psi_{\min'}\rangle$, VQE provides an estimate λ_θ bounding $\lambda_{\min'}$:

$$\lambda_{\min} \leq \lambda_\theta \equiv \langle \psi(\theta) | H | \psi(\theta) \rangle$$

where $|\psi(\theta)\rangle$ is the eigenstate associated with λ_θ . By applying a parameterized circuit, represented by $U(\theta)$, to some arbitrary starting state $|\psi\rangle$, the algorithm obtains an estimate $U(\theta)|\psi\rangle \equiv |\psi(\theta)\rangle$ on $|\psi_{\min}\rangle$. The estimate is iteratively optimized by a classical controller changing the parameter θ minimizing the expectation value of $\langle\psi(\theta)|H|\psi(\theta)\rangle$.

The Variational Method of Quantum Mechanics

Mathematical Background

VQE is an application of the variational method of quantum mechanics. To better understand the variational method, some preliminary mathematical background is provided. An eigenvector, $|\psi_i\rangle$, of a matrix A is invariant under transformation by A up to a scalar multiplicative constant (the eigenvalue λ_i). That is,

$$A|\psi_i\rangle = \lambda_i|\psi_i\rangle$$

Furthermore, a matrix H is Hermitian when it is equal to its own conjugate transpose.

$$H = H^\dagger$$

The spectral theorem states that the eigenvalues of a Hermitian matrix must be real. Thus, any eigenvalue of H has the property that $\lambda_i = \lambda_{*i}$. As any measurable quantity must be real, Hermitian matrices are suitable for describing the Hamiltonians of quantum systems. Moreover, H may be expressed as

$$H = N \sum i=1 \lambda_i |\psi_i\rangle \langle\psi_i|$$

where each λ_i is the eigenvalue corresponding to the eigenvector $|\psi_i\rangle$. Furthermore, the expectation value of the observable H on an arbitrary quantum state $|\psi\rangle$ is given by

$$\langle H \rangle_\psi \equiv \langle \psi | H | \psi \rangle$$

Substituting H with its representation as a weighted sum of its eigenvectors,

$$\langle H \rangle_\psi = \langle \psi | H | \psi \rangle = \langle \psi \left(N \sum i=1 \lambda_i |\psi_i\rangle \langle\psi_i| \right) | \psi \rangle = N \sum i=1 \lambda_i \langle \psi | \psi_i \rangle \langle \psi_i | \psi \rangle = N \sum i=1 \lambda_i |\langle \psi_i | \psi \rangle|^2$$

The last equation demonstrates that the expectation value of an observable on any state can be expressed as a linear combination using the eigenvalues associated with H as the weights. Moreover, each of the weights in the linear combination is greater than or equal to 0, as $|\langle \psi_i | \psi \rangle|^2 \geq 0$ and so it is clear that

$$\lambda_{\min} \leq \langle H \rangle_\psi = \langle \psi | H | \psi \rangle = N \sum i=1 \lambda_i |\langle \psi_i | \psi \rangle|^2$$

The above equation is known as the **variational method** (in some texts it is also known as the variational principle) [2]. It is important to note that this implies that the expectation value of any wave function will always be at least the minimum eigenvalue associated with H. Moreover, the expectation value of state $|\psi_{\min}\rangle$ is given by $\langle\psi_{\min}|H|\psi_{\min}\rangle = \langle\psi_{\min}|\lambda_{\min}|\psi_{\min}\rangle = \lambda_{\min}$. Thus, as expected, $\langle H \rangle_{\psi_{\min}} = \lambda_{\min}$.

Bounding the Ground State

When the Hamiltonian of a system is described by the Hermitian matrix H the ground state energy of that system, E_{gs} , is the smallest eigenvalue associated with H. By arbitrarily selecting a wave function $|\psi\rangle$ (called an *ansatz*) as an initial guess approximating $|\psi_{\min}\rangle$, calculating its expectation value, $\langle H \rangle_{\psi}$, and iteratively updating the wave function, arbitrarily tight bounds on the ground state energy of a Hamiltonian may be obtained.

The Variational Quantum Eigensolver

Variational Forms

A systematic approach to varying the ansatz is required to implement the variational method on a quantum computer. VQE does so through the use of a parameterized circuit with a fixed form. Such a circuit is often called a *variational form*, and its action may be represented by the linear transformation $U(\theta)$. A variational form is applied to a starting state $|\psi\rangle$ (such as the vacuum state $|0\rangle$, or the Hartree Fock state) and generates an output state $U(\theta)|\psi\rangle \equiv |\psi(\theta)\rangle$. Iterative optimization over $|\psi(\theta)\rangle$ aims to yield an expectation value $\langle\psi(\theta)|H|\psi(\theta)\rangle \approx E_{gs} \equiv \lambda_{\min}$. Ideally, $|\psi(\theta)\rangle$ will be close to $|\psi_{\min}\rangle$ (where 'closeness' is characterized by either state fidelity, or Manhattan distance) although in practice, useful bounds on E_{gs} can be obtained even if this is not the case.

Moreover, a fixed variational form with a polynomial number of parameters can only generate transformations to a polynomially sized subspace of all the states in an exponentially sized Hilbert space. Consequently, various variational forms exist. Some, such as Ry and RyRz are heuristically designed, without consideration of the target domain. Others, such as UCCSD, utilize domain specific knowledge to generate close approximations based on the problem's structure. The structure of common variational forms is discussed in greater depth later in this document.

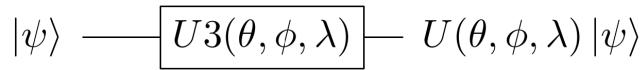
Simple Variational Forms

When constructing a variational form we must balance two opposing goals. Ideally, our n qubit variational form would be able to generate any possible state $|\psi\rangle$ where $|\psi\rangle \in \mathbb{C}^N$ and $N = 2^n$. However, we would like the variational form to use as few parameters as possible. Here, we aim to give intuition for the construction of variational forms satisfying our first goal, while disregarding the second goal for the sake of simplicity.

Consider the case where $n = 1$. The U3 gate takes three parameters, θ , ϕ and λ , and represents the following transformation:

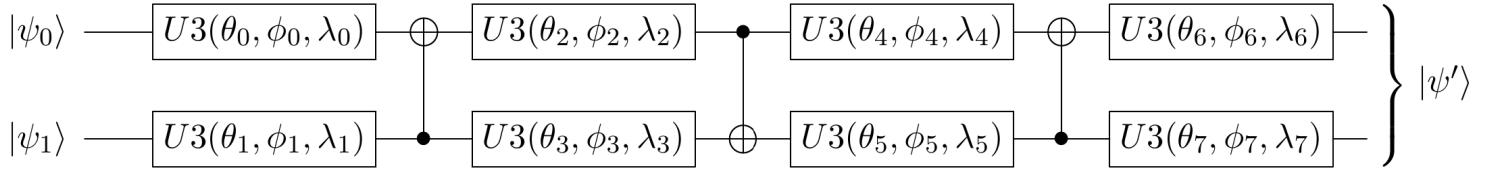
$$U3(\theta, \phi, \lambda) = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2)e^{i\phi}\sin(\theta/2) & e^{i\lambda+i\phi}\cos(\theta/2) \end{pmatrix}$$

Up to a global phase, any possible single qubit transformation may be implemented by appropriately setting these parameters. Consequently, for the single qubit case, a variational form capable of generating any possible state is given by the circuit:



Moreover, this universal 'variational form' only has 3 parameters and thus can be efficiently optimized. It is worth emphasising that the ability to generate an arbitrary state ensures that during the optimization process, the variational form does not limit the set of attainable states over which the expectation value of H can be taken. Ideally, this ensures that the minimum expectation value is limited only by the capabilities of the classical optimizer.

A less trivial universal variational form may be derived for the 2 qubit case, where two body interactions, and thus entanglement, must be considered to achieve universality. Based on the work presented by *Shende et al.* [3] the following is an example of a universal parameterized 2 qubit circuit:



Allow the transformation performed by the above circuit to be represented by $U(\theta)$. When optimized variationally, the expectation value of H is minimized when $U(\theta)|\psi\rangle \equiv |\psi(\theta)\rangle \approx |\psi_{\min}\rangle$. By formulation, $U(\theta)$ may produce a transformation to any possible state, and so this variational form may obtain an arbitrarily tight bound on two qubit ground state energies, only limited by the capabilities of the classical optimizer.

Parameter Optimization

Once an efficiently parameterized variational form has been selected, in accordance with the variational method, its parameters must be optimized to minimize the expectation value of the target Hamiltonian. The parameter optimization process has various challenges. For example, quantum hardware has various types of noise and so objective function evaluation (energy calculation) may not necessarily reflect the true

objective function. Additionally, some optimizers perform a number of objective function evaluations dependent on cardinality of the parameter set. An appropriate optimizer should be selected by considering the requirements of a application.

A popular optimization strategy is gradient decent where each parameter is updated in the direction yielding the largest local change in energy. Consequently, the number of evaluations performed depends on the number of optimization parameters present. This allows the algorithm to quickly find a local optimum in the search space. However, this optimization strategy often gets stuck at poor local optima, and is relatively expensive in terms of the number of circuit evaluations performed. While an intuitive optimization strategy, it is not recommended for use in VQE.

An appropriate optimizer for optimizing a noisy objective function is the *Simultaneous Perturbation Stochastic Approximation* optimizer (SPSA). SPSA approximates the gradient of the objective function with only two measurements. It does so by concurrently perturbing all of the parameters in a random fashion, in contrast to gradient decent where each parameter is perturbed independently. When utilizing VQE in either a noisy simulator or on real hardware, SPSA is a recommended as the classical optimizer.

When noise is not present in the cost function evaluation (such as when using VQE with a statevector simulator), a wide variety of classical optimizers may be useful. Two such optimizers supported by Qiskit Aqua are the *Sequential Least Squares Programming* optimizer (SLSQP) and the *Constrained Optimization by Linear Approximation* optimizer (COBYLA). It is worth noting that COBYLA only performs one objective function evaluation per optimization iteration (and thus the number of evaluations is independent of the parameter set's cardinality). Therefore, if the objective function is noise-free and minimizing the number of performed evaluations is desirable, it is recommended to try COBYLA.

Example with a Single Qubit Variational Form

We will now use the simple single qubit variational form to solve a problem similar to ground state energy estimation. Specifically, we are given a random probability vector $-x$ and wish to determine a possible parameterization for our single qubit variational form such that it outputs a probability distribution that is close to $-x$ (where closeness is defined in terms of the Manhattan distance between the two probability vectors).

We first create the random probability vector in python:

```
import numpy as np
np.random.seed(999999)
target_distr = np.random.rand(2)
# We now convert the random vector into a valid probability vector
target_distr /= sum(target_distr)
```

We subsequently create a function that takes the parameters of our single U3 variational form as arguments and returns the corresponding quantum circuit:

```

from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
def get_var_form(params):
    qr = QuantumRegister(1, name="q")
    cr = ClassicalRegister(1, name='c')
    qc = QuantumCircuit(qr, cr)
    qc.u3(params[0], params[1], params[2], qr[0])
    qc.measure(qr, cr[0])
    return qc

```

Now we specify the objective function which takes as input a list of the variational form's parameters, and returns the cost associated with those parameters:

```

from qiskit import Aer, execute
backend = Aer.get_backend("qasm_simulator")
NUM_SHOTS = 10000

def get_probability_distribution(counts):
    output_distr = [v / NUM_SHOTS for v in counts.values()]
    if len(output_distr) == 1:
        output_distr.append(0)
    return output_distr

def objective_function(params):
    # Obtain a quantum circuit instance from the parameters
    qc = get_var_form(params)
    # Execute the quantum circuit to obtain the probability distribution associated with the circuit
    result = execute(qc, backend, shots=NUM_SHOTS).result()
    # Obtain the counts for each measured state, and convert those counts into a probability vector
    output_distr = get_probability_distribution(result.get_counts(qc))
    # Calculate the cost as the distance between the output distribution and the target distribution
    cost = sum([np.abs(output_distr[i] - target_distr[i]) for i in range(2)])
    return cost

```

Finally, we create an instance of the COBYLA optimizer, and run the algorithm. Note that the output varies from run to run. Moreover, while close, the obtained distribution might not be exactly the same as the target distribution, however, increasing the number of shots taken will increase the accuracy of the output.

```

from qiskit.aqua.components.optimizers import COBYLA

# Initialize the COBYLA optimizer
optimizer = COBYLA(maxiter=500, tol=0.0001)

# Create the initial parameters (noting that our single qubit variational form has 3 parameters)
params = np.random.rand(3)
ret = optimizer.optimize(num_vars=3, objective_function=objective_function, initial_point=params)

```

```

# Obtain the output distribution using the final parameters
qc = get_var_form(ret[0])
counts = execute(qc, backend, shots=NUM_SHOTS).result().get_counts(qc)
output_distr = get_probability_distribution(counts)

print("Target Distribution:", target_distr)
print("Obtained Distribution:", output_distr)
print("Output Error (Manhattan Distance):", ret[1])
print("Parameters Found:", ret[0])

```

```

Target Distribution: [0.51357006 0.48642994]
Obtained Distribution: [0.5182, 0.4818]
Output Error (Manhattan Distance): 0.0001401187388391789
Parameters Found: [1.59966854 0.66273002 0.28432001]

```

Structure of Common Variational Forms

As already discussed, it is not possible for a polynomially parameterized variational form to generate a transformation to any state. Variational forms can be grouped into two categories, depending on how they deal with this limitation. The first category of variational forms use domain or application specific knowledge to limit the set of possible output states. The second approach uses a heuristic circuit without prior domain or application specific knowledge.

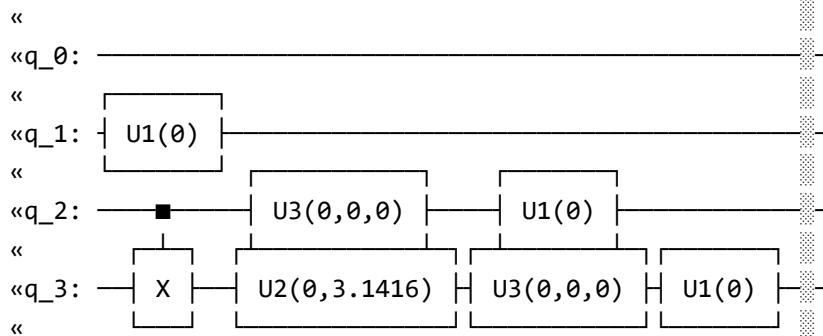
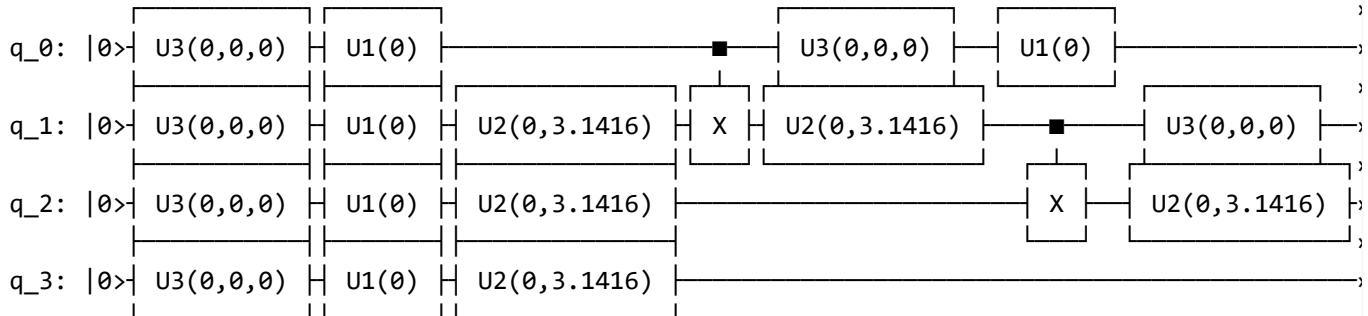
The first category of variational forms exploit characteristics of the problem domain to restrict the set of transformations that may be required. For example, when calculating the ground state energy of a molecule, the number of particles in the system is known *a priori*. Therefore, if a starting state with the correct number of particles is used, by limiting the variational form to only producing particle preserving transformations, the number of parameters required to span the new transformation subspace can be greatly reduced. Indeed, by utilizing similar information from Coupled-Cluster theory, the variational form UCCSD can obtain very accurate results for molecular ground state energy estimation when starting from the Hartree Fock state. Another example illustrating the exploitation of domain-specific knowledge follows from considering the set of circuits realizable on real quantum hardware. Extant quantum computers, such as those based on super conducting qubits, have limited qubit connectivity. That is, it is not possible to implement 2-qubit gates on arbitrary qubit pairs (without inserting swap gates). Thus, variational forms have been constructed for specific quantum computer architectures where the circuits are specifically tuned to maximally exploit the natively available connectivity and gates of a given quantum device. Such a variational form was used in 2017 to successfully implement VQE for the estimation of the ground state energies of molecules as large as BeH₂ on an IBM quantum computer [4].

In the second approach, gates are layered such that good approximations on a wide range of states may be obtained. Qiskit Aqua supports three such variational forms: RyRz, Ry and SwapRz (we will only discuss the first two). All of these variational forms accept multiple user-specified configurations. Three essential configurations are the number of qubits in the system, the depth setting, and the entanglement setting. A

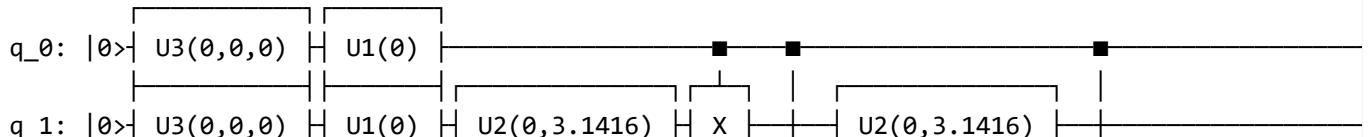
single layer of a variational form specifies a certain pattern of single qubit rotations and CX gates. The depth setting says how many times the variational form should repeat this pattern. By increasing the depth setting, at the cost of increasing the number of parameters that must be optimized, the set of states the variational form can generate increases. Finally, the entanglement setting selects the configuration, and implicitly the number, of CX gates. For example, when the entanglement setting is linear, CX gates are applied to adjacent qubit pairs in order (and thus $n - 1$ CX gates are added per layer). When the entanglement setting is full, a CX gate is applied to each qubit pair in each layer. The circuits for RyRz corresponding to `entanglement="full"` and `entanglement="linear"` can be seen by executing the following code snippet:

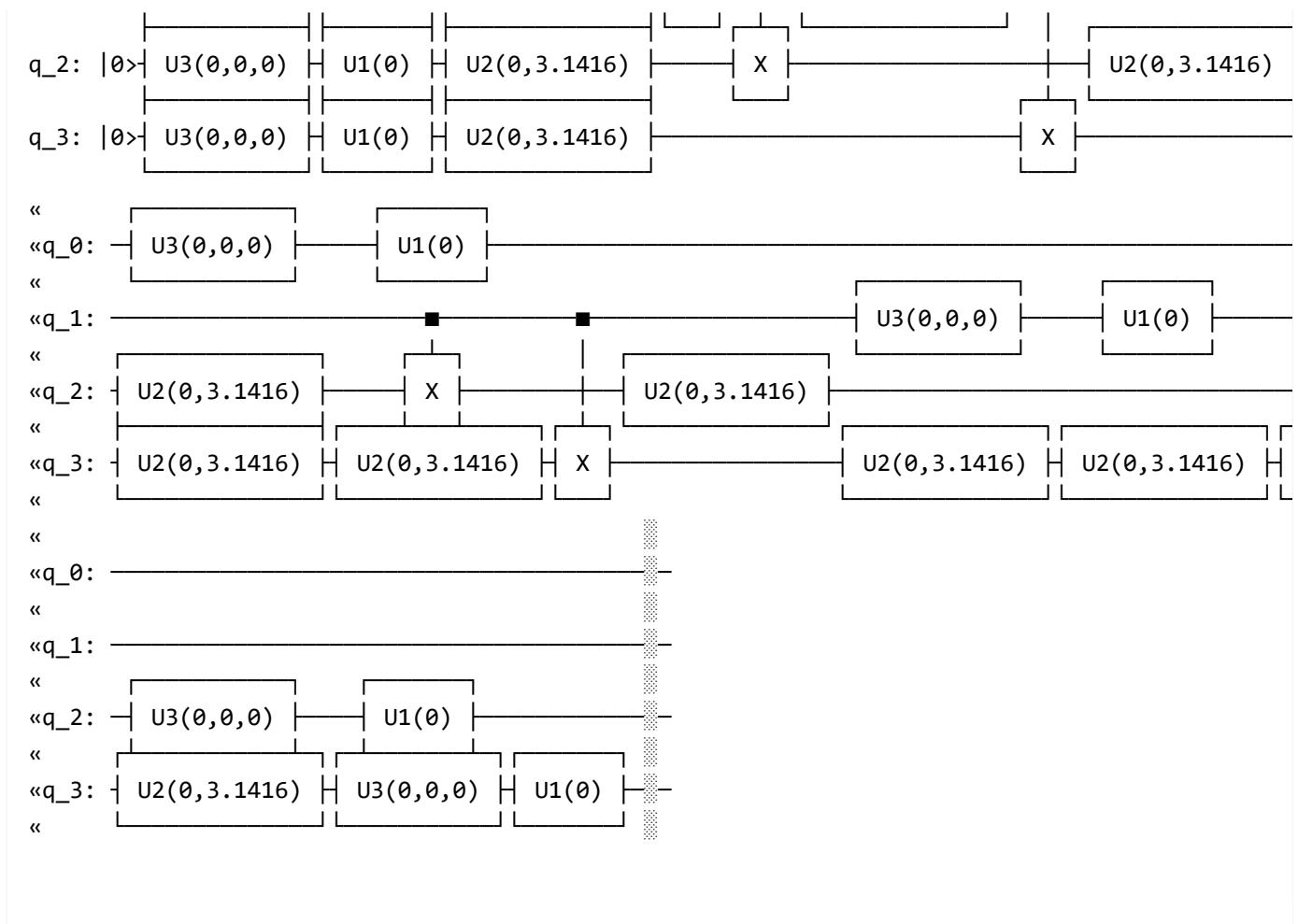
```
from qiskit.aqua.components.variational_forms import RYRZ
entanglements = ["linear", "full"]
for entanglement in entanglements:
    form = RYRZ(num_qubits=4, depth=1, entanglement=entanglement)
    if entanglement == "linear":
        print("=====Linear Entanglement====")
    else:
        print("=====Full Entanglement====")
    # We initialize all parameters to 0 for this demonstration
    print(form.construct_circuit([0] * form.num_parameters).draw(line_length=100))
    print()
```

=====Linear Entanglement=====



=====Full Entanglement=====





Assume the depth setting is set to d . Then, RyRz has $n \times (d + 1) \times 2$ parameters, Ry with linear entanglement has $2n \times (d + 1)^2$ parameters, and Ry with full entanglement has $d \times n \times (n+1)^2 + n$ parameters.

VQE Implementation in Qiskit

This section illustrates an implementation of VQE using the programmatic approach. Qiskit Aqua also enables a declarative implementation, however, it reveals less information about the underlying algorithm. This code, specifically the preparation of qubit operators, is based on the code found in the Qiskit Tutorials repository (and as of July 2019, may be found at: <https://github.com/Qiskit/qiskit-tutorials>).

The following libraries must first be imported.

```
from qiskit.aqua.algorithms import VQE, ExactEigensolver
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
from qiskit.chemistry.aqua_extensions.components.variational_forms import UCCSD
from qiskit.aqua.components.variational_forms import RYRZ
from qiskit.chemistry.aqua_extensions.components.initial_states import HartreeFock
```

```

from qiskit.aqua.components.optimizers import COBYLA, SPSA, SLSQP
from qiskit import IBMQ, BasicAer, Aer
from qiskit.chemistry.drivers import PySCFDriver, UnitsType
from qiskit.chemistry import FermionicOperator
from qiskit import IBMQ
from qiskit.providers.aer import noise
from qiskit.aqua import QuantumInstance
from qiskit.ignis.mitigation.measurement import CompleteMeasFitter

```

Running VQE on a Statevector Simulator

We demonstrate the calculation of the ground state energy for LiH at various interatomic distances. A driver for the molecule must be created at each such distance. Note that in this experiment, to reduce the number of qubits used, we freeze the core and remove two unoccupied orbitals. First, we define a function that takes an interatomic distance and returns the appropriate qubit operator, H, as well as some other information about the operator.

```

def get_qubit_op(dist):
    driver = PySCFDriver(atom="Li .0 .0 .0; H .0 .0 " + str(dist), unit=UnitsType.ANGSTROM,
                         charge=0, spin=0, basis='sto3g')
    molecule = driver.run()
    freeze_list = [0]
    remove_list = [-3, -2]
    repulsion_energy = molecule.nuclear_repulsion_energy
    num_particles = molecule.num_alpha + molecule.num_beta
    num_spin_orbitals = molecule.num_orbitals * 2
    remove_list = [x % molecule.num_orbitals for x in remove_list]
    freeze_list = [x % molecule.num_orbitals for x in freeze_list]
    remove_list = [x - len(freeze_list) for x in remove_list]
    remove_list += [x + molecule.num_orbitals - len(freeze_list) for x in remove_list]
    freeze_list += [x + molecule.num_orbitals for x in freeze_list]
    ferOp = FermionicOperator(h1=molecule.one_body_integrals, h2=molecule.two_body_integrals)
    ferOp, energy_shift = ferOp.fermion_mode_freezing(freeze_list)
    num_spin_orbitals -= len(freeze_list)
    num_particles -= len(freeze_list)
    ferOp = ferOp.fermion_mode_elimination(remove_list)
    num_spin_orbitals -= len(remove_list)
    qubitOp = ferOp.mapping(map_type='parity', threshold=0.0000001)
    qubitOp = qubitOp.two_qubit_reduced_operator(num_particles)
    shift = energy_shift + repulsion_energy
    return qubitOp, num_particles, num_spin_orbitals, shift

```

First, the exact ground state energy is calculated using the qubit operator and a classical exact eigensolver. Subsequently, the initial state $|\psi\rangle$ is created, which the VQE instance uses to produce the final ansatz $\min_{\theta}(|\psi(\theta)\rangle)$. The exact result and the VQE result at each interatomic distance is stored. Observe that the

result given by `vqe.run(backend)['energy'] + shift` is equivalent the quantity $\min_{\theta}(\langle \psi(\theta) | H | \psi(\theta) \rangle)$, where the minimum is not necessarily the global minimum.

When initializing the VQE instance with `VQE(qubitOp, var_form, optimizer, 'matrix')` the expectation value of H on $|\psi(\theta)\rangle$ is directly calculated through matrix multiplication. However, when using an actual quantum device, or a true simulator such as the `qasm_simulator` with `VQE(qubitOp, var_form, optimizer, 'paulis')` the calculation of the expectation value is more complicated. A Hamiltonian may be represented as a sum of a Pauli strings, with each Pauli term acting on a qubit as specified by the mapping being used. Each Pauli string has a corresponding circuit appended to the circuit corresponding to $|\psi(\theta)\rangle$.

Subsequently, each of these circuits is executed, and all of the results are used to determine the expectation value of H on $|\psi(\theta)\rangle$. In the following example, we initialize the VQE instance with `matrix` mode, and so the expectation value is directly calculated through matrix multiplication.

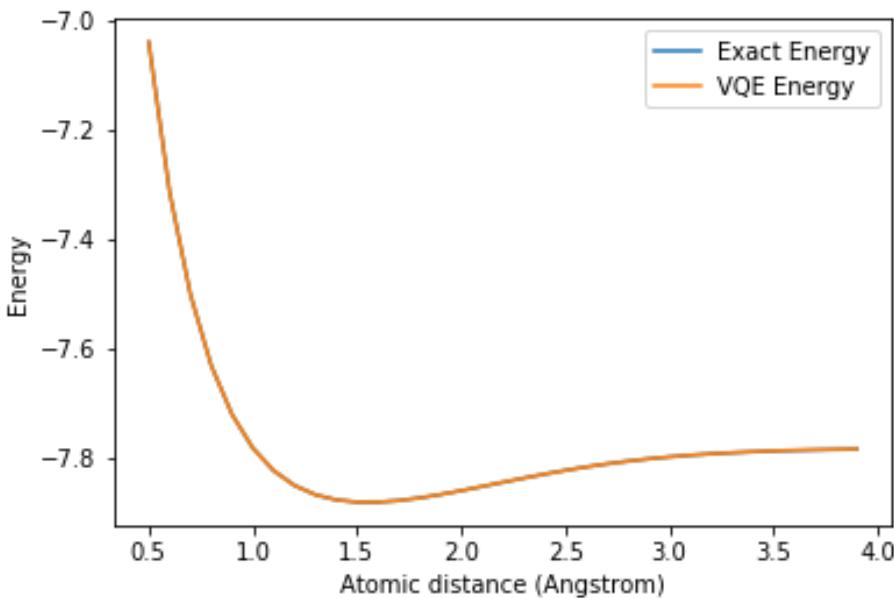
Note that the following code snippet may take a few minutes to run to completion.

```
backend = BasicAer.get_backend("statevector_simulator")
distances = np.arange(0.5, 4.0, 0.1)
exact_energies = []
vqe_energies = []
optimizer = SLSQP(maxiter=5)
for dist in distances:
    qubitOp, num_particles, num_spin_orbitals, shift = get_qubit_op(dist)
    result = ExactEigensolver(qubitOp).run()
    exact_energies.append(result[ 'energy' ] + shift)
    initial_state = HartreeFock(
        qubitOp.num_qubits,
        num_spin_orbitals,
        num_particles,
        'parity'
    )
    var_form = UCCSD(
        qubitOp.num_qubits,
        depth=1,
        num_orbitals=num_spin_orbitals,
        num_particles=num_particles,
        initial_state=initial_state,
        qubit_mapping='parity'
    )
    vqe = VQE(qubitOp, var_form, optimizer, 'matrix')
    results = vqe.run(backend)[ 'energy' ] + shift
    vqe_energies.append(results)
    print("Interatomic Distance:", np.round(dist, 2), "VQE Result:", results, "Exact Energy:",
          results)
print("All energies have been calculated")
```

Interatomic Distance: 0.5 VQE Result: -7.039710219020506 Exact Energy: -7.039732521635202
Interatomic Distance: 0.6 VQE Result: -7.313344302334236 Exact Energy: -7.313345828761008
Interatomic Distance: 0.7 VQE Result: -7.500921095743192 Exact Energy: -7.500922090905936
Interatomic Distance: 0.8 VQE Result: -7.630976914468914 Exact Energy: -7.630978249333209
Interatomic Distance: 0.9 VQE Result: -7.7208107952020795 Exact Energy: -7.720812412134773
Interatomic Distance: 1.0 VQE Result: -7.782240655298441 Exact Energy: -7.782242402637011
Interatomic Distance: 1.1 VQE Result: -7.823597493320795 Exact Energy: -7.82359927636281
Interatomic Distance: 1.2 VQE Result: -7.850696622934822 Exact Energy: -7.850698377596024
Interatomic Distance: 1.3 VQE Result: -7.867561602181376 Exact Energy: -7.867563290110052
Interatomic Distance: 1.4 VQE Result: -7.876999876757721 Exact Energy: -7.877001491818373
Interatomic Distance: 1.5 VQE Result: -7.8810141736656405 Exact Energy: -7.881015715646992
Interatomic Distance: 1.6 VQE Result: -7.881070662952161 Exact Energy: -7.88107204403092
Interatomic Distance: 1.7 VQE Result: -7.878267162143656 Exact Energy: -7.878268167584993
Interatomic Distance: 1.8 VQE Result: -7.873440112155302 Exact Energy: -7.873440293132828
Interatomic Distance: 1.9 VQE Result: -7.86723366674701 Exact Energy: -7.8672339648160285
Interatomic Distance: 2.0 VQE Result: -7.860152327529411 Exact Energy: -7.86015320737878
Interatomic Distance: 2.1 VQE Result: -7.852595105536979 Exact Energy: -7.852595827876738
Interatomic Distance: 2.2 VQE Result: -7.844878726366329 Exact Energy: -7.844879093009722
Interatomic Distance: 2.3 VQE Result: -7.837257439448259 Exact Energy: -7.8372579676155025
Interatomic Distance: 2.4 VQE Result: -7.829935045088515 Exact Energy: -7.829937002623394
Interatomic Distance: 2.5 VQE Result: -7.823070191557451 Exact Energy: -7.82307664213409
Interatomic Distance: 2.6 VQE Result: -7.816782591999657 Exact Energy: -7.816795150472929
Interatomic Distance: 2.7 VQE Result: -7.8111534373726 Exact Energy: -7.811168284803366
Interatomic Distance: 2.8 VQE Result: -7.806218299266321 Exact Energy: -7.806229560089845
Interatomic Distance: 2.9 VQE Result: -7.801962397475152 Exact Energy: -7.8019736023325486
Interatomic Distance: 3.0 VQE Result: -7.798352412318197 Exact Energy: -7.7983634309151295
Interatomic Distance: 3.1 VQE Result: -7.795326815750017 Exact Energy: -7.795340451637537
Interatomic Distance: 3.2 VQE Result: -7.792800698225245 Exact Energy: -7.792834806738612
Interatomic Distance: 3.3 VQE Result: -7.790603799019874 Exact Energy: -7.790774009971014
Interatomic Distance: 3.4 VQE Result: -7.788715354695274 Exact Energy: -7.789088897991478
Interatomic Distance: 3.5 VQE Result: -7.787215781080283 Exact Energy: -7.787716973466144
Interatomic Distance: 3.6 VQE Result: -7.786080393658009 Exact Energy: -7.786603763673838
Interatomic Distance: 3.7 VQE Result: -7.785203497342158 Exact Energy: -7.785702912499886
Interatomic Distance: 3.8 VQE Result: -7.7844795319924325 Exact Energy: -7.784975591698873
Interatomic Distance: 3.9 VQE Result: -7.783853361693722 Exact Energy: -7.7843896116723315

All energies have been calculated

```
plt.plot(distances, exact_energies, label="Exact Energy")
plt.plot(distances, vqe_energies, label="VQE Energy")
plt.xlabel('Atomic distance (Angstrom)')
plt.ylabel('Energy')
plt.legend()
plt.show()
```



Note that the VQE results are very close to the exact results, and so the exact energy curve is hidden by the VQE curve.

Running VQE on a Noisy Simulator

Here, we calculate the ground state energy for H_2 using a noisy simulator and error mitigation.

First, we prepare the qubit operator representing the molecule's Hamiltonian:

```

driver = PySCFDriver(atom='H .0 .0 -0.3625; H .0 .0 0.3625', unit=UnitsType.ANGSTROM, charge=0)
molecule = driver.run()
num_particles = molecule.num_alpha + molecule.num_beta
qubitOp = FermionicOperator(h1=molecule.one_body_integrals, h2=molecule.two_body_integrals).map
qubitOp = qubitOp.two_qubit_reduced_operator(num_particles)

```

Now, we load a device coupling map and noise model from the IBMQ provider and create a quantum instance, enabling error mitigation:

Finally, we must configure the optimizer, the variational form, and the VQE instance. As the effects of noise increase as the number of two qubit gates circuit depth increase, we use a heuristic variational form (RYRZ) rather than UCCSD as RYRZ has a much shallower circuit than UCCSD and uses substantially fewer two qubit gates.

The following code may take a few minutes to run to completion.

```
exact_solution = ExactEigensolver(qubitOp).run()
print("Exact Result:", exact_solution['energy'])
optimizer = SPSA(max_trials=100)
var_form = RYRZ(qubitOp.num_qubits, depth=1, entanglement="linear")
vqe = VQE(qubitOp, var_form, optimizer=optimizer, operator_mode="grouped_paulis")
ret = vqe.run(quantum_instance)
print("VQE Result:", ret['energy'])
```

```
Exact Result: -1.86712097834127
VQE Result: -1.8220854070067132
```

When noise mitigation is enabled, even though the result does not fall within chemical accuracy (defined as being within 0.0016 Hartree of the exact result), it is fairly close to the exact solution.

Problems

1. You are given a Hamiltonian H with the promise that its ground state is close to a maximally entangled n qubit state. Explain which variational form (or forms) is likely to efficiently and accurately learn the the ground state energy of H . You may also answer by creating your own variational form, and explaining why it is appropriate for use with this Hamiltonian.
2. Calculate the number of circuit evaluations performed per optimization iteration, when using the COBYLA optimizer, the `qasm_simulator` with 1000 shots, and a Hamiltonian with 60 Pauli strings.
3. Use VQE to estimate the ground state energy of BeH_2 with an interatomic distance of 1.3\AA . You may re-use the function `get_qubit_op(dist)` by replacing `atom="Li .0 .0 .0; H .0 .0 "` + str(`dist`) with `atom="Be .0 .0 .0; H .0 .0 -"` + str(`dist`) + `"; H .0 .0 "` + str(`dist`) and invoking the function with `get_qubit_op(1.3)`. Note that removing the unoccupied orbitals does not preserve chemical precision for this molecule. However, to get the number of qubits required down to 6 (and thereby allowing efficient simulation on most laptops), the loss of precision is acceptable. While beyond the scope of this exercise, the interested reader may use qubit tapering operations to reduce the number of required qubits to 7, without losing any chemical precision.

References

1. Peruzzo, Alberto, et al. "A variational eigenvalue solver on a photonic quantum processor." *Nature communications* 5 (2014): 4213.
2. Griffiths, David J., and Darrell F. Schroeter. Introduction to quantum mechanics. *Cambridge University Press*, 2018.
3. Shende, Vivek V., Igor L. Markov, and Stephen S. Bullock. "Minimal universal two-qubit cnot-based circuits." arXiv preprint quant-ph/0308033 (2003).
4. Kandala, Abhinav, et al. "Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets." *Nature* 549.7671 (2017): 242.

Solving combinatorial optimization problems using QAOA

In this tutorial, we introduce combinatorial optimization problems, explain approximate optimization algorithms, explain how the Quantum Approximate Optimization Algorithm (QAOA) works and present the implementation of an example that can be run on a simulator or on a 5 qubit quantum chip

Contents

1. [Introduction](#)
2. [Examples](#)
3. [Approximate optimization algorithms](#)
4. [The QAOA algorithm](#)
5. [Qiskit Implementation](#)
 - o 5a [Running QAOA on a simulator](#)
 - o 5b [Running QAOA on a real quantum device](#)
6. [Problems](#)
7. [References](#)

1. Introduction

Combinatorial optimization [1](#) means searching for an optimal solution in a finite or countably infinite set of potential solutions. Optimality is defined with respect to some criterion function, which is to be minimized or maximized, which is typically called the cost function.

There are various types of optimization problems. These include Minimization: cost, distance, length of a traversal, weight, processing time, material, energy consumption, number of objects. Maximization: profit, value, output, return, yield, utility, efficiency, capacity, number of objects. Any maximization problem can be cast in terms of a minimization problem and vice versa. Hence the general form a combinatorial optimization problem is given by

maximize $C(x)$ subject to $x \in S$

where $x \in S$, is a discreet variable and $C: D \rightarrow R$ is the cost function. That maps from some domain S in to the real numbers R . The variable x can be subject to a set of constraints and lies within the set $S \subset D$ of feasible points.

In binary combinatorial optimization problems, the cost function can typically be expressed as a sum of terms that only involve a subsets $Q \subset [n]$ of the n bits in the string $x \in \{0, 1\}^n$. The cost function C typically written in the canonical form

$$C(x) = \sum_{(Q, \bar{Q}) \subset [n]} w_{(Q, \bar{Q})} \prod_{i \in Q} x_i \prod_{j \in \bar{Q}} (1 - x_j),$$

where $x_i \in \{0, 1\}$ and $w_{(Q, \bar{Q})} \in \mathbb{R}$. We want to find the n -bit string x for which $C(x)$ is the maximal.

1.1 Diagonal Hamiltonians

This cost function can be mapped to a Hamiltonian that is diagonal in the computational basis. Given the cost-fucntion C this Hamiltonian is then written as

$$H = \sum_{x \in \{0, 1\}^n} C(x) |x\rangle\langle x|$$

where $x \in \{0, 1\}^n$ labels the computational basis states $|x\rangle \in \mathbb{C}^{2^n}$. If the cost function only has has at most weight k terms, i.e. when only Q contribute that involve at most $Q \leq k$ bits, then this diagonal Hamiltonian is also only a sum of weight k Pauli Z operators.

The expansion of H in to Pauli Z operators can be obtained from the cannonical expansion expantion of the cost-function C by substituting for every binary variable $x_i \in \{0, 1\}$ the matrix

$x_i \rightarrow 2^{-1}(1 - Z_i)$. Here Z_i is read as the Pauli Z operator that acts on qubit i and trivial on all others, i.e.

$$Z_i = \begin{pmatrix} 1 & 00 & -1 \end{pmatrix}.$$

This means that the spin - Hamiltonian encoding the classical cost funtion is written as a $|Q|$ - local quantum spin Hamiltonain only involving Pauli Z- operators.

$$H = \sum_{(Q, \bar{Q}) \subset [n]} w_{(Q, \bar{Q})} 12^{|Q|+|\bar{Q}|} \prod_{i \in Q} (1 - Z_i) \prod_{j \in \bar{Q}} (1 + Z_j).$$

Now, we will assume that only a few (polynomially many in n) $w_{(Q, \bar{Q})}$ will be non-zero. Morover we will assume that the set $|(Q, \bar{Q})|$ is bounded and not too large. This means we can write the cost function as well as the Hamiltonain H as sum of m local terms \hat{C}_k

$$H = n \sum_{k=1}^m \hat{C}_k$$

where both m and the support of \hat{C}_k is reasonably bounded.

2 Examples:

We consider 2 examples to illustrate combinatorial optimization problems. We will only implement the first example as in Qiskit, but provide a sequence of exercises that give the instructions to implement the second example as well.

