

Notes Qiskit UF

Chapter 1

see Qiskit_Notes-Chapter_1-UF_211129

Multiple Qubits and Entanglement

Introduction

Behavior of multi-qubit systems

Most basic multi-qubit gates

Multiple Qubits and Entangled States

Representing Multi_qubit States

Single Qubit: two possible states, two complex amplitudes

Two qubits: four possible states: 00 01 10 11, and to describe the state we require 4 C amplitudes, stored in a 4D-vector:

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

and the role of measurement still work the same way

$$p(|00\rangle) = |\langle 00|a\rangle|^2 = |a_{00}|^2$$

and the normalization holds:

$$|a_{00}|^2 + |a_{01}|^2 + |a_{10}|^2 + |a_{11}|^2 = 1$$

if we have two-qubits we can describe the collective state using the Kronecker product

$$|a\rangle = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}, \quad |b\rangle = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$$

$$|ba\rangle = |b\rangle \otimes |a\rangle = \begin{bmatrix} b_0 \times \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \\ b_1 \times \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} b_0 a_0 \\ b_0 a_1 \\ b_1 a_0 \\ b_1 a_1 \end{bmatrix}$$

with similar things done for multiple qubits if we need to keep track of the 2^n qubits:

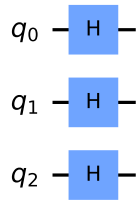
$$|cba\rangle = \begin{bmatrix} c_0 b_0 a_0 \\ c_0 b_0 a_1 \\ c_0 b_1 a_0 \\ c_0 b_1 a_1 \\ c_1 b_0 a_0 \\ c_1 b_0 a_1 \\ c_1 b_1 a_0 \\ c_1 b_1 a_1 \end{bmatrix}$$

```
In [52]: from qiskit import QuantumCircuit, Aer, assemble
from qiskit.circuit import Gate
from qiskit.visualization import plot_histogram, plot_bloch_multivector
import numpy as np
```

```
In [3]: qc = QuantumCircuit(3)
#Apply H-gate to each qubit
for qubit in range(3):
```

```
qc.h(qubit)
qc.draw()
```

Out[3]:



Each qubit is in the $|+\rangle$, so the statevector should be:

$$|++\rangle = \frac{1}{\sqrt{8}} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

```
In [4]: svsim = Aer.get_backend('aer_simulator')
qc.save_statevector()
final_state = svsim.run(qc).result().get_statevector()
# In Jupyter Notebooks we can display this nicely using Latex.
from qiskit.visualization import array_to_latex
array_to_latex(final_state, prefix="\\text{Statevector}=")
```

Out[4]:

$$\text{Statevector} = \left[\frac{1}{\sqrt{8}}, \frac{1}{\sqrt{8}}, \frac{1}{\sqrt{8}}, \frac{1}{\sqrt{8}}, \frac{1}{\sqrt{8}}, \frac{1}{\sqrt{8}}, \frac{1}{\sqrt{8}}, \frac{1}{\sqrt{8}} \right]$$

Quick Exercises:

1. Write down the Kronecker products of qubits

$$|0\rangle|1\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

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

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

$$|-\rangle|+\rangle = \frac{1}{\sqrt{4}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{4}} \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}$$

2. Write the state $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + i|01\rangle)$ as two separate qubits.

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + i|01\rangle) = |0\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$$

In []:

Single Qubit Gates and Multi-Qubit Statevectors

Single gate: $X|0\rangle = |1\rangle$

But how does it act on a multi-qubit statevector?

Namely, how do we represent simultaneous operations (H & X): using the Kronecker product:

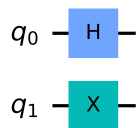
$$X|q_1\rangle \otimes H|q_0\rangle = (X \otimes H)|q_1q_0\rangle$$

$$X \otimes H = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 0 \times \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} & 1 \times \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \\ 1 \times \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} & 0 \times \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 0 & H \\ H & 0 \end{bmatrix}$$

Instead of calculating this by hand, we can use Qiskit's aer_simulator to calculate this for us. The Aer simulator multiplies all the gates in our circuit together to compile a single unitary matrix that performs the whole quantum circuit.

```
In [5]: qc = QuantumCircuit(2)
        qc.h(0)
        qc.x(1)
        qc.draw()
```

Out[5]:



```
In [6]: usim = Aer.get_backend('aer_simulator')
        qc.save_unitary() ##
        unitary = usim.run(qc).result().get_unitary() ## Saves the unitary matrix equivalent to the Kronecker product

        from qiskit.visualization import array_to_latex
        array_to_latex(unitary, prefix="\\text{Circuit}=")
```

Out[6]:

$$\text{Circuit} = \begin{bmatrix} 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \end{bmatrix}$$

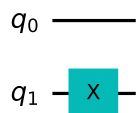
We can also apply one circuit at a time using the identity matrix

$$X \otimes I = \begin{bmatrix} 0 & I \\ I & 0 \end{bmatrix}$$

We can use Qiskit to perform this product:

```
In [7]: qc = QuantumCircuit(2)
        qc.x(1)
        qc.draw()
```

Out[7]:



```
In [8]: usim = Aer.get_backend('aer_simulator')
        qc.save_unitary()
        unitary = usim.run(qc).result().get_unitary()

        array_to_latex(unitary, prefix="\\text{Circuit}=")
```

Out[8]:

$$\text{Circuit} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

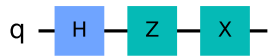
Quick Exercises:

1. Calculate the single qubit unitary U created by the sequence of gates $U = XZH$.

2. Try changing the gates in the circuits above. Calculate the Kronecker product and check using Aer sim.

```
In [9]: qc = QuantumCircuit(1)
        qc.h(0)
        qc.z(0)
        qc.x(0)
        qc.draw()
```

Out[9]:



```
In [10]: usim = Aer.get_backend("aer_simulator")
        qc.save_unitary()
        unitary = usim.run(qc).result().get_unitary()

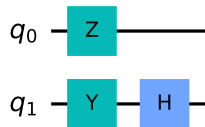
        array_to_latex(unitary, prefix="\\textbf{Single U} = ")
```

Out[10]:

$$\text{Single U} = \begin{bmatrix} -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

```
In [11]: qc = QuantumCircuit(2)
        qc.y(1)
        qc.z(0)
        qc.h(1)
        qc.draw()
```

Out[11]:



```
In [12]: usim = Aer.get_backend('aer_simulator')
        qc.save_unitary()
        unitary = usim.run(qc).result().get_unitary()

        array_to_latex(unitary, prefix="\\text{Test}=")
```

Out[12]:

$$\text{Test} = \begin{bmatrix} \frac{1}{\sqrt{2}}i & 0 & -\frac{1}{\sqrt{2}}i & 0 \\ 0 & -\frac{1}{\sqrt{2}}i & 0 & \frac{1}{\sqrt{2}}i \\ -\frac{1}{\sqrt{2}}i & 0 & -\frac{1}{\sqrt{2}}i & 0 \\ 0 & \frac{1}{\sqrt{2}}i & 0 & \frac{1}{\sqrt{2}}i \end{bmatrix}$$

In []:

Note: Different books, softwares and websites order their qubits differently. This means the kronecker product of the same circuit can look very different. Try to bear this in mind when consulting other sources.