2.1 (weighted) MAXCUT

Consider an n -node non-directed graph $G = (V, E)$ where $|V| = n$ with edge weights $w_{ij} > 0$, $w_{ij} = w_{ji}$ for $(j, k) \in E$. A cut is defined as a partition of the original set V into two subsets. The cost function to be optimized is in this case the sum of weights of edges connecting points in the two different subsets, *crossing* the cut. By assigning $x_i = 0$ or $x_i = 1$ to each node i , one tries to maximize the global profit function (here and in the following summations run over indices $0, 1, \dots, n-1$)

$$C(x) = n \sum_{i,j=1}^n w_{ij} x_i (1 - x_j).$$

To simplify notation, we assume uniform weights $w_{ij} = 1$ for $(i, j) \in E$. In order to find a solution to this problem on a quantum computer, one needs first to map it to a diagonal Hamiltonian as discussed above. We write the sum as a sum over edges in the set $(i, j) = E$

$$C(x) = n \sum_{i,j=1}^n w_{ij} x_i (1 - x_j) = \sum_{(i,j) \in E} (x_i (1 - x_j) + x_j (1 - x_i))$$

To map it to a spin Hamiltonian we make the assignment $x_i \rightarrow (1 - Z_i)/2$, where Z_i is the Pauli Z operator that has eigenvalues ± 1 and obtain $X \rightarrow H$

$$H = \sum_{(j,k) \in E} (1 - Z_j Z_k).$$

This means that the Hamiltonian can be written as a sum of $m = |E|$ local terms

$$\hat{C}_e = (1 - Z_{e1} Z_{e2}) \text{ with } e = (e1, e2) \in E.$$

2.2 Constraint satisfaction problems and MAX3 – SAT.

Another set of examples of a combinatorial optimization problem is 3 – SAT. Here the cost function $C(x) = \sum_{mk=1}^m c_k(x)$ is a sum of clauses $c_k(x)$ that constrain the values of 3 bits of some $x \in \{0, 1\}^n$ that participate in the clause. Consider for instance this example of a 3 – SAT clause

$$c_1(x) = (1 - x_1)(1 - x_3)x_{132}$$

for a bit string $x \in \{0, 1\}^{133}$.¹³³ The clause can only be satisfied by setting the bits $x_1 = 0, x_3 = 0$ and $x_{132} = 1$. The 3 – SAT problem now asks whether there is a bit -string that satisfies all of the m clauses or whether no such string exists. This decision problem is the prime example of a problem that is NP- complete.

The closely related optimization problem MAX3 – SAT asks to find the bit string x that satisfies the maximal number of of clauses in $C(x)$. This can of course be turned again in to a decision problem if we ask where there exists a bit string that satisfies more than \tilde{m} of the m clauses, which is again NP-complete.

3. Approximate optimization algorithms

Both the previsously considered problems MAXCUT and MAX3 – SAT are actually known to be a NP-hard problems [1](#). In fact it turns out that many combinatorial optimization problems are computationally hard to solve in general. In light of this fact, we can't expect to find a provably efficient algorithm, i.e. an algorithm with polynomial runtime in the problem size, that solves these problems. This also applies to quantum algorithms. One possible approach to such problems is to develop heuristic algortihm that don't have a polynomial runtime gurantee but appear to perform well on some instances of such problems. Another alternative are approximate algorithms.

Approximate optimization algorithms find approximate solutions to NP-hard optimization problems. These algorithms are efficient and provide a provable guarantee on how close the approximate solution is to the actual optimum of the problem.

The guarantee typicall comes in form of an approximation ratio, $\alpha \leq 0$. A probabilistic approximate optimization algorithm guarantees that it produces a bit-string $x^* \in \{0, 1\}^n$ so that *with high probability* we have that with a postive $C_{\max} = \max_x C(x)$

$$C_{\max} \geq C(x^*) \geq \alpha C_{\max}$$

For the MAXCUT problem there is a famous approximate algorithm due to Goemans and Williamson [2](#) . This algorithm is based on an SDP relaxation of the original problem combined with a probabilistic rounding techinque that yields an with high probabiltiy approxiamte solution x^* that has an approximation ratio of $\alpha \approx 0.868$. This approximation ratio is actually believed to optimal so that we do not expect to see an improvement by using a quantum algorithm.

4. The QAOA algorithm

The Quantum approximate optimization algorithm (QAOA) by Farhi, Goldsone and Gutmann takes the approach of classical approximate algorithms and looks for a quantum analogue that will likewise produce a classical bit -string x^* that with high probability is expected to have a good approximation ratio α . Before discussing the details, let us first present the general idea of this approach.

4.1 Overview:

We want to find a quantum state $|\psi_p(-\gamma, -\beta)\rangle$, that depends on some real parameters $-\gamma, -\beta \in \mathbb{R}^p$, which has the property that it maximizes the expectation value with respect to the problem Hamiltonian H . Given this trial state we search for parameters $-\gamma^*, -\beta^*$ that maximize $F_p(-\gamma, -\beta) = \langle \psi_p(-\gamma, -\beta) | H | \psi_p(-\gamma, -\beta) \rangle$.

Once we have such a state and the corresponding parameters we prepare the state $|\psi_p(-\gamma^*, -\beta^*)\rangle$ on a quantum computer and measure the state in the Z basis $|x\rangle = |x_1, \dots, x_n\rangle$ to obtain a random outcome x^* .

We will see that this random x^* is going to be a bit-string that is with high probability close to the expected value $M_p = F_p(-\gamma^*, -\beta^*)$. Hence, if M_p is close to C_{\max} so is $C(x^*)$.

4.2 The components of the QAOA algorithm.

4.2.1 The QAOA trial state

Central to QAOA is the trial state $|\psi_p(-\gamma, -\beta)\rangle$ that will be prepared on the quantum computer.

Ideally we want this state to give rise to a large expectation value $F_p(-\gamma, -\beta) = \langle \psi_p(-\gamma, -\beta) | H | \psi_p(-\gamma, -\beta) \rangle$ with respect to the problem Hamiltonian H . In CITE Farhi, the trial states $|\psi_p(-\gamma, -\beta)\rangle$ are constructed from the problem Hamiltonian H together with single qubit Pauli X rotations. That means, given a problems Hamiltonian $H = m \sum k=1^m C_k$ diagonal in the computational basis and a transverse field Hamiltonian $B = n \sum i=1^n X_i$ the trial state is prepared by applying p alternating unitaries

$$|\psi_p(-\gamma, -\beta)\rangle = e^{-i\beta_p B} e^{-i\gamma_p H} \dots e^{-i\beta_1 B} e^{-i\gamma_1 H} |+\rangle^n$$

to the product state $| + \rangle^n$ with $X| + \rangle = | + \rangle$.

This particular ansatz has the advantage that there exists an explicit choice for the vectors $-\gamma^*, -\beta^*$ such that for $M_p = F_p(-\alpha^*, -\beta^*)$ when we take the limit $\lim_{p \rightarrow \infty} M_p = C_{\max}$. This follows by viewing the trial state $|\psi_p(-\gamma, -\beta)\rangle$ as the state that follows from Trotterizing the adiabatic evolution with respect to H and the transverse field Hamiltonian B , c.f. Ref 3.

Conversely the disadvantage of this trial state is one would typically want a state that has been generated from a quantum circuit that is not too deep. Here depth is measured with respect to the gates that can be applied directly on the quantum chip. Hence there are other proposals that suggest using Ansatz trial states that are more tailored to the hardware of the quantum chip Ref. 4, Ref. 5.

4.2.2 Computing the expectation value

An important component of this approach is that we will have to compute or estimate the expectation value $F_p(-\gamma, -\beta) = \langle \psi_p(-\gamma, -\beta) | H | \psi_p(-\gamma, -\beta) \rangle$ so we can optimize the parameters $\vec{\gamma}, \vec{\beta}$. We will be considering two scenarios here.

Classical evaluation

Note that when the circuit to prepare $|\psi_p(\vec{\alpha}, \vec{\beta})\rangle$ is not too deep it may be possible to evaluate the expectation value F_p classically.

This happens for instance when one considers MAXCUT for graphs with bounded degree and one considers a circuit with $p=1$. We will see an example of this in the Qiskit implementation below (section 5.2) and provide an exercise to compute the expectation value.

To illustrate the idea, recall that the Hamiltonian can be written as a sum of individual terms $H = \sum_{k=1}^m \hat{C}_k$. Due to the linearity of the expectation value, it is sufficient to consider the expectation values of the individual summands. For $p=1$ one has that

$$\langle \psi_1(\vec{\gamma}, \vec{\beta}) | \hat{C}_k | \psi_1(\vec{\alpha}, \vec{\beta}) \rangle = \langle \psi_1(\vec{\gamma}, \vec{\beta}) | e^{i\gamma_1 H} e^{i\beta_1 B} | \hat{C}_k | e^{-i\beta_1 B} e^{-i\gamma_1 H} | \psi_1(\vec{\alpha}, \vec{\beta}) \rangle$$

Observe that with $B = \sum_{i=1}^n X_i$ the unitary $e^{i\beta_1 B}$ is actually a product of single qubit rotations about X with an angle β for which we will write $X(\beta)_k = \exp(i\beta X_k)$.

All the individual rotations that don't act on the qubits where \hat{C}_k is supported commute with \hat{C}_k and therefore cancel. This does not increase the support of the operator \hat{C}_k . This

means that the second set of unitary gates $e^{\{-i\gamma_1 H\}} = \prod_{l=1}^m U_l(\gamma)$ have a large set of gates $U_l(\gamma) = e^{\{-i\gamma_1 \hat{C}_l\}}$ that commute with the operator $e^{\{i\beta_1 B\}} \hat{C}_k e^{\{-i\beta_1 B\}}$. The only gates $U_l(\gamma) = e^{\{-i\gamma_1 \hat{C}_l\}}$ that contribute to the expectation value are those which involve qubits in the support of the original \hat{C}_k .

Hence, for bounded degree interaction the support of $e^{\{i\gamma_1 H\}} e^{\{i\beta_1 B\}} \hat{C}_k e^{\{-i\beta_1 B\}} e^{\{-i\gamma_1 H\}}$ only expands by an amount given by the degree of the interaction in H and is therefore independent of the system size. This means that for these smaller sub problems the expectation values are independent of n and can be evaluated classically. The case of a general degree 3 is considered in [3](#).

This is a general observation, which means that if we have a problem where the circuit used for the trial state preparation only increases the support of each term in the Hamiltonian by a constant amount the cost function can be directly evaluated.

When this is the case, and only a few parameters β, γ are needed in the preparation of the trial state, these can be found easily by a simple grid search. Furthermore, an exact optimal value of M_p can be used to bound the approximation ratio

$$\frac{M_p}{C_{\max}} \leq \alpha$$

to obtain an estimate of α . For this case the QAOA algorithm has the same characteristics as a conventional approximate optimization algorithm that comes with a guaranteed approximation ratio that can be obtained with polynomial efficiency in the problem size.

Evaluation on a quantum computer

When the quantum circuit becomes too deep to be evaluated classically, or when the connectivity of the Problem Hamiltonian is too high we can resort to other means of estimating the expectation value. This involves directly estimating $F_p(\vec{\gamma}, \vec{\beta})$ on the quantum computer. The approach here follows the path of the conventional expectation value estimation as used in - CITE - VQE, where a trial state $|\psi_p(\vec{\gamma}, \vec{\beta})\rangle$ is prepared directly on the quantum computer and the expectation value is obtained from sampling.

Since QAOA has a diagonal Hamiltonian H it is actually straight forward to estimate the expectation value. We only need to obtain samples from the trial state in the computational basis. Recall that $H = \sum_{x \in \{0,1\}^n} C(x) |x\rangle\langle x|$ so that we can obtain the sampling estimate of

$$\langle \psi_p(\vec{\gamma}, \vec{\beta}) | H | \psi_p(\vec{\gamma}, \vec{\beta}) \rangle = \sum_{x \in \{0,1\}^n} C(x) |\langle x | \psi_p(\vec{\gamma}, \vec{\beta})\rangle|^2$$

by repeated single qubit measurements of the state $|\psi_p(\vec{\gamma}, \vec{\beta})\rangle$ in the Z - basis. For every bit - string x obtained from the distribution $|\langle x |$

$\langle \psi_p(\vec{\gamma}, \vec{\beta}) | H^2 | \psi_p(\vec{\gamma}, \vec{\beta}) \rangle$ we evaluate the cost function $C(x)$ and average it over the total number of samples. The resulting empirical average approximates the expectation value up to an additive sampling error that lies within the variance of the state. The variance will be discussed below.

With access to the expectation value, we can now run a classical optimization algorithm, such as [6](#), to optimize the F_p .

While this approach does not lead to an a-priori approximation guarantee for x^* , the optimized function value can be used later to provide an estimate for the approximation ratio α .

4.3.3 Obtaining a good approximate solution with high probability

The algorithm is naturally probabilistic in nature and produces random bit strings from the distribution $\langle \psi_p(\vec{\gamma}, \vec{\beta}) | H^2 | \psi_p(\vec{\gamma}, \vec{\beta}) \rangle$. So how can we be sure that we will sample an approximation x^* that is close to the value of the optimized expectation value M_p ? Note that this question is also relevant to the estimation of M_p on a quantum computer in the first place. If the samples drawn from $\langle \psi_p(\vec{\gamma}, \vec{\beta}) | H^2 | \psi_p(\vec{\gamma}, \vec{\beta}) \rangle$ have too much variance, many samples are necessary to determine the mean.

We will draw a bit string x^* that is close to the mean M_p with high probability when the energy as variable has little variance.

Note that the number of terms in the Hamiltonian $H = \sum_{k=1}^m \hat{C}_k$ are bounded by m . Say each individual summand \hat{C}_k has an operator norm that can be bounded by a universal constant $\|\hat{C}_k\| \leq \tilde{C}$ for all $k = 1 \dots m$. Then consider

```
\begin{eqnarray} \langle \psi_p(\vec{\gamma}, \vec{\beta}) | H^2 | \psi_p(\vec{\gamma}, \vec{\beta}) \rangle &= \langle \psi_p(\vec{\gamma}, \vec{\beta}) | (\hat{C}_1 + \hat{C}_2 + \dots + \hat{C}_m)^2 | \psi_p(\vec{\gamma}, \vec{\beta}) \rangle \\ &\leq \|\hat{C}_1\|^2 + \|\hat{C}_2\|^2 + \dots + \|\hat{C}_m\|^2 \leq m \tilde{C}^2 \end{eqnarray}
```

where we have used that $\langle \psi_p(\vec{\gamma}, \vec{\beta}) | \hat{C}_k | \psi_p(\vec{\gamma}, \vec{\beta}) \rangle = \langle \psi_p(\vec{\gamma}, \vec{\beta}) | \hat{C}_k^2 | \psi_p(\vec{\gamma}, \vec{\beta}) \rangle / \langle \psi_p(\vec{\gamma}, \vec{\beta}) | \psi_p(\vec{\gamma}, \vec{\beta}) \rangle$.

This means that the variance of any expectation $F_p(\vec{\gamma}, \vec{\beta})$ is bounded by $m^2 \tilde{C}^2$. Hence this in particular applies for M_p . Furthermore if m only grows polynomially in the number of qubits n , we know that taking polynomially growing number of samples $s =$

$O(\frac{\tilde{C}^2 m^2 \epsilon^2}{\lambda})$ from $\langle \psi_p(\vec{\gamma}, \vec{\beta}) | \lambda \rangle$ will be sufficient to obtain a x^* that leads to an $C(x^*)$ that will be close to M_p .

5. Qiskit Implementation

As the example implementation we consider the MAXCUT problem on the butterfly graph of the openly available IBMQ 5-qubit chip. The graph will be defined below and corresponds to the native connectivity of the device. This allows us to implement the original version of the QAOA algorithm, where the cost - function C and the Hamiltonian H that is used to generate the state coincide. Moreover, for such a simple graph the exact cost function can be calculated analytically. To implement the circuit, we follow the notation and gate definitions from the [Qiskit Documentation](#).

As the first step will load Qiskit and additional python packages.

```
%matplotlib inline
# useful additional packages

#import math tools
import numpy as np

# We import the tools to handle general Graphs
import networkx as nx

# We import plotting tools
import matplotlib.pyplot as plt
from matplotlib import cm
from matplotlib.ticker import LinearLocator, FormatStrFormatter

# importing Qiskit
from qiskit import Aer, IBMQ
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit, execute

from qiskit.providers.ibmq      import least_busy
from qiskit.tools.monitor       import job_monitor
from qiskit.visualization import plot_histogram
```

5.1 Problem definition

We define the cost function in terms of the butterfly graph of the superconducting chip. The graph has $n = 5$ vertices $V = \{0,1,2,3,4,5\}$ and six edges $E = \{(0,1),(0,2),(1,2),(3,2),(3,4),(4,2)\}$, which will all

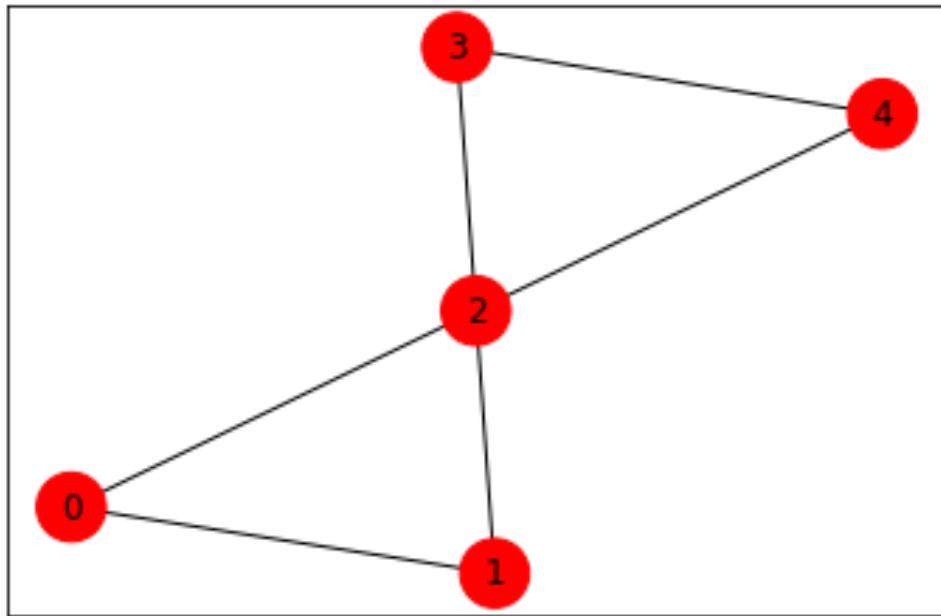
carry the same unit weight $w_{ij} = 1$. We load an additional network package to encode the graph and plot connectivity below.

```
# Generating the butterfly graph with 5 nodes
n      = 5
V      = np.arange(0,n,1)
E      =[(0,1,1.0),(0,2,1.0),(1,2,1.0),(3,2,1.0),(3,4,1.0),(4,2,1.0)]

G      = nx.Graph()
G.add_nodes_from(V)
G.add_weighted_edges_from(E)

# Generate plot of the Graph
colors     = ['r' for node in G.nodes()]
default_axes = plt.axes(frameon=True)
pos        = nx.spring_layout(G)

nx.draw_networkx(G, node_color=colors, node_size=600, alpha=1, ax=default_axes, pos=pos)
```



5.2 Optimal trial state parameters

In this example we consider the case for $p = 1$, i.e. only layer of gates. The expectation value $F_1(\gamma, \beta) = \langle \psi_1(\beta, \gamma) | H | \psi_1(\beta, \gamma) \rangle$ can be calculated analytically for this simple setting. Let us discuss the steps explicitly for the Hamiltonian $H = \sum_{(j,k) \in E} \frac{1}{2} \left(1 - Z_j Z_k \right)$. Due to the linearity of the expectation value we can compute the expectation value for the edges individually

$$f_{\{i,k\}}(\beta, \alpha) = \langle \psi_1(\gamma, \beta) | \frac{1}{2} \left(1 - Z_k \right) | \psi_1(\gamma, \beta) \rangle$$

For the butterfly graph as plotted above, we observe that there are only two kinds of edges A = {(0,1),(3,4)} and B = {(0,2),(1,2),(2,3),(2,4)}. The edges in A only have two neighboring edges, while the edges in B have four. You can convince yourself that we only need to compute the expectation of a single edge in each set since the other expectation values will be the same. This means that we can compute $F_1(\gamma, \beta) = 2 f_A(\gamma, \beta) + 4 f_B(\gamma, \beta)$ by evaluating only computing two expectation values. Note, that following the argument as outlined in [section 4.2.2](#), all the gates that do not intersect with the Pauli operator Z_0Z_1 or Z_0Z_2 commute and cancel out so that we only need to compute

$$f_A(\gamma, \beta) = \frac{1}{2} \left(1 - \langle \psi_1(\gamma, \beta) | X^{dagger}(0)X(1)Z_0Z_1 | \psi_1(\gamma, \beta) \rangle + \langle \psi_1(\gamma, \beta) | X^{dagger}(0)X^{dagger}(1)X(0)X(1)Z_0Z_1 | \psi_1(\gamma, \beta) \rangle \right)$$

and

$$f_B(\gamma, \beta) = \frac{1}{4} \left(1 - \langle \psi_1(\gamma, \beta) | X^{dagger}(0)X^{dagger}(1)X^{dagger}(2)X(0)X(1)Z_0Z_1 | \psi_1(\gamma, \beta) \rangle + \langle \psi_1(\gamma, \beta) | X^{dagger}(0)X^{dagger}(1)X^{dagger}(2)X^{dagger}(3)X(0)X(1)Z_0Z_1 | \psi_1(\gamma, \beta) \rangle + \langle \psi_1(\gamma, \beta) | X^{dagger}(0)X^{dagger}(1)X^{dagger}(2)X^{dagger}(3)X^{dagger}(4)X(0)X(1)Z_0Z_1 | \psi_1(\gamma, \beta) \rangle + \langle \psi_1(\gamma, \beta) | X^{dagger}(0)X^{dagger}(1)X^{dagger}(2)X^{dagger}(3)X^{dagger}(4)X^{dagger}(5)X(0)X(1)Z_0Z_1 | \psi_1(\gamma, \beta) \rangle \right)$$

How complex these expectation values become in general depend only on the degree of the graph we are considering and is independent of the size of the full graph if the degree is bounded. A direct evaluation of this expression with $U_{\{k,l\}}(\gamma) = \exp(i\gamma/2)(1 - Z_k Z_l)$ and $X_k(\beta) = \exp(i\beta X_k)$ yields

$$f_A(\gamma, \beta) = \frac{1}{2} \left(\sin(4\gamma) \sin(4\beta) + \sin^2(2\beta) \sin^2(2\gamma) \right)$$

and

$$f_B(\gamma, \beta) = \frac{1}{4} \left(\sin^2(2\beta) \sin^2(2\gamma) \cos^2(4\gamma) - \frac{1}{4} \sin(4\beta) \sin(4\gamma) (1 + \cos^2(4\gamma)) \right)$$

These results can now be combined as described above, and we the expectation value is therefore given by

$$F_1(\gamma, \beta) = 3 - \left(\sin^2(2\beta) \sin^2(2\gamma) - \frac{1}{2} \sin(4\beta) \sin(4\gamma) \right) / \left(1 + \cos^2(4\gamma) \right)$$

We plot the function $F_1(\gamma, \beta)$ and use a simple grid search to find the parameters (γ^*, β^*) that maximize the expectation value.



```
# Evaluate the function
step_size    = 0.1;

a_gamma      = np.arange(0, np.pi, step_size)
a_beta       = np.arange(0, np.pi, step_size)
a_gamma, a_beta = np.meshgrid(a_gamma,a_beta)

F1 = 3-(np.sin(2*a_beta)**2*np.sin(2*a_gamma)**2-0.5*np.sin(4*a_beta)*np.sin(4*a_gamma))

# Grid search for the minimizing variables
result = np.where(F1 == np.amax(F1))
a      = list(zip(result[0],result[1]))[0]

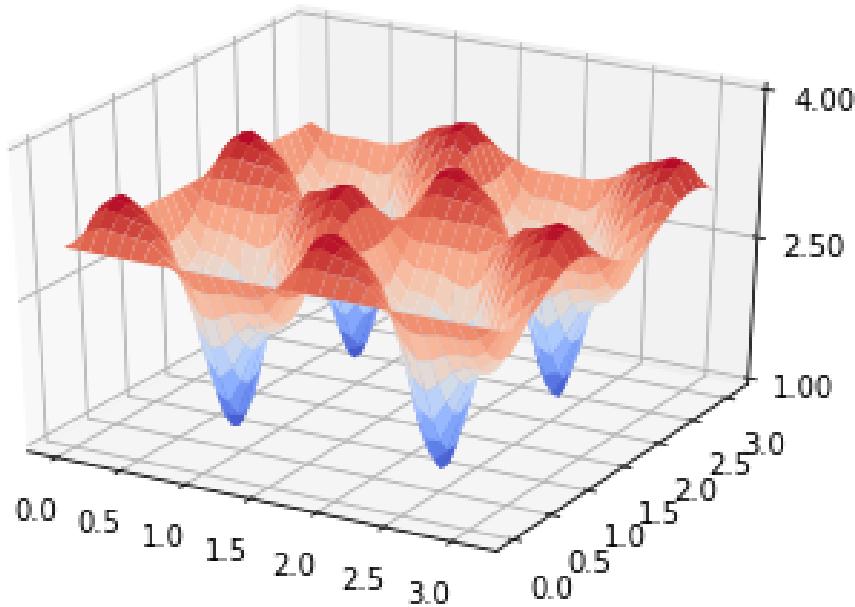
gamma  = a[0]*step_size;
beta   = a[1]*step_size;

# Plot the expectation value F1
fig = plt.figure()
ax  = fig.gca(projection='3d')

surf = ax.plot_surface(a_gamma, a_beta, F1, cmap=cm.coolwarm, linewidth=0, antialiased=
ax.set_zlim(1,4)
ax.xaxis.set_major_locator(LinearLocator(3))
ax.xaxis.set_major_formatter(FormatStrFormatter('%.02f'))

plt.show()

#The smallest paramters and the expectation can be extracted
print('\n --- OPTIMAL PARAMETERS --- \n')
print('The maximal expectation value is: M1 = %.03f' % np.amax(F1))
print('This is attained for gamma = %.03f and beta = %.03f' % (gamma,beta))
```



--- OPTIMAL PARAMETERS ---

The maximal expectation value is: $M_1 = 3.431$
 This is attained for $\gamma = 1.900$ and $\beta = 0.200$

5.3 Quantum circuit

With these parameters we can now construct the circuit that prepares the trial state for the Graph or the Graph $G = (V, E)$ described above with vertex set $V = \{0, 1, 2, 3, 4\}$ and the edges are $E = \{(0, 1), (0, 2), (1, 2), (3, 2), (3, 4), (4, 2)\}$. The circuit is going to require $n = 5$ qubits and we prepare the state

$$|\psi_1(\gamma, \beta)\rangle = e^{-i\beta B} e^{-i\gamma H} |+\rangle^n.$$

Recall that the terms are given by $B = \sum_{k \in V} X_k$ and $H = \sum_{(k,m) \in E} \frac{1}{2} \left(-Z_k Z_m \right)$. To generate the circuit we follow these steps:

- We first implement 5 - Hadamard H gates to generate the uniform superposition.
- This is followed by 6 Ising type gates $U_{\{k,l\}}(\gamma)$ with angle γ along the edges $(k,l) \in E$. This gate can be expressed in terms of the native Qiskit gates as

$$U_{\{k,l\}}(\gamma) = C_{\{u1\}}(-2\gamma)_{\{k,l\}} u1(\gamma)_k u1(\gamma)_l$$

- Lastly we apply single qubit X rotations $X_k(\beta)$ for every vertex $k \in V$ with β as angle. This gate directly parametrized as $X_k(\beta) = R_x(2\beta)_k$ in Qiskit.

- In the last step we measure the qubits in the computational basis, i.e. we perform a Z - measurement and record the resulting bit-string $x \in \{0,1\}^5$.

```
# prepare the quantum and classical registers
QAOA = QuantumCircuit(len(V), len(V))

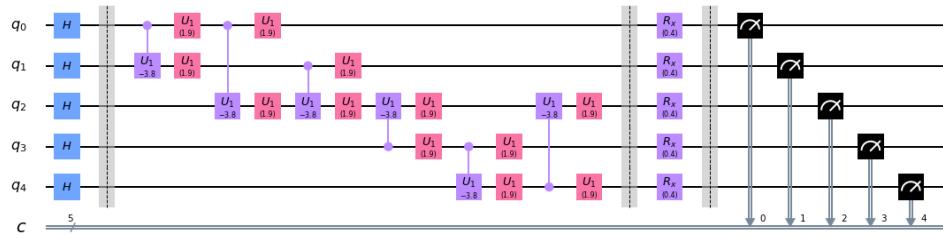
# apply the layer of Hadamard gates to all qubits
QAOA.h(range(len(V)))
QAOA.barrier()

# apply the Ising type gates with angle gamma along the edges in E
for edge in E:
    k = edge[0]
    l = edge[1]
    QAOA.cu1(-2*gamma, k, l)
    QAOA.u1(gamma, k)
    QAOA.u1(gamma, l)

# then apply the single qubit X - rotations with angle beta to all qubits
QAOA.barrier()
QAOA.rx(2*beta, range(len(V)))

# Finally measure the result in the computational basis
QAOA.barrier()
QAOA.measure(range(len(V)), range(len(V)))

### draw the circuit for comparison
QAOA.draw(output='mpl')
```



5.4 Cost function evaluation

Finally, we need a routine to compute the cost function value from the bit string. This is necessary to decide whether we have found a "good candidate" bitstring x but could also be used to estimate the expectation value $F_1(\gamma, \beta)$ in settings where the expectation value can not be evaluated directly.

```
# Compute the value of the cost function
def cost_function_C(x,G):

    E = G.edges()
    if( len(x) != len(G.nodes())):
        return np.nan

    C = 0;
    for index in E:
        e1 = index[0]
        e2 = index[1]

        w      = G[e1][e2]['weight']
        C = C + w*x[e1]*(1-x[e2]) + w*x[e2]*(1-x[e1])

    return C
```

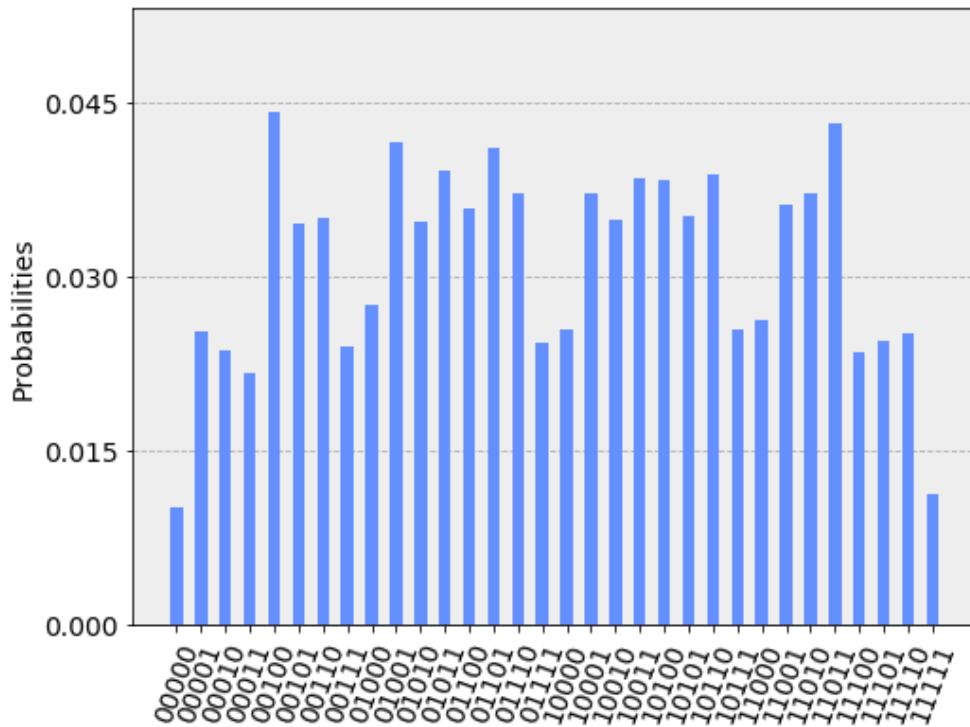
5a. Running QAOA on a simulator

We first run the algorithm on a local QASM simulator.

```
# run on Local simulator
backend      = Aer.get_backend("qasm_simulator")
shots       = 10000

simulate    = execute(QAOA, backend=backend, shots=shots)
QAOA_results = simulate.result()

plot_histogram(QAOA_results.get_counts(), figsize = (8,6), bar_labels = False)
```



Evaluate the date from the simulation

Let us now proceed to calculate the relevant information from the simulated data. We will use the obtained results to

- Compute the mean energy and check whether it agrees with the theoretical prediction
 - Report the sampled bitstring x^* with the largest observed cost function $C(x^*)$
 - Plot the Histogram of the energies to see whether it indeed concentrates around the predicted mean

```

# Evaluate the data from the simulator
counts = QAOA_results.get_counts()

avr_C      = 0
max_C      = [0,0]
hist       = {}

for k in range(len(G.edges())+1):
    hist[str(k)] = hist.get(str(k),0)

for sample in list(counts.keys()):
    # use sampled bit string x to compute cost function C(x,G)
    x          = [int(num) for num in list(sample)]
    tmp_eng   = cost function C(x,G)

```

```

# compute the expectation value and energy distribution
avr_C      = avr_C      + counts[sample]*tmp_eng
hist[str(round(tmp_eng))] = hist.get(str(round(tmp_eng)),0) + counts[sample]

# save best bit string
if( max_C[1] < tmp_eng):
    max_C[0] = sample
    max_C[1] = tmp_eng

M1_sampled   = avr_C/shots

print('\n --- SIMULATION RESULTS ---\n')
print('The sampled mean value is M1_sampled = %.02f while the true value is M1 = %.02f' % (M1_sampled,M1))
print('The approximate solution is x* = %s with C(x*) = %d \n' % (max_C[0],max_C[1]))
print('The cost function is distributed as: \n')
plot_histogram(hist,figsize = (8,6),bar_labels = False)

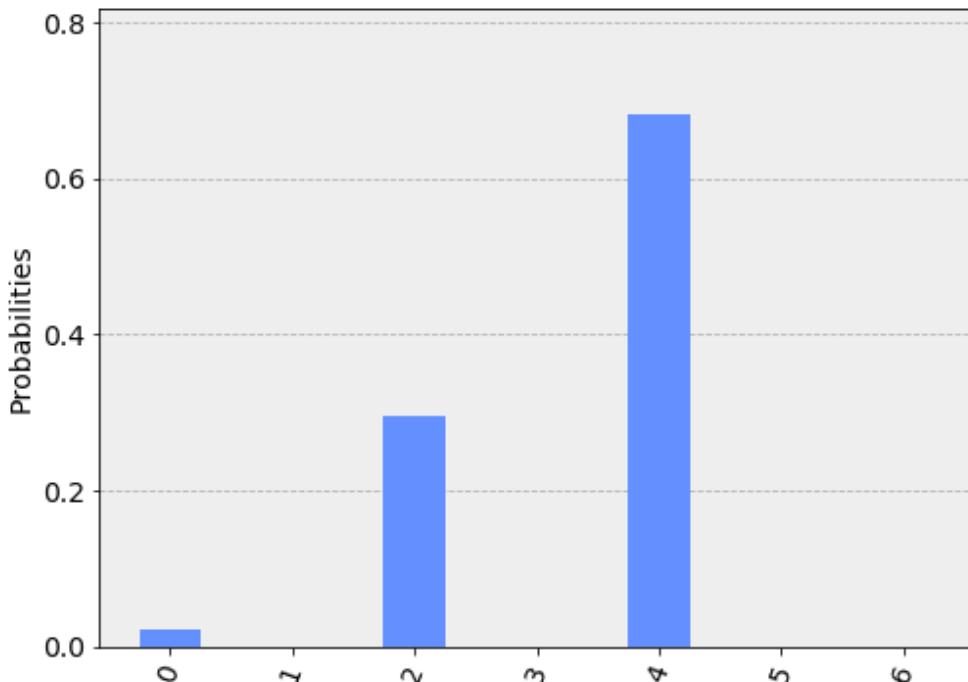
```

--- SIMULATION RESULTS ---

The sampled mean value is M1_sampled = 3.32 while the true value is M1 = 3.43

The approximate solution is $x^* = 10100$ with $C(x^*) = 4$

The cost function is distributed as:



5b. Running QAOA on a real quantum device

We then see how the same circuit can be executed on real-device backends.

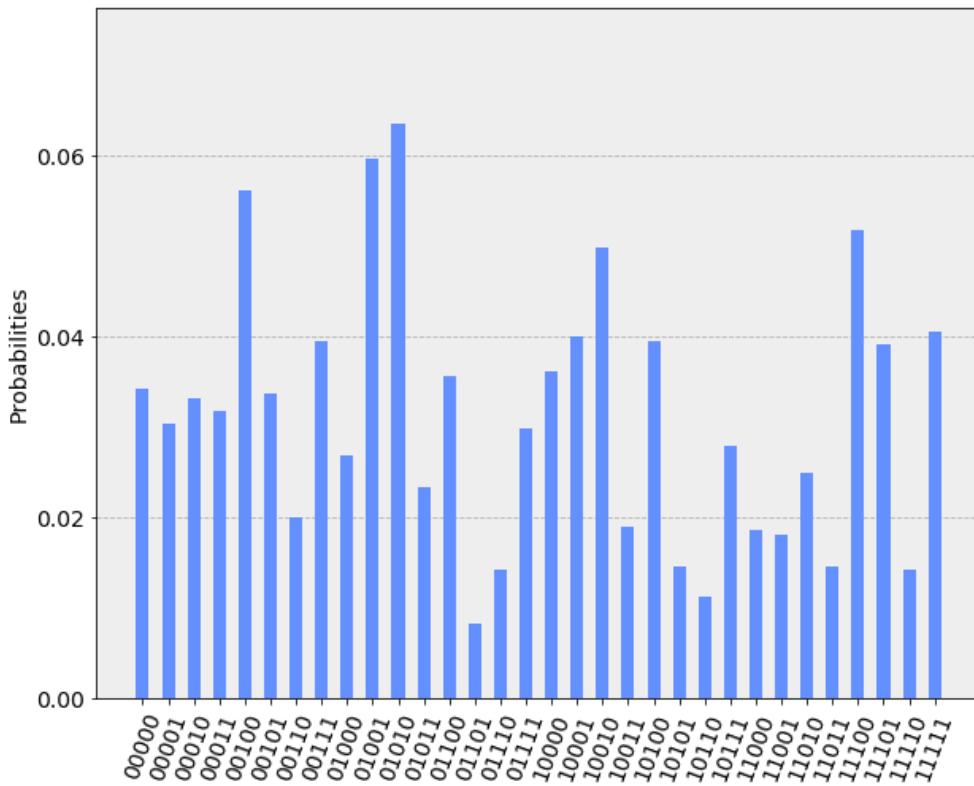
```
# Use the IBMQ essex device
provider = IBMQ.load_account()
backend = provider.get_backend('ibmq_essex')
shots = 2048

job_exp = execute(QAOA, backend=backend, shots=shots)
job_monitor(job_exp)
```



Job Status: job has successfully run

```
exp_results = job_exp.result()
plot_histogram(exp_results.get_counts(), figsize = (10,8), bar_labels = False)
```



Evaluate the data from the experiment

We can now repeat the same analysis as before and compare the experimental result.

```
# Evaluate the data from the experiment
counts = exp_results.get_counts()
```



```

avr_C      = 0
max_C      = [0,0]
hist       = {}

for k in range(len(G.edges())+1):
    hist[str(k)] = hist.get(str(k),0)

for sample in list(counts.keys()):

    # use sampled bit string x to compute C(x)
    x      = [int(num) for num in list(sample)]
    tmp_eng = cost_function_C(x,G)

    # compute the expectation value and energy distribution
    avr_C      = avr_C      + counts[sample]*tmp_eng
    hist[str(round(tmp_eng))] = hist.get(str(round(tmp_eng)),0) + counts[sample]

    # save best bit string
    if( max_C[1] < tmp_eng):
        max_C[0] = sample
        max_C[1] = tmp_eng

M1_sampled = avr_C/shots

print('\n --- EXPERIMENTAL RESULTS ---\n')
print('The sampled mean value is M1_sampled = %.02f while the true value is M1 = %.02f\n' % (M1_sampled,M1))
print('The approximate solution is x* = %s with C(x*) = %d \n' % (max_C[0],max_C[1]))
print('The cost function is distributed as: \n')
plot_histogram(hist,figsize = (8,6),bar_labels = False)

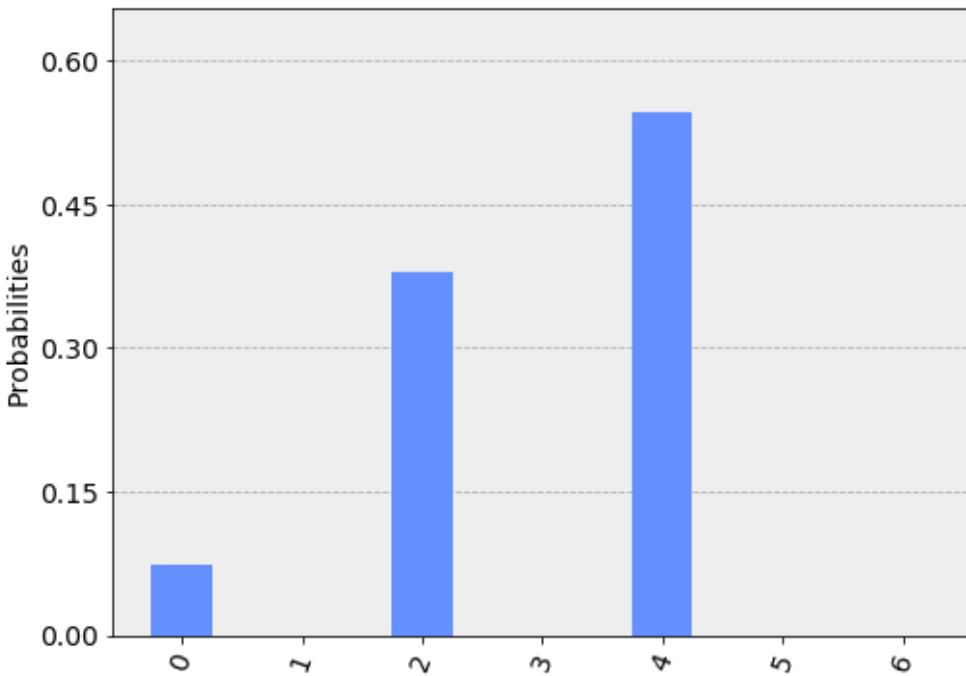
```

--- EXPERIMENTAL RESULTS ---

The sampled mean value is M1_sampled = 2.94 while the true value is M1 = 3.43

The approximate solution is x* = 10100 with C(x*) = 4

The cost function is distributed as:



6. Problems

1. The QAOA algortihm produces a bit string, is this string the optimal solution for this graph? Compare the experimental results from the superconducting chip with the results from the local QASM simulation.
1. We have computed the cost function F_1 analytically in [section 5.2](#). Verify the steps and compute $f_A(\gamma, \beta)$ as well $f_B(\gamma, \beta)$.
 - Write a routine to estimate the expectation value $F_1(\gamma, \beta)$ from the samples obtained in the result (hint: use the function `cost_function_C(x, G)` from [section 5.4](#) and the evaluation of the data in both section [5.a / 5.b](#))
 - Use an optimizaion routine,e.g. SPSA from the VQE example in this tutorial, to optimize the parmeteres in the sampled $F_1(\gamma, \beta)$ numerically. Do you find the same values for γ^*, β^* ?
1. The Trial circuit in [section 5.3](#) corresponds to depth $p=1$ and was directly aimed at being compatible with the Hardware.
 - Use the routine from exercise 2 to evaluate the the cost functions $F_p(\gamma, \beta)$ for $p=2,3$. What do you expect to see in the actual Hardware?

-Generalize this class of trial state to other candidate wave fuctions, such as the Hardware efficient ansatz of Ref. 4.

1. Consider an example of MAX \setminus 3-SAT as discussed in the example section and modify the function cost_function_C(c,G) from [section 5.4](#) you have used to compute F_p accordingly. Run the QAOA algorithm for this instance of MAX \setminus 3-SAT using the hardware efficient algorithm and analyze the results.

7. References

1. Garey, Michael R.; David S. Johnson (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman. ISBN 0-7167-1045-5
2. Goemans, Michel X., and David P. Williamson. [Journal of the ACM \(JACM\) 42.6 \(1995\): 1115-1145](#).
3. Farhi, Edward, Jeffrey Goldstone, and Sam Gutmann. "A quantum approximate optimization algorithm." arXiv preprint [arXiv:1411.4028 \(2014\)](#).
4. Kandala, Abhinav, et al. "Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets." [Nature 549.7671 \(2017\): 242](#).
5. Farhi, Edward, et al. "Quantum algorithms for fixed qubit architectures." arXiv preprint [arXiv:1703.06199 \(2017\)](#).
6. Spall, J. C. (1992), [IEEE Transactions on Automatic Control, vol. 37\(3\), pp. 332–341](#).

```
import qiskit
qiskit.__qiskit_version__
```



```
{'qiskit-terra': '0.10.0',
 'qiskit-aer': '0.3.2',
 'qiskit-ignis': '0.2.0',
 'qiskit-ibmq-provider': '0.3.3',
 'qiskit-aqua': '0.6.1',
 'qiskit': '0.13.0'}
```

Solving Satisfiability Problems using Grover's Algorithm

In this section, we demonstrate how to solve satisfiability problems using the implementation of Grover's algorithm in Qiskit Aqua.

Contents

1. [Introduction](#)
2. [3-Satisfiability Problem](#)
3. [Qiskit Implementation](#)
4. [Problems](#)
5. [References](#)

1. Introduction

Grover's algorithm for unstructured search was introduced in an [earlier section](#), with an example and implementation using Qiskit Terra. We saw that Grover search is a quantum algorithm that can be used search for correct solutions quadratically faster than its classical counterparts. Here, we are going to illustrate the use of Grover's algorithm to solve a particular combinatorial Boolean satisfiability problem.

In computer science, the Boolean satisfiability problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. In other words, it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE. If this is the case, the formula is called satisfiable. On the other hand, if no such assignment exists, the function expressed by the formula is FALSE for all possible variable assignments and the formula is unsatisfiable. This can be seen as a search problem, where the solution is the assignment where the Boolean formula is satisfied.

2. 3-Satisfiability Problem

The 3-Satisfiability (3SAT) Problem is best explained with the following concrete problem. Let us consider a Boolean function f with three Boolean variables v_1, v_2, v_3 as below:

$$f(v_1, v_2, v_3) = (\neg v_1 \vee \neg v_2 \vee \neg v_3) \wedge (v_1 \vee \neg v_2 \vee v_3) \wedge (v_1 \vee v_2 \vee \neg v_3) \\ \wedge (v_1 \vee \neg v_2 \vee \neg v_3) \wedge (\neg v_1 \vee v_2 \vee v_3)$$

In the above function, the terms on the right-hand side equation which are inside $()$ are called clauses; this function has 5 clauses. Being a 3SAT problem, each clause has exactly three literals. For instance, the first clause has $\neg v_1 \neg v_2$, $\neg v_2 \neg v_3$ and $\neg v_3 \neg v_1$ as its literals. The symbol \neg is the Boolean NOT that negates (or, flips) the value of its succeeding literal. The symbols \vee and \wedge are, respectively, the Boolean OR and AND. The Boolean f is satisfiable if there is an assignment of v_1, v_2, v_3 that evaluates to $f(v_1, v_2, v_3) = 1$ ($f(v_1, v_2, v_3) = 1$ that is, f evaluates to True).

A naive way to find such an assignment is by trying every possible combinations of input values of f . Below is the table obtained from trying all possible combinations of v_1, v_2, v_3 . For ease of explanation, we interchangably use 00 and False, as well as 11 and True.

$v_1 v_1$	$v_2 v_2$	$v_3 v_3$	$f f$	Comment
0	0	0	1	Solution
0	0	1	0	Not a solution because f is False
0	1	0	0	Not a solution because f is False
0	1	1	0	Not a solution because f is False
1	0	0	0	Not a solution because f is False
1	0	1	1	Solution
1	1	0	1	Solution
1	1	1	0	Not a solution because f is False

From the table above, we can see that this 3-SAT problem instance has three satisfying solutions: $(v_1, v_2, v_3) = (T, F, T)$ ($v_1, v_2, v_3) = (T, F, T)$ or (F, F, F) (F, F, F) or (T, T, F) (T, T, F)).

In general, the Boolean function f can have many clauses and more Boolean variables. Note that 3SAT problems can be always written in what is known as conjunctive normal form (CNF), that is, a conjunction of one or more clauses, where a clause is a disjunction of three literals; otherwise put, it is an AND of 3 ORs.

3. Qiskit Implementation

Let's now use Qiskit Aqua to solve the example 3SAT problem:

$$f(v_1, v_2, v_3) = (\neg v_1 \vee \neg v_2 \vee \neg v_3) \wedge (v_1 \vee \neg v_2 \vee v_3) \wedge (v_1 \vee v_2 \vee \neg v_3) \\ \wedge (v_1 \vee \neg v_2 \vee \neg v_3) \wedge (\neg v_1 \vee v_2 \vee v_3)$$

First we need to understand the input [DIMACS CNF](#) format that Qiskit Aqua uses for such problem, which looks like the following for the problem:

```
c example DIMACS CNF 3-SAT
p cnf 3 5
-1 -2 -3 0
1 -2 3 0
1 2 -3 0
1 -2 -3 0
-1 2 3 0
```

- Lines that start with `c` are comments
 - eg. `c example DIMACS CNF 3-SAT`
- The first non-comment line needs to be of the form `p cnf nbvar nbclauses`, where
 - `cnf` indicates that the input is in CNF format
 - `nbvar` is the exact number of variables appearing in the file
 - `nbclauses` is the exact number of clauses contained in the file
 - eg. `p cnf 3 5`
- Then there is a line for each clause, where
 - each clause is a sequence of distinct non-null numbers between `-nbvar` and `nbvar` ending with `0` on the same line
 - it cannot contain the opposite literals `i` and `-i` simultaneously
 - positive numbers denote the corresponding variables
 - negative numbers denote the negations of the corresponding variables
 - eg. `-1 2 3 0` corresponds to the clause $\neg v_1 \vee v_2 \vee v_3 \vee \neg v_1 \vee v_2 \vee v_3$

Similarly the solutions to the problem $(v_1, v_2, v_3) = (T, F, T)$ ($v_1, v_2, v_3) = (T, F, T)$ or (F, F, F) (F, F, F) or (T, T, F) (T, T, F) can be written as `1 -2 3`, or `-1 -2 -3`, or `1 2 -3`.

With this example problem input, we create the corresponding oracle for our Grover search. In particular, we use the `LogicalExpressionOracle` component provided by Aqua, which supports parsing DIMACS CNF format strings and constructing the corresponding oracle circuit.

```
import numpy as np
from qiskit import BasicAer
from qiskit.tools.visualization import plot_histogram
from qiskit.aqua import QuantumInstance, run_algorithm
from qiskit.aqua.algorithms import Grover
from qiskit.aqua.components.oracles import LogicalExpressionOracle, TruthTableOracle
```

```
input_3sat = """
c example DIMACS-CNF 3-SAT
p cnf 3 5
-1 -2 -3 0
1 -2 3 0
1 2 -3 0
1 -2 -3 0
-1 2 3 0
..."""
```

```
oracle = LogicalExpressionOracle(input_3sat)
```

The `oracle` can now be used to create an Grover instance:

```
grover = Grover(oracle)
```

We can then configure a simulator backend and run the Grover instance to get the result:

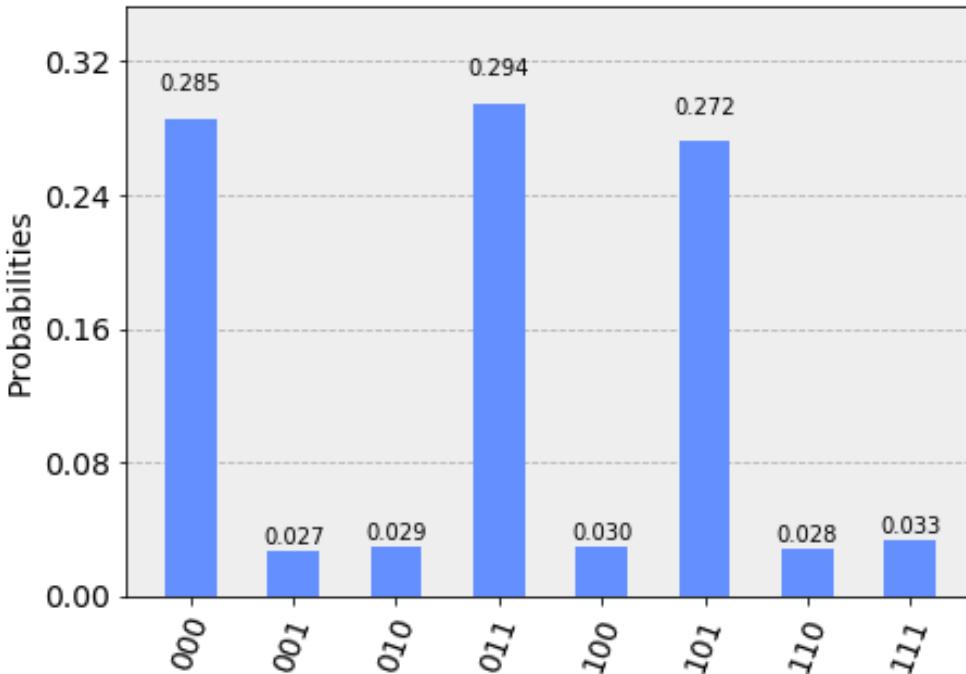
```
backend = BasicAer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024)
result = grover.run(quantum_instance)
print(result['result'])
```

```
[1, 2, -3]
```

As seen above, a satisfying solution to the specified 3-SAT problem is obtained. And it is indeed one of the three satisfying solutions.

Since we used a simulator backend, the complete measurement result is also returned, as shown in the plot below, where it can be seen that the binary strings `000`, `011`, and `101` (note the bit order in each string), corresponding to the three satisfying solutions all have high probabilities associated with them.

```
plot_histogram(result['measurement'])
```



We have seen that the simulator can find the solutions to the example problem. We would like to see what happens if we use the real quantum devices that have noise and imperfect gates.

However, due to the restriction on the length of strings that can be sent over the network to the real devices (there are more than sixty thousands characters of QASM of the circuit), at the moment the above circuit cannot be run on real device backends. We can see the compiled QASM on real-device `ibmq_16_melbourne` backend as follows:

```
# Load our saved IBMQ accounts and get the ibmq_16_melbourne backend
from qiskit import IBMQ
IBMQ.load_account()
provider = IBMQ.get_provider(hub='ibm-q')
backend = provider.get_backend('ibmq_16_melbourne')
```

```
from qiskit.compiler import transpile

# transpile the circuit for ibmq_16_melbourne
grover_compiled = transpile(result['circuit'], backend=backend, optimization_level=3)

print('gates = ', grover_compiled.count_ops())
print('depth = ', grover_compiled.depth())
```

```
gates = {'u3': 966, 'u2': 1192, 'cx': 479, 'u1': 5, 'barrier': 2, 'measure': 3}
depth = 1107
```

The number of gates is in the order of thousands which is above the limits of decoherence time of the current near-term quantum computers. It is a challenge to design a quantum circuit for Grover search to solve satisfiability and other optimization problems.

4. Problems

1. Use Qiskit Aqua to solve the following 3SAT problem:

$$f(x_1, x_2, x_3) = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \quad f(x_1, x_2, x_3) = \\ (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$$

Are the results what you expect?

5. References

1. Giacomo Nannicini (2017), "An Introduction to Quantum Computing, Without the Physics",
[arXiv:1708.03684](https://arxiv.org/abs/1708.03684)

```
import qiskit
qiskit.__qiskit_version__
```



```
{'qiskit': '0.10.4',
 'qiskit-terra': '0.8.2',
 'qiskit-ignis': '0.1.1',
 'qiskit-aer': '0.2.1',
 'qiskit-ibmq-provider': '0.2.2',
 'qiskit-aqua': '0.5.1'}
```

Calibrating qubits using Qiskit and OpenPulse

Qiskit is an open-source framework for programming quantum computers (Ref. 1). Using Qiskit, quantum circuits can be built, simulated and executed on quantum devices.

OpenPulse provides a language for specifying pulse level control (i.e. control of the continuous time dynamics of input signals) of a general quantum device independent of the specific hardware implementation (Ref. 2).

In this tutorial, we show how to implement typical single-qubit calibration and characterization experiments using Qiskit and OpenPulse. These are typically the first round of experiments that would be done in the lab immediately after a device has been fabricated and installed into a system. The presentation is pedagogical, and allows students to explore two-level-system dynamics experimentally.

Each experiment gives us more information about the system, which is typically used in subsequent experiments. For this reason, this notebook has to be mostly executed in order.

Contents

Part 0. [Getting started](#)

Part 1. [Finding the qubit frequency using a frequency sweep](#)

Part 2. Calibrating and using a π pulse

A. [Calibrating \$\pi\$ pulses using a Rabi experiment](#)

B. [Determining 0 vs 1](#)

C. [Measuring \$T_1\$ using inversion recovery](#)

Part 3. Determining qubit coherence

A. [Measuring the qubit frequency precisely using a Ramsey experiment](#)

B. [Measuring \$T_2\$ using Hahn echoes](#)

Part 4. [References](#)

0. Getting started

We'll first get our basic dependencies set up and ready to go. Since we want to use real, noisy devices for our calibration experiments in this notebook, we need to load our IBMQ account and set the appropriate backend.

```
import warnings
warnings.filterwarnings('ignore')
from qiskit.tools.jupyter import *

from qiskit import IBMQ
IBMQ.load_account()
provider = IBMQ.get_provider(hub='ibm-q', group='open', project='main')
backend = provider.get_backend('ibmq_armonk')
```

We verify that the backend supports OpenPulse features by checking the backend configuration. The config provides us with general information about the structure of the backend setup.

```
backend_config = backend.configuration()
assert backend_config.open_pulse, "Backend doesn't support OpenPulse"
```

For instance, we can find the sampling time for the backend pulses within the backend configuration. This will be a very useful value to us as we build and execute our calibration routines.

```
dt = backend_config.dt
print(f"Sampling time: {dt*1e9} ns")    # The configuration returns dt in seconds
```

```
Sampling time: 0.2222222222222222 ns
```

The backend defaults provide a starting point for how to use the backend. It contains estimates for qubit frequencies and default programs to enact basic quantum operators. We can access them with the following:

```
backend_defaults = backend.defaults()
```

Part 1

1. Finding the qubit frequency using a frequency sweep

We begin by searching for the qubit frequency. The qubit frequency is the difference in energy between the ground and excited states, which we label the $|0\rangle$ and $|1\rangle$ states, respectively. This frequency will be crucial for creating pulses which enact particular quantum operators on the qubit -- the final goal of our calibration!

With superconducting qubits, higher energy levels are also available, but we fabricate the systems to be anharmonic so that we can control which transition we are exciting. That way, we are able to isolate two energy levels and treat each qubit as a basic two-level system, ignoring higher energy states.

In a typical lab setting, the qubit frequency can be found by sweeping a range of frequencies and looking for signs of absorption using a tool known as a Network Analyzer. This measurement gives a rough estimate of the qubit frequency. Later on, we will see how to do a more precise measurement using a Ramsey pulse sequence.

First, we define the frequency range that will be swept in search of the qubit. Since this can be arbitrarily broad, we restrict ourselves to a window of 40 MHz around the estimated qubit frequency in `backend_defaults`. We step the frequency in units of 1 MHz.

```
import numpy as np

GHz = 1.0e9
MHz = 1.0e6

# We will find the qubit frequency for the following qubit.
qubit = 0

# The sweep will be centered around the estimated qubit frequency.
center_frequency_Hz = backend_defaults.qubit_freq_est[qubit]           # The default frequ
                                                                     # warning: this wil
print(f"Qubit {qubit} has an estimated frequency of {center_frequency_Hz / GHz}.") # warning: this wil

# We will sweep 40 MHz around the estimated frequency
frequency_span_Hz = 40 * MHz
# in steps of 1 MHz.
frequency_step_Hz = 1 * MHz

# We will sweep 20 MHz above and 20 MHz below the estimated frequency
frequency_min = center_frequency_Hz - frequency_span_Hz / 2
frequency_max = center_frequency_Hz + frequency_span_Hz / 2
# Construct an np array of the frequencies for our experiment
frequencies_GHz = np.arange(frequency_min / GHz,
                             frequency_max / GHz,
```

```
frequency_step_Hz / GHz)

print(f"The sweep will go from {frequency_min / GHz} GHz to {frequency_max / GHz} GHz \
in steps of {frequency_step_Hz / MHz} MHz.")
```

Qubit 0 has an estimated frequency of 4.974265348122223 GHz.
The sweep will go from 4.954265348122224 GHz to 4.994265348122224 GHz in steps of 1.0

Next, we define the pulses we will use for our experiment. We will start with the drive pulse, which is a Gaussian pulse.

Remember the value `dt` from earlier? All durations in pulse are given in terms of `dt`. In the next cell, we define the length of the drive pulse in terms of `dt`.

```
def get_closest_multiple_of_16(num):
    return (int(num) - (int(num)%16))
```

```
from qiskit import pulse          # This is where we access all of our Pulse features
from qiskit.pulse import pulse_lib # This Pulse module helps us build sampled pulses f

# Drive pulse parameters
drive_sigma_us = 0.075           # This determines the actual width of the gaussian
drive_samples_us = drive_sigma_us*8 # This is a truncating parameter, because gaussian has a natural finite length

drive_sigma = get_closest_multiple_of_16(drive_sigma_us * 1e-6/dt)      # The width of the pulse
drive_samples = get_closest_multiple_of_16(drive_samples_us * 1e-6/dt)      # The truncation
drive_amp = 0.3

# Drive pulse samples
drive_pulse = pulse_lib.gaussian(duration=drive_samples,
                                    sigma=drive_sigma,
                                    amp=drive_amp,
                                    name='freq_sweep_excitation_pulse')
```

Next, we will create the instructions we need to measure our qubit. This actually consists of two pulses: one stimulates the readout with a Gaussian-Square pulse applied at the readout resonator frequency, and the other triggers the data acquisition instrument to acquire data for the duration of the pulse.

```

### Construct the measurement pulse
# Measurement pulse parameters

meas_samples_us = 3.0
meas_sigma_us = 0.014      # The width of the gaussian part of the rise and fall
meas_risefall_us = 0.1     # and the truncating parameter: how many samples to dedicate

meas_samples = get_closest_multiple_of_16(meas_samples_us * 1e-6/dt)
meas_sigma = get_closest_multiple_of_16(meas_sigma_us * 1e-6/dt)          # The width of
meas_risefall = get_closest_multiple_of_16(meas_risefall_us * 1e-6/dt)      # and the tr

# meas_samples = 12800
# meas_sigma = 64            # The width of the gaussian part of the rise and fall
# meas_risefall = 400         # and the truncating parameter: how many samples to dedicate

meas_amp = 0.25
# Measurement pulse samples
meas_pulse = pulse_lib.gaussian_square(duration=meas_samples,
                                         sigma=meas_sigma,
                                         amp=meas_amp,
                                         risefall=meas_risefall,
                                         name='measurement_pulse')

### Construct the acquire pulse to trigger the acquisition
# Acquire pulse samples
acq_cmd = pulse.Acquire(duration=meas_samples)

```

We have to check one additional thing in order to properly measure our qubits: the measurement map. This is a hardware constraint. When acquisition is done for one qubit, it is also done on other qubits. We have to respect this constraint when building our program in OpenPulse. Let's check which group of qubits our qubit is in:

```

# Find out which group of qubits need to be acquired with this qubit
meas_map_idx = None
for i, measure_group in enumerate(backend_config.meas_map):
    if qubit in measure_group:
        meas_map_idx = i
        break
assert meas_map_idx is not None, f"Couldn't find qubit {qubit} in the meas_map!"

```

Lastly, we specify the channels on which we will apply our pulses. Drive, measure, and acquire channels are indexed by qubit index.

```
### Collect the necessary channels
drive_chan = pulse.DriveChannel(qubit)
meas_chan = pulse.MeasureChannel(qubit)
acq_chan = pulse.AcquireChannel(qubit)
```



Now that the pulse parameters have been defined, and we have created the pulse shapes for our experiments. We can proceed to creating the pulse schedules.

At each frequency, we will send a drive pulse of that frequency to the qubit and measure immediately after the pulse. The pulse envelopes are independent of frequency, so we will build a reusable schedule , and we will specify the drive pulse frequency with a frequency configuration array.

```
# Create the base schedule
# Start with drive pulse acting on the drive channel
schedule = pulse.Schedule(name='Frequency sweep')
schedule += drive_pulse(drive_chan)

# In a new schedule, which we will use again later, add a measurement stimulus on the
# measure channel pulse to trigger readout
measure_schedule = meas_pulse(meas_chan)
# Trigger data acquisition, and store measured values into respective memory slots
measure_schedule += acq_cmd([pulse.AcquireChannel(i) for i in backend_config.meas_map[m
                                         [pulse.MemorySlot(i) for i in backend_config.meas_map[meas_
# The left shift `<<` is special syntax meaning to shift the start time of the schedule
schedule += measure_schedule << schedule.duration

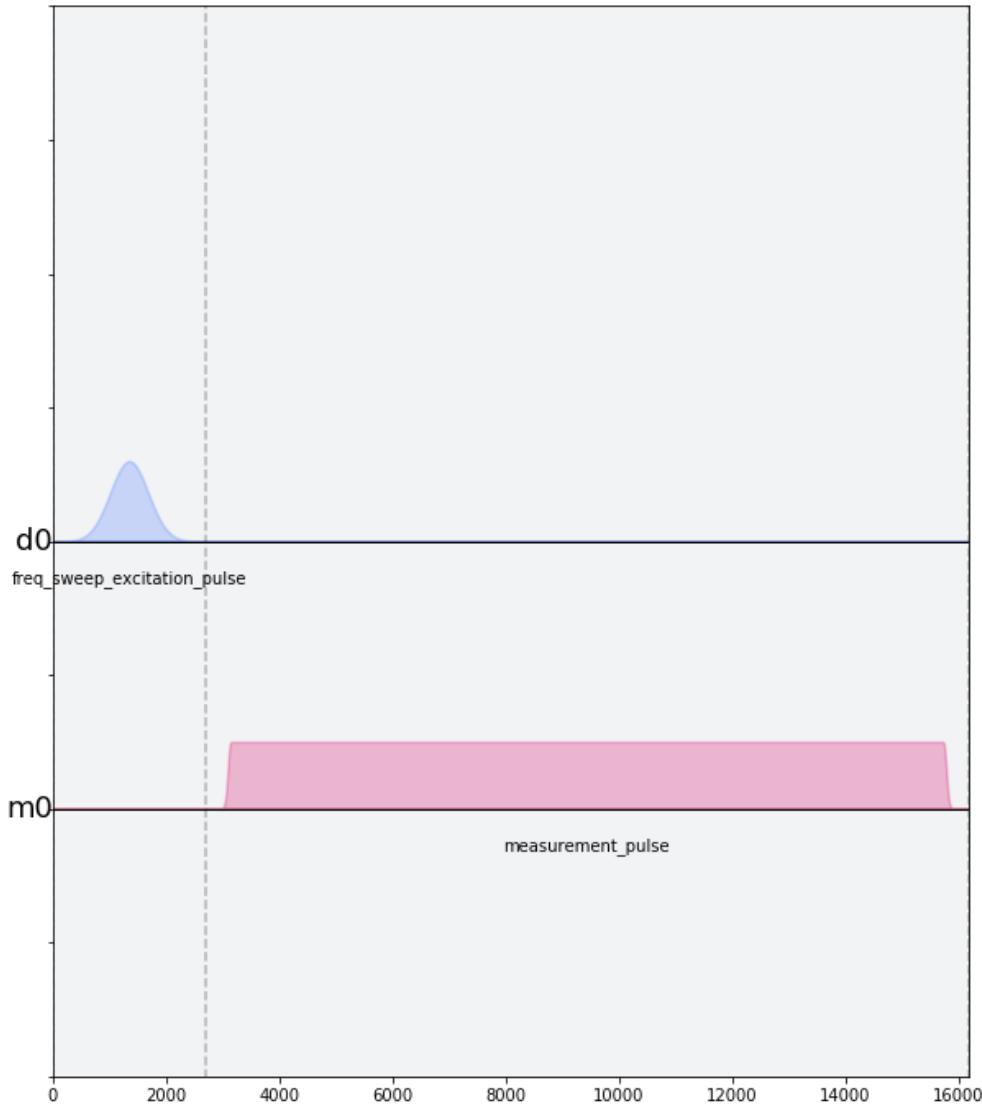
# Create the frequency settings for the sweep
schedule_frequencies = [{drive_chan: freq} for freq in frequencies_GHz]
```



As a sanity check, it's always a good idea to look at the pulse schedule. This is done using `schedule.draw()` as shown below.

```
schedule.draw(channels_to_plot=[drive_chan, meas_chan], label=True, scaling=1.0)
```





We assemble the `schedules` and `schedule_frequencies` above into a program object, called a Qobj, that can be sent to the quantum device. We request that each schedule (each point in our frequency sweep) is repeated `num_shots_per_frequency` times in order to get a good estimate of the qubit response.

We also specify measurement settings. `meas_level=0` returns raw data (an array of complex values per shot), `meas_level=1` returns kerneled data (one complex value per shot), and `meas_level=2` returns classified data (a 0 or 1 bit per shot). We choose `meas_level=1` to replicate what we would be working with if we were in the lab, and hadn't yet calibrated the discriminator to classify 0s and 1s. We ask for the '`avg`' of the results, rather than each shot individually.

```
from qiskit import assemble

num_shots_per_frequency = 1024
frequency_sweep_program = assemble(schedule,
                                    backend=backend,
                                    meas_level=1,
```



```
    meas_return='avg',
    shots=num_shots_per_frequency,
    schedule_los=schedule_frequencies)
```

Finally, we can run the assembled program on the backend using:

```
job = backend.run(frequency_sweep_program)
```



It is always a good idea to print the `job_id` for later retrieval, and to monitor the job status by using `job_monitor()`

```
# print(job.job_id())
from qiskit.tools.monitor import job_monitor
job_monitor(job)
```



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

Once the job is run, the results can be retrieved using:

```
frequency_sweep_results = job.result(timeout=120) # timeout parameter set to 60 seconds
```



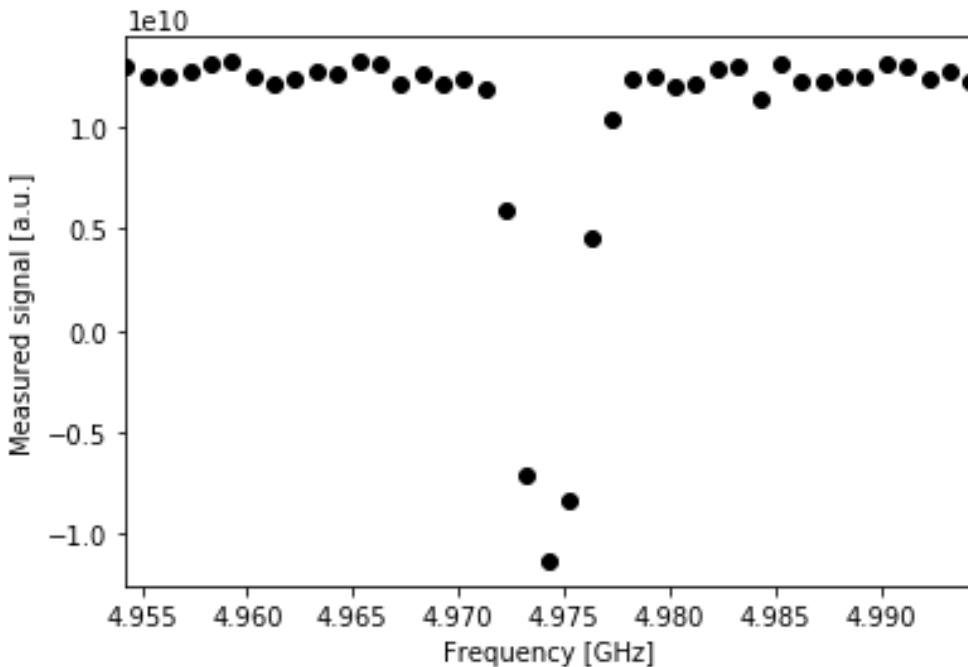
We will extract the results and plot them using `matplotlib`:

```
%matplotlib inline
import matplotlib.pyplot as plt

sweep_values = []
for i in range(len(frequency_sweep_results.results)):
    # Get the results from the ith experiment
    res = frequency_sweep_results.get_memory(i)
    # Get the results for `qubit` from this experiment
    sweep_values.append(res[qubit])

plt.scatter(frequencies_GHz, sweep_values, color='black')
plt.xlim([min(frequencies_GHz), max(frequencies_GHz)])
plt.xlabel("Frequency [GHz]")
plt.ylabel("Measured signal [a.u.]")
plt.show()
```





As you can see above, the peak near the center corresponds to the location of the qubit frequency. The signal shows power-broadening, which is a signature that we are able to drive the qubit off-resonance as we get close to the center frequency. To get the value of the peak frequency, we will fit the values to a resonance response curve, which is typically a Lorentzian shape.

```
from scipy.optimize import curve_fit

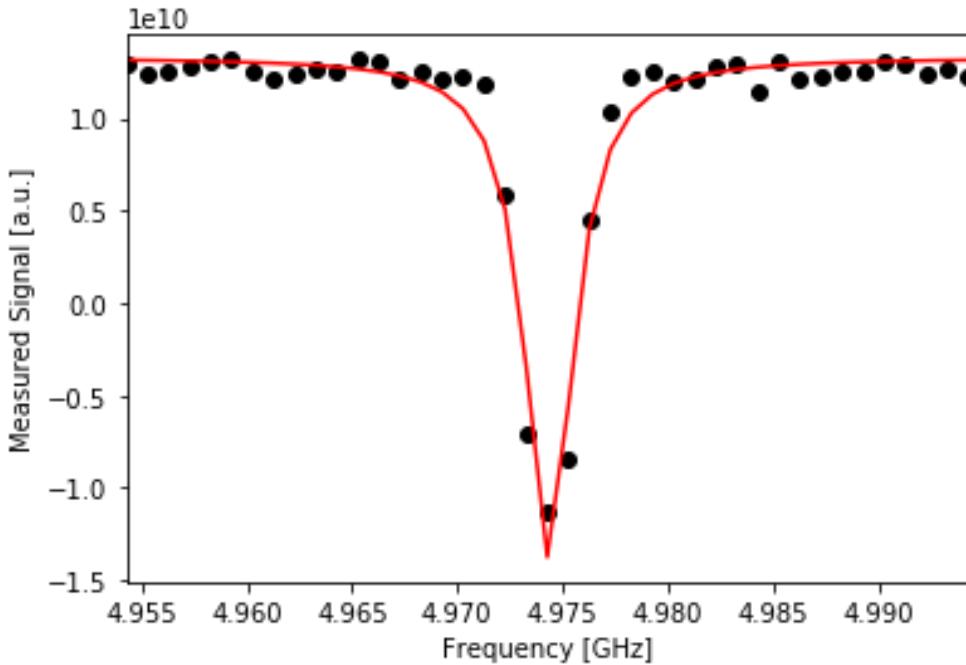
def fit_function(x_values, y_values, function, init_params):
    fitparams, conv = curve_fit(function, x_values, y_values, init_params)
    y_fit = function(x_values, *fitparams)

    return fitparams, y_fit
```

```
fit_params, y_fit = fit_function(frequencies_GHz,
                                  sweep_values,
                                  lambda x, A, q_freq, B, C: (A / np.pi) * (B / ((x - q_
[-2e10, 4.975, 1, 3e10] # initial parameters for curve
) )
```

```
plt.scatter(frequencies_GHz, sweep_values, color='black')
plt.plot(frequencies_GHz, y_fit, color='red')
plt.xlim([min(frequencies_GHz), max(frequencies_GHz)])

plt.xlabel("Frequency [GHz]")
plt.ylabel("Measured Signal [a.u.]")
plt.show()
```



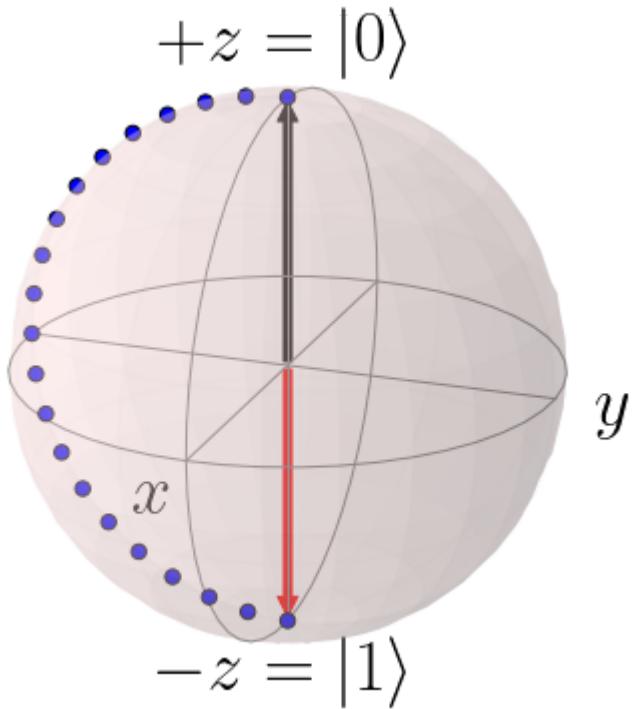
```
A, rough_qubit_frequency, B, C = fit_params
rough_qubit_frequency = round(rough_qubit_frequency, 5)
print(f"We've updated our qubit frequency estimate from "
      f"{round(backend_defaults.qubit_freq_est[qubit] / GHz, 5)} GHz to {rough_qubit_fr
```

We've updated our qubit frequency estimate from 4.97427 GHz to 4.97435 GHz.

Part 2. Calibrating and using a π pulse

A. Calibrating π pulses using a Rabi experiment

Once we know the frequency of our qubit, the next step is to determine the strength of a π pulse. Strictly speaking of the qubit as a two-level system, a π pulse is one that takes the qubit from $|0\rangle$ to $|1\rangle$, and vice versa. This is also called the X or X₁₈₀ gate, or bit-flip operator. We already know the microwave frequency needed to drive this transition from the previous frequency sweep experiment, and we now seek the amplitude needed to achieve a π rotation from $|0\rangle$ to $|1\rangle$. The desired rotation is shown on the Bloch sphere in the figure below -- you can see that the π pulse gets its name from the angle it sweeps over on a Bloch sphere.



We will change the drive amplitude in small increments and measuring the state of the qubit each time. We expect to see oscillations which are commonly named Rabi oscillations, as the qubit goes from $|0\rangle$ to $|1\rangle$ and back.

```
# This experiment uses these values from the previous experiment:
# `qubit`,
# `measure_schedule`, and
# `rough_qubit_frequency`.

# Rabi experiment parameters
num_rabi_points = 50

# Drive amplitude values to iterate over: 64 amplitudes evenly spaced from 0 to 0.1
drive_amp_min = 0
drive_amp_max = 0.75
drive_amps = np.linspace(drive_amp_min, drive_amp_max, num_rabi_points)
```

```
# Build the Rabi experiments:
#   A drive pulse at the qubit frequency, followed by a measurement,
#   where we vary the drive amplitude each time.
rabi_schedules = []
for drive_amp in drive_amps:
    rabi_pulse = pulse_lib.gaussian(duration=drive_samples, amp=drive_amp,
                                      sigma=drive_sigma, name=f"Rabi drive amplitude = {drive_amp}")
    this_schedule = pulse.Schedule(name=f"Rabi drive amplitude = {drive_amp}")
```

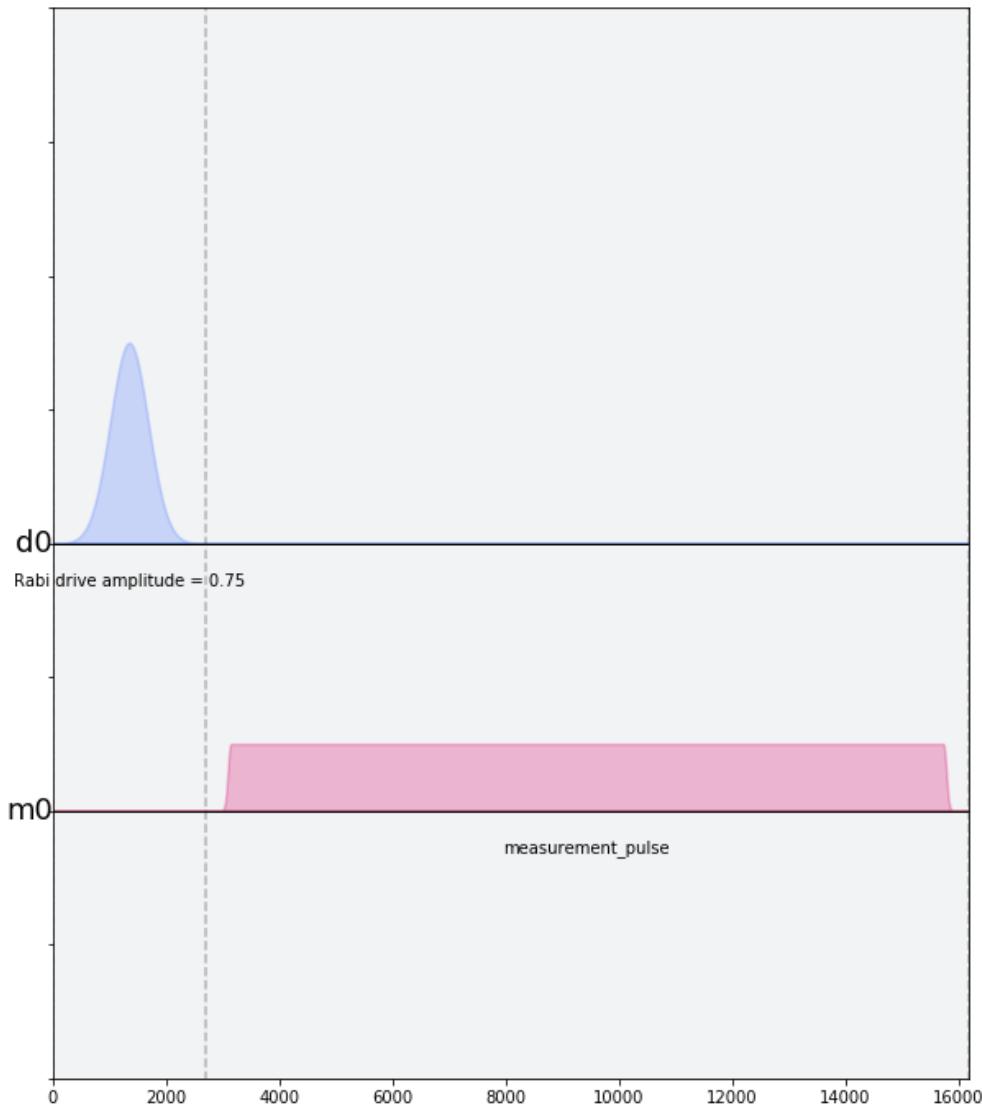
```

this_schedule += rabi_pulse(drive_chan)
# Reuse the measure_schedule from the frequency sweep experiment
this_schedule += measure_schedule << this_schedule.duration
rabi_schedules.append(this_schedule)

```

The schedule will look essentially the same as the frequency sweep experiment. The only difference is that we are running a set of experiments which vary the amplitude of the drive pulse, rather than its modulation frequency.

```
rabi_schedules[-1].draw(channels_to_plot=[drive_chan, meas_chan], label=True, scaling=1)
```



```

# Assemble the schedules into a Qobj
num_shots_per_point = 1024

rabi_experiment_program = assemble(rabi_schedules,
                                    backend=backend,

```

```
        meas_level=1,  
        meas_return='avg',  
        shots=num_shots_per_point,  
        schedule_los=[{drive_chan: rough_qubit_frequency}]  
                    * num_rabi_points)
```

```
job = backend.run(rabi_experiment_program)  
# print(job.job_id())  
job_monitor(job)
```

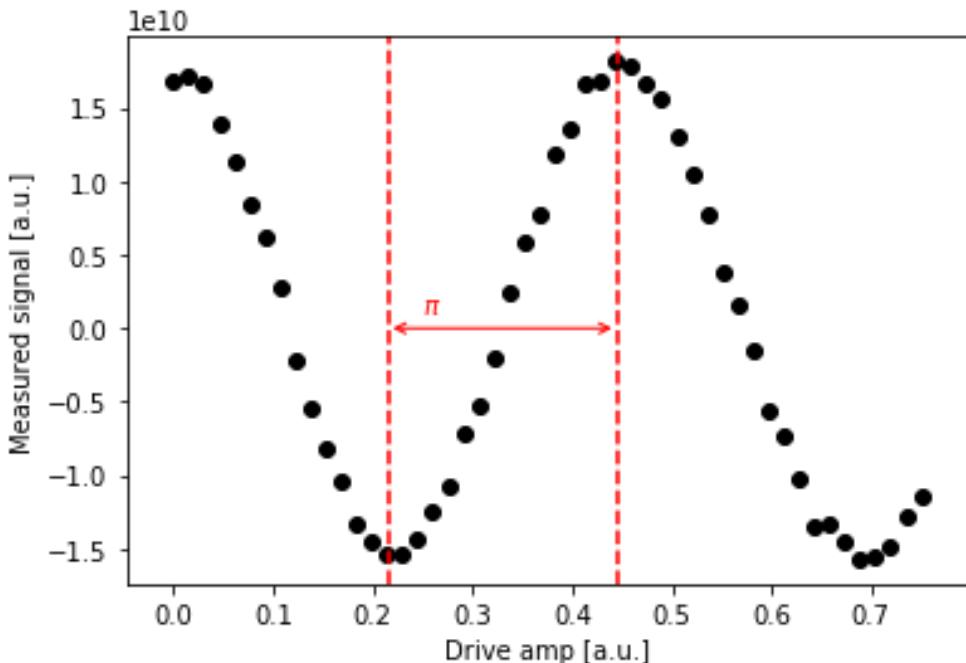
Job Status: job has successfully run

```
rabi_results = job.result(timeout=120)
```

Now that we have our results, we will extract them and fit them to a sinusoidal curve. For the range of drive amplitudes we selected, we expect that we will rotate the qubit several times completely around the Bloch sphere, starting from $|0\rangle$. The amplitude of this sinusoid tells us the fraction of the shots at that Rabi drive amplitude which yielded the $|1\rangle$ state. We want to find the first amplitude for which that fraction state reaches a maximum -- that is the calibrated amplitude that enacts a π pulse.

```
def baseline_remove(values):  
    return np.array(values) - np.mean(values)
```

```
rabi_values = []  
for i in range(num_rabi_points):  
    # Get the results for `qubit` from the ith experiment  
    rabi_values.append(rabi_results.get_memory(i)[qubit])  
  
rabi_values = baseline_remove(rabi_values)  
  
plt.xlabel("Drive amp [a.u.]")  
plt.ylabel("Measured signal [a.u.]")  
plt.scatter(driveamps, rabi_values, color='black')  
plt.axvline(0.215, color='red', linestyle='--')  
plt.axvline(0.445, color='red', linestyle='--')  
plt.annotate("", xy=(0.445, 0), xytext=(0.215, 0), arrowprops=dict(arrowstyle="<->", col  
plt.annotate("\u03c0", xy=(0.25, 0.1e10), color='red')  
plt.show()
```

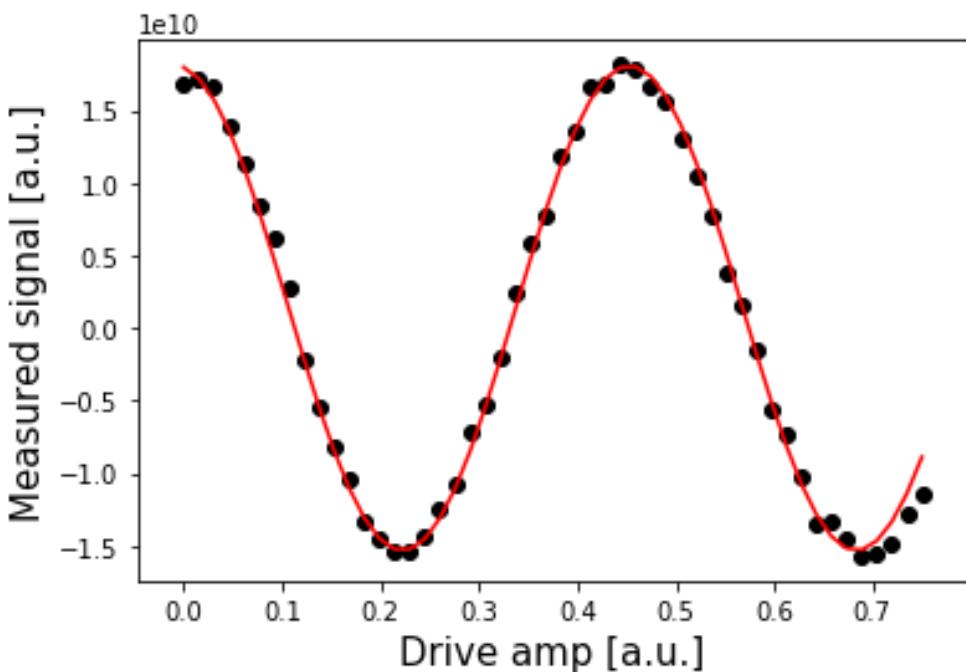


```

fit_params, y_fit = fit_function(drive_amps,
                                  rabi_values,
                                  lambda x, A, B, drive_period, phi: (A*np.cos(2*np.pi*x
[1.5e10, 0.1e10, 0.5, 0])

plt.scatter(drive_amps, rabi_values, color='black')
plt.plot(drive_amps, y_fit, color='red')
plt.xlabel("Drive amp [a.u.]", fontsize=15)
plt.ylabel("Measured signal [a.u.]", fontsize=15)
plt.show()

```



```
_,_,drive_period,_ = fit_params  
pi_amp = abs(drive_period / 2)  
print(f"Pi Amplitude = {pi_amp}")
```



```
Pi Amplitude = 0.23060434214516984
```

Our π pulse!