Multi-Qubit Gates

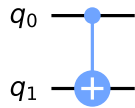
How qubits interact with each other

CNOT-gate

Conditional gate: X-gate on the second qubit (target), if the state of the first qubit is $|1\rangle$: q_0 is control, q_1 target

```
In [13]: qc = QuantumCircuit(2)
          qc.cx(0,1)
          qc.draw()
```

Out[13]:



If qubits are not in superposition, easy as they behave as classical qubits:

Input(t,c)	Output(t,c)
00	00
01	11
10	10
11	01

and has the vector

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \quad \text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

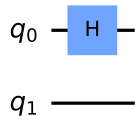
depending on which qubit is the control and which is the target, which varies among references. In our case, the leftmost. This matrix swaps the amplitudes of our statevector:

$$|a\rangle = \begin{bmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{bmatrix}, \quad \text{CNOT}|a\rangle = \begin{bmatrix} a_{00} \\ a_{11} \\ a_{10} \\ a_{01} \end{bmatrix} \begin{matrix} \leftarrow \\ \leftarrow \end{matrix},$$

But **how about a superposition??**

```
In [14]: qc = QuantumCircuit(2)
          qc.h(0)
          qc.draw()
```

Out[14]:



```
In [15]: svsim = Aer.get_backend('aer_simulator')
          qc.save_statevector()
          final_state = svsim.run(qc).result().get_statevector()

          array_to_latex(final_state, prefix="\\text{Statevector = }")
```

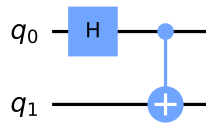
Out[15]:

$$\text{Statevector} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \end{bmatrix}$$

As expected, this produces the state $|0\rangle \otimes |+\rangle = |0+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle)$.

```
In [16]: qc = QuantumCircuit(2)
          qc.h(0)
          qc.cx(0,1)
          qc.draw()
```

Out[16]:



In [17]:

```
qc.save_statevector()
result = svsim.run(qc).result()

final_state = result.get_statevector()
array_to_latex(final_state, prefix="\text{Statevector = }")
```

Out[17]:

$$\text{Statevector} = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} \end{bmatrix}$$

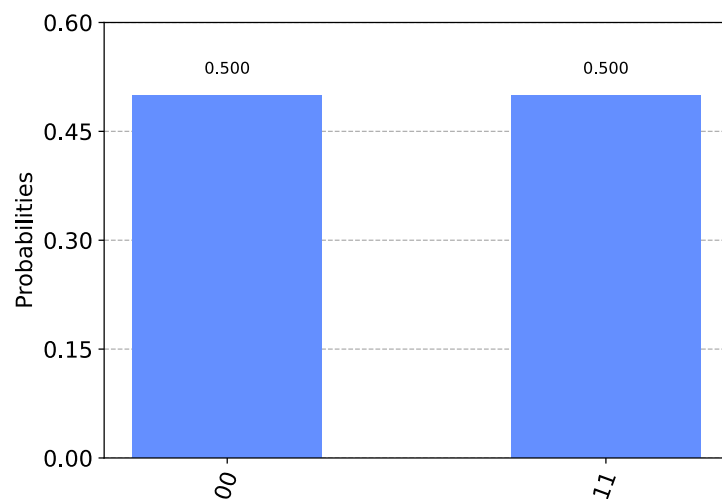
Entangled States

Which corresponds to $\text{CNOT}|0+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, known as a **Bell State** (!!!), with 50% chance of being measured as $|00\rangle$ or $|11\rangle$.

In [18]:

```
plot_histogram(result.get_counts())
```

Out[18]:



This combined state cannot be written as two separate qubit states. Although the qubits are in superposition, measuring one of the states will collapse the other. For example, if we measure the top qubit and get $|1\rangle$, the collective state of our system changes to:

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \xrightarrow{\text{measure}} |11\rangle$$

but random: *no-communication theorem*, Asher Peres, Daniel R. Terno, Quantum Information and Relativity Theory, 2004, <https://arxiv.org/abs/quant-ph/0212023>.

In []:

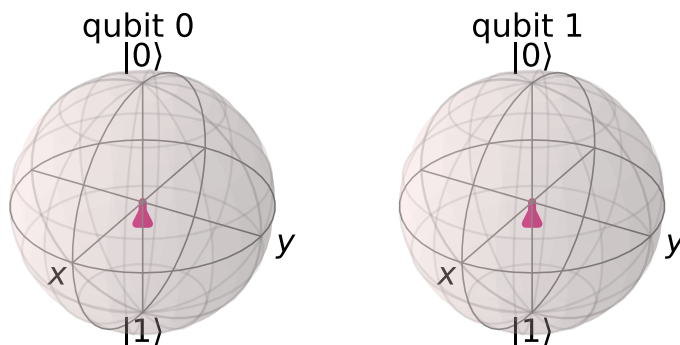
Visualizing Entangled States

We have seen that the state above cannot be written as two separate qubits, meaning we lose information if we try to plot our separate states on Bloch spheres:

In [19]:

```
plot_bloch_multivector(final_state)
```

Out[19]:



No specific measurement for which a specific result is guaranteed, in any basis, in contrast with single qubits. We miss the important correlation between the qubits. In

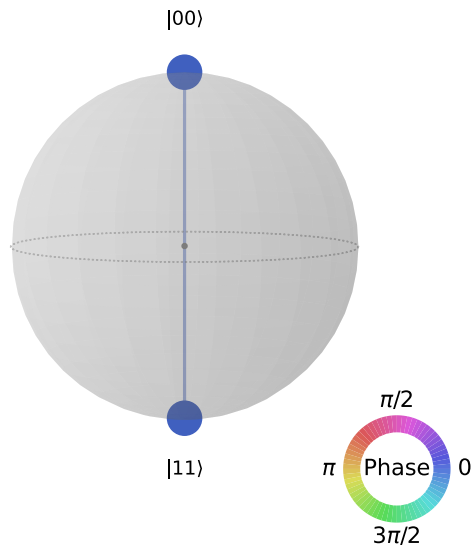
fact, both $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ and $\frac{1}{\sqrt{2}}(|10\rangle + |01\rangle)$ will look the same, despite being very different states, with different measurement outcomes.

Alternative visualization: **Q-sphere**.

- Each amplitude is represented by a blob on the surface of a sphere
- size \propto magnitude of the amplitude
- color \propto phase of the amplitude

```
In [20]: from qiskit.visualization import plot_state_qsphere
         plot_state_qsphere(final_state)
```

Out[20]:



Exercises:

1. Create a quantum circuit that produces the Bell state: $\frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$. Use the statevector simulator to verify your result.

$$|0\rangle \otimes |0\rangle \rightarrow H|0\rangle \otimes X|0\rangle = |+\rangle \otimes |1\rangle \rightarrow CNOT(|+\rangle \otimes |1\rangle) = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$

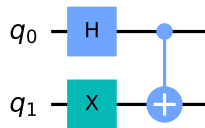
2. The circuit above transforms $|00\rangle$ to $\frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$. Calculate the unitary using the Qiskit simulator. Verify that this unitary perform the correct transformation.

$$\frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & -1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

3. Think about other ways you could represent a statevector visually. Can you design an interesting visualization from which you can read the magnitude and phase of each amplitude?

```
In [21]: qc = QuantumCircuit(2)
         qc.x(1)
         qc.h(0)
         qc.cx(0,1)
         qc.draw()
```

Out[21]:



```
In [22]: qc.save_statevector()
result = svsim.run(qc).result()

final_state = result.get_statevector()
array_to_latex(final_state, prefix="\\text{Statevector} = ")
```

Out[22]:

$$\text{Statevector} = \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \end{bmatrix}$$

To obtain the unitary

```
In [23]: qc = QuantumCircuit(2)
qc.x(1)
qc.h(0)
qc.cx(0,1)

usim = Aer.get_backend('aer_simulator')
qc.save_unitary()
unitary = usim.run(qc).result().get_unitary()

array_to_latex(unitary, prefix="\\text{Unitary} = ")
```

Out[23]:

$$\text{Unitary} = \begin{bmatrix} 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

```
In [24]: #qc = QuantumCircuit(2)
#unitary.
```

In []:

Phase Kickback

Exploring the CNOT-gate

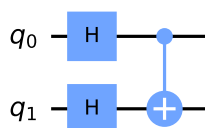
In the previous section, we saw that the CNOT gate can entangle states,

$$\text{CNOT} |0+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

What if we put the second qubit in a superposition as well?

```
In [25]: qc = QuantumCircuit(2)
qc.h(0)
qc.h(1)
qc.cx(0,1)
qc.draw()
```

Out[25]:



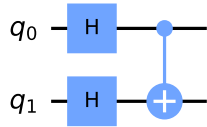
$$\text{CNOT} |++\rangle = \frac{1}{\sqrt{2}}\text{CNOT}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle + |10\rangle + |01\rangle) = |++\rangle$$