Let's define our pulse, with the amplitude we just found, so we can use it in later experiments.

```
pi_pulse = pulse_lib.gaussian(duration=drive_samples,  
                                amp=pi_amp,  
                                sigma=drive_sigma,  
                                name='pi_pulse')
```



B. Determining 0 vs 1

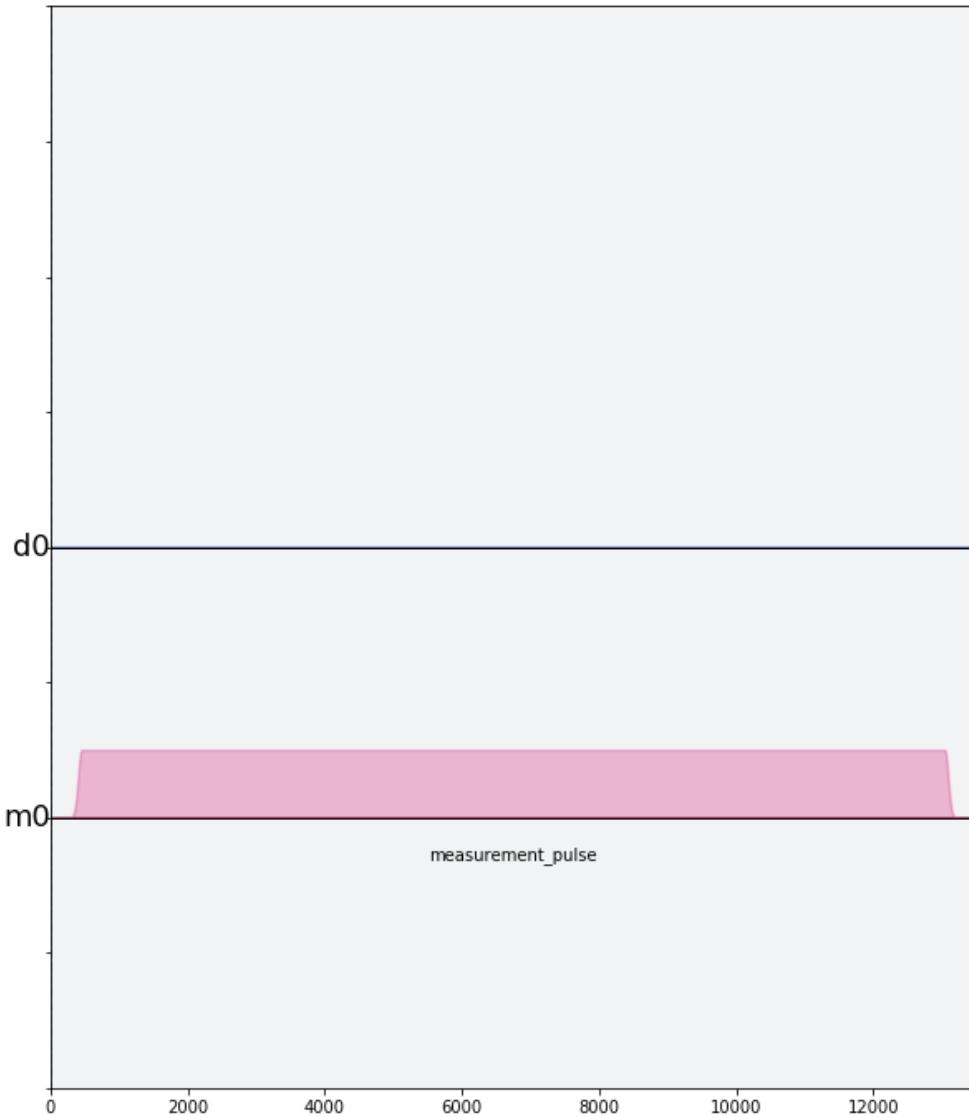
Once our π pulses have been calibrated, we can now create the state $|1\rangle$ with good probability. We can use this to find out what the states $|0\rangle$ and $|1\rangle$ look like in our measurements, by repeatedly preparing them and plotting the measured signal. This is what we use to build a discriminator, which is simply a function which takes a measured and kernelized complex value and classifies it as a 0 or a 1.

```
# Create two schedules  
  
# Ground state schedule  
gnd_schedule = pulse.Schedule(name="ground state")  
gnd_schedule += measure_schedule  
  
# Excited state schedule  
exc_schedule = pulse.Schedule(name="excited state")  
exc_schedule += pi_pulse(drive_chan) # We found this in Part 2A above  
exc_schedule += measure_schedule << exc_schedule.duration
```

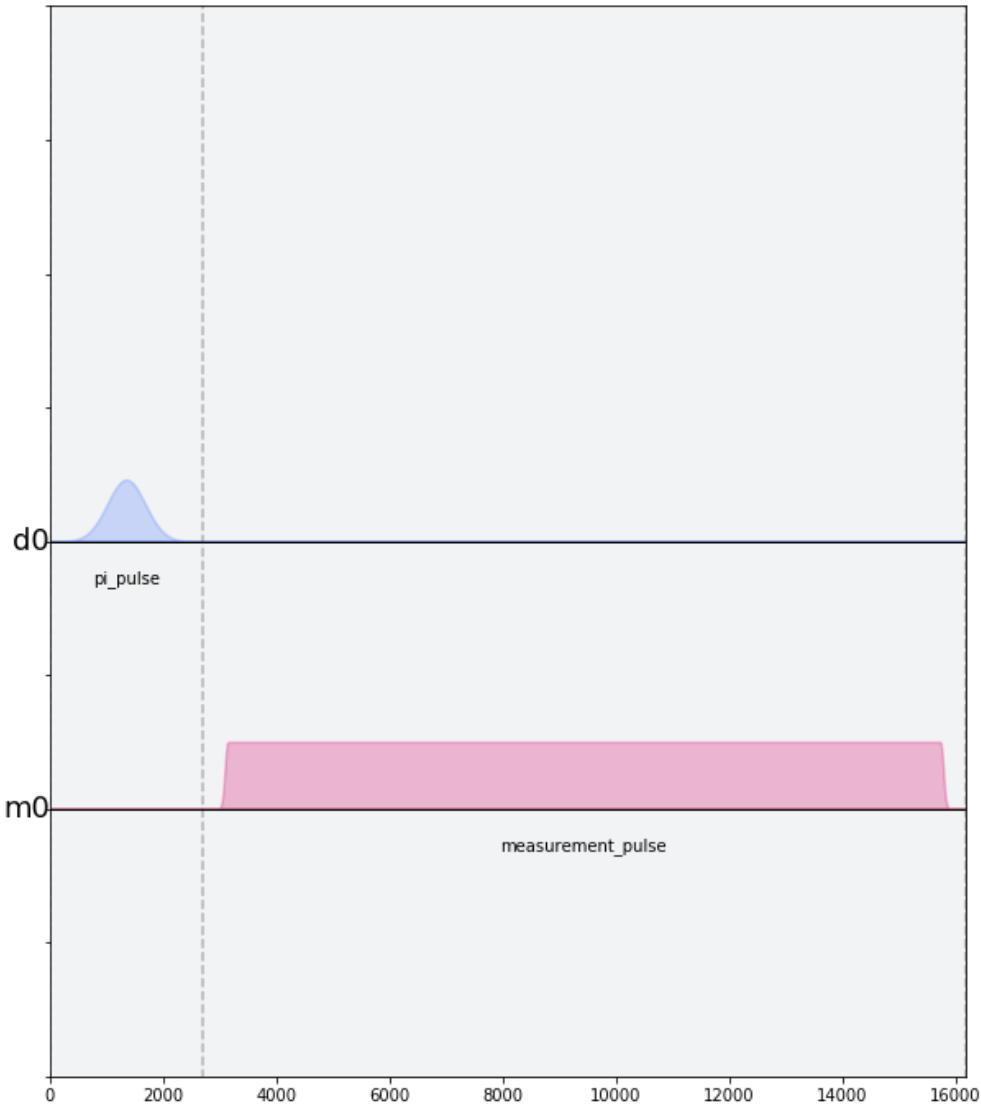


```
gnd_schedule.draw(channels_to_plot=[drive_chan, meas_chan], label=True, scaling=1.0)
```





```
exc_schedule.draw(channels_to_plot=[drive_chan, meas_chan], label=True, scaling=1.0) 
```



We assemble the ground and excited state preparation schedules into one Qobj. Each of these will run `num_shots` times. We choose `meas_level=1` this time, because we do not want the results already classified for us as $|0\rangle$ or $|1\rangle$. Instead, we want kernelized data: raw acquired data that has gone through a kernel function to yield a single complex value for each shot. (You can think of a kernel as a dot product applied to the raw measurement data.) We pass the same frequency for both schedules, although it is only used by the `exc_schedule`.

```
# Execution settings
num_shots = 1024

gnd_exc_program = assemble([gnd_schedule, exc_schedule],
                           backend=backend,
                           meas_level=1,
                           meas_return='single',
                           shots=num_shots,
                           schedule_los=[{drive_chan: rough_qubit_frequency}] * 2)
```

```
job = backend.run(gnd_exc_program)
# print(job.job_id())
job_monitor(job)
```



Job Status: job has successfully run

```
gnd_exc_results = job.result(timeout=120)
```



Now that we have the results, we can visualize the two populations which we have prepared on a simple scatter plot, showing results from the ground state program in blue and results from the excited state preparation program in red.

```
gnd_results = gnd_exc_results.get_memory(0)[:, qubit]
exc_results = gnd_exc_results.get_memory(1)[:, qubit]

plt.figure(figsize=[4,4])
# Plot all the results
# All results from the gnd_schedule are plotted in blue
plt.scatter(np.real(gnd_results), np.imag(gnd_results),
            s=5, cmap='viridis', c='blue', alpha=0.5, label='state_0')
# All results from the exc_schedule are plotted in red
plt.scatter(np.real(exc_results), np.imag(exc_results),
            s=5, cmap='viridis', c='red', alpha=0.5, label='state_1')

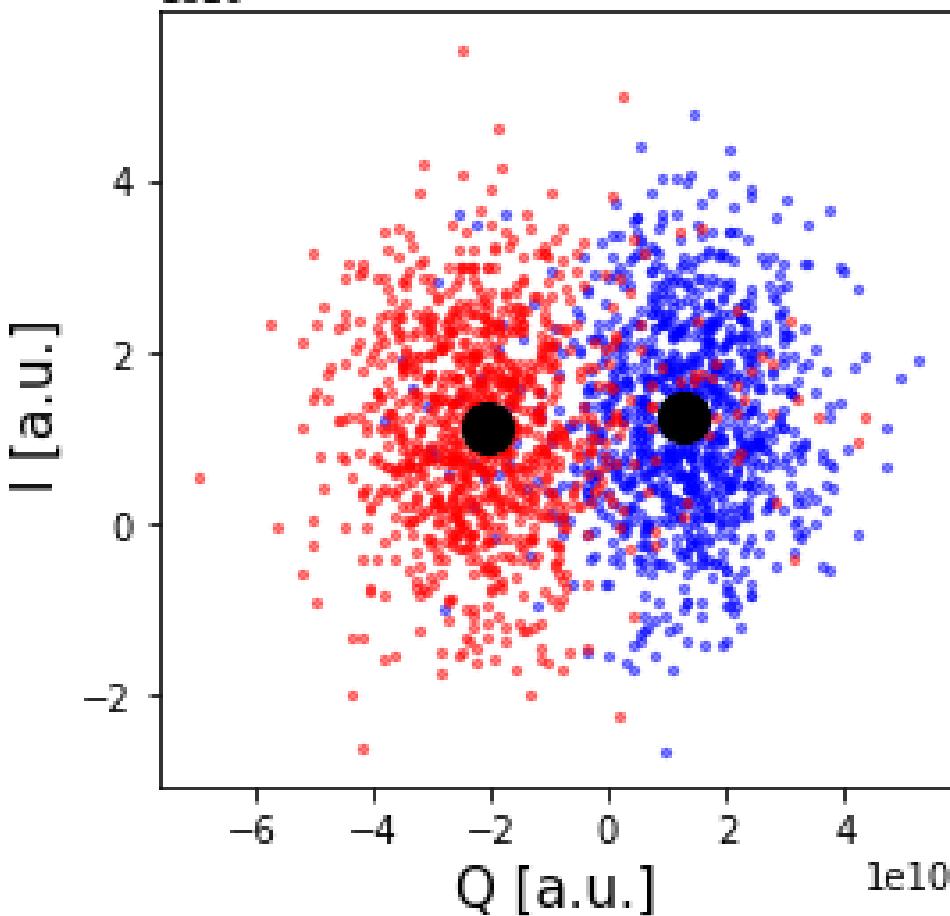
# Plot a Large dot for the average result of the 0 and 1 states.
mean_gnd = np.mean(gnd_results) # takes mean of both real and imaginary parts
mean_exc = np.mean(exc_results)
plt.scatter(np.real(mean_gnd), np.imag(mean_gnd),
            s=200, cmap='viridis', c='black', alpha=1.0, label='state_0_mean')
plt.scatter(np.real(mean_exc), np.imag(mean_exc),
            s=200, cmap='viridis', c='black', alpha=1.0, label='state_1_mean')

plt.ylabel('I [a.u.]', fontsize=15)
plt.xlabel('Q [a.u.]', fontsize=15)
plt.title("0-1 discrimination", fontsize=15)

plt.show()
```



0-1 discrimination



We can clearly see that the two populations of $|0\rangle$ and $|1\rangle$ form their own clusters. Kerneled measurement results (from `meas_level=1`) are classified (into `meas_level=2`) by applying a discriminator which optimally separates these two clusters. Optimal separation is simply a line in the IQ plane, equidistant from the average results we plotted above in the large dot, and normal to the line connecting the two dots.

We can set up a quick classifier function by returning 0 if a given point is closer to the mean of the ground state results, and returning 1 if the point is closer to the average excited state results.

```
import math

def classify(point: complex):
    """Classify the given state as |0> or |1>."""
    def distance(a, b):
        return math.sqrt((np.real(a) - np.real(b))**2 + (np.imag(a) - np.imag(b))**2)
    return int(distance(point, mean_exc) < distance(point, mean_gnd))
```

C. Measuring T_1 using inversion recovery

The T_1 time of a qubit is the time it takes for a qubit to decay from the excited state to the ground state. It is important because it limits the duration of meaningful programs we can run on the quantum computer.

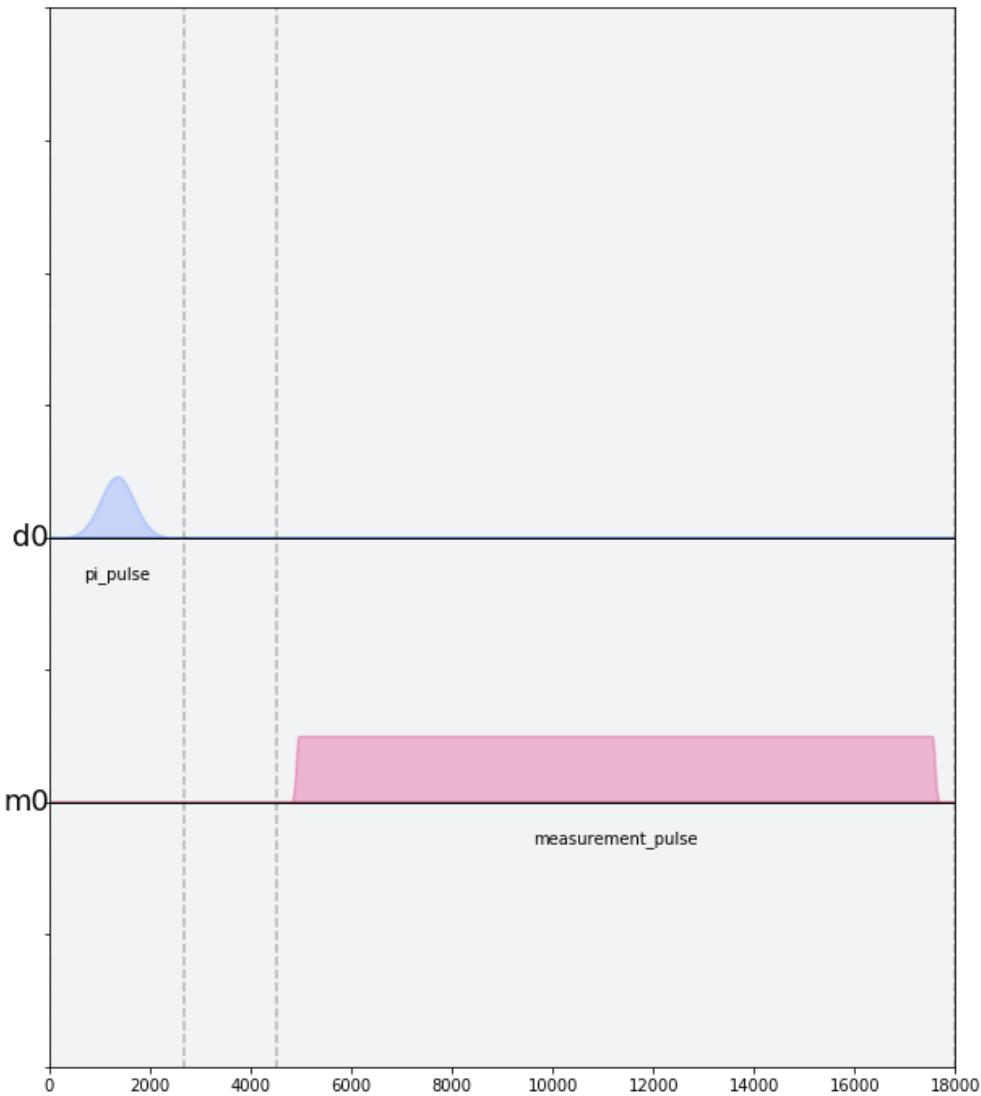
Measuring T_1 is similar to our previous experiments, and uses the π pulse we've calibrated. We again apply a single drive pulse, our π pulse, then apply a measure pulse. However, this time we do not apply the measurement immediately. We insert a delay, and vary that delay between experiments. When we plot the measured signal against delay time, we will see a signal that decays exponentially as the qubit relaxes in energy. The decay time is the T_1 , or relaxation time, of the qubit!

```
# T1 experiment parameters
time_max_us = 450
time_step_us = 6
times_us = np.arange(1, time_max_us, time_step_us)
# Convert to ns first (1 us = 1000 ns) and then convert to units of dt
delay_times_dt = times_us * 1e-6 / dt
# We will use the same `pi_pulse` and qubit frequency that we calibrated and used before
```

```
# Create schedules for the experiment
t1_schedules = []
for delay in delay_times_dt:
    this_schedule = pulse.Schedule(name=f"T1 delay = {delay} us")
    this_schedule += pi_pulse(drive_chan)
    this_schedule |= measure_schedule << int(delay)
    t1_schedules.append(this_schedule)
```

We can check out our T_1 schedule, too. To really get a sense of this experiment, try looking at a couple of the schedules by running the next cell multiple times, with different values of `sched_idx`. You will see the measurement pulse start later as you increase `sched_idx`.

```
sched_idx = 0
t1_schedules[sched_idx].draw(channels_to_plot=[drive_chan, meas_chan], label=True, scal
```



```
# Execution settings
num_shots = 256

t1_experiment = assemble(t1_schedules,
                         backend=backend,
                         meas_level=1,
                         meas_return='avg',
                         shots=num_shots,
                         schedule_los=[{drive_chan: rough_qubit_frequency}] * len(t1_sc
```

```
job = backend.run(t1_experiment)
# print(job.job_id())
job_monitor(job)
```

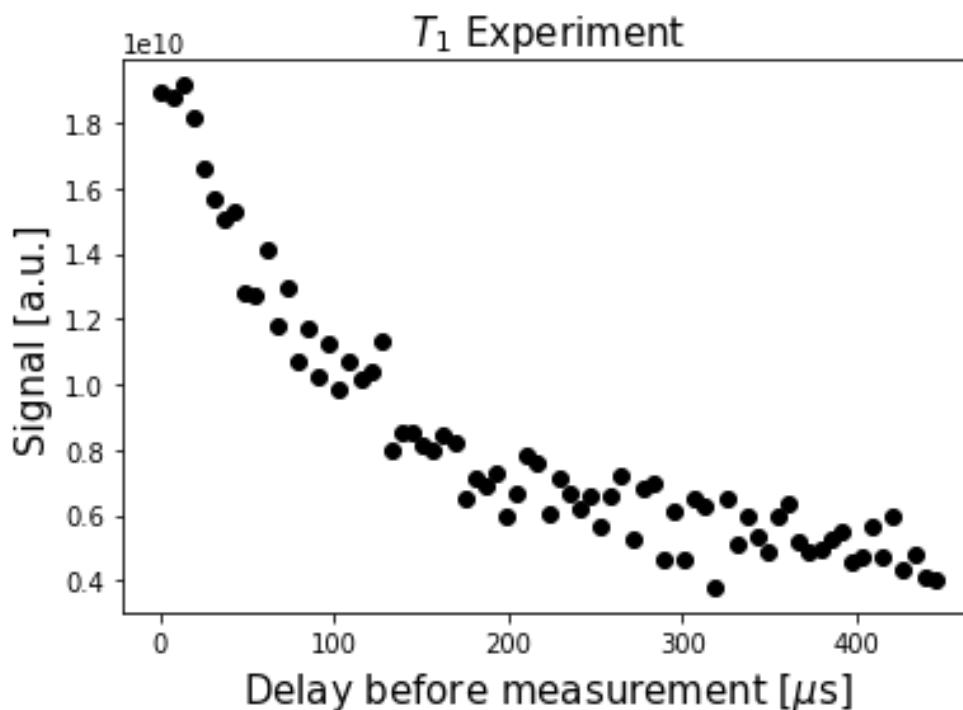
Job Status: job has successfully run

```
t1_results = job.result(timeout=120)
```



```
t1_values = []
for i in range(len(times_us)):
    t1_values.append(t1_results.get_memory(i)[qubit])

plt.scatter(times_us, t1_values, color='black')
plt.title("$T_1$ Experiment", fontsize=15)
plt.xlabel('Delay before measurement [$\mu s$]', fontsize=15)
plt.ylabel('Signal [a.u.]', fontsize=15)
plt.show()
```



We can then fit the data to a decaying exponential, giving us T1!

```
# Fit the data
fit_params, y_fit = fit_function(times_us, t1_values,
                                 lambda x, A, C, T1: (A * np.exp(-x / T1) + C),
                                 [-3e10, 3e10, 100]
)

_, _, T1 = fit_params

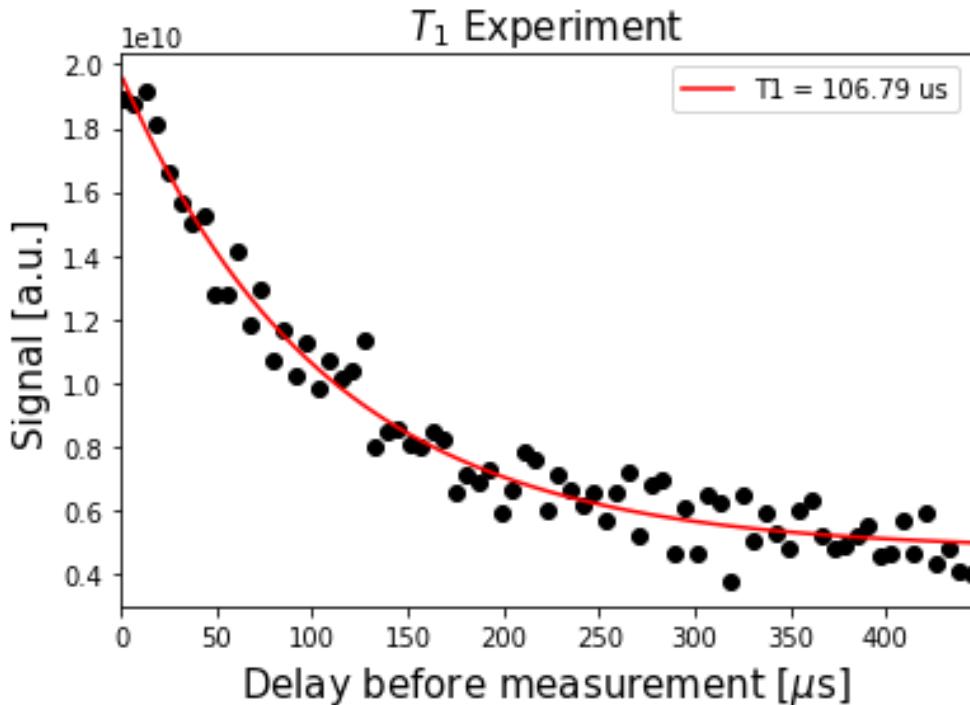
plt.scatter(times_us, t1_values, color='black')
plt.plot(times_us, y_fit, color='red', label=f"T1 = {T1:.2f} us")
plt.xlim(0, np.max(times_us))
plt.title("$T_1$ Experiment", fontsize=15)
```



```

plt.xlabel('Delay before measurement [$\mu s]', fontsize=15)
plt.ylabel('Signal [a.u.]', fontsize=15)
plt.legend()
plt.show()

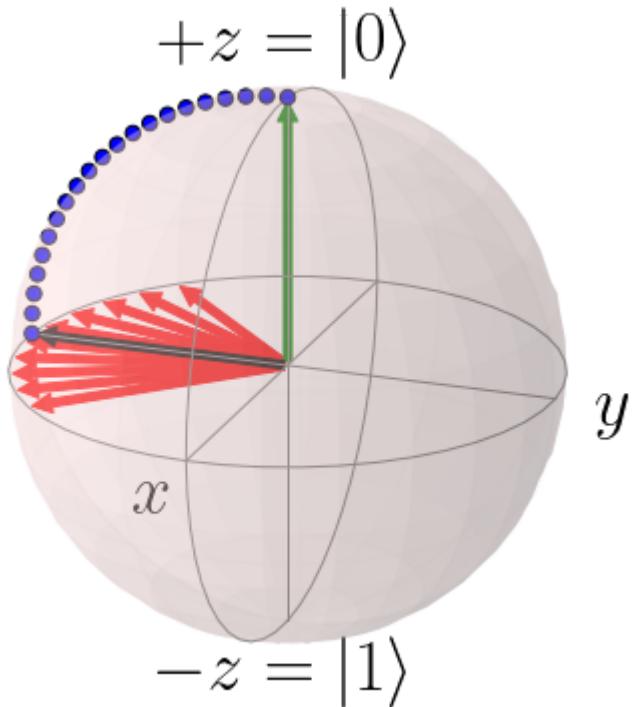
```



Part 3. Determining qubit coherence

A. Measuring the qubit frequency precisely using a Ramsey experiment

Now, we determine the qubit frequency to better precision. This is done using a Ramsey pulse sequence. In this pulse sequence, we first apply a $\pi/2$ ("pi over two") pulse, wait some time Δt , and then apply another $\pi/2$ pulse. Since we are measuring the signal from the qubit at the same frequency as the pulses, we should observe oscillations at the difference in frequency between the applied pulses and the qubit.



```
# Ramsey experiment parameters
time_max_us = 1.8
time_step_us = 0.025
times_us = np.arange(0.1, time_max_us, time_step_us)
# Convert to ns first (1 us = 1000 ns) and then convert to units of dt
delay_times_dt = times_us * 1e-6 / dt

# Drive parameters
# The drive amplitude for pi/2 is simply half the amplitude of the pi pulse
drive_amp = pi_amp / 2
# x_90 is a concise way to say pi_over_2; i.e., an X rotation of 90 degrees
x90_pulse = pulse_lib.gaussian(duration=drive_samples,
                                  amp=drive_amp,
                                  sigma=drive_sigma,
                                  name='x90_pulse')
```

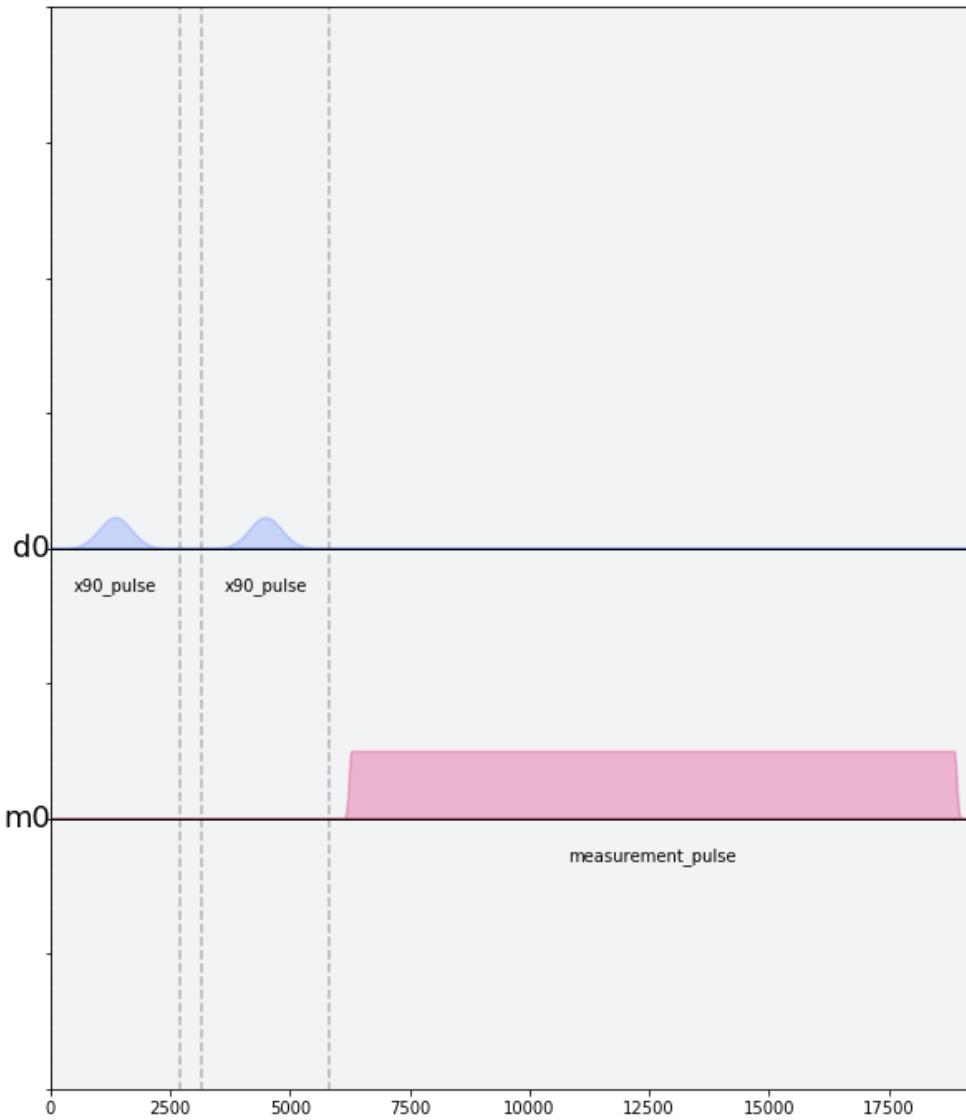
```
# create schedules for Ramsey experiment
ramsey_schedules = []

for delay in delay_times_dt:
    this_schedule = pulse.Schedule(name=f"Ramsey delay = {delay * dt / 1e3} us")
    this_schedule |= x90_pulse(drive_chan)
    this_schedule |= x90_pulse(drive_chan) << int(this_schedule.duration + delay)
    this_schedule |= measure_schedule << int(this_schedule.duration)

    ramsey_schedules.append(this_schedule)
```

Just like for T_1 schedules, it will be illuminating to execute the next cell multiple times to inspect a few of the schedules we've made. As you look at increasing indices of `ramsey_schedules`, the delay between the two $\pi/2$ pulses will increase.

```
ramsey_schedules[0].draw(channels_to_plot=[drive_chan, meas_chan], label=True, scaling=
```



Here, we will apply a commonly used experimental trick. We will drive the pulses off-resonance by a known amount, which we will call `detuning_MHz`. The measured Ramsey signal should show oscillations with frequency near `detuning_MHz`, with a small offset. This small offset is exactly how far away `rough_qubit_frequency` was from the qubit frequency.

```
# Execution settings  
num_shots = 256
```

```
detuning_MHz = 2
ramsey_frequency = round(rough_qubit_frequency + detuning_MHz/1.0e3, 6)
ramsey_program = assemble(ramsey_schedules,
                           backend=backend,
                           meas_level=1,
                           meas_return='avg',
                           shots=num_shots,
                           schedule_los=[{drive_chan: ramsey_frequency}]*len(ramsey_s
                           ))
```

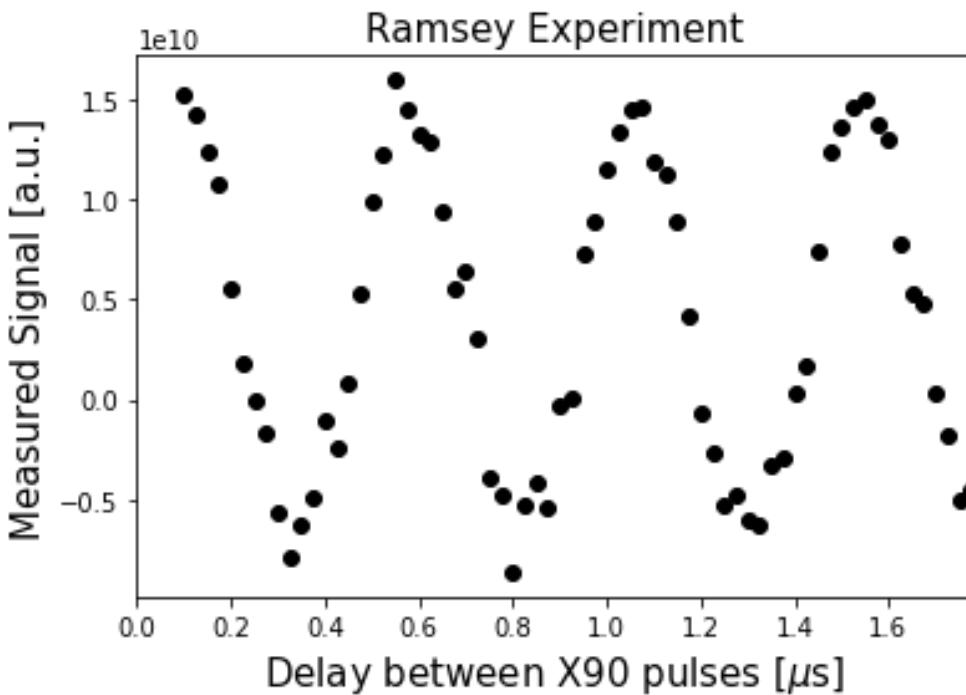
```
job = backend.run(ramsey_program)
# print(job.job_id())
job_monitor(job)
```

Job Status: job has successfully run

```
ramsey_results = job.result(timeout=120)
```

```
ramsey_values = []
for i in range(len(times_us)):
    ramsey_values.append(ramsey_results.get_memory(i)[qubit])

plt.scatter(times_us, ramsey_values, color='black')
plt.xlim(0, np.max(times_us))
plt.title("Ramsey Experiment", fontsize=15)
plt.xlabel('Delay between X90 pulses [μs]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.show()
```



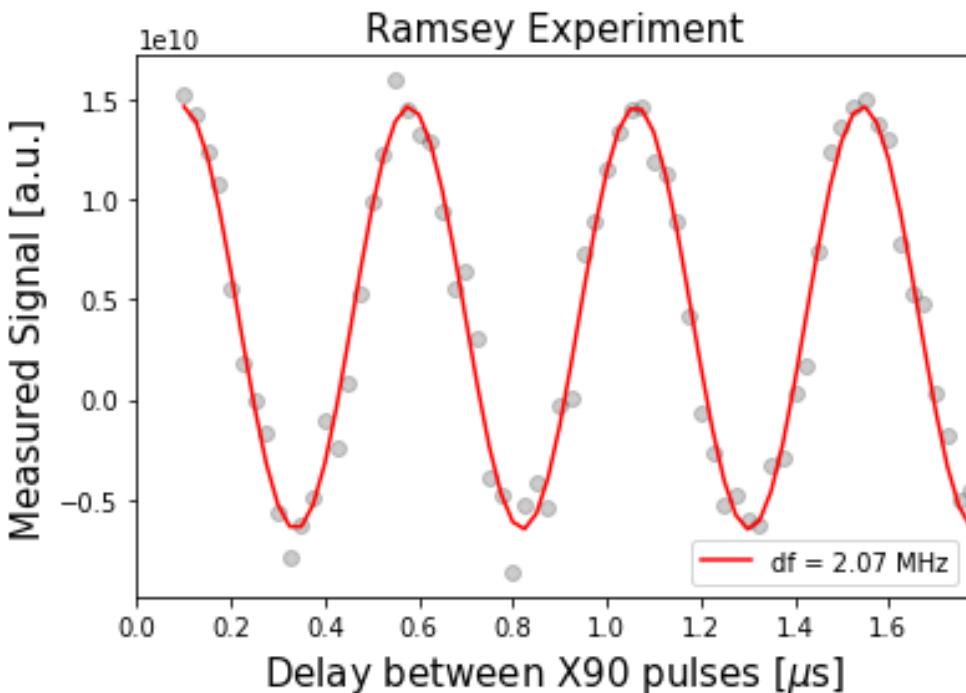
We will fit the data to a sinusoid, and extract the information we are interested in -- namely, Δf .

```

fit_params, y_fit = fit_function(times_us, ramsey_values,
                                  lambda x, A, del_f_MHz, C, B: (
                                      A * np.cos(2*np.pi*del_f_MHz*x - C) + B
                                  ),
                                  [2e10, 1./0.4, 0, 0.25e10]
)
# Off-resonance component
_, del_f_MHz, _, _ = fit_params

plt.scatter(times_us, ramsey_values, color='black', alpha=0.2)
plt.plot(times_us, y_fit, color='red', label=f"df = {del_f_MHz:.2f} MHz")
plt.xlim(0, np.max(times_us))
plt.xlabel('Delay between X90 pulses [$\mu\text{s}$]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Ramsey Experiment', fontsize=15)
plt.legend()
plt.show()

```



Now that we know `del_f_MHz`, we can update our estimate of the qubit frequency.

```
precise_qubit_freq = round(rough_qubit_frequency + (del_f_MHz - detuning_MHz) / 1e3, 6)
print(f"Our updated qubit frequency is now {precise_qubit_freq} GHz. It used to be {rou
```

Our updated qubit frequency is now 4.974421 GHz. It used to be 4.97435 GHz

B. Measuring T_2 using Hahn echoes

Next, we can measure the coherence time, T_2 , of our qubit. The pulse sequence used to do this experiment is known as a Hahn echo, a term that comes from the NMR community. A Hahn echo experiment is very similar to the Ramsey experiment above, with an additional π pulse between the two $\pi/2$ pulses. The π pulse at time τ reverses the accumulation of phase, and results in an echo at time 2τ , where we apply the last $\pi/2$ pulse to do our measurement. The Wikipedia page for Hahn echoes

The decay time for the Hahn echo experiment gives us the coherence time, T_2 .

```
# T2 experiment parameters
tau_max_us = 200
tau_step_us = 4
taus_us = np.arange(2, tau_max_us, tau_step_us)
```

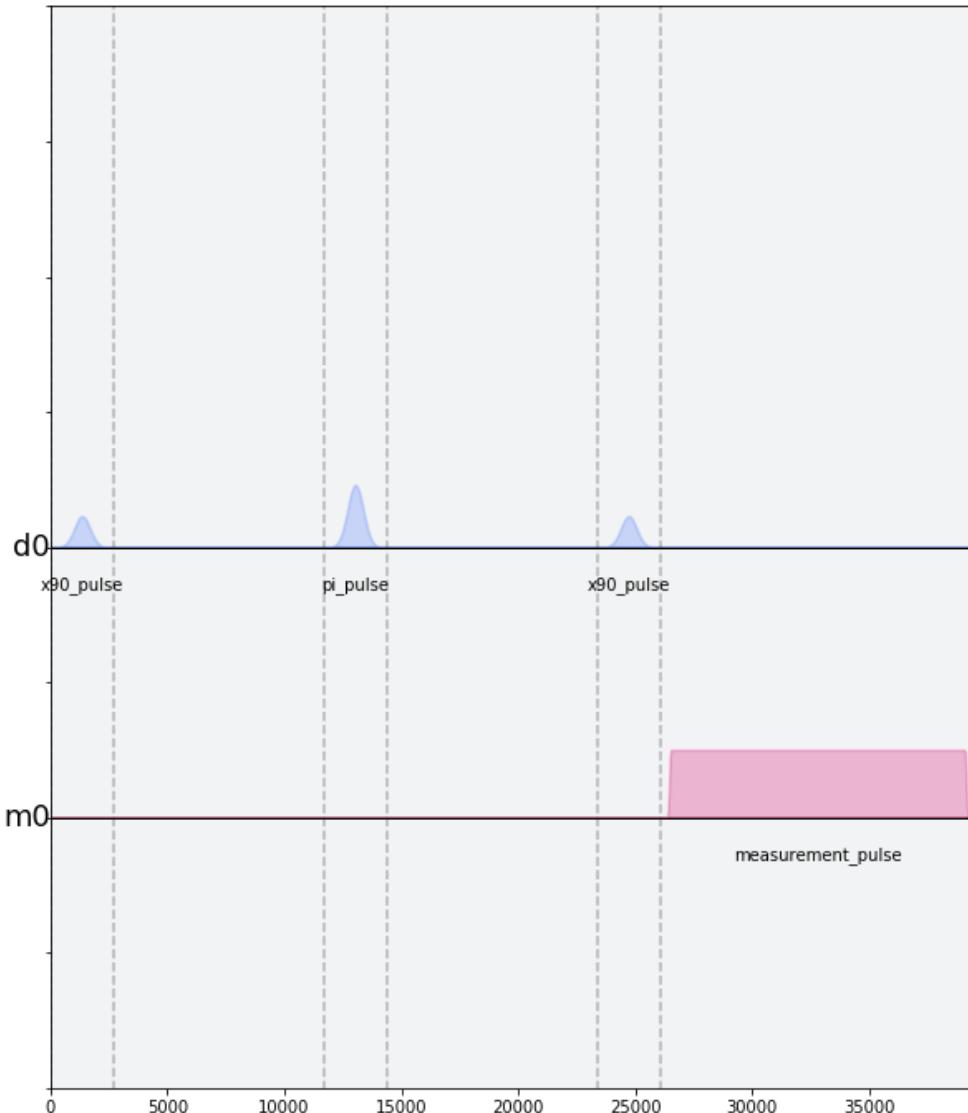
```
# Convert to ns first (1 us = 1000 ns) and then convert to units of dt
delay_times_dt = taus_us * 1e-6 / dt
```

```
# We will use the pi_pulse and x90_pulse from previous experiments
```

```
t2_schedules = []
for tau in delay_times_dt:
    this_schedule = pulse.Schedule(name=f"T2 delay = {tau} us")
    this_schedule |= x90_pulse(drive_chan)
    this_schedule |= pi_pulse(drive_chan) << int(this_schedule.duration + tau)
    this_schedule |= x90_pulse(drive_chan) << int(this_schedule.duration + tau)
    this_schedule |= measure_schedule << int(this_schedule.duration)

t2_schedules.append(this_schedule)
```

```
t2_schedules[0].draw(channels_to_plot=[drive_chan, meas_chan], label=True, scaling=100)
```



```
# Execution settings
num_shots_per_point = 512

t2_experiment = assemble(t2_schedules,
                         backend=backend,
                         meas_level=1,
                         meas_return='avg',
                         shots=num_shots_per_point,
                         schedule_los=[{drive_chan: precise_qubit_freq}]
                           * len(t2_schedules))
```

```
job = backend.run(t2_experiment)
# print(job.job_id())
job_monitor(job)
```

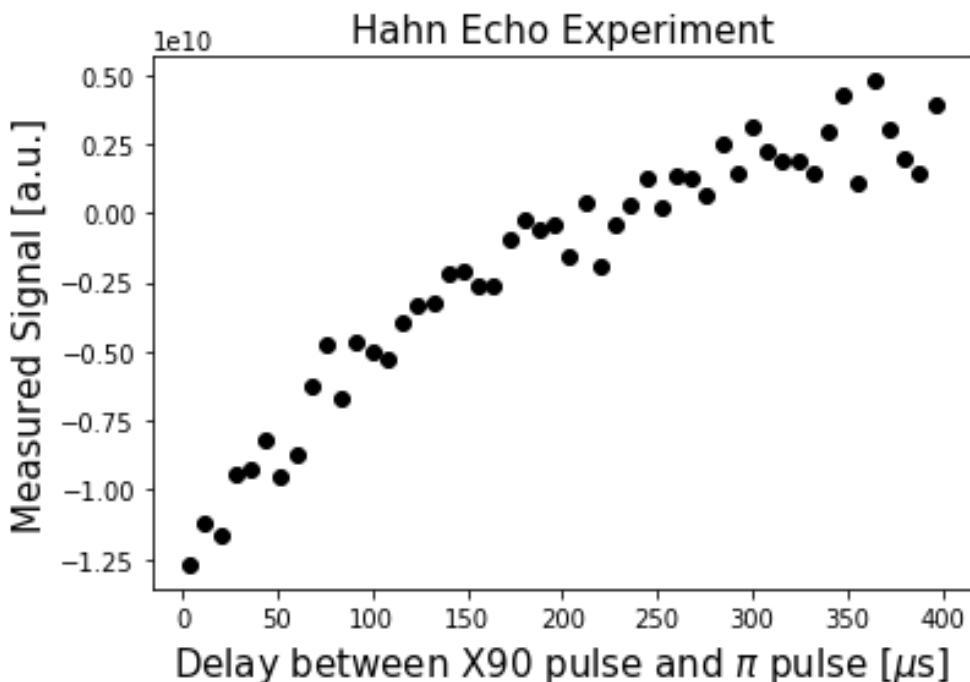
Job Status: job has successfully run

```
t2_results = job.result(timeout=120)
```



```
t2_values = []
for i in range(len(taus_us)):
    t2_values.append(t2_results.get_memory(i)[qubit])

plt.scatter(2*taus_us, t2_values, color='black')
plt.xlabel('Delay between X90 pulse and  $\pi$  pulse [ $\mu s$ ]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Hahn Echo Experiment', fontsize=15)
plt.show()
```

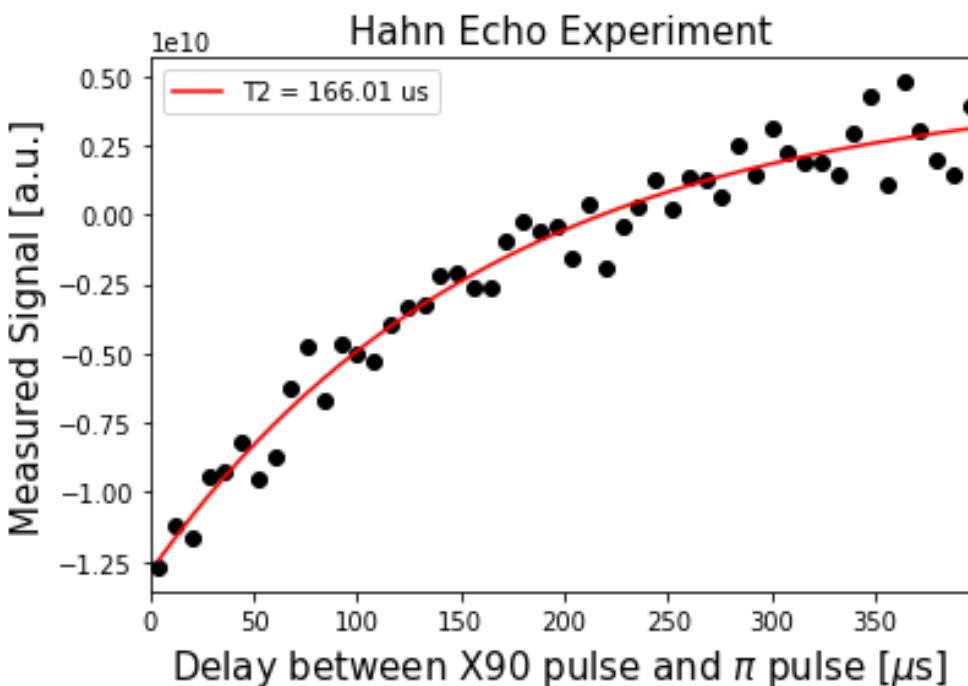


```
fit_params, y_fit = fit_function(2*taus_us, t2_values,
    lambda x, A, B, T2: (A * np.exp(-x / T2) + B),
    [-1.2e15, -2.4e15, 20])

_, _, T2 = fit_params
print()

plt.scatter(2*taus_us, t2_values, color='black')
plt.plot(2*taus_us, y_fit, color='red', label=f"T2 = {T2:.2f} us")
plt.xlim(0, np.max(2*taus_us))
plt.xlabel('Delay between X90 pulse and  $\pi$  pulse [ $\mu s$ ]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Hahn Echo Experiment', fontsize=15)
plt.legend()
plt.show()
```





C. Dynamical decoupling

A single π pulse is able to eliminate quasi-static noise due to the reversal of phase accumulation. This concept can be extended to noise that cannot be approximated as quasi-static by applying several π pulses in succession. This technique, commonly known as dynamical decoupling, allows us to cancel different frequencies of noise and is used to extract longer coherence times from qubits.

```
# DD experiment parameters
tau_us_min = 1
tau_us_max = 40
tau_step_us = 1.5
taus_us = np.arange(tau_us_min, tau_us_max, tau_step_us)
# Convert to dt
taus_dt = taus_us * 1e-6 / dt
numpulses = 6 # apply two pi pulses
print(f"Total time ranges from {2.*numpulses*taus_us[0]} to {2.*numpulses*taus_us[-1]}
```

Total time ranges from 12.0 to 462.0 us

```
T2DD_schedules = []
for delay in taus_dt:
    this_schedule = pulse.Schedule(name=f"T2DD delay = {delay} us")
```

```

this_schedule |= x90_pulse(drive_chan)
this_schedule |= pi_pulse(drive_chan) << int(this_schedule.duration + delay)

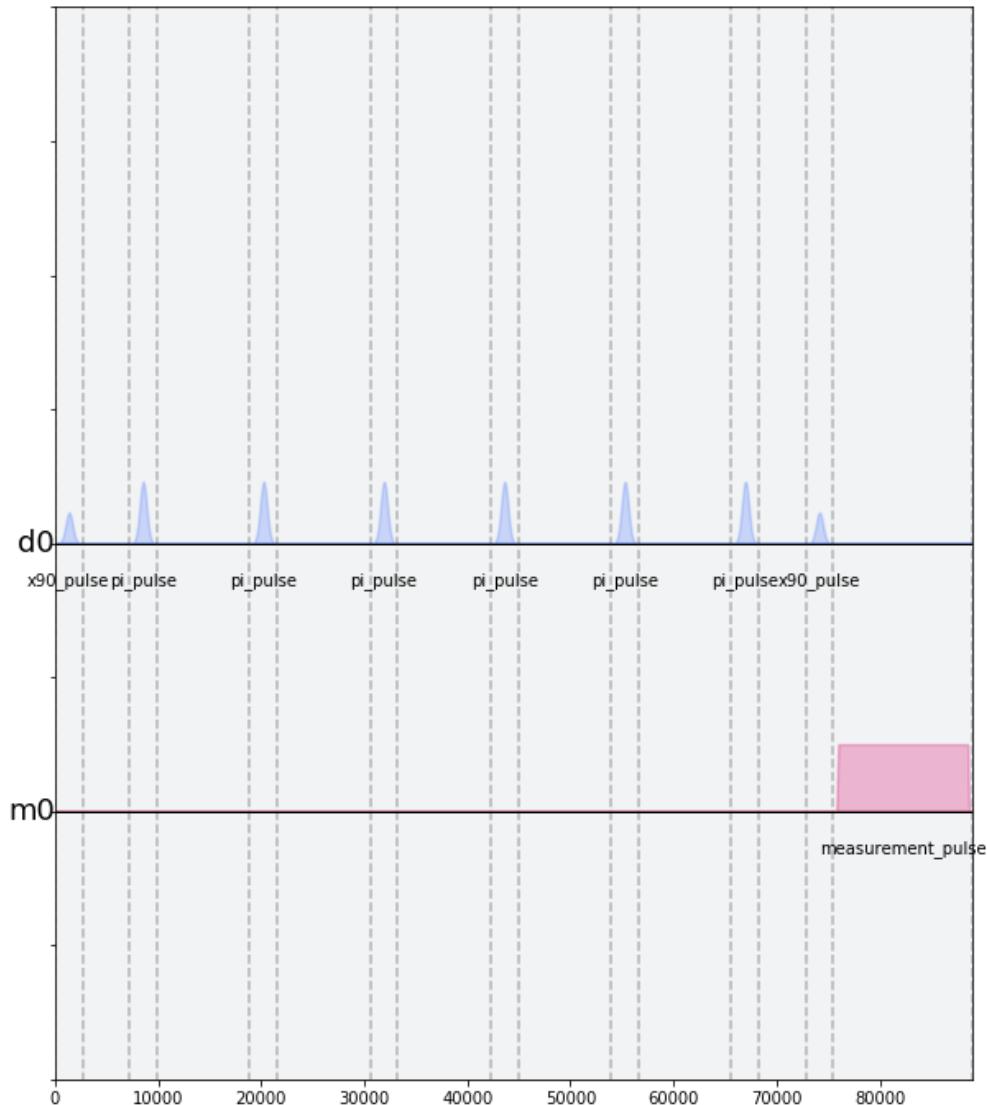
for _ in range(num_pulses - 1):
    this_schedule |= pi_pulse(drive_chan) << int(this_schedule.duration + 2*delay)

this_schedule |= x90_pulse(drive_chan) << int(this_schedule.duration + delay)
this_schedule |= measure_schedule << int(this_schedule.duration)

T2DD_schedules.append(this_schedule)

```

T2DD_schedules[0].draw(channels_to_plot=[drive_chan, meas_chan], label=True, scaling=1.



num_shots_per_point = 1024

T2DD_experiment = assemble(T2DD_schedules,

```
backend=backend,
meas_level=1,
meas_return='avg',
shots=num_shots_per_point,
schedule_los=[{drive_chan: precise_qubit_freq}]
* len(T2DD_schedules))
```

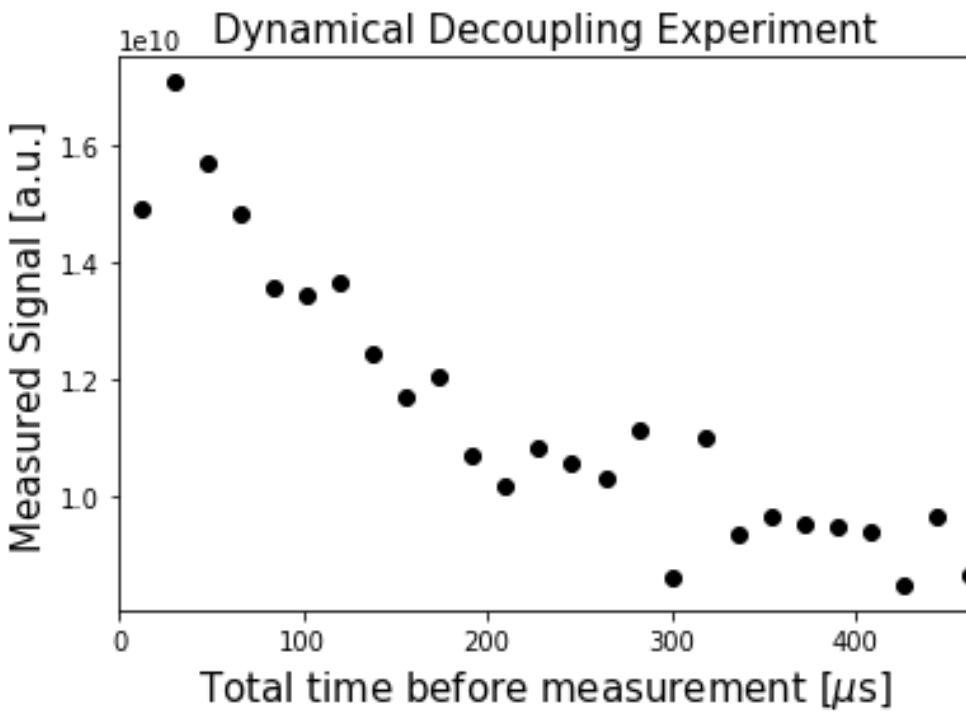
```
job = backend.run(T2DD_experiment)
# print(job.job_id())
job_monitor(job)
```

Job Status: job has successfully run

```
T2DD_results = job.result(timeout=120)
```

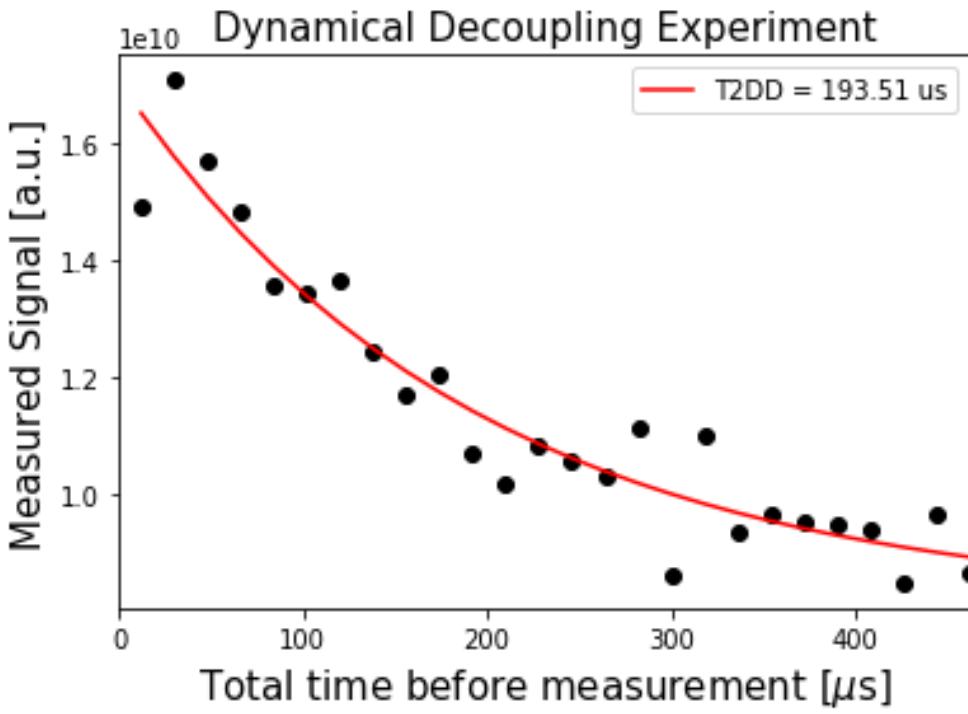
```
times_us = 2.*numpipulses*taus_us
DD_values = []
for i in range(len(taus_us)):
    DD_values.append(T2DD_results.get_memory(i)[qubit])
```

```
plt.scatter(times_us, DD_values, color='black')
plt.xlim(0, np.max(times_us))
plt.xlabel('Total time before measurement [$\mu s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Dynamical Decoupling Experiment', fontsize=15)
plt.show()
```



```
# Fit the data
fit_func = lambda x, A, B, T2DD: (A * np.exp(-x / T2DD) + B)
fitparams, conv = curve_fit(fit_func, times_us, DD_values, [1.7e10, 0.8e10, 150])

_, _, T2DD = fitparams
plt.scatter(times_us, DD_values, color='black')
plt.plot(times_us, fit_func(times_us, *fitparams), color='red', label=f"T2DD = {T2DD:.2f} \u00b5s")
plt.xlim([0, np.max(times_us)])
plt.xlabel('Total time before measurement [$\mu s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Dynamical Decoupling Experiment', fontsize=15)
plt.legend()
plt.show()
```



Part 4. References

1. H. Abraham, I. Y. Akhalwaya, G. Aleksandrowicz, T. Alexander, G. Alexandrowics, E. Arbel, A. Asfaw, C. Azaustre, P. Barkoutsos, G. Barron, L. Bello, Y. Ben-Haim, L. S. Bishop, S. Bosch, D. Bucher, CZ, F. Cabrera, P. Calpin, L. Capelluto, J. Carballo, C.-F. Chen, A. Chen, R. Chen, J. M. Chow, C. Claus, A. W. Cross, A. J. Cross, J. Cruz- Benito, C. Culver, A. D. C órcoles-Gonzales, S. Dague, M. Dartailh, A. R. Davila, D. Ding, E. Dumitrescu, K. Dumon, I. Duran, P. Eendebak, D. Egger, M. Everitt, P. M. Fern ández, A. Frisch, A. Fuhrer, J. Gacon, Gadi, B. G. Gago, J. M. Gambetta, L. Garcia, S. Garion, Gawel-Kus, L. Gil, J. Gomez-Mosquera, S. de la Puente Gonz ález, D. Green- berg, J.A. Gunnels, I. Haide, I. Hamamura, V. Havlicek, J. Hellmers, L. Herok, H. Horii, C. Howington, W. Hu, S. Hu, H. Imai, T. Imamichi, R. Iten, T. Itoko, A. Javadi-Abhari, Jessica, K. Johns, N. Kanazawa, A. Karazeev, P. Kassebaum, V. Krishnan, K. Kr- sulich, G. Kus, R. LaRose, R. Lambert, J. Latone, S. Lawrence, P. Liu, P. B. Z. Mac, Y. Maeng, A. Malyshev, J. Marecek, M. Marques, D. Mathews, A. Matsuo, D. T. Mc- Clure, C. McGarry, D. McKay, S. Meesala, A. Mezzacapo, R. Midha, Z. Minev, P. Mu- rali, J. Mu üggenburg, D. Nadlinger, G. Nannicini, P. Nation, Y. Naveh, Nick- Singstock, P. Niroula, H. Norlen, L. J. O'Riordan, S. Oud, D. Padilha, H. Paik, S. Perriello, A. Phan, M. Pistoia, A. Pozas-iKerstjens, V. Prutyanov, J. P érez, Quintiii, R. Raymond, R. M.-C. Redondo, M. Reuter, D. M. Rodr íquez, M. Ryu, M. Sandberg, N. Sathaye, B. Schmitt, C. Schnabel, T. L. Scholten, E. Schoute, I. F. Sertage, Y. Shi, A. Silva, Y. Siraichi, S. Sivarajah, J. A. Smolin, M. Soeken, D. Steenken, M. Stypulkoski, H. Takahashi, C. Taylor, P. Taylor, S. Thomas, M. Tillet, M. Tod, E. de la Torre, K. Trabing, M. Treinish, TrishaPe, W. Turner, Y. Vaknin, C. R. Valcarce, F. Varchon, D. Vogt- Lee, C. Vuillot, J. Weaver, R. Wieczorek, J. A. Wildstrom, R. Wille, E. Winston, J. J. Woehr, S. Woerner, R. Woo, C. J. Wood, R. Wood, S. Wood, J. Wootton, D. Yerlin, J. Yu, L. Zdanski, Zoufalc, azulehner, drholmie, fanizzamarco, kanejess, klinvill, merav aharoni, ordmoj, tigerjack, yang.luh, and yotamvakninibm, "Qiskit: An open-source framework for quantum computing," 2019.

2. D. C. McKay, T. Alexander, L. Bello, M. J. Biercuk, L. Bishop, J. Chen, J. M. Chow, A. D. Córcoles, D. Egger, S. Filipp, J. Gomez, M. Hush, A. Javadi-Abhari, D. Moreda, P. Nation, B. Paulovicks, E. Winston, C. J. Wood, J. Wootton, and J. M. Gambetta, "Qiskit backend specifications for OpenQASM and OpenPulse experiments," 2018.

```
import qiskit.tools.jupyter  
%qiskit_version_table
```



Version Information

Qiskit Software	Version
Qiskit	0.14.0
Terra	0.11.0
Aer	0.3.4
Ignis	0.2.0
Aqua	0.6.1
IBM Q Provider	0.4.4
System information	
Python	3.7.3 (default, Mar 27 2019, 16:54:48) [Clang 4.0.1 (tags/RELEASE_401/final)]
OS	Darwin
CPUs	6
Memory (Gb)	16.0
Wed Dec 11 01:59:49 2019 SAST	

Introduction to Quantum Error Correction

```
from qiskit import *
```



Correcting errors in phone calls

In this chapter, we will introduce the idea of quantum error correction. This will be done via the quantum repetition code: a simple quantum error correcting code that has all the basic features of more complex methods. Unfortunately, it is not a fully useful code in itself (which is why the more complex methods are needed). Nevertheless it is a great starting point, and a good first test of how well quantum error correction might work on a given device.

Before all this, it would be helpful to first introduce the idea of error correction in general.

Imagine you are speaking on the phone. Someone asks you a question to which the answer is 'yes' or 'no'. The way you give your response will depend on two factors:

- How important is it that you are understood correctly?
- How good is your connection?

Both of these can be parameterized with probabilities. For the first, we can use P_a , the maximum acceptable probability of being misunderstood. If you are being asked to confirm a preference for ice cream flavours, and don't mind too much if you get vanilla rather than chocolate, P_a might be quite high. If you are being asked a question on which someone's life depends, however, P_a will be much lower.

For the second we can use pp , the probability that your answer is garbled by a bad connection. For simplicity, let's imagine a case where a garbled 'yes' doesn't simply sound like nonsense, but sounds like a 'no'. And similarly a 'no' is transformed into 'yes'. Then pp is the probability that you are completely misunderstood.

If your connection is very good, or the answer is not very important, we will have $p < P_a$. The probability of being misunderstood in this case is so low that you don't care. The way you answer will then be simple: you just say 'yes' or 'no'.

If, however, your connection is poor and your answer is important, we will have $p > P_a$. A single 'yes' or 'no' is not enough in this case. The probability of being misunderstood would be too high.

So how do we overcome this problem? The answer is something you would probably do without thinking: repeat yourself. Instead of 'yes', say 'yes, yes, yes'. Instead of 'no', say 'no, no no'.

If the person you are talking to hears 'yes, yes, yes', they will of course conclude that you meant 'yes'. If they hear 'no, yes, yes', 'yes, no, yes' or 'yes, yes, no', they will probably conclude the same thing, since there is more positivity than negativity in the answer. To be misunderstood in this case, at least two of your replies need to be garbled. The probability for this, P_P , will be less than pp . So your message will be more likely to be understood.

```
p = 0.01
P = 3 * p**2 * (1-p) + p**3
print('Probability of a single reply being garbled:',p)
print('Probability of a the majority of three replies being garbled:',P)
```



```
Probability of a single reply being garbled: 0.01
Probability of a the majority of three replies being garbled: 0.00029800000000000003
```

If $P < P_a$, this technique solves our problem. If not, we can simply add more repetitions. The fact that $P < p$ above comes from the fact that we need at least two replies to be garbled to flip the majority, and so even the most likely possibilities have a probability of $\sim p^2 \sim p^2$. For five repetitions we'd need at least three replies to be garbled to flip the majority, which happens with probability $\sim p^3 \sim p^3$. The value for P_P in this case would then be even lower. Indeed, as we increase the number of repetitions, P_P will decrease exponentially. No matter how bad the connection, or how certain we need to be of our message getting through correctly, we can achieve it by just repeating our answer enough times.

Though this is a simple example, it contains all the aspects of error correction.

- There is some information to be sent or stored: In this case, a 'yes' or 'no'.
- The information is encoded in a larger system to protect it against noise: In this case, by repeating the message.
- The information is finally decoded, mitigating for the effects of noise: In this case, by trusting the majority of the transmitted messages.

It is using these steps that we correct errors in quantum computers. But first, let's implement this simple repetition encoding using qubits.

Correcting errors in qubits

Let's just try to store a simple `0` or a `1` in some qubits. Of course, we know that noise exists in our system. This can have the effect of causing us to readout a `1` for a qubit that should be `0`, and vice versa.

Here is some Qiskit code that provides a simple example of such noise. In this we go beyond the simple case of a single noise event which happens with a probability `pp`. Instead we consider two forms of error that can occur. One is a gate error: an imperfection in any operation we perform. We model this here in a simple way, using so-called depolarizing noise. The effect of this will be, with probability `p_gate`, to replace the state of any qubit with a completely random state. For two qubit gates, it is applied independently to each qubit.

The other form of noise is that for measurement. This simply flips between a `0` to a `1` and vice-versa immediately before measurement with probability `p_meas`.

```
from qiskit.providers.aer.noise import NoiseModel
from qiskit.providers.aer.noise.errors import pauli_error, depolarizing_error

def get_noise(p_meas,p_gate):

    error_meas = pauli_error([(‘X’,p_meas), (‘I’, 1 - p_meas)])
    error_gate1 = depolarizing_error(p_gate, 1)
    error_gate2 = error_gate1.tensor(error_gate1)

    noise_model = NoiseModel()
    noise_model.add_all_qubit_quantum_error(error_meas, “measure”) # measurement error
    noise_model.add_all_qubit_quantum_error(error_gate1, [“x”]) # single qubit gate err
    noise_model.add_all_qubit_quantum_error(error_gate2, [“cx”]) # two qubit gate error

    return noise_model
```

As an example, we'll now create such a noise model with a probability of 1% for each type of error.

```
noise_model = get_noise(0.01,0.01)
```

Let's see what affect this has when try to store a '0' using three qubits in state $|0\rangle|0\rangle$. We'll repeat the process `shots=10000` times to see how likely different results are.

```
qc0 = QuantumCircuit(3,3,name='0') # initialize circuit with three qubits in the 0 state
qc0.measure(qc0.qregs[0],qc0.cregs[0]) # measure the qubits

# run the circuit with th noise model and extract the counts
counts = execute( qc0, Aer.get_backend('qasm_simulator'),noise_model=noise_model,shots=10000)

print(counts)
```

```
{'010': 70, '100': 100, '011': 1, '000': 9731, '001': 98}
```

As you can see, almost all results still come out '000', as they would if there was no noise. Of the remaining possibilities, those with a majority of 0 s are most likely. In total, much less than 100 samples come out with a majority of 1 s, so $P < 1\%$.

Now let's try the same for storing a '1' using three qubits in state $|1\rangle|1\rangle$.

```
qc1 = QuantumCircuit(3,3,name='0') # initialize circuit with three qubits in the 0 state
qc1.x(qc1.qregs[0]) # flip each 0 to 1

qc1.measure(qc1.qregs[0],qc1.cregs[0]) # measure the qubits

# run the circuit with th noise model and extract the counts
counts = execute( qc1, Aer.get_backend('qasm_simulator'),noise_model=noise_model,shots=10000)

print(counts)
```

```
{'010': 3, '100': 5, '101': 168, '110': 150, '111': 9530, '011': 140, '001': 4}
```

The number of samples that come out with a majority in the wrong state (0 in this case) is again much less than 100, so $P < 1\%$. Whether we store a 0 or a 1, we can retrieve the information with a smaller probability of error than either of our sources of noise.

This was possible because the noise we considered was relatively weak. As we increase p_{meas} and p_{gate} , the higher the probability P will be. The extreme case of this is for either of them

to have a 50/50 50/50 chance of applying the bit flip error, x . For example, let's run the same circuit as before but with $p_{\text{meas}} = 0.5$ $p_{\text{meas}}=0.5$ and $p_{\text{gate}} = 0$ $p_{\text{gate}}=0$.

```
noise_model = get_noise(0.5,0.0)
counts = execute( qc1, Aer.get_backend('qasm_simulator'),noise_model=noise_model,shots=
print(counts)
```

```
{'010': 1215, '100': 1278, '101': 1304, '111': 1173, '011': 1255, '001': 1258, '110':
```

With this noise, all outcomes occur with equal probability. No trace of the encoded state remains. This is an important point to consider for error correction: sometimes the noise is too strong to be corrected. The optimal approach is to combine a good way of encoding the information you require, with hardware that whose noise is not too strong.

Storing qubits

So far, we have considered cases where there is no delay between encoding and decoding. For qubits, this means that there is no significant amount of time that passes between initializing the circuit, and making the final measurements.

However, there are many cases for which there will be a significant delay. As an obvious example, one may wish to encode a quantum state and store it for a long time, like a quantum hard drive. A less obvious but much more important example is performing fault-tolerant quantum computation itself. For this, we need to store quantum states and preserve their integrity during the computation. This must also be done in a way that allows us to manipulate the stored information in any way we need, and which corrects any errors we may introduce when performing the manipulations.

In all cases, we need account for the fact that errors do not only occur when something happens (like a gate or measurement), they also occur when the qubits are idle. Such noise is due to the fact that the qubits interact with each other and their environment. The longer we leave our qubits idle for, the greater the effects of this noise becomes. If we leave them for long enough, we'll encounter a situation like the $p_{\text{meas}} = 0.5$ $p_{\text{meas}}=0.5$ case above, where the noise is too strong for errors to be reliably corrected.

The solution is to keep measuring throughout. Don't leave your qubit idle for too long, but keep extracting information to keep track of the errors that have occurred.

For the case of classical information, where we simply wish to store a `0` or `1`, this can be done by just constantly measuring the value of each qubit. Then we will keep track of when the values change due to noise, and can easily deduce a history of when errors occurred.

For quantum information, however, it is not so easy. For example, consider the case that we wish to encode the state $|+\rangle |+\rangle$. Our encoding is such that

$$|0\rangle \rightarrow |000\rangle, \quad |1\rangle \rightarrow |111\rangle.$$

To encode the $|+\rangle |+\rangle$ state we therefore need

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \rightarrow \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle).$$

It will be convenient to now introduce some terminology. In the expression above, there are two different concepts that we can refer to as a 'qubit'. One is that of the 'physical qubit'. These are the actual physical qubits that exist on a chip. There are three physical qubits in the above expression. The other type of qubit is the 'logical' qubit. This is a qubit state, encoded across multiple physical qubits, with the hope that our physical qubit will experience less noise than the physical qubits they are made up of. In the above expression there is a single logical qubit, which is in the state $|+\rangle |+\rangle$ and encoded across three physical qubits.

With the repetition encoding that we are using, a z measurement of the logical qubit is done using a z measurement of each physical qubit. The final result for the logical measurement is decoded from the physical qubit measurement results by simply looking which output is in the majority.

As mentioned earlier, we can keep track of errors on logical qubits that are stored for a long time by constantly performing z measurements of the physical qubits. However, note that this effectively corresponds to constantly performing z measurements of the physical qubits. This is fine if we are simply storing a `0` or `1`, but it has undesired effects if we are storing a superposition. Specifically: the first time we do such a check for errors, we will collapse the superposition!

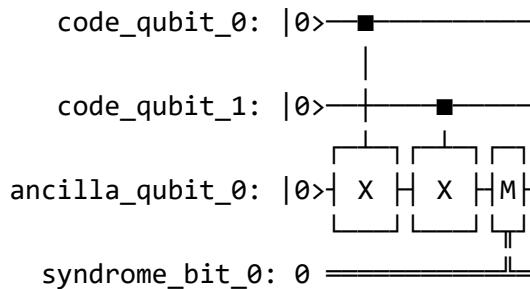
This is not ideal. If we wanted to do some computation on our logical qubit, or if we wish to perform a basis change before final measurement, we need to preserve the superposition. Destroying it is an error. But this is not an error caused by imperfections in our devices. It is an error that we have introduced as part of our attempts to correct errors. And since we cannot hope to recreate any arbitrary superposition stored in our quantum computer, it is an error that cannot be corrected.

For this reason, we must find another way of keeping track of the errors that occur when our logical qubit is stored for long times. This should give us the information we need to detect and correct errors, and to decode the final measurement result with high probability. However, it should not

cause uncorrectable errors to occur during the process by collapsing superpositions that we need to preserve.

The way to do this is with the following circuit element.

```
from qiskit import *
cq = QuantumRegister(2,'code_qubit')
lq = QuantumRegister(1,'ancilla_qubit')
sb = ClassicalRegister(1,'syndrome_bit')
qc = QuantumCircuit(cq,lq,sb)
qc.cx(cq[0],lq[0])
qc.cx(cq[1],lq[0])
qc.measure(lq,sb)
print(qc)
```



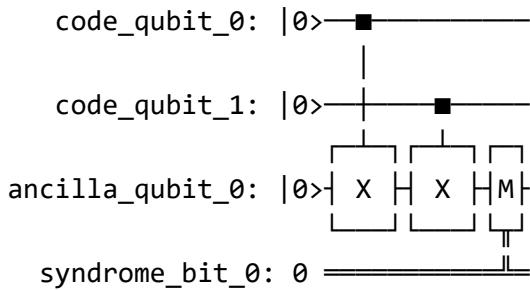
Here we have three physical qubits. Two are called 'code qubits', and the other is called an 'ancilla qubit'. One bit of output is extracted, called the syndrome bit.

The ancilla qubit will always be initialized in state $|0\rangle|0\rangle$. The code qubits, however, can be initialized in different states. Let's see what output we get for different initial states of the code qubits. We'll do this by creating a circuit `qc_init` that prepares the code qubits in some state, and then run the circuit `qc_init+qc`.

First, the trivial case: `qc_init` does nothing, and so the code qubits are initially $|00\rangle|00\rangle$.

```
qc_init = QuantumCircuit(cq)
print(qc_init+qc)

counts = execute( qc_init+qc, Aer.get_backend('qasm_simulator'), shots=10000).result().g
print('\nResults:',counts)
```



Results: {'0': 10000}

The outcome, in all cases, is '0'.

Now let's try an initial state of $|11\rangle|11\rangle$.

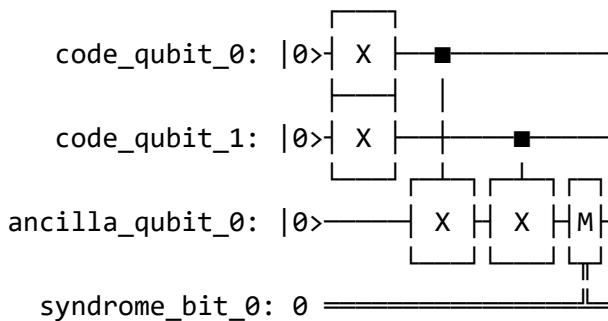
```

qc_init = QuantumCircuit(cq)
qc_init.x(cq)

print(qc_init+qc)

counts = execute( qc_init+qc, Aer.get_backend('qasm_simulator'), shots=10000).result().get_counts()
print('\nResults:', counts)

```



Results: {'0': 10000}

The outcome is also always '0'.

Now let's try a superposition of $|00\rangle|00\rangle$ and $|11\rangle|11\rangle$.

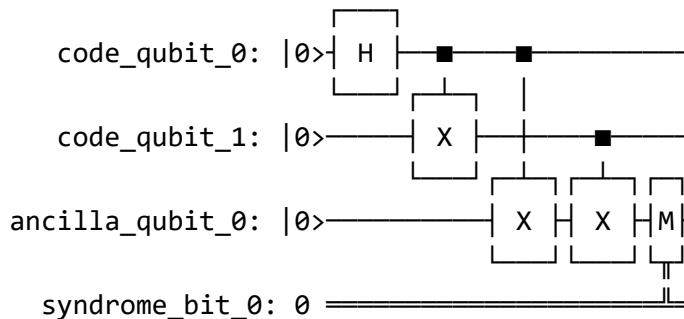
```

qc_init = QuantumCircuit(cq)
qc_init.h(cq[0])
qc_init.cx(cq[0],cq[1])

print(qc_init+qc)

counts = execute( qc_init+qc, Aer.get_backend('qasm_simulator'),shots=10000).result().get_counts()
print('\nResults:',counts)

```



Results: {'0': 10000}

Again, the outcome is always '0'.