In [26]:


```
qc = QuantumCircuit(2)
qc.h(0)
qc.h(1)
qc.cx(0,1)
display(qc.draw())

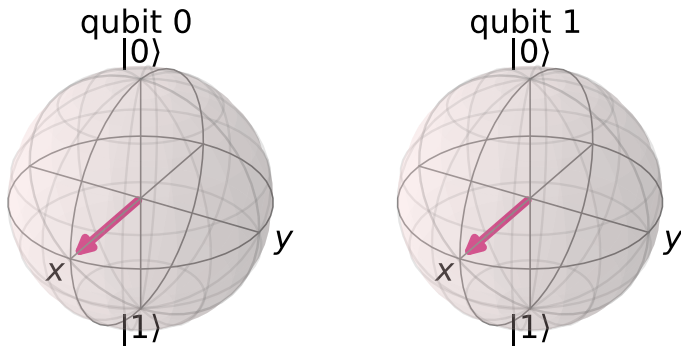
svsim = Aer.get_backend('aer_simulator')
qc.save_statevector()
final_state = svsim.run(qc).result().get_statevector()
display(array_to_latex(final_state, prefix="\text{Statevector} = "))

plot_bloch_multivector(final_state)
```



$$\text{Statevector} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

Out[26]:



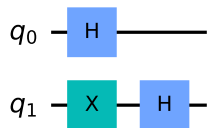
If we now put the target on the $|-\rangle$ state, it will create the state:

$$\text{CNOT} |-\rangle = \frac{1}{\sqrt{2}} \text{CNOT}(|00\rangle + |01\rangle - |10\rangle - |11\rangle) = \frac{1}{\sqrt{2}}(|00\rangle - |01\rangle - |10\rangle + |11\rangle) = |--\rangle$$

In [27]:

```
qc = QuantumCircuit(2)
qc.h(0)
qc.x(1)
qc.h(1)
qc.draw()
```

Out[27]:

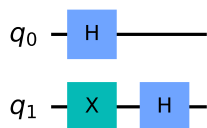


In [28]:

```
qc = QuantumCircuit(2)
qc.h(0)
qc.x(1)
qc.h(1)
display(qc.draw())

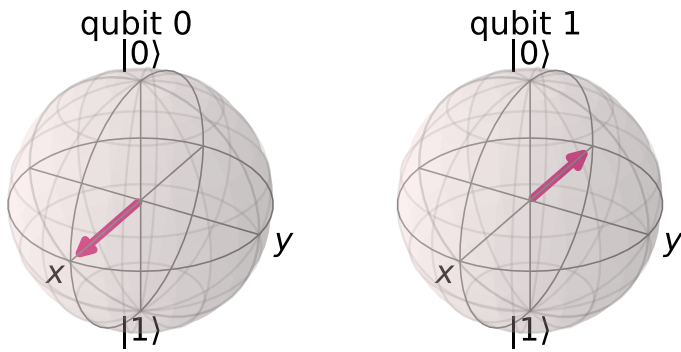
svsim = Aer.get_backend('aer_simulator')
qc.save_statevector()
final_state = svsim.run(qc).result().get_statevector()
display(array_to_latex(final_state, prefix="\text{Statevector} = "))

plot_bloch_multivector(final_state)
```



$$\text{Statevector} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} \end{bmatrix}$$

Out[28]:



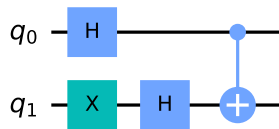
Notice that CNOT $| - + \rangle = | - - \rangle$ affects the state of the *control* qubit while leaving the target unchanged.

In [29]:

```
qc = QuantumCircuit(2)
qc.h(0)
qc.x(1)
qc.h(1)
qc.cx(0,1)
display(qc.draw())

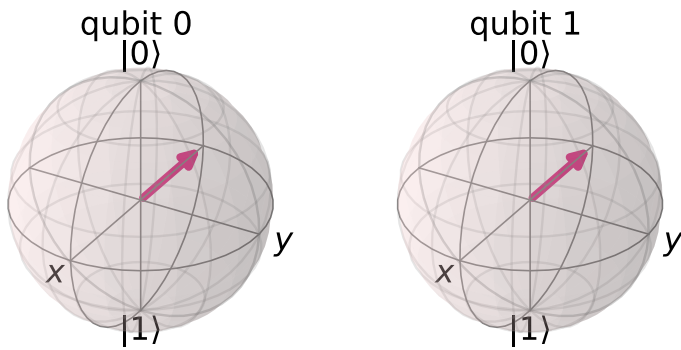
qc.save_statevector()
final_state = svsim.run(qc).result().get_statevector()
display(array_to_latex(final_state, prefix="\text{Statevector} = "))

plot_bloch_multivector(final_state)
```

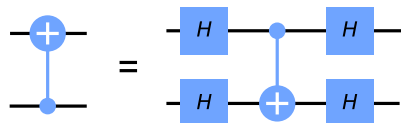


$$\text{Statevector} = \begin{bmatrix} \frac{1}{2} & -\frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

Out[29]:



In the same way that H transforms $| + \rangle \rightarrow | 0 \rangle$ and $| - \rangle \rightarrow | 1 \rangle$, wrapping the CNOT in H-gates is equivalent to switching control and target qubits:

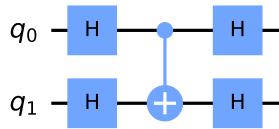


Let's verify this using the Qiskit Ar simulator:

In [30]:

```
qc = QuantumCircuit(2)
qc.h(0)
qc.h(1)
qc.cx(0,1)
qc.h(0)
qc.h(1)
display(qc.draw())
```

```
qc.save_unitary()
usim = Aer.get_backend('aer_simulator')
unitary = usim.run(qc).result().get_unitary()
array_to_latex(unitary, prefix="\\text{U} =")
```



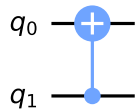
Out[30]:

$$U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

In [31]:

```
qc = QuantumCircuit(2)
qc.cx(1,0)
display(qc.draw())

qc.save_unitary()
usim = Aer.get_backend("aer_simulator")
unitary2 = usim.run(qc).result().get_unitary()
array_to_latex(unitary2, prefix="\\text{U2} =")
```



Out[31]:

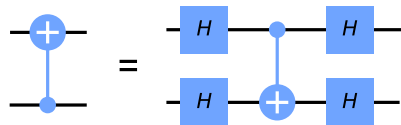
$$U2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

In []:

Phase Kickback

Explaining the CNOT Circuit Identity

We just saw that



This is an example of **kickback** (or **phase kickback**), which is very important and used in almost every quantum algorithm. *Kickback is where the eigenvalue added by a gate is 'kicked back' into a different operation via a **controlled** operation.*

For example,

$$X|-\rangle = -|-\rangle$$

If you control qubit is $|0\rangle$ or $|1\rangle$, it adds an overall, global phase, with no observable effects:

$$\text{CNOT}|-\rangle|0\rangle = |-\rangle \otimes |0\rangle = |-\rangle|0\rangle$$

$$\text{CNOT}|-\rangle|1\rangle = X|-\rangle \otimes |1\rangle = -|-\rangle|1\rangle$$

but an interesting effect happens when the state is in superposition.

$$\text{CNOT}|-\rangle|+\rangle = \frac{1}{\sqrt{2}}(\text{CNOT}|-\rangle|0\rangle + \text{CNOT}|-\rangle|1\rangle) = \frac{1}{\sqrt{2}}(|-\rangle|0\rangle - |-\rangle|1\rangle)$$

That is,

$$\text{CNOT}|-+\rangle = |- \rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |- - \rangle$$

Wrapping the CNOT in H-gates transforms the computation basis $\{|0\rangle, |1\rangle\}$ to $\{|+\rangle, |-\rangle\}$, which can be useful for hardware.