Now let's try a superposition of $|01\rangle|01\rangle$ and $|10\rangle|10\rangle$ instead.

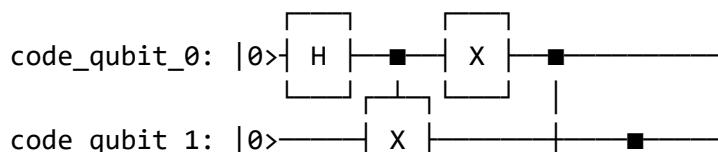
```

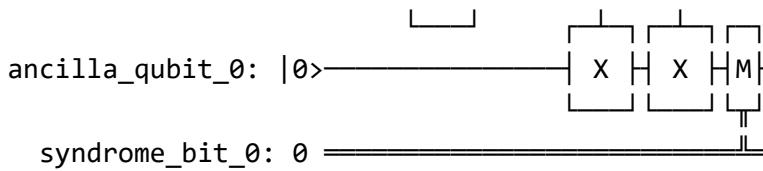
qc_init = QuantumCircuit(cq)
qc_init.h(cq[0])
qc_init.cx(cq[0],cq[1])
qc_init.x(cq[0])

print(qc_init+qc)

counts = execute( qc_init+qc, Aer.get_backend('qasm_simulator'),shots=10000).result().get_counts()
print('\nResults:',counts)

```





Results: {'1': 10000}

For this the output is always '1'. If you try the input $|01\rangle|01\rangle$ or $|10\rangle|10\rangle$ you will find the same.

This measurement is therefore telling us about a collective property of multiple qubits. Specifically, it looks at the two code qubits and determines whether their state is the same or different in the z basis. For basis states that are the same in the z basis, like $|00\rangle|00\rangle$ and $|11\rangle|11\rangle$, the measurement simply returns '0'. It also does so for any superposition of these. Since it does not distinguish between these states in any way, it also does not collapse such a superposition.

Similarly, For basis states that are different in the z basis it returns a '1'. This occurs for $|01\rangle|01\rangle$, $|10\rangle|10\rangle$ or any superposition thereof.

Now suppose we apply such a 'syndrome measurement' on all pairs of physical qubits in our repetition code. If their state is described by a repeated $|0\rangle|0\rangle$, a repeated $|1\rangle|1\rangle$, or any superposition thereof, all the syndrome measurements will return '0'. Given this result, we will know that our states are indeed encoded in the repeated states that we want them to be, and can deduce that no errors have occurred. If some syndrome measurements return '1', however, it is a signature of an error. We can therefore use these measurement results to determine how to decode the result.

Quantum repetition code

We now know enough to understand exactly how the quantum version of the repetition code is implemented

We can use it in Qiskit by importing the required tools from Ignis.

```
from qiskit.ignis.verification.topological_codes import RepetitionCode
from qiskit.ignis.verification.topological_codes import lookuptable_decoding
```



We are free to choose how many physical qubits we want the logical qubit to be encoded in. So far we have focussed on three repetitions, so let's continue with that.

```
n = 3
```



We can also choose how many times the syndrome measurements will be applied while we store our logical qubit, before the final readout measurement. Let's just go for one round.

```
T = 1
```



The circuits for our code can then be created automatically from the using a `RepetitionCode` object from Ignis.

```
code = RepetitionCode(n,T)
```



With this we can inspect various properties of the code, such as the names of the qubit registers used for the code and ancilla qubits.

```
code.qubit_registers
```



```
{'code_qubit', 'link_qubit'}
```

These registers are also attributes of the `repetition_code` object.

```
code.code_qubit
```



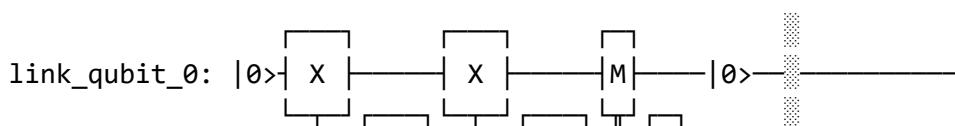
```
QuantumRegister(3, 'code_qubit')
```

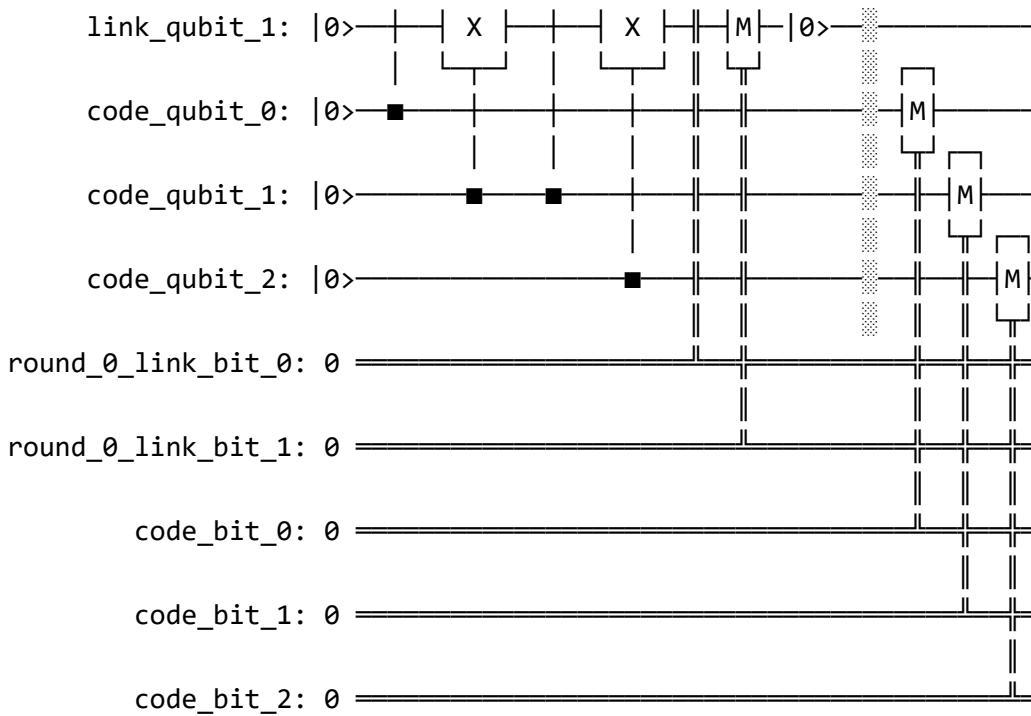
The `RepetitionCode` contains two quantum circuits that implement the code: One for each of the two possible logical bit values.

```
for log in ['0','1']:
    print('===== logical',log,'=====\n')
    print( code.circuit[log] )
```

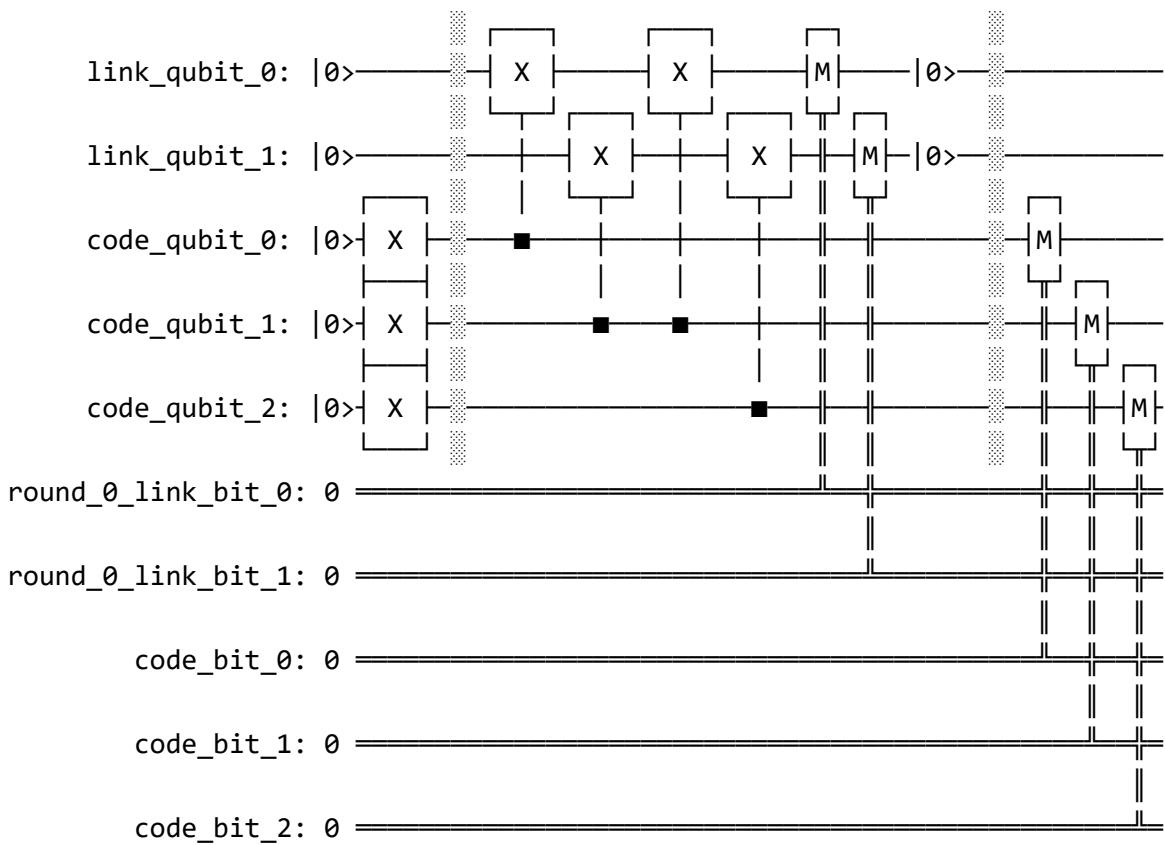


```
===== logical 0 =====
```





===== logical 1 =====



In these circuits, we have two types of physical qubits. There are the 'code qubits', which are the three physical qubits across which the logical state is encoded. There are also the 'link qubits', which serve as the ancilla qubits for the syndrome measurements.

Our single round of syndrome measurements in these circuits consist of just two syndrome measurements. One compares code qubits 0 and 1, and the other compares code qubits 1 and 2. One might expect that a further measurement, comparing code qubits 0 and 2, should be required to create a full set. However, these two are sufficient. This is because the information on whether 0 and 2 have the same z basis state can be inferred from the same information about 0 and 1 with that for 1 and 2. Indeed, for n qubits, we can get the required information from just $n - 1$ syndrome measurements of neighbouring pairs of qubits.

Let's now run these circuits without noise and see what happens.

```
circuits = code.get_circuit_list()
job = execute( circuits, Aer.get_backend('qasm_simulator') )
for log in ['0','1']:
    print('\nLogical',log,':',job.result().get_counts(log))
```

```
Logical 0 : {'000 00': 1024}
```

```
Logical 1 : {'111 00': 1024}
```

Here we see that the output comes in two parts. The part on the right holds the outcomes of the two syndrome measurements. That on the left holds the outcomes of the three final measurements of the code qubits.

For more measurement rounds, $T = 4$ for example, we would have the results of more syndrome measurements on the right.

```
code = RepetitionCode(n,4)

circuits = code.get_circuit_list()
job = execute( circuits, Aer.get_backend('qasm_simulator') )
for log in ['0','1']:
    print('\nLogical',log,':',job.result().get_counts(log))
```

```
Logical 0 : {'000 00 00 00 00': 1024}
```

```
Logical 1 : {'111 00 00 00 00': 1024}
```

For more repetition, $n = 5$ for example, each set of measurements would be larger. The final measurement on the left would be of n qubits. The T syndrome measurements would each be of the $n - 1$ possible neighbouring pairs.

```

code = RepetitionCode(5,4)

circuits = code.get_circuit_list()
job = execute( circuits, Aer.get_backend('qasm_simulator') )
for log in ['0','1']:
    print('\nLogical',log,':',job.result().get_counts(log))

```

Logical 0 : {'00000 0000 0000 0000 0000': 1024}

Logical 1 : {'11111 0000 0000 0000 0000': 1024}

```

code = RepetitionCode(n,4)

circuits = code.get_circuit_list()
job = execute( circuits, Aer.get_backend('qasm_simulator') )
for log in ['0','1']:
    print('\nLogical',log,':',job.result().get_counts(log))

```

Logical 0 : {'000 00 00 00 00': 1024}

Logical 1 : {'111 00 00 00 00': 1024}

Now let's return to the $n = 3$ $n=3$, $T = 1$ $T=1$ example and look at a case with some noise.

```

code = RepetitionCode(3,1)

noise_model = get_noise(0.2,0.2)

circuits = code.get_circuit_list()
job = execute( circuits, Aer.get_backend('qasm_simulator'), noise_model=noise_model )
raw_results = {}
for log in ['0','1']:
    raw_results[log] = job.result().get_counts(log)
    print('\n===== logical',log,'=====\n')
    print(raw_results[log])

```

===== logical 0 =====

{'000 10': 80, '001 00': 68, '010 01': 45, '110 01': 14, '111 00': 13, '111 11': 2, '(

===== logical 1 =====

```
{'000 10': 7, '001 00': 19, '010 01': 14, '110 01': 40, '111 00': 145, '111 11': 38,
```

Here we have created `raw_results`, a dictionary that holds both the results for a circuit encoding a logical `0` and `1` encoded for a logical `1`.

Our task when confronted with any of the possible outcomes we see here is to determine what the outcome should have been, if there was no noise. For an outcome of `'000 00'` or `'111 00'`, the answer is obvious. These are the results we just saw for a logical `0` and logical `1`, respectively, when no errors occur. The former is the most common outcome for the logical `0` even with noise, and the latter is the most common for the logical `1`. We will therefore conclude that the outcome was indeed that for logical `0` whenever we encounter `'000 00'`, and the same for logical `1` when we encounter `'111 00'`.

Though this tactic is optimal, it can nevertheless fail. Note that `'111 00'` typically occurs in a handful of cases for an encoded `0`, and `'00 00'` similarly occurs for an encoded `1`. In this case, through no fault of our own, we will incorrectly decode the output. In these cases, a large number of errors conspired to make it look like we had a noiseless case of the opposite logical value, and so correction becomes impossible.

We can employ a similar tactic to decode all other outcomes. The outcome `'001 00'`, for example, occurs far more for a logical `0` than a logical `1`. This is because it could be caused by just a single measurement error in the former case (which incorrectly reports a single `0` to be `1`), but would require at least two errors in the latter. So whenever we see `'001 00'`, we can decode it as a logical `0`.

Applying this tactic over all the strings is a form of so-called 'lookup table decoding'. This is where every possible outcome is analyzed, and the most likely value to decode it as is determined. For many qubits, this quickly becomes intractable, as the number of possible outcomes becomes so large. In these cases, more algorithmic decoders are needed. However, lookup table decoding works well for testing out small codes.

We can use tools in Qiskit to implement lookup table decoding for any code. For this we need two sets of results. One is the set of results that we actually want to decode, and for which we want to calculate the probability of incorrect decoding, P_{P} . We will use the `raw_results` we already have for this.

The other set of results is one to be used as the lookup table. This will need to be run for a large number of samples, to ensure that it gets good statistics for each possible outcome. We'll use

```
shots=10000 .
```

```
job = execute( circuits, Aer.get_backend('qasm_simulator'), noise_model=noise_model, sh  
table_results = {}  
for log in ['0','1']:  
    table_results[log] = job.result().get_counts(log)
```

With this data, which we call `table_results`, we can now use the `lookupable_decoding` function from Qiskit. This takes each outcome from `raw_results` and decodes it with the information in `table_results`. Then it checks if the decoding was correct, and uses this information to calculate P_P .

```
P = lookupable_decoding(raw_results,table_results)  
print('P =' ,P)
```

```
P = {'0': 0.201, '1': 0.2073}
```

Here we find P_P to be similar to p_{meas} and p_{gate} . The value of P_P for an encoded `1` is higher than that for `0`. This is because the encoding of `1` requires the application of `x` gates, which are an additional source of noise.

As always with quantum error correction, more impressive results can be found when the noise is not so strong. For example, $p_{\text{meas}} = p_{\text{gate}} = 0.1$ `pmeas=pgate=0.1`

```
code = RepetitionCode(3,1)  
  
noise_model = get_noise(0.1,0.1)  
  
circuits = code.get_circuit_list()  
  
job = execute( circuits, Aer.get_backend('qasm_simulator'), noise_model=noise_model )  
raw_results = {}  
for log in ['0','1']:  
    raw_results[log] = job.result().get_counts(log)  
  
job = execute( circuits, Aer.get_backend('qasm_simulator'), noise_model=noise_model, sh  
table_results = {}  
for log in ['0','1']:  
    table_results[log] = job.result().get_counts(log)
```

```
P = lookuptable_decoding(raw_results,table_results)
print('P =',P)
```

```
P = {'0': 0.0678, '1': 0.0733}
```

Here P is less than either p_{meas} or p_{gate} , and much less than the two combined. We therefore have a considerable improvement. If you try it with larger values of n and you'll see even better results. Though note that the value of `shots` used may need to be increased also.

Basic principles of quantum error correction

As mentioned at the start of this section, the repetition code is a simple example of the basic principles of quantum error correction. These are as follows.

1. The information we wish to store and process takes the form of 'logical qubits'. The states of these are encoded across many of the actual 'physical qubits' of a device.
2. Information about errors is extracted constantly through a process of 'syndrome' measurement. These consist of measurements that extract no information about the logical stored information. Instead they assess collective properties of groups of physical qubits, in order to determine when faults arise in the encoding of the logical qubits.
3. The information from syndrome measurements allows the effects of errors to be identified and mitigated for with high probability. This requires a decoding method.

There is another basic principle for which the repetition code is not such a good example.

1. Manipulating stored information must require action on multiple physical qubits. The minimum number required for any code is known as the distance of the code, d . Possible manipulations include performing an x operation on the logical qubit (flipping an encoded $|0\rangle|0\rangle$ to an encoded $|1\rangle|1\rangle$, and vice-versa), or performing a logical z measurement (distinguishing an encoded $|0\rangle|0\rangle$ from an encoded $|1\rangle|1\rangle$).

This makes it harder to perform operations on logical qubits when required: both for us, and for errors. The latter is, of course, the reason why this behaviour is required. If logical information could be accessed using only a single physical qubit, it would always be possible for single stray errors to disturb the logical qubit. The aim is usually to make it relatively straightforward for us to perform logical operations, given that we know how to do it, but hard for noise to achieve it by random chance.

In terms of making it hard for noise to perform a logical x , the repetition code cannot be beaten: All code qubits must be flipped to flip the logical value. From this perspective, $d = n$ $d=n$. For a z measurement, however, the repetition code is very poor. In the ideal case of no errors, the logical z basis information is repeated across every code qubit. Measuring any single code qubit is therefore sufficient to deduce the logical value. For this logical operation, and the overall distance, is therefore $d = 1$ $d=1$ for the repetition code. This is also reflected by the fact that the code is unable to detect and correct logical z errors.

For a better example of quantum error correction, we therefore need to find alternatives to the repetition approach. One of the foremost examples is the surface code, which will be added to this textbook as soon as it is implemented in Ignis.

Measurement Error Mitigation

```
from qiskit import *
```



Introduction

The effect of noise is to give us outputs that are not quite correct. The effect of noise that occurs throughout a computation will be quite complex in general, as one would have to consider how each gate transforms the effect of each error.

A simpler form of noise is that occurring during final measurement. At this point, the only job remaining in the circuit is to extract a bit string as an output. For an n qubit final measurement, this means extracting one of the 2^n possible n bit strings. As a simple model of the noise in this process, we can imagine that the measurement first selects one of these outputs in a perfect and noiseless manner, and then noise subsequently causes this perfect output to be randomly perturbed before it is returned to the user.

Given this model, it is very easy to determine exactly what the effects of measurement errors are. We can simply prepare each of the 2^n possible basis states, immediately measure them, and see what probability exists for each outcome.

As an example, we will first create a simple noise model, which randomly flips each bit in an output with probability p .

```
from qiskit.providers.aer.noise import NoiseModel
from qiskit.providers.aer.noise.errors import pauli_error, depolarizing_error

def get_noise(p):
    error_meas = pauli_error([(X,p), (I, 1 - p)])
    noise_model = NoiseModel()
    noise_model.add_all_qubit_quantum_error(error_meas, "measure") # measurement error
    return noise_model
```



Let's start with an instance of this in which each bit is flipped 1% of the time.

```
noise_model = get_noise(0.01)
```



Now we can test out its effects. Specifically, let's define a two qubit circuit and prepare the states $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$. Without noise, these would lead to the definite outputs '00', '01', '10' and '11', respectively. Let's see what happens with noise. Here, and in the rest of this section, the number of samples taken for each circuit will be `shots=10000`.

```
for state in ['00', '01', '10', '11']:
    qc = QuantumCircuit(2,2)
    if state[0]=='1':
        qc.x(1)
    if state[1]=='1':
        qc.x(0)
    qc.measure(qc.qregs[0],qc.cregs[0])
    print(state+' becomes',
          execute(qc,Aer.get_backend('qasm_simulator'),noise_model=noise_model,shots=10000).result().get_counts(qc))
```



```
00 becomes {'11': 2, '00': 9788, '10': 116, '01': 94}
01 becomes {'11': 92, '00': 102, '01': 9806}
10 becomes {'11': 102, '00': 96, '10': 9802}
11 becomes {'11': 9805, '10': 94, '01': 101}
```

Here we find that the correct output is certainly the most dominant. Ones that differ on only a single bit (such as '01', '10' in the case that the correct output is '00' or '11'), occur around 1% of the time. Those than differ on two bits occur only a handful of times in 10000 samples, if at all.

So what about if we ran a circuit with this same noise model, and got an result like the following?

```
{'10': 98, '11': 4884, '01': 111, '00': 4907}
```

Here '01' and '10' occur for around 1% of all samples. We know from our analysis of the basis states that such a result can be expected when these outcomes should in fact never occur, but instead the result should be something that differs from them by only one bit: '00' or '11'. When we look at the results for those two outcomes, we can see that they occur with roughly equal probability. We can therefore conclude that the initial state was not simply $|00\rangle$, or $|11\rangle$, but an

equal superposition of the two. If true, this means that the result should have been something along the lines of.

```
{'11': 4977, '00': 5023}
```

Here is a circuit that produces results like this (up to statistical fluctuations).

```
qc = QuantumCircuit(2,2)
qc.h(0)
qc.cx(0,1)
qc.measure(qc.qregs[0],qc.cregs[0])
print(execute(qc,Aer.get_backend('qasm_simulator')),noise_model=noise_model,shots=10000)
```

```
{'11': 4946, '00': 4877, '10': 101, '01': 76}
```

In this example we first looked at results for each of the definite basis states, and used these results to mitigate the effects of errors for a more general form of state. This is the basic principle behind measurement error mitigation.

Error mitigation in with linear algebra

Now we just need to find a way to perform the mitigation algorithmically rather than manually. We will do this by describing the random process using matrices. For this we need to rewrite our counts dictionaries as column vectors. For example, the dictionary `{'10': 96, '11': 1, '01': 95, '00': 9808}` describing would be rewritten as

$$C = \begin{pmatrix} 9808 \\ 95 \\ 96 \\ 1 \end{pmatrix}.$$

Here the first element is that for '`00`' , the next is that for '`01`' , and so on.

The information gathered from the basis states $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$ can then be used to define a matrix, which rotates from an ideal set of counts to one affected by measurement noise. This is done by simply taking the counts dictionary for $|00\rangle$, normalizing it so that all elements sum to one, and then using it as the first column of the matrix. The next column is similarly defined by the counts dictionary obtained for $|01\rangle$, and so on.

There will be statistical variations each time the circuit for each basis state is run. In the following, we will use the data obtained when this section was written, which was as follows.

```
00 becomes {'10': 96, '11': 1, '01': 95, '00': 9808}
01 becomes {'10': 2, '11': 103, '01': 9788, '00': 107}
10 becomes {'10': 9814, '11': 90, '01': 1, '00': 95}
11 becomes {'10': 87, '11': 9805, '01': 107, '00': 1}
```

This gives us the following matrix.

$$M = \begin{pmatrix} 0.9808 & 0.0107 & 0.0095 & 0.0001 \\ 0.0095 & 0.9788 & 0.0001 & 0.0107 \\ 0.0096 & 0.0002 & 0.9814 & 0.0087 \\ 0.0001 & 0.0103 & 0.0090 & 0.9805 \end{pmatrix}$$

If we now take the vector describing the perfect results for a given state, applying this matrix gives us a good approximation of the results when measurement noise is present.

$$C_{\text{noisy}} = M C_{\text{ideal}}$$

As an example, let's apply this process for the state $(|00\rangle + |11\rangle)/\sqrt{2}$,

$$\begin{pmatrix} 0.9808 & 0.0107 & 0.0095 & 0.0001 \\ 0.0095 & 0.9788 & 0.0001 & 0.0107 \\ 0.0096 & 0.0002 & 0.9814 & 0.0087 \\ 0.0001 & 0.0103 & 0.0090 & 0.9805 \end{pmatrix} \begin{pmatrix} 0 \\ 5000 \\ 5000 \\ 0 \end{pmatrix} = \begin{pmatrix} 101 \\ 4895.5 \\ 4908 \\ 96.5 \end{pmatrix}.$$

In code, we can express this as follows.

```
import numpy as np

M = [[0.9808, 0.0107, 0.0095, 0.0001],
      [0.0095, 0.9788, 0.0001, 0.0107],
      [0.0096, 0.0002, 0.9814, 0.0087],
      [0.0001, 0.0103, 0.0090, 0.9805]]

Cideal = [[0],
           [5000],
           [5000],
           [0]]
```

```
Cnoisy = np.dot( M, Cideal)
print('C_noisy =\n',Cnoisy)
```

```
C_noisy =
[[ 101. ]
[4894.5]
[4908. ]
[ 96.5]]
```

Either way, the resulting counts found in C_{noisy} , for measuring the $(|00\rangle + |11\rangle)/\sqrt{2}$ with measurement noise, come out quite close to the actual data we found earlier. So this matrix method is indeed a good way of predicting noisy results given a knowledge of what the results should be.

Unfortunately, this is the exact opposite of what we need. Instead of a way to transform ideal counts data into noisy data, we need a way to transform noisy data into ideal data. In linear algebra, we do this for a matrix M by finding the inverse matrix M^{-1} ,

$$C_{\text{ideal}} = M^{-1} C_{\text{noisy}}.$$

The trouble is, such an inverse does not always exist. To obtain an inverse, we need the columns of the original matrix to be orthogonal to each other. This is not the case for the matrix above. Indeed, it will not hold for any matrix describing measurement noise.

One possible method is to use the so-called 'pseudo inverse'. In Python, this can be applied using one of the linear algebra tools from Scipy.

```
import scipy.linalg as la

M = [[0.9808,0.0107,0.0095,0.0001],
      [0.0095,0.9788,0.0001,0.0107],
      [0.0096,0.0002,0.9814,0.0087],
      [0.0001,0.0103,0.0090,0.9805]]

Minv = la.pinv(M)

print(Minv)
```

```
[[ 1.01978044e+00 -1.11470783e-02 -9.87135367e-03  1.05228426e-04]
 [-9.89772783e-03  1.02188470e+00  9.39504466e-05 -1.11514471e-02]
```

```
[ -9.97422955e-03 -4.05845410e-06  1.01913199e+00 -9.04172099e-03]
[  9.15212840e-05 -1.07335657e-02 -9.35458279e-03  1.02008794e+00]]
```

```
/usr/local/lib/python3.7/site-packages/scipy/linalg/basic.py:1321: RuntimeWarning: in-
x, resids, rank, s = lstsq(a, b, cond=cond, check_finite=False)
```

Applying this inverse to C_{noisy} , we can obtain an approximation of the true counts.

```
Cmitigated = np.dot( Minv, Cnoisy)
print('C_mitigated =\n',Cmitigated)
```



```
C_mitigated =
[[ 1.35003120e-13]
 [ 5.00000000e+03]
 [ 5.00000000e+03]
 [-2.00373051e-12]]
```

Of course, counts should be integers, and so these values need to be rounded. This gives us a very nice result.

$$C_{\text{mitigated}} = \begin{pmatrix} 0 \\ 5000 \\ 5000 \\ 0 \end{pmatrix}$$

This is exactly the true result we desire. Our mitigation worked extremely well!

Error mitigation in Qiskit

```
from qiskit.ignis.mitigation.measurement import (complete_meas_cal,CompleteMeasFitter)
```



The process of measurement error mitigation can also be done using tools from Qiskit. This handles the collection of data for the basis states, the construction of the matrices and the calculation of the inverse. The latter can be done using the pseudo inverse, as we saw above. However, the default is an even more sophisticated method using least squares fitting.

As an example, let's stick with doing error mitigation for a pair of qubits. For this we define a two qubit quantum register, and feed it into the function `complete_meas_cal`.

```
qr = qiskit.QuantumRegister(2)
meas_calibs, state_labels = complete_meas_cal(qr=qr, circlabel='mcal')
```

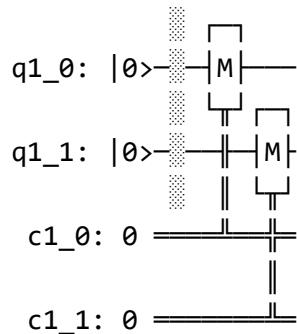


This creates a set of circuits to take measurements for each of the four basis states for two qubits: $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$.

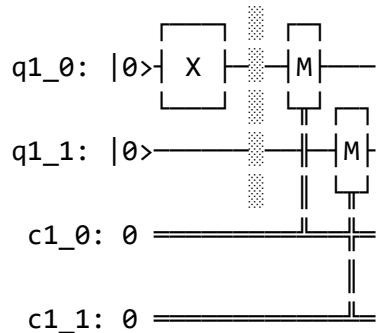
```
for circuit in meas_calibs:
    print('Circuit',circuit.name)
    print(circuit)
    print()
```



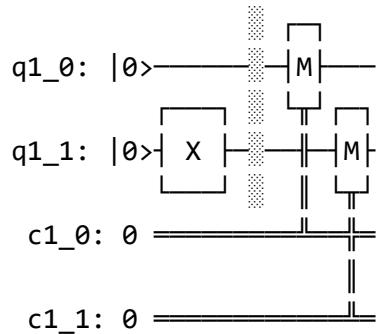
Circuit mcalcal_00



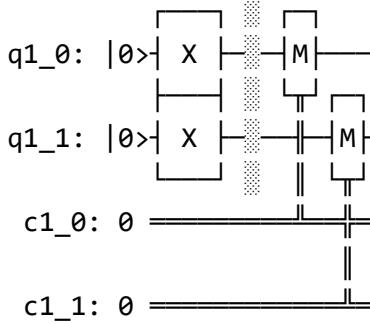
Circuit mcalcal_01



Circuit mcalcal_10



```
Circuit mcalcal_11
```



Let's now run these circuits without any noise present.

```
# Execute the calibration circuits without noise
backend = qiskit.Aer.get_backend('qasm_simulator')
job = qiskit.execute(meas_calibs, backend=backend, shots=1000)
cal_results = job.result()
```

With the results we can construct the calibration matrix, which we have been calling M .

```
meas_fitter = CompleteMeasFitter(cal_results, state_labels, circlabel='mcal')
print(meas_fitter.cal_matrix)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

With no noise present, this is simply the identity matrix.

Now let's create a noise model. And to make things interesting, let's have the errors be ten times more likely than before.

```
noise_model = get_noise(0.1)
```

Again we can run the circuits, and look at the calibration matrix, M .

```
backend = qiskit.Aer.get_backend('qasm_simulator')
job = qiskit.execute(meas_calibs, backend=backend, shots=1000, noise_model=noise_model)
cal_results = job.result()
```

```
meas_fitter = CompleteMeasFitter(cal_results, state_labels, circlabel='mcal')
print(meas_fitter.cal_matrix)
```

```
[[0.812 0.107 0.092 0.008]
 [0.097 0.81  0.01  0.097]
 [0.078 0.01  0.81  0.101]
 [0.013 0.073 0.088 0.794]]
```

This time we find a more interesting matrix, and one that is not invertible. Let's see how well we can mitigate for this noise. Again, let's use the Bell state $(|00\rangle + |11\rangle)/\sqrt{2}$ for our test.

```
qc = QuantumCircuit(2,2)
qc.h(0)
qc.cx(0,1)
qc.measure(qc.qregs[0],qc.cregs[0])

results = qiskit.execute(qc, backend=backend, shots=10000, noise_model=noise_model).res

noisy_counts = results.get_counts()
print(noisy_counts)
```

```
{'11': 4086, '00': 4085, '10': 908, '01': 921}
```

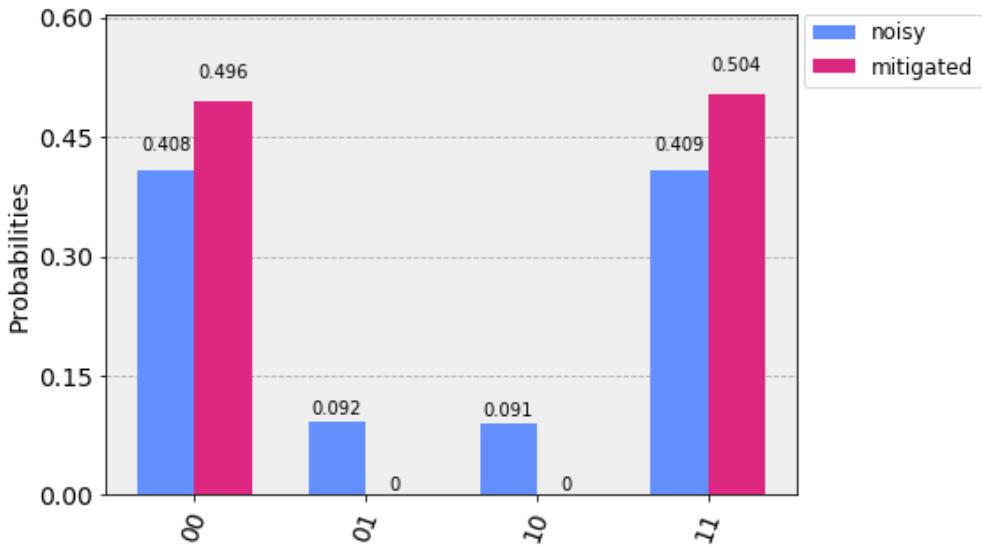
In Qiskit we mitigate for the noise by creating a measurement filter object. Then, taking the results from above, we use this to calculate a mitigated set of counts. Qiskit returns this as a dictionary, so that the user doesn't need to use vectors themselves to get the result.

```
# Get the filter object
meas_filter = meas_fitter.filter

# Results with mitigation
mitigated_results = meas_filter.apply(results)
mitigated_counts = mitigated_results.get_counts(0)
```

To see the results most clearly, let's plot both the noisy and mitigated results.

```
from qiskit.tools.visualization import *
plot_histogram([noisy_counts, mitigated_counts], legend=['noisy', 'mitigated'])
```



Here we have taken results for which almost 20% of samples are in the wrong state, and turned it into an exact representation of what the true results should be. However, this example does have just two qubits with a simple noise model. For more qubits, and more complex noise models or data from real devices, the mitigation will have more of a challenge. Perhaps you might find methods that are better than those Qiskit uses!

Randomized Benchmarking

Introduction

One of the main challenges in building a quantum information processor is the non-scalability of completely characterizing the noise affecting a quantum system via process tomography. In addition, process tomography is sensitive to noise in the pre- and post rotation gates plus the measurements (SPAM errors). Gateset tomography can take these errors into account, but the scaling is even worse. A complete characterization of the noise is useful because it allows for the determination of good error-correction schemes, and thus the possibility of reliable transmission of quantum information.

Since complete process tomography is infeasible for large systems, there is growing interest in scalable methods for partially characterizing the noise affecting a quantum system. A scalable (in the number n of qubits comprising the system) and robust algorithm for benchmarking the full set of Clifford gates^[1] by a single parameter using randomization techniques was presented in [1]. The concept of using randomization methods for benchmarking quantum gates is commonly called **Randomized Benchmarking (RB)**.

References

1. Easwar Magesan, J. M. Gambetta, and Joseph Emerson, *Robust randomized benchmarking of quantum processes*, <https://arxiv.org/pdf/1009.3639>
2. Easwar Magesan, Jay M. Gambetta, and Joseph Emerson, *Characterizing Quantum Gates via Randomized Benchmarking*, <https://arxiv.org/pdf/1109.6887>
3. A. D. C'orcoles, Jay M. Gambetta, Jerry M. Chow, John A. Smolin, Matthew Ware, J. D. Strand, B. L. T. Plourde, and M. Steffen, *Process verification of two-qubit quantum gates by randomized benchmarking*, <https://arxiv.org/pdf/1210.7011>
4. Jay M. Gambetta, A. D. C'orcoles, S. T. Merkel, B. R. Johnson, John A. Smolin, Jerry M. Chow, Colm A. Ryan, Chad Rigetti, S. Poletto, Thomas A. Ohki, Mark B. Ketchen, and M. Steffen, *Characterization of addressability by simultaneous randomized benchmarking*, <https://arxiv.org/pdf/1204.6308>

The Randomized Benchmarking Protocol

A RB protocol (see [1,2]) consists of the following steps:

(We should first import the relevant qiskit classes for the demonstration).

```
#Import general Libraries (needed for functions)
import numpy as np
import matplotlib.pyplot as plt
from IPython import display

#Import the RB Functions
import qiskit.ignis.verification.randomized_benchmarking as rb

#Import Qiskit classes
import qiskit
from qiskit.providers.aer.noise import NoiseModel
from qiskit.providers.aer.noise.errors.standard_errors import depolarizing_error, therm
```



Step 1: Generate RB sequences

The RB sequences consist of random Clifford elements chosen uniformly from the Clifford group on n-qubits, including a computed reversal element, that should return the qubits to the initial state.

n

More precisely, for each length m, we choose K_m RB sequences. Each such sequence contains m random elements C_{i_j} chosen uniformly from the Clifford group on n-qubits, and the $m + 1$ element $C_{i_{m+1}}$ is defined as follows: $C_{i_{m+1}} = (C_{i_1} \cdot \dots \cdot C_{i_m})^{-1}$. It can be found efficiently by the Gottesmann-Knill theorem.

For example, we generate below several sequences of 2-qubit Clifford circuits.

```
#Generate RB circuits (2Q RB)
#number of qubits
```



```

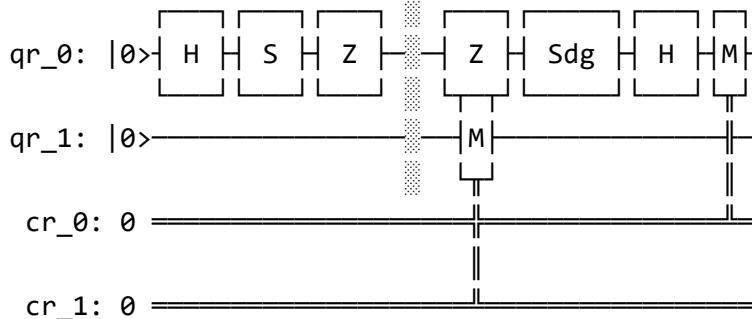
nQ=2
rb_opts = {}
#Number of Cliffords in the sequence
rb_opts['length_vector'] = [1, 10, 20, 50, 75, 100, 125, 150, 175, 200]
#Number of seeds (random sequences)
rb_opts['nseeds'] = 5
#Default pattern
rb_opts['rb_pattern'] = [[0,1]]


rb_circs, xdata = rb.randomized_benchmarking_seq(**rb_opts)

```

As an example, we print the circuit corresponding to the first RB sequence

```
print(rb_circs[0][0])
```



One can verify that the Unitary representing each RB circuit should be the identity (with a global phase). We simulate this using Aer unitary simulator.

```

# Create a new circuit without the measurement
qregs = rb_circs[0][-1].qregs
cregs = rb_circs[0][-1].cregs
qc = qiskit.QuantumCircuit(*qregs, *cregs)
for i in rb_circs[0][-1][0:-nQ]:
    qc.data.append(i)

```



```

# The Unitary is an identity (with a global phase)
backend = qiskit.Aer.get_backend('unitary_simulator')
basis_gates = ['u1', 'u2', 'u3', 'cx'] # use U,CX for now
job = qiskit.execute(qc, backend=backend, basis_gates=basis_gates)
print(np.around(job.result().get_unitary(),3))

```



```
[[ -1.+0.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j -1.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j -1.+0.j  0.+0.j]
 [ 0.+0.j  0.+0.j  0.+0.j -1.+0.j]]
```

Step 2: Execute the RB sequences (with some noise)

We can execute the RB sequences either using Qiskit Aer Simulator (with some noise model) or using IBMQ provider, and obtain a list of results.

By assumption each operation C_{i_j} is allowed to have some error, represented by $\Lambda_{i_j,j}$, and each sequence can be modeled by the operation: $S_{i_m} = \bigcirc_{m+1, j=1} (\Lambda_{i_j,j} \circ C_{i_j})$
where $i_m = (i_1, \dots, i_m)$ and i_{m+1} is uniquely determined by i_m .

```
# Run on a noisy simulator
noise_model = NoiseModel()
# Depolarizing_error
dp = 0.005
noise_model.add_all_qubit_quantum_error(depolarizing_error(dp, 1), ['u1', 'u2', 'u3'])
noise_model.add_all_qubit_quantum_error(depolarizing_error(2*dp, 2), 'cx')

backend = qiskit.Aer.get_backend('qasm_simulator')
```

Step 3: Get statistics about the survival probabilities

For each of the K_m sequences the survival probability $\text{Tr}[E_\Psi S_{i_m}(\rho_\Psi)]$ is measured. Here ρ_Ψ is the initial state taking into account preparation errors and E_Ψ is the POVM element that takes into account measurement errors. In the ideal (noise-free) case $\rho_\Psi = E_\Psi = |\Psi\rangle\langle\Psi|$.

In practice one can measure the probability to go back to the exact initial state, i.e. all the qubits in the ground state $|00\dots0\rangle$ or just the probability for one of the qubits to return back to the ground state. Measuring the qubits independently can be more convenient if a correlated measurement scheme is not possible. Both measurements will fit to the same decay parameter according to the properties of the *twirl*.

Step 4: Find the averaged sequence fidelity

Average over the K_m random realizations of the sequence to find the averaged sequence **fidelity**,

$$F_{\text{seq}}(m, |\Psi\rangle) = \text{Tr}[E_\Psi S_{K_m}(\rho_\Psi)]$$

$$\text{where } S_{K_m} = \frac{1}{K_m} \sum_i i_m S_{i_m}$$

is the average sequence operation.

Step 5: Fit the results

Repeat Steps 1 through 4 for different values of m and fit the results for the averaged sequence

$$fidelity_{(0)\text{seq}}(m, |\Psi\rangle) = A_0 \alpha^m + B_0$$

where A_0 and B_0 absorb state preparation and measurement errors as well as an edge effect from the error on the final gate.

α determines the average error-rate r , which is also called **Error per Clifford (EPC)** according to the relation $r = 1 - \alpha = 1 - \alpha 2^n = 2^n - 12^n(1 - \alpha)$ (where $n = nQ$ is the number of qubits).

As an example, we calculate the average sequence fidelity for each of the RB sequences, fit the results to the exponential curve, and compute the parameters α and EPC.

```
# Create the RB fitter
backend = qiskit.Aer.get_backend('qasm_simulator')
basis_gates = ['u1', 'u2', 'u3', 'cx']
shots = 200
qobj_list = []
rb_fit = rb.RBFitter(None, xdata, rb_opts['rb_pattern'])
for rb_seed, rb_circ_seed in enumerate(rb_circs):
    print('Compiling seed %d' % rb_seed)
    new_rb_circ_seed = qiskit.compiler.transpile(rb_circ_seed, basis_gates=basis_gates)
    qobj = qiskit.compiler.assemble(new_rb_circ_seed, shots=shots)
    print('Simulating seed %d' % rb_seed)
    job = backend.run(qobj, noise_model=noise_model, backend_options={'max_parallel_experiments': 1})
    qobj_list.append(job)
# Add data to the fitter
rb_fit.add_data(job.result())
print('After seed %d, alpha: %f, EPC: %f' % (rb_seed, rb_fit.fit[0]['params'][1], rb_fit.eperclifford))
```

```
Compiling seed 0
Simulating seed 0
After seed 0, alpha: 0.970829, EPC: 0.021879
Compiling seed 1
Simulating seed 1
After seed 1, alpha: 0.970255, EPC: 0.022309
Compiling seed 2
Simulating seed 2
After seed 2, alpha: 0.972152, EPC: 0.020886
Compiling seed 3
Simulating seed 3
After seed 3, alpha: 0.971834, EPC: 0.021124
Compiling seed 4
Simulating seed 4
After seed 4, alpha: 0.971461, EPC: 0.021404
```

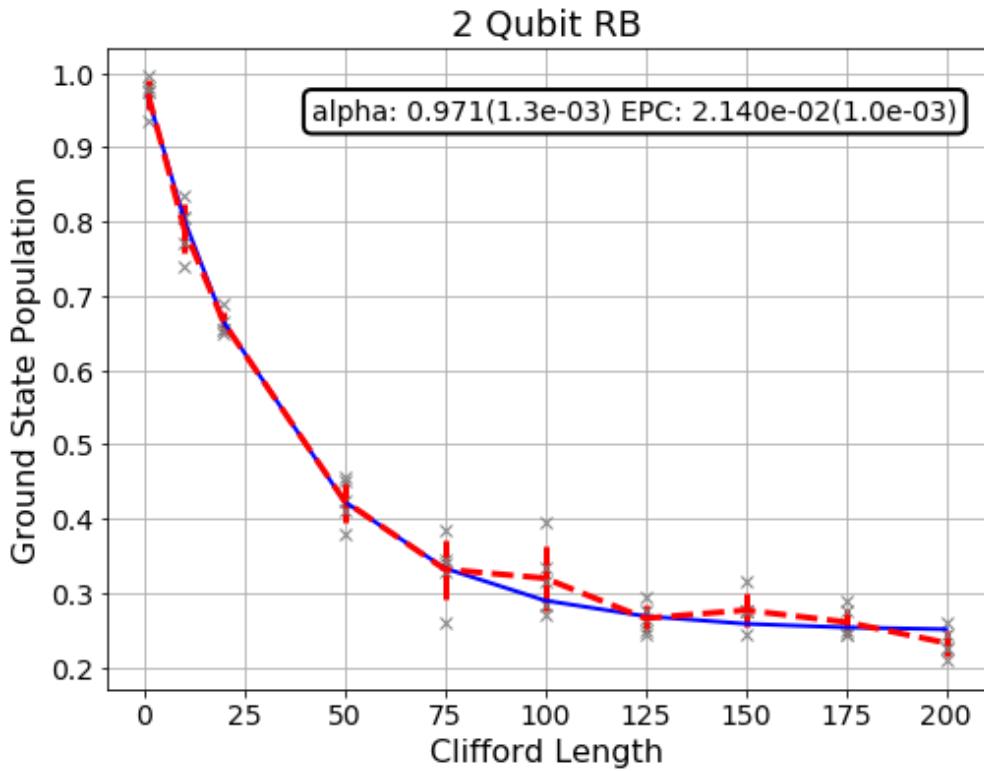
Plot the results

```
plt.figure(figsize=(8, 6))
ax = plt.subplot(1, 1, 1)

# Plot the essence by calling plot_rb_data
rb_fit.plot_rb_data(0, ax=ax, add_label=True, show_plt=False)

# Add title and label
ax.set_title('%d Qubit RB'%(nQ), fontsize=18)

plt.show()
```



The intuition behind RB

The depolarizing quantum channel has a parameter α , and works like this: with probability α , the state remains the same as before; with probability $1 - \alpha$, the state becomes the totally mixed state, namely:

$$\rho_f = \alpha \rho_i + 1 - \alpha 2^n * I$$

Suppose that we have a sequence of m gates, not necessarily Clifford gates, where the error channel of the gates is a depolarizing channel with parameter α (same α for all the gates). Then with probability α^m the state is correct at the end of the sequence, and with probability $1 - \alpha^m$ it becomes the totally mixed state, therefore:

$$\rho_{mf} = \alpha^m \rho_i + 1 - \alpha^m 2^n * I$$

Now suppose that in addition we start with the ground state; that the entire sequence amounts to the identity; and that we measure the state at the end of the sequence with the standard basis. We derive that the probability of success at the end of the sequence is:

$$\alpha^m + 1 - \alpha^m 2^n = 2^n - 12^n \alpha^m + 12^n = A_0 \alpha^m + B_0$$

It follows that the probability of success, aka fidelity, decays exponentially with the sequence length, with exponent α .

The last statement is not necessarily true when the channel is other than the depolarizing channel. However, it turns out that if the gates are uniformly-randomized Clifford gates, then the noise of each gate behaves on average as if it was the depolarizing channel, with some parameter that can be computed from the channel, and we obtain the exponential decay of the fidelity.

Formally, taking an average over a finite group G (like the Clifford group) of a quantum channel Λ is also called a *twirl*: $W_G(\Lambda) = \frac{1}{|G|} \sum_{U \in G} U^\dagger \circ \Lambda \circ U$

Twirling over the entire unitary group yields exactly the same result as the Clifford group. The Clifford group is a *2-design* of the unitary group.

Simultaneous Randomized Benchmarking

RB is designed to address fidelities in multiqubit systems in two ways. For one, RB over the full n -qubit space can be performed by constructing sequences from the n -qubit Clifford group. Additionally, the n -qubit space can be subdivided into sets of qubits $\{n_i\}$ and n_i -qubit RB performed in each subset simultaneously [4]. Both methods give metrics of fidelity in the n -qubit space.

For example, it is common to perform 2Q RB on the subset of two-qubits defining a CNOT gate while the other qubits are quiescent. As explained in [4], this RB data will not necessarily decay exponentially because the other qubit subspaces are not twirled. Subsets are more rigorously characterized by simultaneous RB, which also measures some level of crosstalk error since all qubits are active.

An example of simultaneous RB (1Q RB and 2Q RB) can be found in:

https://github.com/Qiskit/qiskit-tutorials/blob/master/qiskit/ignis/randomized_benchmarking.ipynb

Predicted Gate Fidelity

If we know the errors on the underlying gates (the gateset) we can predict the fidelity. First we need to count the number of these gates per Clifford.

Then, the two qubit Clifford gate error function gives the error per 2Q Clifford. It assumes that the error in the underlying gates is depolarizing. This function is derived in the supplement to [5].

```
#Count the number of single and 2Q gates in the 2Q Cliffords
gates_per_cliff = rb.rb_utils.gates_per_clifford(qobj_list, xdata[0], basis_gates, rb_op
```

```
for i in range(len(basis_gates)):
    print("Number of %s gates per Clifford: %f"%(basis_gates[i],
                                                np.mean([gates_per_cliff[0][i],gates_p
```

Number of u1 gates per Clifford: 0.266266
Number of u2 gates per Clifford: 0.934934
Number of u3 gates per Clifford: 0.487336
Number of cx gates per Clifford: 1.519432

```
# Prepare lists of the number of qubits and the errors
ngates = np.zeros(7)
ngates[0:3] = gates_per_cliff[0][0:3]
ngates[3:6] = gates_per_cliff[1][0:3]
ngates[6] = gates_per_cliff[0][3]
gate_qubits = np.array([0, 0, 0, 1, 1, 1, -1], dtype=int)
gate_errs = np.zeros(len(gate_qubits))
gate_errs[[1, 4]] = dp/2 #convert from depolarizing error to epg (1Q)
gate_errs[[2, 5]] = 2*dp/2 #convert from depolarizing error to epg (1Q)
gate_errs[6] = dp*3/4 #convert from depolarizing error to epg (2Q)

#Calculate the predicted epc
pred_epc = rb.rb_utils.twoQ_clifford_error(ngates,gate_qubits,gate_errs)
print("Predicted 2Q Error per Clifford: %e"%pred_epc)
```

Predicted 2Q Error per Clifford: 1.700920e-02

Measuring Quantum Volume

Introduction

Quantum Volume (QV) is a single-number metric that can be measured using a concrete protocol on near-term quantum computers of modest size. The QV method quantifies the largest random circuit of equal width and depth that the computer successfully implements. Quantum computing systems with high-fidelity operations, high connectivity, large calibrated gate sets, and circuit rewriting toolchains are expected to have higher quantum volumes.

References

[1] Andrew W. Cross, Lev S. Bishop, Sarah Sheldon, Paul D. Nation, and Jay M. Gambetta, *Validating quantum computers using randomized model circuits*, <https://arxiv.org/pdf/1811.12926>

The Quantum Volume Protocol

A QV protocol (see [1]) consists of the following steps:

(We should first import the relevant qiskit classes for the demonstration).

```
#Import general Libraries (needed for functions)
import numpy as np
import matplotlib.pyplot as plt

#Import Qiskit classes classes
import qiskit
from qiskit.providers.aer.noise import NoiseModel
from qiskit.providers.aer.noise.errors.standard_errors import depolarizing_error, therm

#Import the qv function.
import qiskit.ignis.verification.quantum_volume as qv
```

Step 1: Generate QV sequences

It is well-known that quantum algorithms can be expressed as polynomial-sized quantum circuits built from two-qubit unitary gates. Therefore, a model circuit consists of d layers of random permutations of the qubit labels, followed by random two-qubit gates (from $SU(4)$). When the circuit width m is odd, one of the qubits is idle in each layer.

More precisely, a **QV circuit** with **depth d** and **width m**, is a sequence $U = U^{(d)} \dots U^{(2)} U^{(1)}$ of d layers:

$$U^{(t)} = U_{\pi_t(m'-1), \pi_t(m)}^{(t)} \otimes \dots \otimes U_{\pi_t(1), \pi_t(2)}^{(t)}$$

each labeled by times $t = 1 \dots d$ and acting on $m' = 2\lfloor n/2 \rfloor$ qubits. Each layer is specified by choosing a uniformly random permutation $\pi_t \in S_m$ of the m qubit indices and sampling each $U_{a,b}^{(t)}$, acting on qubits a and b , from the Haar measure on $SU(4)$.

In the following example we have 6 qubits Q0,Q1,Q3,Q5,Q7,Q10. We are going to look at subsets up to the full set (each volume circuit will be depth equal to the number of qubits in the subset)

```
# qubit_lists: List of list of qubit subsets to generate QV circuits
qubit_lists = [[0,1,3],[0,1,3,5],[0,1,3,5,7],[0,1,3,5,7,10]]
# ntrials: Number of random circuits to create for each subset
ntrials = 50
```

We generate the quantum volume sequences. We start with a small example (so it doesn't take too long to run).

```
qv_circs, qv_circs_nameas = qv.qv_circuits(qubit_lists, ntrials)
```

As an example, we print the circuit corresponding to the first QV sequence. Note that the ideal circuits are run on the first n qubits (where n is the number of qubits in the subset).

```
#pass the first trial of the nameas through the transpiler to illustrate the circuit
qv_circs_nameas[0] = qiskit.compiler.transpile(qv_circs_nameas[0], basis_gates=['u1', 'u2', 'u3'])
```

```
print(qv_circs_nameas[0][0])
```



```

qr_2: |0> U3(1.9895,0.0074621,5.0423) X U3(1.5708,0,-5.713) »
cr_0: 0 »
cr_1: 0 »
cr_2: 0 »
« »
«qr_0: »
« »
«qr_1: ─■ U3(0.33219,-1.7764e-15,4.7124) ─■ »
« »
«qr_2: ┌ X ─ U3(1.5708,-3.1416,-1.5708) ─ X ─ »
« »
«cr_0: »
« »
«cr_1: »
« »
«cr_2: »
« »
« »
«qr_0: »
« »
«qr_1: ┌ U3(1.9521,5.8535,0.045854) ┌ U3(1.1882,3.6057,-0.89911) ─■ »
« »
«qr_2: ┌ U3(0.51454,2.5314,-2.1542) ┌ U3(0.86532,0.8319,2.0513) ─ X ─ »
« »
«cr_0: »
« »
«cr_1: »
« »
«cr_2: »
« »
« »
«qr_0: »
« »
«qr_1: ┌ U3(0.94961,1.5708,4.7124) ─■ ┌ U3(0.0076131,3.1416,1.5708) »
« »
«qr_2: ┌ U3(1.5708,-4.4409e-16,-5.938) ┌ X ┌ U3(1.5708,-3.1416,-1.5708) »
« »
«cr_0: »
« »
«cr_1: »
« »
«cr_2: »

```

Step 2: Simulate the ideal QV circuits

The quantum volume method requires that we know the ideal output for each circuit, so we use the statevector simulator in Aer to get the ideal result.

```
#The Unitary is an identity (with a global phase)
backend = qiskit.Aer.get_backend('statevector_simulator')
ideal_results = []
for trial in range(ntrials):
    print('Simulating trial %d'%trial)
    ideal_results.append(qiskit.execute(qv_circs_nameas[trial], backend=backend).result)
```



```
Simulating trial 0
Simulating trial 1
Simulating trial 2
Simulating trial 3
Simulating trial 4
Simulating trial 5
Simulating trial 6
Simulating trial 7
Simulating trial 8
Simulating trial 9
Simulating trial 10
Simulating trial 11
Simulating trial 12
Simulating trial 13
Simulating trial 14
Simulating trial 15
Simulating trial 16
Simulating trial 17
Simulating trial 18
Simulating trial 19
Simulating trial 20
Simulating trial 21
Simulating trial 22
Simulating trial 23
Simulating trial 24
Simulating trial 25
Simulating trial 26
Simulating trial 27
Simulating trial 28
Simulating trial 29
Simulating trial 30
Simulating trial 31
Simulating trial 32
Simulating trial 33
Simulating trial 34
Simulating trial 35
Simulating trial 36
```

```
Simulating trial 37
Simulating trial 38
Simulating trial 39
Simulating trial 40
Simulating trial 41
Simulating trial 42
Simulating trial 43
Simulating trial 44
Simulating trial 45
Simulating trial 46
Simulating trial 47
Simulating trial 48
Simulating trial 49
```

Next, we load the ideal results into a quantum volume fitter

```
qv_fitter = qv.QVFitter(qubit_lists=qubit_lists)
qv_fitter.add_statevectors(ideal_results)
```



Step 3: Calculate the heavy outputs

To define when a model circuit U has been successfully implemented in practice, we use the *heavy output* generation problem. The ideal output distribution is $p_U(x) = |\langle x|U|0\rangle|^2$, where $x \in \{0, 1\}^m$ is an observable bit-string.

Consider the set of output probabilities given by the range of $p_U(x)$ sorted in ascending order $p_0 \leq p_1 \leq \dots \leq p_{2^m-1}$. The median of the set of probabilities is $p_{\text{med}} = (p_{2^{m-1}} + p_{2^{m-1}-1})/2$, and the *heavy outputs* are

$$H_U = \{x \in \{0, 1\}^m \text{ such that } p_U(x) > p_{\text{med}}\}.$$

The heavy output generation problem is to produce a set of output strings such that more than two-thirds are heavy.

As an illustration, we print the heavy outputs from various depths and their probabilities (for trial 0):

```
for qubit_list in qubit_lists:
    l = len(qubit_list)
    print ('qv_depth_'+str(l)+'_trial_0:', qv_fitter._heavy_outputs['qv_depth_'+str(l)+
```



```
qv_depth_3_trial_0: ['000', '001', '100', '111']
qv_depth_4_trial_0: ['0000', '0100', '0101', '1000', '1001', '1011', '1100', '1101']
```

```
qv_depth_5_trial_0: ['00000', '00011', '00100', '00101', '00111', '01001', '01011', '01100']
qv_depth_6_trial_0: ['000011', '000110', '000111', '001000', '001100', '001110', '001111']
```

```
for qubit_list in qubit_lists:
    l = len(qubit_list)
    print ('qv_depth_'+str(l) + '_trial_0:', qv_fitter._heavy_output_prob_ideal['qv_depth_
```

```
qv_depth_3_trial_0: 0.8364265965659454
qv_depth_4_trial_0: 0.8954287202366203
qv_depth_5_trial_0: 0.8721440467960914
qv_depth_6_trial_0: 0.8499058961161158
```

Step 4: Define the noise model

We define a noise model for the simulator. To simulate decay, we add depolarizing error probabilities to the CNOT and U gates.

```
noise_model = NoiseModel()
p1Q = 0.002
p2Q = 0.02
noise_model.add_all_qubit_quantum_error(depolarizing_error(p1Q, 1), 'u2')
noise_model.add_all_qubit_quantum_error(depolarizing_error(2*p1Q, 1), 'u3')
noise_model.add_all_qubit_quantum_error(depolarizing_error(p2Q, 2), 'cx')
#noise_model = None
```

We can execute the QV sequences either using Qiskit Aer Simulator (with some noise model) or using IBMQ provider, and obtain a list of exp_results.

```
backend = qiskit.Aer.get_backend('qasm_simulator')
basis_gates = ['u1', 'u2', 'u3', 'cx'] # use U,CX for now
shots = 1024
exp_results = []
for trial in range(ntrials):
    print('Running trial %d'%trial)
    exp_results.append(qiskit.execute(qv_circs[trial], basis_gates=basis_gates, backend
```

```
Running trial 0
Running trial 1
```

Running trial 2
Running trial 3
Running trial 4
Running trial 5
Running trial 6
Running trial 7
Running trial 8
Running trial 9
Running trial 10
Running trial 11
Running trial 12
Running trial 13
Running trial 14
Running trial 15
Running trial 16
Running trial 17
Running trial 18
Running trial 19
Running trial 20
Running trial 21
Running trial 22
Running trial 23
Running trial 24
Running trial 25
Running trial 26
Running trial 27
Running trial 28
Running trial 29
Running trial 30
Running trial 31
Running trial 32
Running trial 33
Running trial 34
Running trial 35
Running trial 36
Running trial 37
Running trial 38
Running trial 39
Running trial 40
Running trial 41
Running trial 42
Running trial 43
Running trial 44
Running trial 45
Running trial 46
Running trial 47

Running trial 48

Running trial 49

Step 5: Calculate the average gate fidelity

The *average gate fidelity* between the m -qubit ideal unitaries U and the executed U' is:

$$F_{\text{avg}}(U, U') = \frac{|\text{Tr}(U^\dagger U')|^2/2^m + 1}{2^m + 1}$$

The observed distribution for an implementation U' of model circuit U is $q_U(x)$, and the probability of sampling a heavy output is:

$$h_U = \sum_{x \in H_U} q_U(x)$$

As an illustration, we print the heavy output counts from various depths (for trial 0):

```
qv_fitter.add_data(exp_results)
for qubit_list in qubit_lists:
    l = len(qubit_list)
    #print (qv_fitter._heavy_output_counts)
    print ('qv_depth_'+str(l)+'_trial_0:', qv_fitter._heavy_output_counts['qv_depth_'+str(l)])
```



```
qv_depth_3_trial_0: 792
qv_depth_4_trial_0: 788
qv_depth_5_trial_0: 736
qv_depth_6_trial_0: 651
```

Step 6: Calculate the achievable depth

The probability of observing a heavy output by implementing a randomly selected depth d model circuit is:

$$h_d = \int_U h_U dU$$

The *achievable depth* $d(m)$ is the largest d such that we are confident that $h_d > 2/3$. In other words,

$$h_1, h_2, \dots, h_{d(m)} > 2/3 \text{ and } h_{d(m+1)} \leq 2/3$$

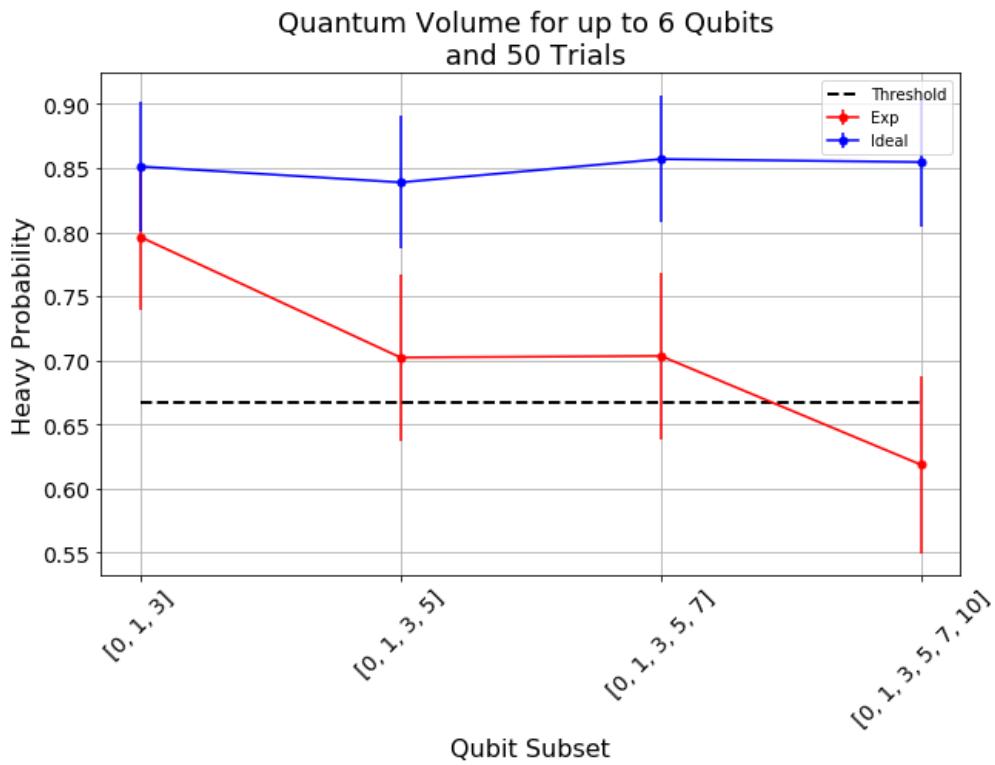
We now convert the heavy outputs in the different trials and calculate the mean h_d and the error for plotting the graph.

```
plt.figure(figsize=(10, 6))
ax = plt.gca()

# Plot the essence by calling plot_rb_data
qv_fitter.plot_qv_data(ax=ax, show_plt=False)

# Add title and label
ax.set_title('Quantum Volume for up to %d Qubits \n and %d Trials' %(len(qubit_lists[-1])))

plt.show()
```



Step 7: Calculate the Quantum Volume

The quantum volume treats the width and depth of a model circuit with equal importance and measures the largest square-shaped (i.e., $m = d$) model circuit a quantum computer can implement successfully on average.

The *quantum volume* V_Q is defined as

$$\log_2 V_Q = \arg \max_m \min(m, d(m))$$

We list the statistics for each depth. For each depth we list if the depth was successful or not and with what confidence interval. For a depth to be successful the confidence interval must be > 97.5%.

```
qv_success_list = qv_fitter.qv_success()
qv_list = qv_fitter.ydata
QV = 1
for qidx, qubit_list in enumerate(qubit_lists):
    if qv_list[0][qidx]>2/3:
        if qv_success_list[qidx][0]:
            print("Width/depth %d greater than 2/3 (%f) with confidence %f (successful)
                  (%d,%d,%d,%d,%d)" % (len(qubit_list),qv_list[0][qidx],qv_success_list[qidx][1],qv_fitter.QV,qv_fitter.quantum_volume()[qidx])
        else:
            print("Width/depth %d greater than 2/3 (%f) with confidence %f (unsuccessful)
                  (%d,%d,%d,%d,%d)" % (len(qubit_list),qv_list[0][qidx],qv_success_list[qidx][1]))
    else:
        print("Width/depth %d less than 2/3 (unsuccessful).%d" % (len(qubit_list)))
```

Width/depth 3 greater than 2/3 (0.796309) with confidence 0.988582 (successful). Quantum Volume is: 8
Width/depth 4 greater than 2/3 (0.702207) with confidence 0.708690 (unsuccessful).
Width/depth 5 greater than 2/3 (0.703359) with confidence 0.714988 (unsuccessful).
Width/depth 6 less than 2/3 (unsuccessful).

```
print ("The Quantum Volume is:", QV)
```

The Quantum Volume is: 8

The Variational Quantum Linear Solver

```
import qiskit
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit import Aer, execute
import math
import random
import numpy as np
from scipy.optimize import minimize
```



Introduction

The Variational Quantum Linear Solver, or the VQLS is a variational quantum algorithm that utilizes VQE in order to solve systems of linear equations more efficiently than classical computational algorithms. Specifically, if we are given some matrix A , such that $A|x\rangle = |b\rangle$, where $|b\rangle$ is some known vector, the VQLS algorithm is theoretically able to find a normalized $|x\rangle$ that is proportional to $|x\rangle$, which makes the above relationship true.

The output of this algorithm is identical to that of the HHL Quantum Linear-Solving Algorithm, except, while HHL provides a much more favourable computation speedup over VQLS, the variational nature of our algorithm allows for it to be performed on NISQ quantum computers, while HHL would require much more robust quantum hardware, and many more qubits.

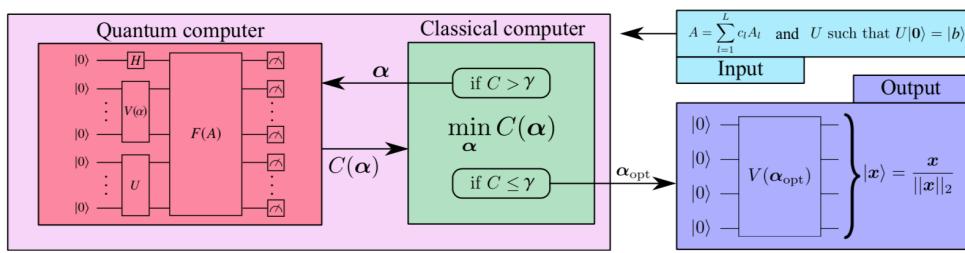
The Algorithm

To begin, the inputs into this algorithm are evidently the matrix A , which we have to decompose into a linear combination of unitaries with complex coefficients:

$$A = \sum n c_n A_n$$

Where each A_n is some unitary, and some unitary U that prepares state $|b\rangle$ from $|0\rangle$. Now, recall the general structure of a variational quantum algorithm. We have to construct a quantum cost function, which can be evaluated with a low-depth parametrized quantum circuit, then output to the classical optimizer. This allows us to search a parameter space for some set of parameters α , such that $|\psi(\alpha)\rangle = |x\rangle\|x\|$, where $|\psi(k)\rangle$ is the output of our quantum circuit corresponding to some parameter set k .

Before we actually begin constructing the cost function, let's take a look at a "high level" overview of the sub-routines within this algorithm, as illustrated in this image from the original paper:



So essentially, we start off with a qubit register, with each qubit initialized to $|0\rangle$. Our algorithm takes its inputs, then prepares and evaluates the cost function, starting with the creation of some ansatz $V(\alpha)$. If the computed cost is greater than some parameter γ , the algorithm is run again with updated parameters, and if not, the algorithm terminates, and the ansatz is calculated with the optimal parameters (determined at termination). This gives us the state vector that minimizes our cost function, and therefore the normalized form of $|x\rangle$.

Let's start off by considering the ansatz $V(\alpha)$, which is just a circuit that prepares some arbitrary state $|\psi(k)\rangle$. This allows us to "search" the state space by varying some set of parameters, k . Anyways, the ansatz that we will use for this implementation is given as follows:

```
def apply_fixed_ansatz(qubits, parameters):
    for iz in range (0, len(qubits)):
        circ.ry(parameters[0][iz], qubits[iz])

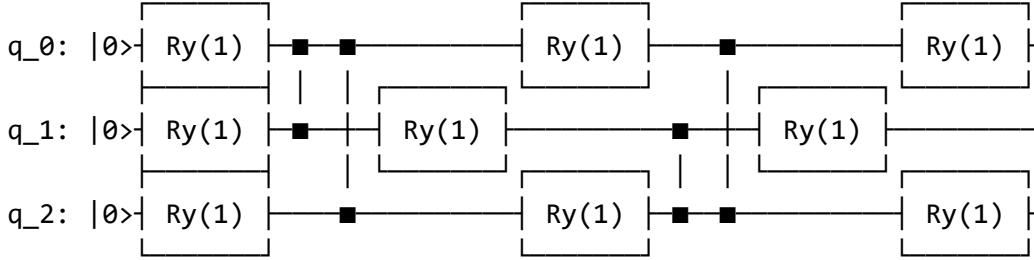
        circ.cz(qubits[0], qubits[1])
        circ.cz(qubits[2], qubits[0])

    for iz in range (0, len(qubits)):
        circ.ry(parameters[1][iz], qubits[iz])

        circ.cz(qubits[1], qubits[2])
        circ.cz(qubits[2], qubits[0])

    for iz in range (0, len(qubits)):
        circ.ry(parameters[2][iz], qubits[iz])

circ = QuantumCircuit(3)
apply_fixed_ansatz([0, 1, 2], [[1, 1, 1], [1, 1, 1], [1, 1, 1]])
print(circ)
```



This is called a **fixed hardware ansatz**: the configuration of quantum gates remains the same for each run of the circuit, all that changes are the parameters. Unlike the QAOA ansatz, it is not composed solely of Trotterized Hamiltonians. The applications of Ry gates allows us to search the state space, while the CZ gates create "interference" between the different qubit states.

Now, it makes sense for us to consider the actual **cost function**. The goal of our algorithm will be to minimize cost, so when $|\Phi\rangle = A|\psi(k)\rangle$ is very close to $|b\rangle$, we want our cost function's output to be very small, and when the vectors are close to being orthogonal, we want the cost function to be very large. Thus, we introduce the "projection" Hamiltonian:

$$H_P = I - |b\rangle\langle b|$$

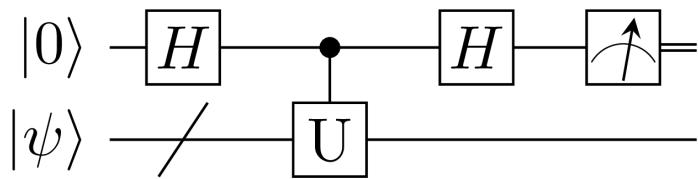
Where we have:

$$C_P = \langle\Phi|H_P|\Phi\rangle = \langle\Phi|(I - |b\rangle\langle b|)|\Phi\rangle = \langle\Phi|\Phi\rangle - \langle\Phi|b\rangle\langle b|\Phi\rangle$$

Notice how the second term tells us "how much" of $|\Phi\rangle$ lies along $|b\rangle$. We then subtract this from another number to get the desired low number when the inner product of $|\Phi\rangle$ and $|b\rangle$ is greater (they agree more), and the opposite for when they are close to being orthogonal. This is looking good so far! However, there is still one more thing we can do to increase the accuracy of the algorithm: normalizing the cost function. This is due to the fact that if $|\Phi\rangle$ has a small norm, then the cost function will still be low, even if it does not agree with $|b\rangle$. Thus, we replace $|\Phi\rangle$ with $|\Phi\rangle/\sqrt{\langle\Phi|\Phi\rangle}$:

$$\hat{C}_P = \langle\Phi|\Phi\rangle\langle\Phi|\Phi\rangle - \langle\Phi|b\rangle\langle b|\Phi\rangle\langle\Phi|\Phi\rangle = 1 - \langle\Phi|b\rangle\langle b|\Phi\rangle\langle\Phi|\Phi\rangle = 1 - |\langle b|\Phi\rangle|^2\langle\Phi|\Phi\rangle$$

Ok, so, we have prepared our state $|\psi(k)\rangle$ with the ansatz. Now, we have two values to calculate in order to evaluate the cost function, namely $|\langle b|\Phi\rangle|^2$ and $\langle\Phi|\Phi\rangle$. Luckily, a nifty little quantum subroutine called the **Hadamard Test** allows us to do this! Essentially, if we have some unitary U and some state $|\phi\rangle$, and we want to find the expectation value of U with respect to the state, $\langle\phi|U|\phi\rangle$, then we can evaluate the following circuit:



Then, the probability of measuring the first qubit to be 0 is equal to $\frac{1}{2}(1 + \text{Re}(U))$ and the probability of measuring 1 is $\frac{1}{2}(1 - \text{Re}(U))$, so subtracting the two probabilities gives us $\text{Re}(U)$. Luckily, the matrices we will be dealing with when we test this algorithm are completely real, so $\text{Re}(U) = \langle U \rangle$, for this specific implementation. Here is how the Hadamard test works. By the circuit diagram, we have as our general state vector:

$$|0\rangle + |1\rangle\sqrt{2} \otimes |\psi\rangle = |0\rangle \otimes |\psi\rangle + |1\rangle \otimes |\psi\rangle\sqrt{2}$$

Applying our controlled unitary:

$$|0\rangle \otimes |\psi\rangle + |1\rangle \otimes |\psi\rangle\sqrt{2} \rightarrow |0\rangle \otimes |\psi\rangle + |1\rangle \otimes U|\psi\rangle\sqrt{2}$$

Then applying the Hadamard gate to the first qubit:

$$\begin{aligned} |0\rangle \otimes |\psi\rangle + |1\rangle \otimes U|\psi\rangle\sqrt{2} &\rightarrow \frac{1}{2} [|0\rangle \otimes |\psi\rangle + |1\rangle \otimes |\psi\rangle + |0\rangle \otimes U|\psi\rangle \\ &\quad - |1\rangle \otimes U|\psi\rangle] \end{aligned}$$

$$\Rightarrow |0\rangle \otimes (I + U)|\psi\rangle + |1\rangle \otimes (I - U)|\psi\rangle$$

When we take a measurement of the first qubit, remember that in order to find the probability of measuring 0, we must take the inner product of the state vector with $|0\rangle$, then multiply by its complex conjugate (see the quantum mechanics section if you are not familiar with this). The same follows for the probability of measuring 1. Thus, we have:

$$P(0) = \frac{1}{4} \langle \psi | (I + U)(I + U^\dagger) | \psi \rangle = \frac{1}{4} \langle \psi | (I^2 + U + U^\dagger + U^\dagger U) | \psi \rangle = \frac{1}{4} \langle \psi | (2I + U + U^\dagger) | \psi \rangle$$

$$\Rightarrow \frac{1}{4} [2 + \langle \psi | U^\dagger | \psi \rangle + \langle \psi | U | \psi \rangle] = \frac{1}{4} [2 + (\langle \psi | U | \psi \rangle)^* + \langle \psi | U | \psi \rangle] = \frac{1}{2}(1 + \operatorname{Re} \langle \psi | U | \psi \rangle)$$

By a similar procedure, we get:

$$P(1) = \frac{1}{2}(1 - \operatorname{Re} \langle \psi | U | \psi \rangle)$$

And so, by taking the difference:

$$P(0) - P(1) = \operatorname{Re} \langle \psi | U | \psi \rangle$$

Cool! Now, we can actually implement this for the two values we have to compute. Starting with $\langle \Phi | \Phi \rangle$, we have:

$$\langle \Phi | \Phi \rangle = \langle \psi(k) | A^\dagger A | \psi(k) \rangle = \langle 0 | V(k)^\dagger A^\dagger A V(k) | 0 \rangle = \langle 0 | V(k)^\dagger \left(\sum n c_n A_n \right)^\dagger \left(\sum n c_n A_n \right) V(k) | 0 \rangle$$

$$\Rightarrow \langle \Phi | \Phi \rangle = \sum_m \sum_n c_m^* c_n \langle 0 | V(k)^\dagger A_{tm} A_n V(k) | 0 \rangle$$

and so our task becomes computing every possible term $\langle 0 | V(k)^\dagger A_{tm} A_n V(k) | 0 \rangle$ using the Hadamard test. This requires us to prepare the state $V(k) | 0 \rangle$, and then perform controlled operations with some control-ancilla qubit for the unitary matrices A_m^\dagger and A_n . We can implement this in code:

```
#Creates the Hadamard test
```



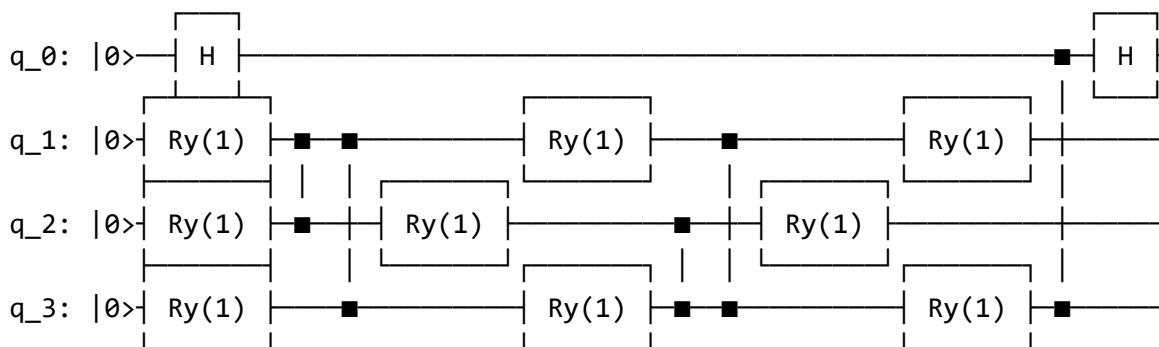
```
def had_test(gate_type, qubits, ancilla_index, parameters):
    circ.h(ancilla_index)
    apply_fixed_ansatz(qubits, parameters)

    for ie in range (0, len(gate_type[0])):
        if (gate_type[0][ie] == 1):
            circ.cz(ancilla_index, qubits[ie])

    for ie in range (0, len(gate_type[1])):
        if (gate_type[1][ie] == 1):
            circ.cz(ancilla_index, qubits[ie])

    circ.h(ancilla_index)

circ = QuantumCircuit(4)
had_test([[0, 0, 0], [0, 0, 1]], [1, 2, 3], 0, [[1, 1, 1], [1, 1, 1], [1, 1, 1]])
print(circ)
```



The reason why we are applying two different "gate_types" is because this represents the pairs of gates shown in the expanded form of $|\langle b | \Phi | \Phi \rangle|^2$.

It is also important to note that for the purposes of this implementation (the systems of equations we will actually be solving, we are only concerned with the gates Z and \mathbb{I} , so I only include support for these gates (The code includes number "identifiers" that signify the application of different gates, 0 for \mathbb{I} and 1 for Z).

Now, we can move on to the second value we must calculate, which is $|\langle b | \Phi | \Phi \rangle|^2$. We get:

$$|\langle b | \Phi | \Phi \rangle|^2 = |\langle b | A V(k) | 0 \rangle|^2 = |\langle 0 | U^{\dagger} A V(k) | 0 \rangle|^2 = |\langle 0 | U^{\dagger} A V(k) | 0 \rangle \langle 0 | V(k)^{\dagger} A^{\dagger} U | 0 \rangle|$$

All we have to do now is the same expansion as before for the product $\langle 0 | U^{\dagger} A V(k) | 0 \rangle \langle 0 | V(k)^{\dagger} A^{\dagger} U | 0 \rangle$:

$$\langle 0 | U^{\dagger} A V(k) | 0 \rangle^2 = \sum_m \sum_n c_m^* c_n \langle 0 | U^{\dagger} A_n V(k) | 0 \rangle \langle 0 | V(k)^{\dagger} A_m^{\dagger} U | 0 \rangle$$

Now, again, for the purposes of this demonstration, we will soon see that all the outputs/expectation values of our implementation will be real, so we have:

$$\Rightarrow \langle 0 | U^{\dagger} A V(k) | 0 \rangle = (\langle 0 | U^{\dagger} A V(k) | 0 \rangle)^* = \langle 0 | V(k)^{\dagger} A^{\dagger} U | 0 \rangle$$

Thus, in this particular implementation:

$$|\langle b | \Phi \rangle|^2 = \sum_m \sum_n c_m c_n |\langle 0 | U^\dagger A_n V(k) | 0 \rangle \langle 0 | U^\dagger A_m V(k) | 0 \rangle$$

There is a sophisticated way of solving for this value, using a newly-proposed subroutine called the **Hadamard Overlap Test** (see cited paper), but for this tutorial, we will just be using a standard Hadamard Test, where we control each matrix. This unfortunately requires the use of an extra ancilla qubit. We essentially just place a control on each of the gates involved in the ancilla, the $|b\rangle$ preparation unitary, and the A_n unitaries. We get something like this for the controlled-ansatz:

```
#Creates controlled ansatz for calculating |<b|psi>|^2 with a Hadamard test
def control_fixed_ansatz(qubits, parameters, ancilla, reg):

    for i in range (0, len(qubits)):
        circ.cry(parameters[0][i], qiskit.circuit.Qubit(reg, ancilla), qiskit.circuit.Q

        circ.ccx(ancilla, qubits[1], 4)
        circ.cz(qubits[0], 4)
        circ.ccx(ancilla, qubits[1], 4)

        circ.ccx(ancilla, qubits[0], 4)
        circ.cz(qubits[2], 4)
        circ.ccx(ancilla, qubits[0], 4)