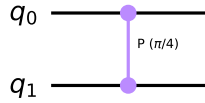
Kickback with the T-gate

Controlled T-gate:

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$$

```
In [32]: qc = QuantumCircuit(2)
qc.cp(np.pi/4,0,1) #phase, control, target
qc.draw()
```

Out[32]:



Controlled-T:

$$\text{Controlled-T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\pi/4} \end{bmatrix}$$

```
In [33]: qc = QuantumCircuit(2)
qc.cp(np.pi/4,0,1)

qc.save_unitary()
usim = Aer.get_backend('aer_simulator')
unitary = usim.run(qc).result().get_unitary()
array_to_latex(unitary,prefix="\text{Controlled-T} = ")
```

Out[33]:

$$\text{Controlled-T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{\sqrt{2}}(1+i) \end{bmatrix}$$

More generally, one can find the matrix for any controlled-U operation using the rule:

$$U = \begin{bmatrix} u_{00} & u_{01} \\ u_{10} & u_{11} \end{bmatrix}$$

$$\text{Controlled-U} = \begin{bmatrix} I & 0 \\ 0 & U \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} \\ 0 & 0 & u_{10} & u_{11} \end{bmatrix}$$

or, using Qiskit's ordering

$$\text{Controlled-U} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & u_{00} & 0 & u_{01} \\ 0 & 0 & 1 & 0 \\ 0 & u_{10} & 0 & u_{11} \end{bmatrix}$$

Remember that applying the T-gate in the state $|1\rangle$ adds a phase $e^{i\pi/4}$, $T|1\rangle = e^{i\pi/4}|1\rangle$, which is global and unobservable.

But if we control using $|+\rangle$:

$$|1+\rangle = |1\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}}(|10\rangle + |11\rangle)$$

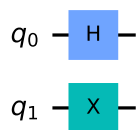
$$\text{Controlled-T}|1+\rangle = \frac{1}{\sqrt{2}}(|10\rangle + e^{i\pi/4}|11\rangle) = |1\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle + e^{i\pi/4}|1\rangle)$$

Which rotates our control qubit around the Z-axis of the Bloch sphere, leaving the target unchanged.

```
In [34]: ## Original
qc = QuantumCircuit(2)
qc.h(0)
qc.x(1)
display(qc.draw())

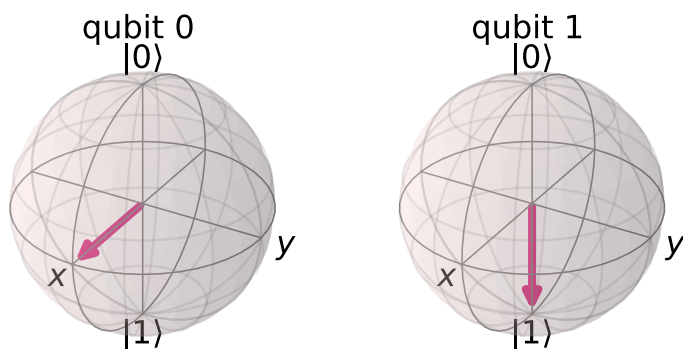
qc.save_statevector()
svsim = Aer.get_backend('aer_simulator')
final_state = svsim.run(qc).result().get_statevector()
display(array_to_latex(final_state, prefix="\\text{Statevector}"))

plot_bloch_multivector(final_state)
```



$$\text{Statevector} \begin{bmatrix} 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

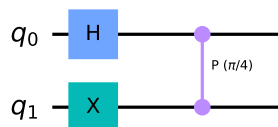
Out[34]:



```
In [35]: #Controlled
qc = QuantumCircuit(2)
qc.h(0)
qc.x(1)
qc.cp(np.pi/4, 0, 1)
display(qc.draw())

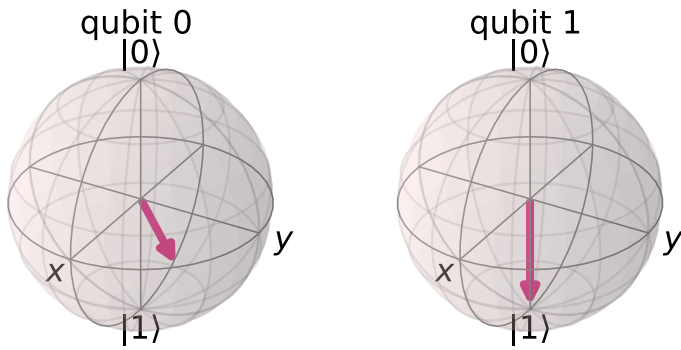
qc.save_statevector()
svsim = Aer.get_backend('aer_simulator')
final_state = svsim.run(qc).result().get_statevector()
display(array_to_latex(final_state, prefix="\\text{Statevector}"))

plot_bloch_multivector(final_state)
```

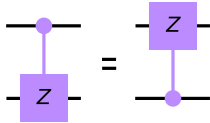


$$\text{Statevector} \begin{bmatrix} 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{2}(1+i) \end{bmatrix}$$

Out[35]:



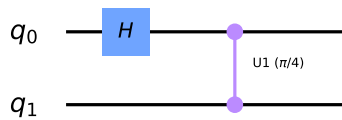
After exploring this behaviour, it may become clear why Qiskit draws the controlled-Z rotation gates in this symmetrical fashion (two controls instead of a control and a target). There is no clear control or target qubit for all cases.



In []:

Exercises:

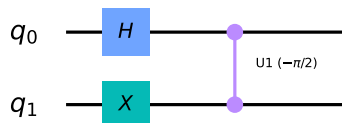
1 What would be the resulting state of the control qubit (q0) if the target qubit (q1) was in the state $|0\rangle$? (as shown in the circuit below)? Use Qiskit to check your answer.



$$\text{Controlled-T} |0+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle) = |0\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |0+\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

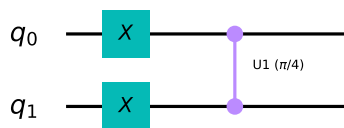
no change

2 What would be the resulting state of the control qubit (q0) if the target qubit (q1) was in the state $|1\rangle$, and the circuit used a controlled-Sdg gate instead of the controlled-T (as shown in the circuit below)?



$$\text{Controlled-Sdg} |1+\rangle = \frac{1}{\sqrt{2}}(|10\rangle + e^{-i\pi/2}|11\rangle) = |1\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle + e^{-i\pi/2}|1\rangle) = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 0 \\ 1 \\ -i \end{bmatrix}$$

3 What would happen to the control qubit (q_0) if it was in the state $|1\rangle$ instead of the state $|+\rangle$ before application of the controlled-T (as shown in the circuit below)?



$$\text{Controlled-T} |11\rangle = e^{i\pi/4} |11\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1+i \end{bmatrix}$$

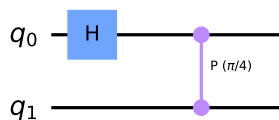
overall phase

```
In [36]: #1
qc = QuantumCircuit(2)
qc.h(0)
qc.cp(np.pi/4,0,1)
display(qc.draw())

qc.save_statevector()
svsim = Aer.get_backend('aer_simulator')

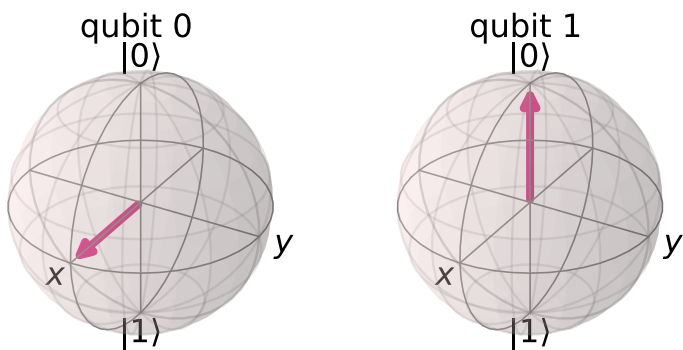
final_state = svsim.run(qc).result().get_statevector()
display(array_to_latex(final_state,prefix="\text{SV} ="))

plot_bloch_multivector(final_state)
```



$$\text{SV} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \end{bmatrix}$$

Out[36]:

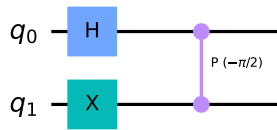


```
In [37]: #2
qc = QuantumCircuit(2)
qc.h(0)
qc.x(1)
qc.cp(-np.pi/2,0,1)
display(qc.draw())

qc.save_statevector()
svsim = Aer.get_backend('aer_simulator')

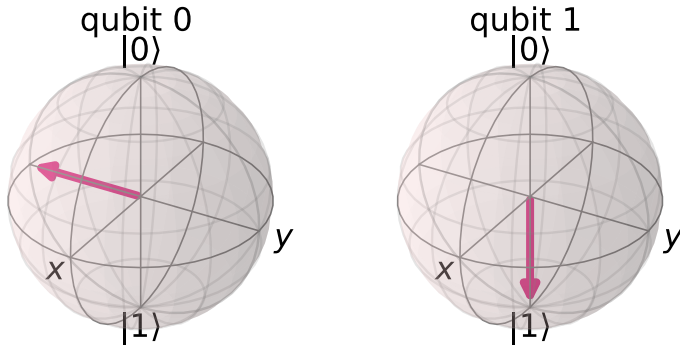
final_state = svsim.run(qc).result().get_statevector()
display(array_to_latex(final_state,prefix="\text{SV} ="))

plot_bloch_multivector(final_state)
```



$$SV = \begin{bmatrix} 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}i \end{bmatrix}$$

Out[37]:



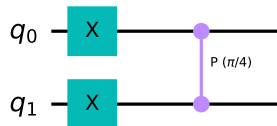
In [38]:

```
#3
#1
qc = QuantumCircuit(2)
qc.x(0)
qc.x(1)
qc.cp(np.pi/4,0,1)
display(qc.draw())

qc.save_statevector()
svsim = Aer.get_backend('aer_simulator')

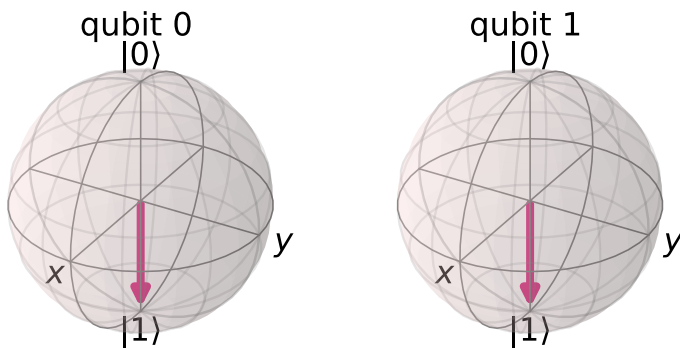
final_state = svsim.run(qc).result().get_statevector()
display(array_to_latex(final_state,prefix="\text{SV} ="))

plot_bloch_multivector(final_state)
```



$$SV = \begin{bmatrix} 0 & 0 & 0 & \frac{1}{\sqrt{2}}(1+i) \end{bmatrix}$$

Out[38]:



In []:

More Circuit Identities

See Barenco + 1995, "Elementary gates for quantum computation", <https://journals.aps.org/pr/abstract/10.1103/PhysRevA.52.3457>

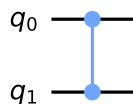
In this section, we'll look at how we can transform basic gates into each other, and how to use them to build some gates that are slightly more complex (but still pretty basic).

Making a Controlled-Z from a CNOT

Controlled-Z, or `cz`, gate. Just as a CNOT applies an X to the target whenever the control is $|1\rangle$, `cz` applies a Z.

```
In [39]: qc = QuantumCircuit(2)
          qc.cz(0,1)
          qc.draw()
```

Out[39]:



However, in IBM Q devices, the only kind of two-qubit gate that can be applied is a CNOT. Therefore, we need a way to transform among them.

We know that

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

and

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

From where it follows that

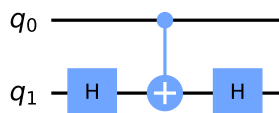
$$\begin{aligned} HXH|+\rangle &= HX|0\rangle = H|1\rangle = |-\rangle, HXH|-\rangle = HX|1\rangle = H|0\rangle = |+\rangle \longrightarrow HXH = Z \\ HZH|0\rangle &= HZ|+\rangle = H|-\rangle = |1\rangle, HZH|1\rangle = HZ|-\rangle = H|+\rangle = |0\rangle \longrightarrow HZH = X \end{aligned}$$

and the same trick can be used to transform a CNOT into a controlled-Z surrounding the target with Hadamards.

$$Hc_zH = CNOT, Hc_xH = CZ$$

```
In [40]: qc = QuantumCircuit(2)
          qc.h(1)
          qc.cx(0,1)
          qc.h(1)
          qc.draw()
```

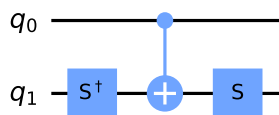
Out[40]:



More generally, a single CNOT can be transformed into any rotation around the Bloch sphere by an angle π by preceding and following by the right rotations. Ex: Controlled-Y:

```
In [41]: qc = QuantumCircuit(2)
          ##Controlled Y
          qc.sdg(1)
          qc.cx(0,1)
          qc.s(1)
          qc.draw()
```

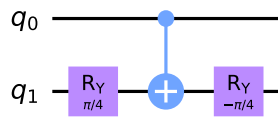
Out[41]:



and Controlled-H (???)

```
In [42]: qc = QuantumCircuit(2)
          ##Controlled Y
          qc.ry(np.pi/4, 1)
          qc.cx(0,1)
          qc.ry(-np.pi/4, 1)
          qc.draw()
```

Out[42]:



In []:

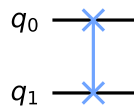
Swapping Qubits

Sometimes we need to move information around in the QC. **Move the states with SWAP gate.**

In [43]:

```
qc = QuantumCircuit(2)
qc.swap(0,1)
qc.draw()
```

Out[43]:



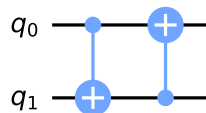
This invokes the gate directly, but how is it physically created?

Let's say qubit a is in state $|1\rangle$ and qubit b in $|0\rangle$:

In [45]:

```
qc = QuantumCircuit(2)
#swap a 1 from a to b
qc.cx(0,1) # copies 1 from a to b
qc.cx(1,0) #uses the 1 on b to rotate a to 0
qc.draw()
```

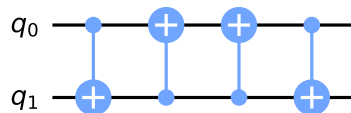
Out[45]:



In [46]:

```
#which can be reversed
qc.cx(1,0) # copies 1 from b to a
qc.cx(0,1) #uses the 1 on a to rotate b to 0
qc.draw()
```

Out[46]:

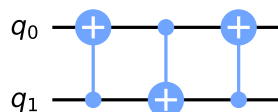


Notice that the first gate of each qubit does not affect the initial state of the control qubit, and the same for the last. We can, therefore, add an ineffective gate from each one:

In [47]:

```
qc = QuantumCircuit(2)
# swaps states of qubits a and b
qc.cx(1,0)
qc.cx(0,1)
qc.cx(1,0)
qc.draw()
```

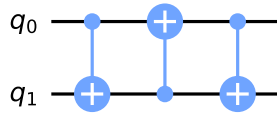
Out[47]:



There is one ineffective gate for each qubit, but it swaps both around. Since it works for all states in the computational basis, **it works for all states**. We can also change the order of the qubits:

```
In [48]: qc = QuantumCircuit(2)
# swaps states of qubits a and b
qc.cx(0,1)
qc.cx(1,0)
qc.cx(0,1)
qc.draw()
```

Out[48]:



Exercise

Find a different circuit that swaps qubits in the states $|+\rangle$ and $|-\rangle$, and show that this is equivalent to the circuit shown above.

In []:

Controlled Rotations

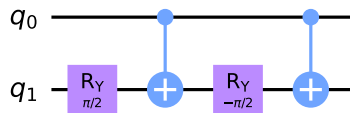
Before: π rotations from a single CNOT (controlled-Y). How about any controlled rotations?

First, arbitrary rotations around the y-axis:

```
In [50]: c=0
t=1

qc = QuantumCircuit(2)
theta = np.pi
qc.ry(theta/2, t)
qc.cx(c,t)
qc.ry(-theta/2,t)
qc.cx(c,t)
qc.draw()
```

Out[50]:



If the control state is $|1\rangle$, $\text{qc.ry}(-\text{theta}/2, t)$ is preceded by an X-gate, flipping the direction of the Y-rotation and making a second $R_Y(\theta/2)$, for a total rotation $R_Y(\theta)$. It works because X and Y are orthogonal, and therefore it also works for $R_Z(\theta)$.

For a controlled version of any single qubit rotation, V , we need three rotations A, B, C and a phase α such that

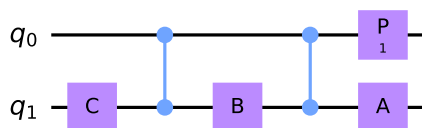
$$ABC = I \quad \text{for control-qubit} = |0\rangle \text{ with controlled-Z} \quad e^{i\alpha}AZBZC = V \quad \text{for control-qubit} = |1\rangle \text{ with controlled-Z.}$$

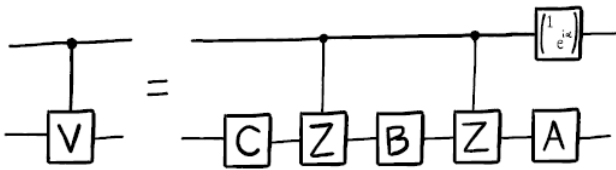
An $R_Z(2\alpha)$ is also used to control the right phase in superposition states.

```
In [53]: A = Gate('A', 1, []) #name, num qubits, params???
B = Gate('B', 1, [])
C = Gate('C', 1, [])
alpha=1 # arbitrary alpha for drawing
```

```
In [54]: qc = QuantumCircuit(2)
qc.append(C,[t]) ## Append gate to circuit
qc.cz(c,t)
qc.append(B, [t])
qc.cz(c,t)
qc.append(A,[t])
qc.p(alpha,c)
qc.draw()
```

Out[54]:





In []:

In []:

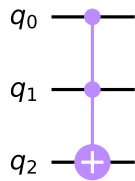
The Toffoli

The Toffoli gate is a three-qubit gate with two controls and one target: **CCNOT** or **CCX** gate.

X on the target only if the two controls are $|1\rangle$.

```
In [55]: qc = QuantumCircuit(3)
a=0
b=1
t=2
# Toffoli with control qubits a and b and target t
qc.ccx(a,b,t)
qc.draw()
```

Out[55]:

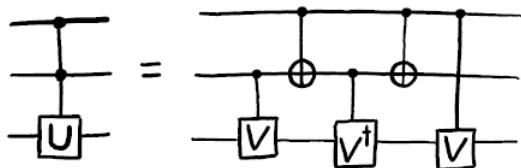
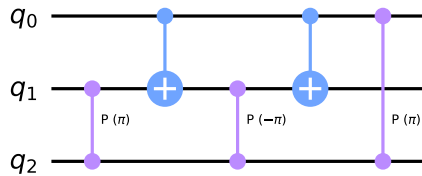


How to build an arbitrary controlled-controlled-U gate for any single rotation U ?

Define controlled versions of $V = \sqrt{U}$ and V^\dagger . (Below, we use $\text{cp}(\theta, c, t)$ and $\text{cp}(-\theta, c, t)$ in place of undefined subroutines cv and cvdg).

```
In [56]: qc = QuantumCircuit(3)
qc.cp(theta, b, t)
qc.cx(a,b)
qc.cp(-theta, b, t)
qc.cx(a,b)
qc.cp(theta, a, t)
qc.draw()
```

Out[56]:



It turns out that the minimum number of CNOT gates required to implement the Toffoli gate is six [Shende and Markov 2009] <https://arxiv.org/abs/0803.2316>

irrationality in action by applying the gate. Keeping in mind that every time we apply a rotation that is larger than 2π , we are doing an implicit modulus by 2π on the rotation angle. Thus, repeating the combined rotation mentioned above n times results in a rotation around the same axis by a different angle. As a hint to a rigorous proof, recall that an irrational number cannot be written as what?

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 due to the pigeonhole principle. 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

$$\theta_{n_2-n_1} \neq 0, \quad -\frac{2\pi}{n} \leq \theta_{n_2-n_1} \leq \frac{2\pi}{n}.$$

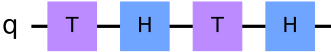
and therefore we can do rotations around small angles by just increasing n , to an accuracy of $2\pi/n$.

To do that on a second axis, we simply do $R_Z(\pi/4)$ and $R_X(\pi/4)$ in the opposite order:

In [61]:

```
qc = QuantumCircuit(1)
qc.t(0)
qc.h(0)
qc.t(0)
qc.h(0)
qc.draw()
```

Out[61]:



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.

In []:

Proving Universality

Introduction

What can any given computer do? What are the limits of what is deemed computable, in general?

To ask this question of 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 corresponding outputs, we call it universal.

Quantum computers similarly take input states and convert them into output states. We will therefore be able to define universality in a similar way. To be more precise, and to be able to prove when universality can and cannot be achieved, it is useful to use the matrix representation of our quantum gates.

Fun with matrices

Matrices as outer products

Inner products: $\langle 0|0\rangle = 1$. **Number**

Outer product: $||0\rangle\langle 0| =$

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

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

$\end{pmatrix}$

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

,

$|0\rangle\langle 1| =$

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

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

$\end{pmatrix}$

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

,

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

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

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

$\end{pmatrix}$

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

,

$$|1\rangle\langle 1| =$$

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

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

$\end{pmatrix}$

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

. \$\$

and therefore we can write:

$$M = \begin{pmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{pmatrix} = m_{00}|0\rangle\langle 0| + m_{01}|0\rangle\langle 1| + m_{10}|1\rangle\langle 0| + m_{11}|1\rangle\langle 1|$$

Unitary and Hermitian matrices

Hermitian conjugate: $M^\dagger = (M^T)^*$

Unitary matrices: $UU^\dagger = U^\dagger U = I \longrightarrow \text{Hermitian} = \text{Inverse}$

Hermitian matrices: $M = M^\dagger$, such as X, Y, Z , and H

In QC, all but measurement and rest ops are unitary.

Unitary preserve inner product: $(\langle \phi_0 | U^\dagger)(U | \phi_1 \rangle) = \langle \phi_0 | U^\dagger U | \phi_1 \rangle = \langle \phi_0 | \phi_1 \rangle$

\longrightarrow for any orthonormal basis $\{|\psi_i\rangle\}$, the set of states $\{|\phi_i\rangle = U|\psi_i\rangle\}$ is also an orthonormal basis, U serving as a rotation between these basis.

$\longrightarrow U = \sum_j |\phi_j\rangle\langle\psi_j|$: quantum version of truth tables.

All unitary and hermitian matrices are diagonalizable: $M = \sum_j \lambda_j |h_j\rangle\langle h_j|$, λ_j : eigenvalues, $|h_j\rangle$ eigenvalues.

\longrightarrow for **unitary**: $UU^\dagger = I \rightarrow \lambda_j \lambda_j^* = 1, \lambda_j \in \mathbb{C}$, which can be expressed as e^{ih_j} , $h_j \in \mathbb{R}$

\longrightarrow for **hermitian**: $H = H^\dagger \rightarrow \lambda_j = \lambda_j^* \rightarrow \lambda_j \in \mathbb{R}$

They only differ in that $\lambda_j \in \text{CorR}$, meaning that _for every unitary, we can define a corresponding Hermitian matrix (and vice-versa), using the same eigenstates h_j from e^{ih_j} _.

\longrightarrow unitary from Hermitian: $U = e^{iH}$ (matrix exponentiation).