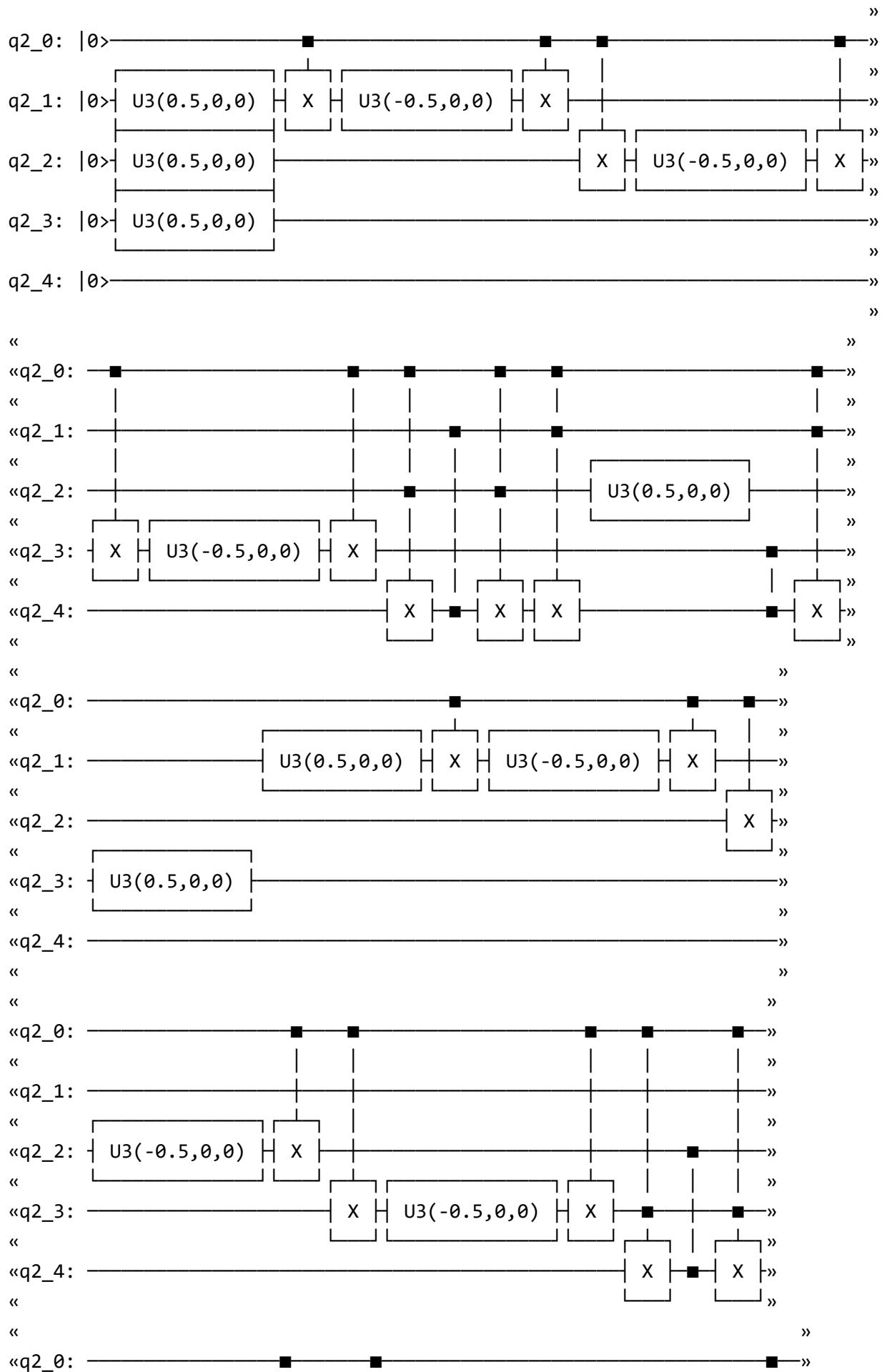
    for i in range (0, len(qubits)):
        circ.cry(parameters[1][i], qiskit.circuit.Qubit(reg, ancilla), qiskit.circuit.Q

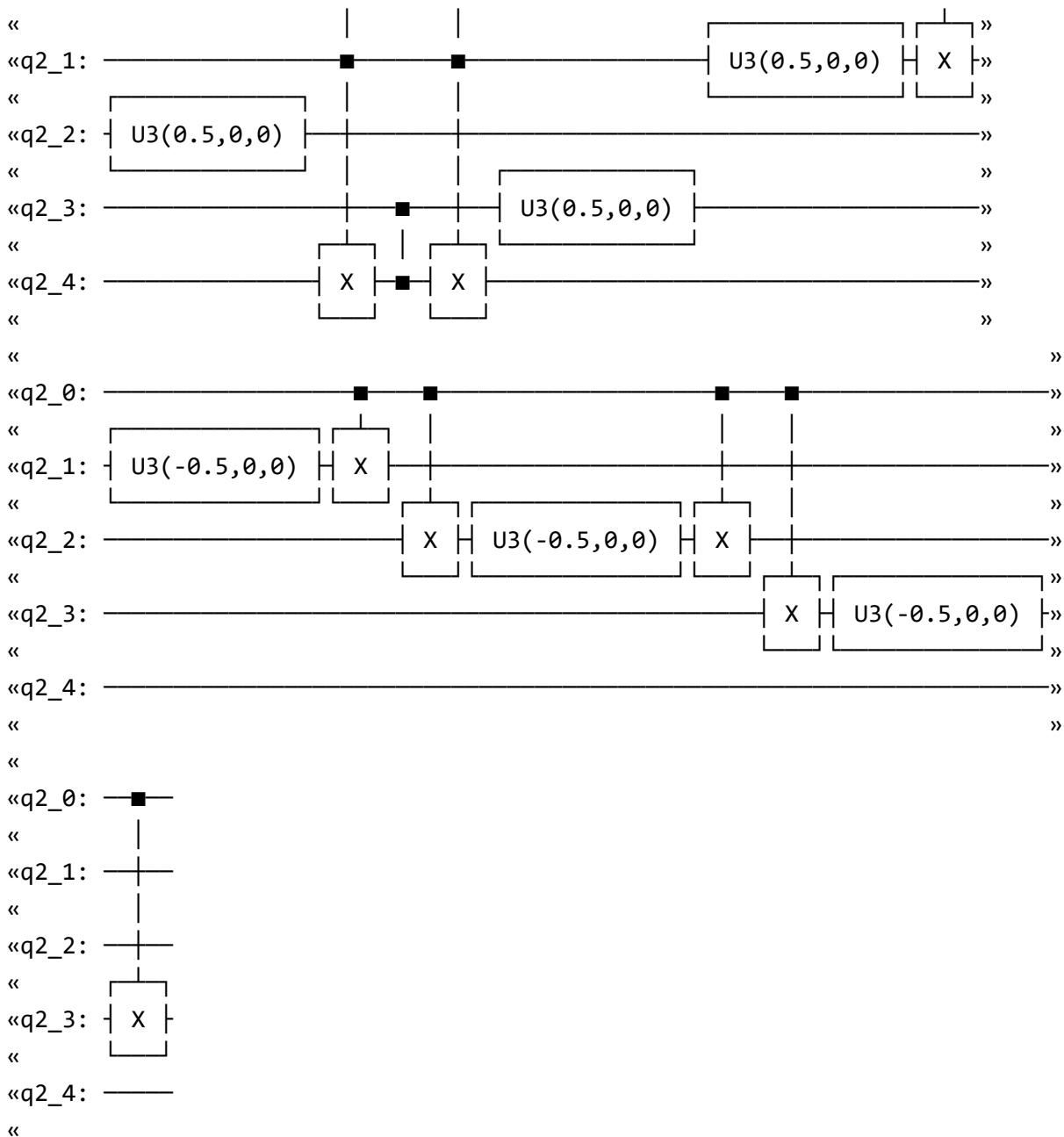
        circ.ccx(ancilla, qubits[2], 4)
        circ.cz(qubits[1], 4)
        circ.ccx(ancilla, qubits[2], 4)

        circ.ccx(ancilla, qubits[0], 4)
        circ.cz(qubits[2], 4)
        circ.ccx(ancilla, qubits[0], 4)

    for i in range (0, len(qubits)):
        circ.cry(parameters[2][i], qiskit.circuit.Qubit(reg, ancilla), qiskit.circuit.Q

q_reg = QuantumRegister(5)
circ = QuantumCircuit(q_reg)
control_fixed_ansatz([1, 2, 3], [[1, 1, 1], [1, 1, 1], [1, 1, 1]], 0, q_reg)
print(circ)
```





Notice the extra qubit, `q0_4`. This is an ancilla, and allows us to create a CCZ gate, as is shown in the circuit. Now, we also have to create the circuit for U. In our implementation, we will pick U as:

$$U = H_1 H_2 H_3$$

Thus, we have:

```
def control_b(ancilla, qubits):
```

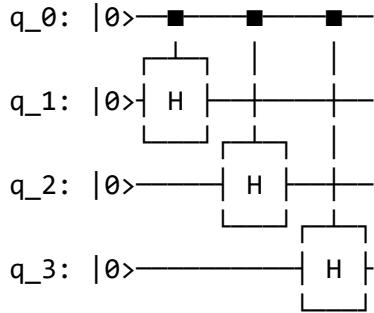


```

for ia in qubits:
    circ.ch(ancilla, ia)

circ = QuantumCircuit(4)
control_b(0, [1, 2, 3])
print(circ)

```



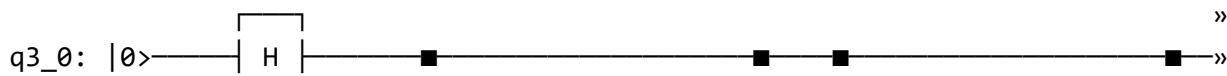
Finally, we construct our new Hadamard test:

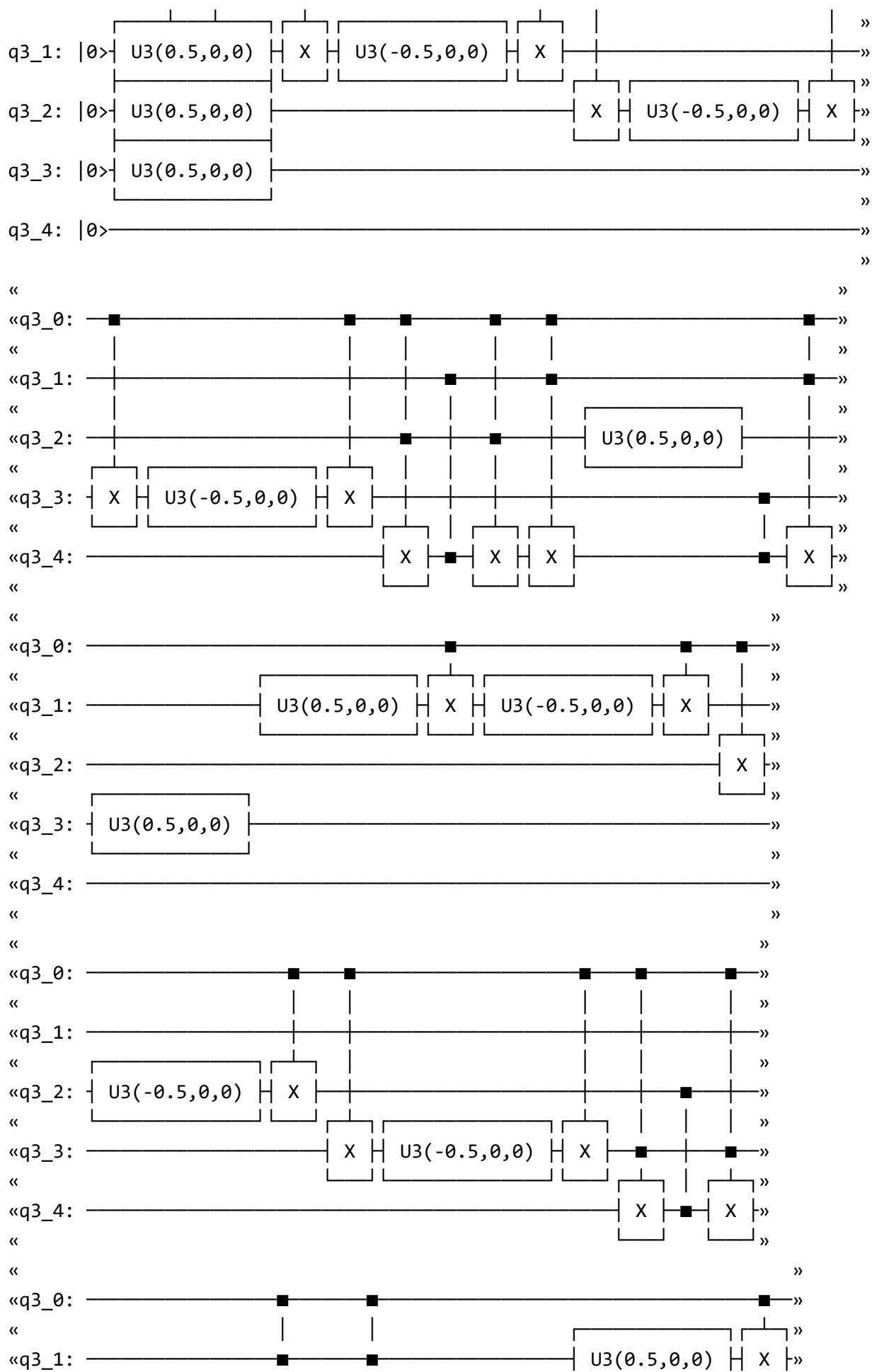
```

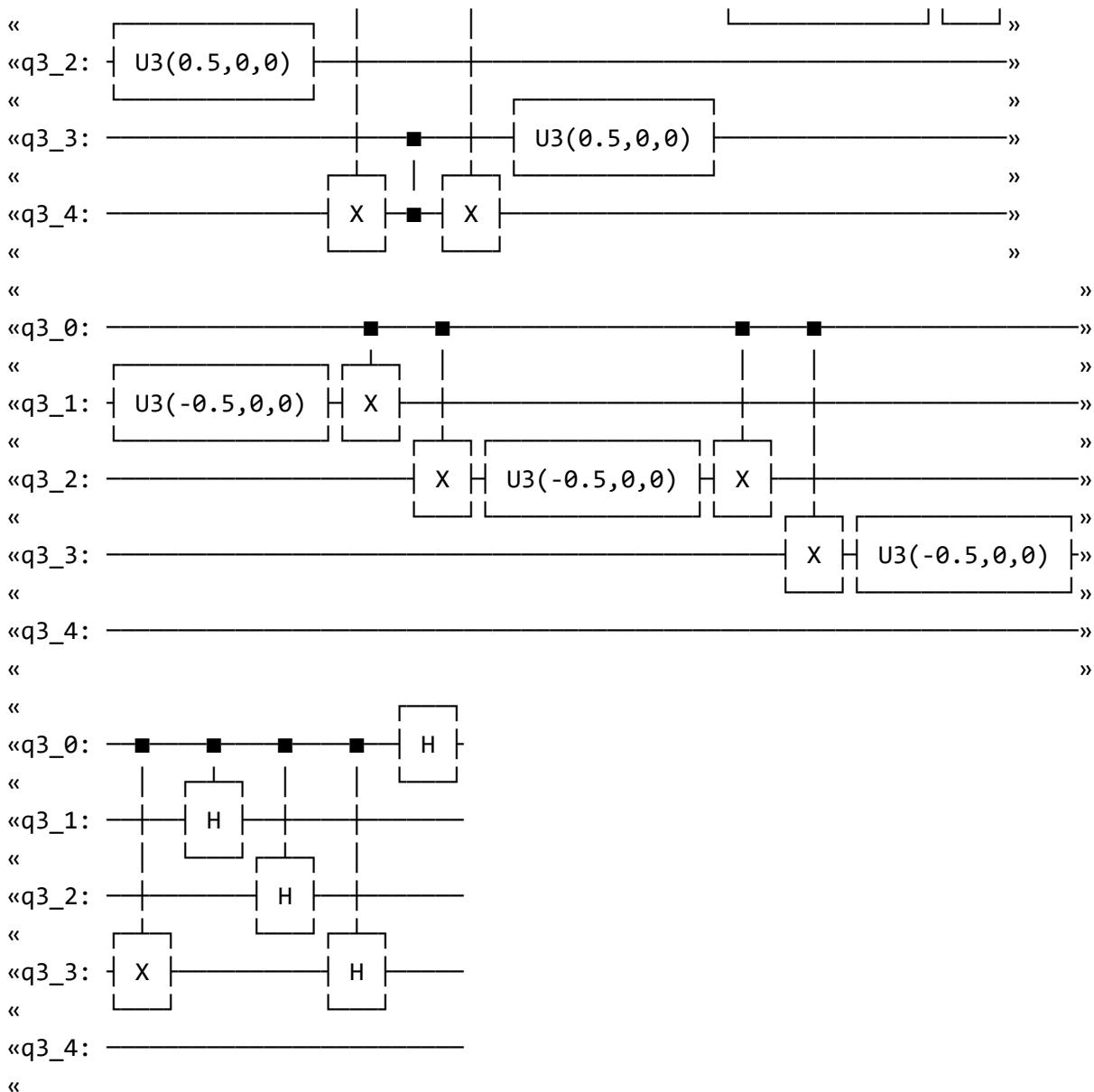
#Create the controlled Hadamard test, for calculating <psi|psi>   

def special_had_test(gate_type, qubits, ancilla_index, parameters, reg):
    circ.h(ancilla_index)
    control_fixed_ansatz(qubits, parameters, ancilla_index, reg)
    for ty in range (0, len(gate_type)):
        if (gate_type[ty] == 1):
            circ.cz(ancilla_index, qubits[ty])
    control_b(ancilla_index, qubits)
    circ.h(ancilla_index)
q_reg = QuantumRegister(5)
circ = QuantumCircuit(q_reg)
special_had_test([[0, 0, 0], [0, 0, 1]], [1, 2, 3], 0, [[1, 1, 1], [1, 1, 1], [1, 1, 1]])
print(circ)

```







This is for the specific implementation when all of our parameters are set to 1, and the set of gates A_n is simply $[0, 0, 0]$, and $[0, 0, 1]$, which corresponds to the identity matrix on all qubits, as well as the Z matrix on the third qubit (with my "code notation").

Now, we are ready to calculate the final cost function. This simply involves us taking the products of all combinations of the expectation outputs from the different circuits, multiplying by their respective coefficients, and arranging into the cost function that we discussed previously!

```
#Implements the entire cost function on the quantum circuit
```



```
def calculate_cost_function(parameters):
```

```
    global opt
```

```

overall_sum_1 = 0

parameters = [parameters[0:3], parameters[3:6], parameters[6:9]]

for i in range(0, len(gate_set)):
    for j in range(0, len(gate_set)):

        global circ

        qctl = QuantumRegister(5)
        qc = ClassicalRegister(5)
        circ = QuantumCircuit(qctl, qc)

        backend = Aer.get_backend('statevector_simulator')

        multiply = coefficient_set[i]*coefficient_set[j]

        had_test([gate_set[i], gate_set[j]], [1, 2, 3], 0, parameters)

        job = execute(circ, backend)

        result = job.result()
        outputstate = result.get_statevector(circ, decimals=100)
        o = outputstate

        m_sum = 0
        for l in range (0, len(o)):
            if (l%2 == 1):
                n = float(o[l])**2
                m_sum+=n

        overall_sum_1+=multiply*(1-(2*m_sum))

overall_sum_2 = 0

for i in range(0, len(gate_set)):
    for j in range(0, len(gate_set)):

        multiply = coefficient_set[i]*coefficient_set[j]
        mult = 1

        for extra in range(0, 2):

            qctl = QuantumRegister(5)
            qc = ClassicalRegister(5)
            circ = QuantumCircuit(qctl, qc)

```

```

backend = Aer.get_backend('statevector_simulator')

if (extra == 0):
    special_had_test(gate_set[i], [1, 2, 3], 0, parameters, qctl)
if (extra == 1):
    special_had_test(gate_set[j], [1, 2, 3], 0, parameters, qctl)

job = execute(circ, backend)

result = job.result()
outputstate = result.get_statevector(circ, decimals=100)
o = outputstate

m_sum = 0
for l in range (0, len(o)):
    if (l%2 == 1):
        n = float(o[l])**2
        m_sum+=n
mult = mult*(1-(2*m_sum))

overall_sum_2+=multiply*mult

print(1-float(overall_sum_2/overall_sum_1))

return 1-float(overall_sum_2/overall_sum_1)

```

This code may look long and daunting, but it isn't! In this simulation, I'm taking a **numerical** approach, where I'm calculating the amplitude squared of each state corresponding to a measurement of the ancilla Hadamard test qubit in the 1 state, then calculating $P(0) - P(1) = 1 - 2P(1)$ with that information. This is very exact, but is not realistic, as a real quantum device would have to sample the circuit many times to generate these probabilities (I'll discuss sampling later). In addition, this code is not completely optimized (it completes more evaluations of the quantum circuit than it has to), but this is the simplest way in which the code can be implemented, and I will be optimizing it in an update to this tutorial in the near future.

The final step is to actually use this code to solve a real linear system. We will first be looking at the example:

$$A = 0.45 Z_3 + \mathbb{I}$$

In order to minimize the cost function, we use the COBYLA optimizer method, which we repeatedly applying. Our search space for parameters is determined by $\frac{k}{1000} \leq k \leq 3000$, which is initially chosen randomly. We will run the optimizer for 200 steps, then terminate and apply the ansatz for our optimal parameters, to get our optimized state vector! In addition, we will compute some post-processing, to see if our algorithm actually works! In order to do this, we will apply A to our optimal vector $|\psi\rangle$, normalize it, then calculate the inner product squared of this vector and the solution vector, $|b\rangle$! We can put this all into code as:

```
coefficient_set = [0.55, 0.45]
gate_set = [[0, 0, 0], [0, 0, 1]]

out = minimize(calculate_cost_function, x0=[float(random.randint(0,3000))/1000 for i in
print(out)

out_f = [out['x'][0:3], out['x'][3:6], out['x'][6:9]]

circ = QuantumCircuit(3, 3)
apply_fixed_ansatz([0, 1, 2], out_f)

backend = Aer.get_backend('statevector_simulator')

job = execute(circ, backend)

result = job.result()
o = result.get_statevector(circ, decimals=10)

a1 = coefficient_set[1]*np.array([[1,0,0,0,0,0,0], [0,1,0,0,0,0,0], [0,0,1,0,0,0,0],
a2 = coefficient_set[0]*np.array([[1,0,0,0,0,0,0], [0,1,0,0,0,0,0], [0,0,1,0,0,0,0,
a3 = np.add(a1, a2)

b = np.array([float(1/np.sqrt(8)),float(1/np.sqrt(8)),float(1/np.sqrt(8)),float(1/np.sq
print((b.dot(a3.dot(o))/(np.linalg.norm(a3.dot(o))))**2)
```

```
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/ipykernel
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/ipykernel
```

```
0.9831625609596678
0.714912455705426
0.7529880773718886
0.7535739217359092
```

0.38887933377570305
0.43787060968759484
0.9995074080630343
0.2725579992425663
0.34530912011383297
0.9811319924317494
0.7407761170344858
0.5630549481302094
0.2619730646995667
0.5389930132093528
0.25184270902020445
0.46908713020131154
0.4422184803430059
0.5893271821377166
0.2855692599105859
0.9116589313008762
0.29141167608440854
0.525824549202996
0.3295799342424929
0.30350926168129244
0.2539859257368158
0.3327968430140976
0.24442108645671845
0.2517400354018866
0.2847112665247806
0.24842609523000503
0.2450669400424792
0.25131199264152904
0.24451533011721582
0.26335589503798007
0.2548752033305517
0.24510223330719405
0.2420596901101676
0.2333295382174383
0.23710075617997817
0.2306670755821555
0.2240817114520406
0.22095423921532442
0.22211478241995253
0.21943556733156278
0.21730082800773365
0.21590344336007627
0.2140042924689879
0.20779024131304558
0.19719922332846085
0.19063091169721802

0.18578926566423037
0.17465465726304008
0.16495804769184486
0.15721213367172704
0.14826522592046276
0.13806348685646574
0.12776798028305525
0.11798170459767032
0.11146257140379046
0.10058460669094504
0.09347679346943738
0.08313862031945529
0.07497083397659365
0.06912244971828985
0.06303824406711311
0.063427225342084
0.05936678092714909
0.055818687403525136
0.05302312997909753
0.05543758840441915
0.054974276427438906
0.05429900370813945
0.052694017684379424
0.05237037425296587
0.05988596588670281
0.05756655792876397
0.04440544552284542
0.04740962048207942
0.04660859805780415
0.04489334591892946
0.04580247191928066
0.0375166325571078
0.03988978231458584
0.0375970231652315
0.036082779501486306
0.040529570149290595
0.039093987789943774
0.035904235417706265
0.03638778154701572
0.03641423421024703
0.03555253652129542
0.037657699533344635
0.037145421392184685
0.03308982948205241
0.03242172992965053
0.034966630919115116

0.03171088893536056
0.032075781550268134
0.03234245039688666
0.0315564587195295
0.03162660915901372
0.03107315383737974
0.02947301627995158
0.02880150404815962
0.029156887615918503
0.028645430241494685
0.02795262891576311
0.028653267891823764
0.027902176216327845
0.026434434570602017
0.025671431131693012
0.02521146097948157
0.024913178149970627
0.024171072224127288
0.02333237280100109
0.023456442818372225
0.023317166504144504
0.024497422245249312
0.023396859798709757
0.022063295677102635
0.022346562291048255
0.022290380173742408
0.02159645358423612
0.022438304991653113
0.021321463584655587
0.020412927478190457
0.020092737071547906
0.020463840191913163
0.019624969597142772
0.01965588161387155
0.01967197447592106
0.019360692343968977
0.018949380033071894
0.0182654724232979
0.018607809610731385
0.0184265748265251
0.018663278534589156
0.018222387329481893
0.01807679908204518
0.019049293822393176
0.016883807674721973
0.01594351338081923

0.01552040946028932
0.01521549832303104
0.015177429132839149
0.015620245241075414
0.015628680106564707
0.014516602093362807
0.01352375716643961
0.013348860659593065
0.013342038519289168
0.013033014608608728
0.013962922928118848
0.012716119050846464
0.012169688416967195
0.01192232728587328
0.011792207814985245
0.01365329146978933
0.01158875950921634
0.011195313935059903
0.010920555584017388
0.01115898577674479
0.010201877155896821
0.010657302475341246
0.010729019137815432
0.00964621886416972
0.009209795683734256
0.00968367826173111
0.009261610502754203
0.009388371177821764
0.009037626965632262
0.010236705431579507
0.009051287134295483
0.008798115231816395
0.008780859100836702
0.009210513132848819
0.008233841209075887
0.008071810329476459
0.007993122880147974
0.007967125451031953
0.007756262468621156
0.007751763376797061
0.007911252121015067
0.007936184277404612
0.007590953476323659
0.007198216193537776
0.007121240970198728
0.0072037497321087995

```

0.007123452876170111
0.0072507436813531445
0.007387319091008426
0.006900090719671237
0.006546974413554674
0.006543424070501902
0.00650636458743542
0.006702224181074357
0.006662711864537707
0.006235439513391672
0.006067529070167055
0.005907904334877201
    fun: 0.005907904334877201
    maxcv: 0.0
    message: 'Maximum number of function evaluations has been exceeded.'
    nfev: 200
    status: 2
    success: False
    x: array([3.17231031, 0.66099822, 0.94311286, 1.65284435, 1.1750949 ,
       0.94517854, 3.04031284, 0.84369061, 2.92924202])
(0.9940920956678407-0j)

```

As you can see, our cost function has achieved a fairly low value of 0.005907904334877201, and when we calculate our classical cost function, we get 0.9940920956678407, which agrees perfectly with what we measured, the vectors $|\psi\rangle_o$ and $|b\rangle$ are very similar!

Let's do another test! This time, we will keep $|b\rangle$ the same, but we will have:

$$A \cdot = 0.55 \mathbf{I} + 0.225 Z_2 + 0.225 Z_3$$

Again, we run our optimization code:

```

coefficient_set = [0.55, 0.225, 0.225]
gate_set = [[0, 0, 0], [0, 1, 0], [0, 0, 1]]

out = minimize(calculate_cost_function, x0=[float(random.randint(0,3000))/1000 for i in
print(out)

out_f = [out['x'][0:3], out['x'][3:6], out['x'][6:9]]

circ = QuantumCircuit(3, 3)
apply_fixed_ansatz([0, 1, 2], out_f)

```

```

backend = Aer.get_backend('statevector_simulator')

job = execute(circ, backend)

result = job.result()
o = result.get_statevector(circ, decimals=10)

a1 = coefficient_set[2]*np.array([[1,0,0,0,0,0,0,0], [0,1,0,0,0,0,0,0], [0,0,1,0,0,0,0,0],
a0 = coefficient_set[1]*np.array([[1,0,0,0,0,0,0,0], [0,1,0,0,0,0,0,0], [0,0,-1,0,0,0,0,0],
a2 = coefficient_set[0]*np.array([[1,0,0,0,0,0,0,0], [0,1,0,0,0,0,0,0], [0,0,1,0,0,0,0,0],
a3 = np.add(np.add(a2, a0), a1)

b = np.array([float(1/np.sqrt(8)),float(1/np.sqrt(8)),float(1/np.sqrt(8)),float(1/np.sqrt(8))])

print((b.dot(a3.dot(o))/(np.linalg.norm(a3.dot(o))))**2)

```

```

/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/ipykernel
/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/site-packages/ipykernel

```

```

0.8833056895221885
0.7404062275757255
0.8091891476664499
0.5951720668847402
0.39062534343372135
0.2931639114976957
0.5604281659812129
0.560080687994932
0.4803820195483641
0.5324588876270797
0.4685752608919278
0.3282579558887714
0.24711074171070913
0.2844236577697423
0.25260564852823797
0.3150509699174163
0.2992825712207584
0.2139070770671957
0.2244990546173472
0.22613139095989798
0.172829276048014
0.1600018428675094
0.19562164408593274

```

0.1587687310399153
0.17179896136502615
0.15123167889056754
0.1310353184387082
0.14748215199832526
0.12955815255339975
0.12938049063121726
0.1507987572368068
0.15614975859029057
0.12583784975341972
0.1370405913395396
0.12190998301674683
0.13850716852312062
0.14937640643261818
0.12490627765037154
0.126062749819002
0.11862847675452592
0.12757317305757454
0.11629015246752805
0.09545897096537115
0.0844605964901024
0.09215242076941255
0.07930000643513602
0.08295022225910409
0.06880125309019802
0.08789085024962462
0.0723822703567113
0.058960738324484896
0.056295204949769584
0.050240678318537846
0.04862239722404582
0.055474029210123676
0.04083178628298989
0.031132847620148896
0.016056639832908615
0.01686491697274073
0.015490388498731411
0.01562984776458587
0.01577706325406525
0.01393481747356018
0.011078923747476321
0.007931871992739037
0.009189516314713653
0.007179211217900883
0.013201983069383583
0.00826294682496198

0.011680689511930464
0.018056710444077972
0.007335967048391123
0.027127591463935397
0.012070902637693615
0.007143062809405132
0.0117395654651401
0.006705055189160314
0.0067266104437182506
0.006754904523539995
0.005294953545512349
0.007544273046776517
0.005458623097653215
0.0050339577062862295
0.0024171197249938103
0.0016098687721065597
0.0032127849094635286
0.001373458196467836
0.0018154527446835322
0.0026063693356888074
0.0018413611793377527
0.0012656961602972583
0.001912613576024902
0.0013032878567653672
0.004780452278314473
0.001248468156262672
0.0012787369680381522
0.0012037205835343512
0.0011819210497052701
0.0016027785143645223
0.0013230294807701215
0.0013609173328812396
0.000993732252732915
0.0017779800894820452
0.0010510907889442755
0.0009585942467664754
0.0008629138723946772
0.0010207518625582335
0.0006872668365267565
0.0007062083333057023
0.0007503080958982666
0.000717567162378363
0.0006974796215615253
0.0008486177438721265
0.000731447125851159
0.0006952466460919959

0.0007171419288046765
0.0006876656453529417
0.0006411945852227152
0.0006275005832183655
0.0006110590451339215
0.0006290570042071231
0.0006190654053689348
0.0006313325807618675
0.0006213165772820384
0.0006300475953494589
0.0006285936302956596
0.0005852557855583473
0.000570849350115421
0.0005452201202065243
0.0005709851427690982
0.0005471887071138992
0.0005232308682776576
0.0005093662530177845
0.0005154825245138328
0.0005145268097240807
0.0005064587932218645
0.00048581679698223024
0.00047943478559497876
0.00046007172799755747
0.00043227019767499986
0.00041912544469857593
0.00042180585008277927
0.000416069680302944
0.000418961174539767
0.00041967137368525975
0.000423278283357309
0.00040882739119119105
0.000418054628174791
0.00040851017821808
0.0003929850801126511
0.00040519315841125447
0.00040283350265402085
0.00039305189321792167
0.00038345878865808647
0.00037134728038545894
0.00037184830193537355
0.000364222816488069
0.00036072429475597634
0.0003651575913563576
0.0003681988225339161
0.0003515884890532561

```
0.00033802764625257
0.00033391666385085284
0.0003339117476462983
0.0003228565502255698
0.00031204476190105357
0.0003024559400898186
0.00029174856775104896
0.00029017513764806324
0.0002888164939010007
0.00028381992889803254
0.00027023425572603177
0.0002584110031729203
0.00024804727164196017
0.00024058938257442986
0.00023633531008837583
0.00023716593024381005
0.00022830894277114933
0.00022049997496431661
0.00021843001411114837
0.00021941925638713222
0.00021066898491861608
0.00019966662854398187
0.0001905276970799452
0.00018340012108675197
0.0001773014823526209
0.00017482860435513725
0.00017237614918474975
0.0001667601670679586
0.00015890940988982916
0.00015275818329618662
0.00014971835524424382
0.00014134077118210797
0.00013506290672615773
0.00012918291166186258
0.00012854441887910628
0.0001322011719223637
0.00012002700169067015
0.0001162558506269118
0.0001115758871370609
    fun: 0.0001115758871370609
    maxcv: 0.0
message: 'Maximum number of function evaluations has been exceeded.'
    nfev: 200
    status: 2
success: False
    x: array([3.14090809, 2.17686771, 2.05968755, 3.16561892, 2.62000129,
```

```
1.76616971, 1.55745791, 2.33026238, 3.02413357])  
(0.9998884241081756-0j)
```

Again, very low error, 0.0001115758871370609, and the classical cost function agrees, being 0.9998884241081756! Great, so it works!

Now, we have found that this algorithm works **in theory**. I tried to run some simulations with a circuit that samples the circuit instead of calculating the probabilities numerically. Now, let's try to **sample** the quantum circuit, as a real quantum computer would do! For some reason, this simulation would only converge somewhat well for a ridiculously high number of "shots" (runs of the circuit, in order to calculate the probability distribution of outcomes). I think that this is mostly to do with limitations in the classical optimizer (COBYLA), due to the noisy nature of sampling a quantum circuit (a measurement with the same parameters won't always yield the same outcome). Luckily, there are other optimizers that are built for noisy functions, such as SPSA, but we won't be looking into that in this tutorial. Let's try our sampling for our second value of A, with the same matrix U:

```
#Implements the entire cost function on the quantum circuit (sampling, 100000 shots)  
  
def calculate_cost_function(parameters):  
  
    global opt  
  
    overall_sum_1 = 0  
  
    parameters = [parameters[0:3], parameters[3:6], parameters[6:9]]  
  
    for i in range(0, len(gate_set)):  
        for j in range(0, len(gate_set)):  
  
            global circ  
  
            qctl = QuantumRegister(5)  
            qc = ClassicalRegister(1)  
            circ = QuantumCircuit(qctl, qc)  
  
            backend = Aer.get_backend('qasm_simulator')  
  
            multiply = coefficient_set[i]*coefficient_set[j]  
  
            had_test([gate_set[i], gate_set[j]], [1, 2, 3], 0, parameters)  
  
            circ.measure(0, 0)
```

```
job = execute(circ, backend, shots=100000)

result = job.result()
outputstate = result.get_counts(circ)

if ('1' in outputstate.keys()):
    m_sum = float(outputstate["1"])/100000
else:
    m_sum = 0

overall_sum_1+=multiply*(1-2*m_sum)

overall_sum_2 = 0

for i in range(0, len(gate_set)):
    for j in range(0, len(gate_set)):

        multiply = coefficient_set[i]*coefficient_set[j]
        mult = 1

        for extra in range(0, 2):

            qctl = QuantumRegister(5)
            qc = ClassicalRegister(1)

            circ = QuantumCircuit(qctl, qc)

            backend = Aer.get_backend('qasm_simulator')

            if (extra == 0):
                special_had_test(gate_set[i], [1, 2, 3], 0, parameters, qctl)
            if (extra == 1):
                special_had_test(gate_set[j], [1, 2, 3], 0, parameters, qctl)

            circ.measure(0, 0)

        job = execute(circ, backend, shots=100000)

        result = job.result()
        outputstate = result.get_counts(circ)

        if ('1' in outputstate.keys()):
            m_sum = float(outputstate["1"])/100000
        else:
            m_sum = 0
```

```

        mult = mult*(1-2*m_sum)

    overall_sum_2+=multiply*mult

print(1-float(overall_sum_2/overall_sum_1))

return 1-float(overall_sum_2/overall_sum_1)

```

```

coefficient_set = [0.55, 0.225, 0.225] 
gate_set = [[0, 0, 0], [0, 1, 0], [0, 0, 1]]

out = minimize(calculate_cost_function, x0=[float(random.randint(0,3000))/1000 for i in
print(out)

out_f = [out['x'][0:3], out['x'][3:6], out['x'][6:9]]

circ = QuantumCircuit(3, 3)
apply_fixed_ansatz([0, 1, 2], out_f)

backend = Aer.get_backend('statevector_simulator')

job = execute(circ, backend)

result = job.result()
o = result.get_statevector(circ, decimals=10)

a1 = coefficient_set[2]*np.array([[1,0,0,0,0,0,0,0], [0,1,0,0,0,0,0,0], [0,0,1,0,0,0,0,0],
a0 = coefficient_set[1]*np.array([[1,0,0,0,0,0,0,0], [0,1,0,0,0,0,0,0], [0,0,-1,0,0,0,0,0],
a2 = coefficient_set[0]*np.array([[1,0,0,0,0,0,0,0], [0,1,0,0,0,0,0,0], [0,0,1,0,0,0,0,0],
a3 = np.add(np.add(a2, a0), a1)

b = np.array([float(1/np.sqrt(8)),float(1/np.sqrt(8)),float(1/np.sqrt(8)),float(1/np.sqrt(8))])

print((b.dot(a3.dot(o))/(np.linalg.norm(a3.dot(o))))**2)

```

0.8367103788436497
0.7148710439728692
0.8472916820290619
0.6684217728672097
0.6204540862003505
0.48320953009803347
0.5524018686660065

0.6600568030865614
0.4543924845264439
0.4331738501571404
0.7223098948085269
0.6389624092262558
0.5610966938622768
0.4900673149774074
0.35956600130107197
0.3172619836354691
0.33956024013263075
0.32595505068299924
0.42619099290041396
0.27262379257243075
0.1056197780159075
0.3649759904214662
0.23234497678669153
0.19641858545457236
0.13691397503815317
0.16597120254045572
0.27694802421951803
0.37485142708483155
0.2544151463886435
0.11246143309087209
0.14181356273765167
0.07015901353669018
0.09920239632875139
0.09948761337998635
0.14184169586474749
0.10569052469756868
0.14532575059767083
0.11480759004021734
0.03750339657564061
0.07119901819539054
0.061502804132651856
0.05318717523193128
0.056058380304796374
0.0746784040917482
0.05441230423108967
0.05467688283319805
0.05791668421945817
0.05223278393765918
0.05939272293632325
0.06921505639866765
0.07390071355663874
0.04105308506641303
0.047763651904778714

0.048921939543102866
0.0736336436245173
0.07429398245696428
0.04635093215684494
0.06647371729144658
0.05444129892130434
0.05703802358920218
0.05576537231308454
0.07311688744602385
0.06765398043935089
0.042505138999036296
0.060181400577737354
0.05086337384429751
0.07011936556261777
0.06462638676549326
0.05256482191909062
0.05821651162494279
0.051202925363357576
0.04961269813116076
0.03711896679680893
0.06785395379937675
0.07175636590040035
0.03700981821914373
0.05248495031608247
0.06354018533943195
0.057949648610918336
0.0663240709996149
0.0628206767716748
0.054605112218120255
0.041664816763141266
0.04464293699392341
0.06340368716046885
0.05965617146012514
0.06523211100374937
0.04153988526258423
0.04408866925717747
0.05990492273102621
0.04555626637126731
0.0358077476140638
0.05466741179158696
0.05143602957514515
0.07689244956525798
0.061283749585028024
0.04705052612849048
0.04094344848698461
0.06655007121072642

```
0.055622878557913924
0.07604577707602356
0.0696804883352129
0.03677467713908489
    fun: 0.03677467713908489
    maxcv: 0.0
    message: 'Optimization terminated successfully.'
    nfev: 103
    status: 1
    success: True
    x: array([3.1092751 , 2.84541129, 2.19177542, 2.26913578, 1.03921624,
       2.24045306, 2.50589403, 4.27790656, 2.48093736])
(0.9465619648128146-0j)
```

So as you can see, not amazing, our solution is still off by a fairly significant margin (3.677\% error isn't awful, but ideally, we want it to be **much** closer to 0). Again, I think this is due to the optimizer itself, not the actual quantum circuit. I will be making an update to this Notebook once I figure out how to correct this problem (likely with the introduction of a noisy optimizer, as I previously mentioned).

Acknowledgements

This implementation is based off of the work presented in the research paper "Variational Quantum Linear Solver: A Hybrid Algorithm for Linear Systems", written by Carlos Bravo-Prieto, Ryan LaRose, M. Cerezo, Yiğit Subaşı, Lukasz Cincio, and Patrick J. Coles, which is available at [this link](#).

Special thanks to Carlos Bravo-Prieto for personally helping me out, by answering some of my questions concerning the paper!