Pauli decomposition

Recall all matrices can be written in terms of outer products.

$$M = \begin{pmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{pmatrix} = m_{00}|0\rangle\langle 0| + m_{01}|0\rangle\langle 1| + m_{10}|1\rangle\langle 0| + m_{11}|1\rangle\langle 1|$$

We will show they can all be written in terms of Pauli matrices, given that

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

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

$$|0\rangle\langle 1| = |0\rangle\langle 0|X = \frac{(1+Z)X}{2} = \frac{X+iY}{2}$$

$$|1\rangle\langle 0| = X|0\rangle\langle 0| = \frac{X(1+Z)}{2} = \frac{X-iY}{2}$$

and we can now write M as $M = \frac{1}{2} \begin{pmatrix} m_{00}+m_{11} & m_{01}-m_{10} \\ m_{01}+m_{10} & m_{00}-m_{11} \end{pmatrix}$

- $\frac{1}{2} (m_{01}+m_{10}) X$
- $\frac{i}{2} (m_{01}-m_{10}) Y$
- $\frac{1}{2} (m_{00}-m_{11}) Z$

While this applies for a general **two-qubit** matrix, it applies for any number of qubits:

$$|00\dots 0\rangle\langle 0\dots 00| = \left(\frac{1+Z}{2}\right) \otimes \left(\frac{1+Z}{2}\right) \dots \otimes \left(\frac{1+Z}{2}\right)$$

etc, and any matrix can be expressed as a tensor product of Pauli matrices.

Defining Universality

Just as each quantum gate can be represented by a unitary, so too **can we describe an entire quantum computation by a (very large) unitary operation**. The effect of this is to rotate the input state to the output state.

Mapping of inputs $|x\rangle$ into outputs $|f(x)\rangle$ can be described by a *reversible* computation $U = \sum |f(x)\rangle\langle x|$.

Another special case is that the input and output states could describe a physical system, and the computation we perform is to simulate the dynamics of that system. This is an important problem that is impractical for classical computers, but is a natural application of quantum computers. *The time evolution of the system in this case corresponds to the unitary that we apply, and the associated Hermitian matrix is the Hamiltonian of the system.* Achieving any unitary would therefore correspond to simulating any time evolution, and engineering the effects of any Hamiltonian.

Combining these insights we can define what it means for quantum computers to be universal. It is simply **the ability to achieve any desired unitary on any arbitrary number of qubits**. If we have this, we know that we can reproduce anything a digital computer can do, simulate any quantum system, and do everything else that is possible for a quantum computer.

Basic gate sets

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 single- and two-qubit gates, most of which correspond to the set of so-called **Clifford gates**. This is a very important set of operations, which do a lot of the heavy-lifting in any quantum algorithm.

Clifford Gates

Hadamard: $H = \frac{1}{\sqrt{2}}(|+\rangle\langle 0| + |-\rangle\langle 1|) = \frac{1}{\sqrt{2}}(|0\rangle\langle +| + |1\rangle\langle -|)$.

→ moves info available for an X measurement to a Z measurement.

Combinations for different operations:

$$HXH = ZHZH = X$$

S-Gate: similar constructions, swaps X and Y (H swaps X and Z)

$$SXS^\dagger = YSYS^\dagger = -XSZS^\dagger = Z$$

Combining S and H , could go from Y to Z .

Pauli: similar, but only adding phase

$$ZXZ = -XZY = -YZZ = Z$$

if we combine them with S , we can make the phase go away.

Multi-qubit Clifford gates: they transform tensor products of Paulis to other tensor products of Paulis. Example: CNOT

$$CX_{j,k}(X \otimes I)CX_{j,k}^\dagger = X \otimes X$$

"copies" an X from the control qubit over 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.

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 $R_{X,Y,Z}(\theta)$.

Ex: $R_X(\theta)$:

$$R_x(\theta) = e^{i\frac{\theta}{2}X}$$

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**. 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.

Expanding the Gate Set

As another example, let's conjugate $R_x(\theta)$ with CNOT:

$$CX_{j,k}(R_x(\theta) \otimes I)CX_{j,k} = CX_{j,k}e^{i\frac{\theta}{2}(X \otimes I)}CX_{j,k} = e^{i\frac{\theta}{2}CX_{j,k}(X \otimes I)CX_{j,k}} = e^{i\frac{\theta}{2}X \otimes X}$$

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**.

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 on the qubit on the right to turn the X there into a Y :

$$(I \otimes S)e^{i\frac{\theta}{2}X \otimes X}(I \otimes S^\dagger) = e^{i\frac{\theta}{2}X \otimes Y}$$

With these techniques, we can make complex entangling operations that act on any arbitrary number of qubits, and combining the single and two-qubit Clifford gates with rotations around the x axis gives us a powerful set of possibilities

Proving Universality

Suppose we wish to implement the unitary:

$$U = e^{i(aX+bZ)}$$

but the only gates available are $R_x(\theta)$ and $R_z(\theta)$. (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 U is simply the sum of the rotations, but since exponentiation of matrices do not commute, we cannot simply apply $R_z(2b) = e^{ibZ}$ and then $R_x(2a) = e^{iaX}$, because $e^{iaX}e^{ibZ} \neq e^{i(aX+bZ)}$.

We could, however, "linearize" the problem having in mind that:

$$U = \lim_{n \rightarrow \infty} \left(e^{iaX/n} e^{ibZ/n} \right)^n, \quad \text{for } e^{iaX/n} e^{ibZ/n} \approx e^{i(aX+bZ)}, \quad O(1/n^2)$$

and by combining the n slices, approximation error scales as $1/n$.

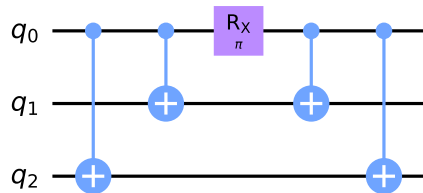
This can be used for single qubits and for more complex cases too. For instance:

$$U = e^{i(aX \otimes X \otimes X + bZ \otimes Z \otimes Z)}$$

We do know how to create $e^{i\frac{\theta}{2}X \otimes X \otimes X}$ and, using a few Hadamards, $e^{i\frac{\theta}{2}Z \otimes Z \otimes Z}$.

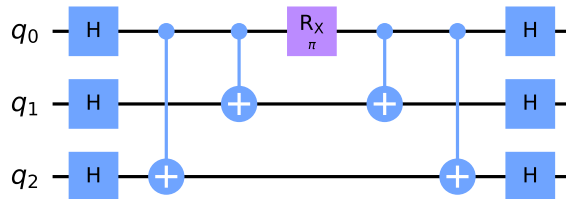
```
In [66]: qc = QuantumCircuit(3)
qc.cx(0,2)
qc.cx(0,1)
qc.rx(theta,0)
qc.cx(0,1)
qc.cx(0,2)
qc.draw()
```

Out[66]:



```
In [69]: qc = QuantumCircuit(3)
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(0,2)
qc.h(2)
qc.h(1)
qc.h(0)
qc.draw()
```

Out[69]:



This gives us the ability to reproduce a small slice of our new, three-qubit U :

$$e^{i(aX \otimes X \otimes X)/n} e^{i(bZ \otimes Z \otimes Z)/n} \approx e^{i(aX \otimes X \otimes X + bZ \otimes Z \otimes Z)/n}$$

which we can then combine the slices together to get an arbitrarily accurate approximation of U .

This gate set is not the only one that can achieve universality. For example **it can be shown that just the Hadamard and Toffoli are sufficient for universality**. Multiple other gates sets have also been considered and been proven universal, each motivated by different routes toward achieving the gates fault-tolerantly.

Everything we have discussed in this book follows the circuit model of computation. However, the circuit model is not the only universal model of quantum computation. Other forms of quantum computation such as **adiabatic quantum computing** or **measurement based quantum computing** exist. The fact that they are universal means that it has been proven that there is a mapping in polynomial time and resources from the circuit model to these other models of computation.

There are other forms of quantum computation that are not universal, but are applicable to specific applications. For example **quantum annealing** may be useful for optimization and sampling problems. Annealing is the process of heating a metal to a high temperature and then allowing it to cool down slowly. This process causes molten metal to flow over the surface of the metal piece and redistribute itself; changing many properties of the metal in question. Quantum annealing is analogous to the physical process of annealing in some sense. It involves encoding problems into an energy landscape of sorts and then letting a quantum state explore the landscape. While normal waves may get trapped in troughs which are lower than their surroundings (local minima), quantum effects increase the speed at which the quantum states find the true lowest point on the landscape (global minima).

In []:

Classical Computation on a Quantum Computer

Introduction

One consequence of having a universal set of quantum gates is the ability to reproduce any classical computation. Quantum computers can do anything that a classical computer can do, and they can do so with at least the same computational complexity.

Problems that require quantum solutions often involve *components that can be tackled using classical algorithms*. Moreover, in many cases, **the classical algorithm must be run on inputs that exist in a superposition state**. This requires the classical algorithm to be run on quantum hardware. In this section we introduce some of the ideas used when doing this.

Consulting an Oracle

Many quantum algorithms are based around the analysis of some function $f(x)$. Often these algorithms simply assume the existence of some 'black box' implementation of this function, which we can give an input x and receive the corresponding output $f(x)$. This is referred to as an **oracle**. The advantage of thinking of the oracle in this abstract way allows us to concentrate on the quantum techniques we use to analyze the function, rather than the function itself.

Boolean Oracles: $U_f|x, \bar{0}\rangle = |x, f(x)\rangle$, where $|x, \bar{0}\rangle = |x\rangle \otimes |\bar{0}\rangle$ represents a multi-qubit state with 2 registers.

→ Register 1: binary representation of x ; Number of qubits = number of bit needed to represent the *inputs*.

→ Register 2: encode output, binary representation of $f(x)$; Number of qubits as many as necessary depending on U_f .

Phase Oracle: $P_f|x\rangle = (-1)^{f(x)}|x\rangle$, $f(x) = \{0, 1\}$ (binary).

→ it can be realized using the phase kickback mechanism, as a Boolean oracle. To see this, consider U_f corresponding to the same function: generaliezd form of CNOT, controlled by the input register: output = $|0\rangle$ if $f(x) = 0$, output = $|1\rangle$ if $f(x) = 1$. If the initial state of the output register were $|- \rangle$,

$$U_f(|x\rangle \otimes |- \rangle) = (P_f \otimes I)(|x\rangle \otimes |- \rangle)$$

Taking out the garbage

The functions evaluated by an oracle are typically those that can be evaluated efficiently on a classical computer. However, the need to implement it as a unitary in one of the forms shown above means that it must instead be implemented using quantum gates. However, __this is not quite as simple as just taking the Boolean gates that can implement the classical algorithm, and replacing them with their quantum counterparts.

Reversibility: One issue that we must take care of is that of reversibility.

A unitary $U = \sum_x |f(x)\rangle\langle x|$ only possible for unique input/output. Alternative is to include a copy of the input in the output, $U_f|x, \bar{0}\rangle = |x, f(x)\rangle$.

Superpositions: Classical algorithms typically do not only compute the desired output, but will also create additional information along the way. Such additional remnants of a computation do not pose a significant problem classically, and the memory they take up can easily be recovered by deleting them. From a quantum perspective, however, things are not so easy. For instance:

$$U_f|x, \bar{0}, \bar{0}\rangle = |x, f(x), g(x)\rangle$$

→ 3rd Register: scratchpad for classical algorithm: **garbage** $g(x)$.

Interference: Quantum algorithms are typically built upon interference effects. The simplest such effect is to create a superposition using some unitary, and then remove it using the inverse of that unitary. The entire effect of this is, of course, trivial. However, we must ensure that our quantum computer is at least able to do such trivial things.

Suppose $U_f|x, \bar{0}\rangle = |x, f(x)\rangle$ and we are required to return to $x, \bar{0}\rangle$. We could easily apply U_f^\dagger , but since we only know how to apply V_f and V_f^\dagger , what do we do?

Example: $f(x) = x$ and $g(x) = x$, single qubit, achieved with a single `cx` :

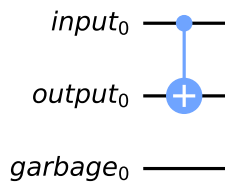
```
In [70]: from qiskit import QuantumCircuit, QuantumRegister

#defining registers
input_bit = QuantumRegister(1, 'input')
output_bit = QuantumRegister(1, 'output')
garbage_bit = QuantumRegister(1, 'garbage')

#aggregating Circuit
Uf = QuantumCircuit(input_bit, output_bit, garbage_bit)
Uf.cx(input_bit[0], output_bit[0])

Uf.draw()
```

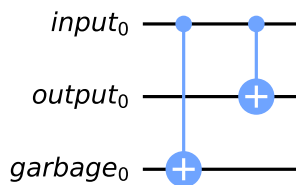
Out[70]:



For V_f , we also need to make a copy of the input for the garbage:

```
In [72]: Vf = QuantumCircuit(input_bit, output_bit, garbage_bit)
Vf.cx(input_bit[0], garbage_bit[0]) # first, before modifying
Vf.cx(input_bit[0], output_bit[0])
Vf.draw()
```

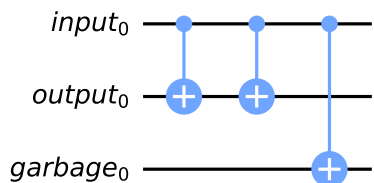
Out[72]:



We can now see the effect of applying U_f and then V_f^\dagger :

```
In [77]: qc = Vf+Vf.inverse()
#qc = QuantumCircuit.combine(Uf,Vf.inverse())
qc.draw()
```

Out[77]:



Notice the two `cx` cancel each other. Mathematically,

$$V_f^\dagger U_f |x, 0, 0\rangle = V_f^\dagger |x, f(x), 0\rangle = |x, 0, g(x)\rangle$$

This circuit does not simply return us to the initial state, but instead leaves the first qubit entangled with unwanted garbage. Any subsequent steps in an algorithm will therefore not run as expected, since the state is not the one that we need. Solution? Remove garbage!

This can be done by a method known as **uncomputation**. We simply need to take another blank variable and apply V_f :

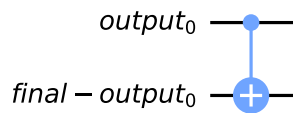
$$|x, 0, 0, 0\rangle \mapsto |x, f(x), g(x), 0\rangle$$

and apply a set of CNOTs, each controlled on one of the qubits used to encode the output, targeted on the corresponding qubit in the extra blank variable. In the previous case:

```
In [80]: final_output_bit = QuantumRegister(1, 'final-output')
```

```
copy = QuantumCircuit(output_bit, final_output_bit)
copy.cx(output_bit, final_output_bit)
copy.draw()
```

Out[80]:



which transforms the state to

$$|x, f(x), g(x), 0\rangle \mapsto |x, f(x), g(x), f(x)\rangle$$

and allows us to apply V_f^\dagger to undo the original computation:

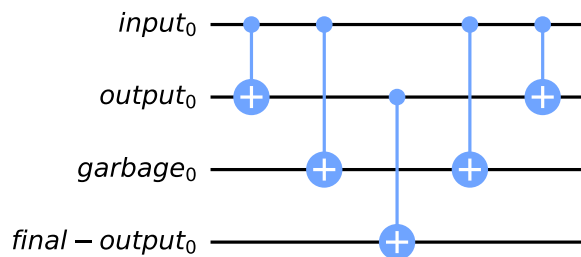
$$|x, f(x), g(x), 0\rangle \mapsto |x, 0, 0, f(x)\rangle$$

performing the computation without garbage.

In [81]:

```
(Vf.inverse() + copy + Vf).draw()
```

Out[81]:



In []:

Exercises

1 Show that the output is correctly written to the 'final output' register (and only to this register) when the 'output' register is initialized as $|0\rangle$.

2 Determine what happens when the 'output' register is initialized as $|1\rangle$.

In []: