Chapter 2 - Multiple Qubits and Entanglement - Full Notes

Grant Talbert

December 6th, 2023

Contents

1 Phase Kickback **2**
1.1 Exploring the CNOT-Gate . . . 2
1.2 Phase Kickback . . . 6
1.2.1 Explaining the CNOT Circuit Identity . . . 6
1.2.2 Kickback with the T-gate . . . 8
1.2.3 Quick Excercises . . . 11

2 Basic Circuit Identities **12**
2.1 Making a Controlled-Z from CNOT . . . 12
2.2 Swapping Qubits . . . 14
2.3 Controlled Rotations . . . 18
2.4 The Toffoli . . . 20
2.5 Arbitrary Rotations from H and T . . . 23

3 Proving Universality **27**
3.1 Introduction . . . 27
3.2 Fun With Matrices . . . 27
3.2.1 Matrices as Outer Products . . . 27
3.2.2 Unitary and Hermitian Matrices . . . 27
3.2.3 Pauli Decomposition . . . 29
3.3 Defining Universality . . . 30
3.4 Basic Gate Sets . . . 30
3.4.1 Clifford Gates . . . 30
3.4.2 Non-Clifford Gates . . . 31
3.4.3 Expanding the Gate Set . . . 32
3.5 Proving Universality . . . 32

4 Classical Computation on a Quantum Computer **36**
4.1 Introduction . . . 36
4.2 Consulting an Oracle . . . 36
4.3 Taking Out the Garbage . . . 37
4.4 Quick Exercises . . . 40

1 Phase Kickback

1.1 Exploring the CNOT-Gate

In the previous section, we saw basic results with the CNOT gate. Here we explore more interesting results. We saw we an entangle the qubits by placing the control qubit in the state $|+\rangle$:

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

But what happens if we put the second qubit in superposition?

```
from qiskit import QuantumCircuit, Aer, assemble
from math import pi
import numpy as np
from qiskit.visualization import plot_bloch_multivector, plot_histogram,
array_to_latex

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

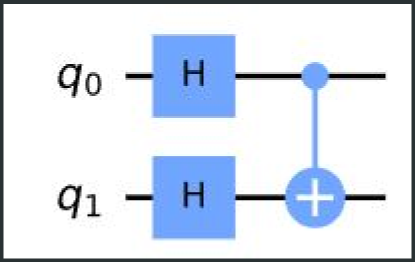


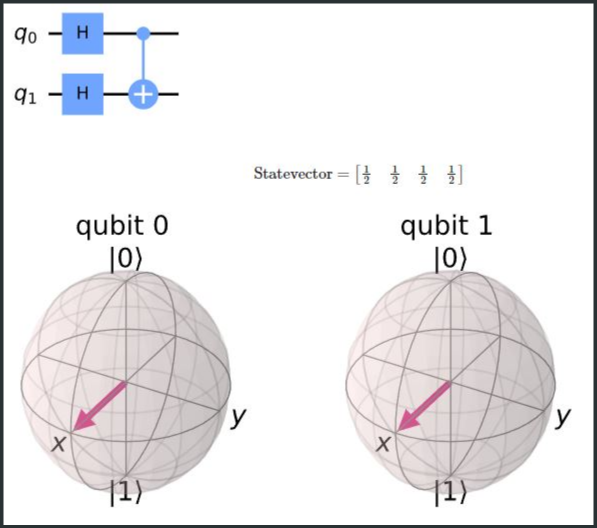
Figure 1: CNOT acting on the state:

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

Since the CNOT simply swaps the amplitudes of $|01\rangle$ and $|10\rangle$, we see no change:

```
qc = QuantumCircuit(2)
qc.h(0)
qc.h(1)
qc.cx(0,1)
display(qc.draw())  # `display` is a command for Jupyter notebooks
# similar to `print`, but for rich content

# Let's see the result
svsim = Aer.get_backend('aer_simulator')
qc.save_statevector()
qobj = assemble(qc)
final_state = svsim.run(qobj).result().get_statevector()
display(array_to_latex(final_state, prefix="\\text{Statevector} = "))
plot_bloch_multivector(final_state)
```



$\text{Statevector} = [\frac{1}{2} \quad \frac{1}{2} \quad \frac{1}{2} \quad \frac{1}{2}]$

Figure 2: Results of CNOT on the superposition

Let's put the target qubit in the state $|-\rangle$, so it has negative phase:

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

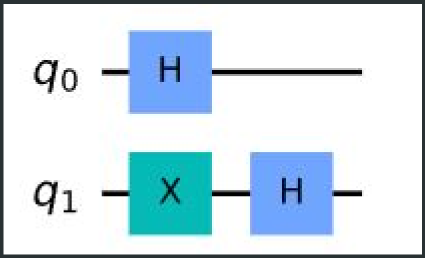


Figure 3: This creates the state:

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

If CNOT acts on this state, it will switch the amplitudes of $|01\rangle$ and $|11\rangle$, resulting in

$$CNOT|-+\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle) = \frac{1}{2}\begin{pmatrix} 1 \\ -1 \\ -1 \\ 1 \end{pmatrix}$$

Notice:

$$|-\rangle \otimes |-\rangle = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ -1 \end{pmatrix} \otimes \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ -1 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1\begin{pmatrix} 1 \\ -1 \end{pmatrix} \\ \\ -1\begin{pmatrix} 1 \\ -1 \end{pmatrix} \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1 \\ -1 \\ -1 \\ 1 \end{pmatrix} = \frac{1}{2}(|00\rangle - |01\rangle - |10\rangle + |11\rangle)$$

$$\Rightarrow CNOT|-+\rangle = |--\rangle$$

Interestingly, this effects the state of the *control* qubit, while leaving the state of the *target* qubit unchanged. Recall that the H-gate transforms $|+\rangle \to |0\rangle$ and $|-\rangle \to |1\rangle$; we see that wrapping a CNOT in H-gates has the equivelant behavior of a CNOT acting in the opposite direction:

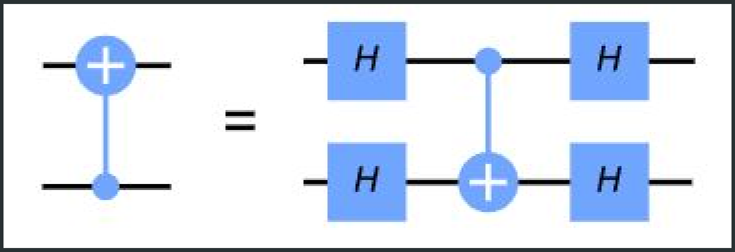


Figure 4: Circuit Diagram

Recall the following:

$$CNOT(q_1,q_0)=\begin{pmatrix}1&0&0&0\\0&0&0&1\\0&0&1&0\\0&1&0&0\end{pmatrix}\qquad CNOT(q_0,q_1)=\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}$$

It follows that

$$\begin{aligned}
&(H\otimes H)\,(CNOT\,(q_1,q_0))\,(H\otimes H)\\
&=\left(\begin{pmatrix}1&1\\1&-1\end{pmatrix}\otimes\begin{pmatrix}1&1\\1&-1\end{pmatrix}\right)\begin{pmatrix}1&0&0&0\\0&0&0&1\\0&0&1&0\\0&1&0&0\end{pmatrix}\left(\begin{pmatrix}1&1\\1&-1\end{pmatrix}\otimes\begin{pmatrix}1&1\\1&-1\end{pmatrix}\right)\\
&=\frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}\frac{1}{\sqrt{2}}\begin{pmatrix}H&H\\H&-H\end{pmatrix}\begin{pmatrix}1&0&0&0\\0&0&0&1\\0&0&1&0\\0&1&0&0\end{pmatrix}\begin{pmatrix}H&H\\H&-H\end{pmatrix}\\
&=\left(\frac{1}{\sqrt{2}}\right)^4\begin{pmatrix}1&1&1&1\\1&-1&1&-1\\1&1&-1&-1\\1&-1&-1&1\end{pmatrix}\begin{pmatrix}1&0&0&0\\0&0&0&1\\0&0&1&0\\0&1&0&0\end{pmatrix}\begin{pmatrix}1&1&1&1\\1&-1&1&-1\\1&1&-1&-1\\1&-1&-1&1\end{pmatrix}\\
&=\frac{1}{\sqrt{2^4}}\begin{pmatrix}1&1&1&1\\1&-1&1&-1\\1&-1&-1&1\\1&1&-1&-1\end{pmatrix}\begin{pmatrix}1&1&1&1\\1&-1&1&-1\\1&1&-1&-1\\1&-1&-1&1\end{pmatrix}\\
&=\frac{1}{4}\begin{pmatrix}1+1+1+1&1-1+1-1&1+1-1-1&1-1-1+1\\1-1+1-1&1+1+1+1&1-1-1+1&1+1-1-1\\1-1-1+1&1+1-1-1&1-1+1-1&1+1+1+1\\1+1-1-1&1-1-1+1&1+1+1+1&1-1+1-1\end{pmatrix}\\
&=\frac{1}{4}\begin{pmatrix}4&0&0&0\\0&4&0&0\\0&0&0&4\\0&0&4&0\end{pmatrix}=\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}=CNOT\,(q_0,q_1)
\end{aligned}$$

We can also verify this with Qiskit's Aer simulator:

```python
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')
qobj = assemble(qc)
unitary = usim.run(qobj).result().get_unitary()
array_to_latex(unitary, prefix="\\text{Circuit = }\n")
```

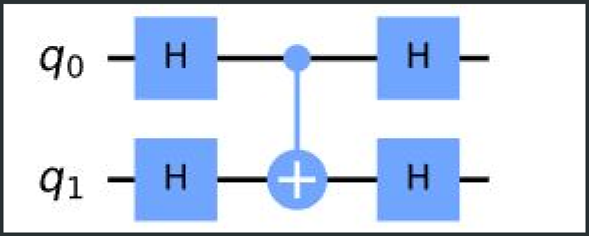


Figure 5: Circuit Diagram

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

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

qobj = assemble(qc)
unitary = usim.run(qobj).result().get_unitary()
array_to_latex(unitary, prefix="\\text{Circuit = }\n")
```

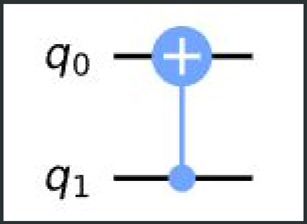


Figure 6: Circuit Diagram

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

This identity is known as a *phase kickback*.

1.2 Phase Kickback

1.2.1 Explaining the CNOT Circuit Identity

Phase kickback is extremely important and is used in almost every quantum algorithm. Kickback is where the eigenvalue added by a gate to a qubit is 'kicked back' into a different qubit via a controlled operation. For example, we saw performing an X-gate on a $|-\rangle$ qubit gives it the phase -1:

$$X|-\rangle = -|1\rangle$$

When our control qubit is either $|0\rangle$ or $|1\rangle$, this phase affects the whole state, however it's a global phase and has no observable effect.

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

This can be shown in more depth:

$$|{-0}\rangle = \begin{pmatrix} 1\begin{pmatrix}1\\0\end{pmatrix} \\ \\ 1\begin{pmatrix}1\\0\end{pmatrix}\end{pmatrix} = \begin{pmatrix}1\\0\\-1\\0\end{pmatrix} = |00\rangle - |10\rangle$$

and CNOT switches the amplitudes of $|01\rangle$ and $|11\rangle$, both of which are 0.

$$\text{CNOT}|{-1}\rangle = \text{CNOT}\left(\frac{1}{\sqrt{2}}(|01\rangle - |11\rangle)\right)$$
$$= \frac{1}{\sqrt{2}}(-|01\rangle + |11\rangle)$$
$$= X|-\rangle \otimes |1\rangle$$
$$= -|-\rangle \otimes |1\rangle$$
$$= -|{-1}\rangle$$

The interesting effect of phase kickback only happens when the control qubit is in superposition. The component of the control qubit that lies in the direction of $|1\rangle$ applies this phase factor to the *corresponding* target qubit. This applied phase factor in turn introduces a relative phase into the control qubit:

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

This can then be written as two seperable qubit states:

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

Wrapping CNOT in H-gates transforms the qubits from the computational basis to the $(|+\rangle, |-\rangle)$ basis, where we see this effect. This identity is very useful in hardware, since some hardwares only allow for CNOTs in one direction. This identity helps overcome these limitations.

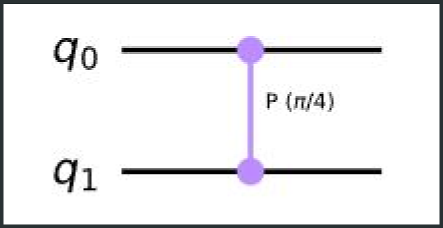


Figure 7: Circuit Diagram

1.2.2 Kickback with the T-gate

Let's look at another controlled operation, the controlled T-gate:

```
qc = QuantumCircuit(2)
qc.cp(pi/4, 0, 1)
qc.draw()
```

Recall the T-gate has the matrix:

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

and the controlled T-gate has the following matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{\frac{i\pi}{4}} \end{pmatrix}$$

We verify this with Qiskit's Aer simulator:

```
qc = QuantumCircuit(2)
qc.cp(pi/4, 0, 1)
display(qc.draw())
# See Results:
qc.save_unitary()
qobj = assemble(qc)
unitary = usim.run(qobj).result().get_unitary()
array_to_latex(unitary, prefix="\\text{Controlled-T} = \n")
```

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

More generally. we can find any controlled-U operation using the rule:

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

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

Or, with Qiskit's qubit ordering:

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

Applying a T-gate on a qubit in state $|1\rangle$ is equal to adding a phase of $e^{\frac{i\pi}{4}}$ to the qubit:

$$T|1\rangle = e^{\frac{i\pi}{4}}|1\rangle$$

This is *global phase* and unobservable. However if we control this qubit using another qubit in the $|+\rangle$ state, the phase is no longer global but relative, which changes the *relative phase* in our control qubit:

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

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

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

$$= |1\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle + e^{\frac{i\pi}{4}}|1\rangle)$$

This has the effect of rotating the control qubit about the Z-axis of the Bloch sphere, while leaving the target qubit untouched. Let's see this in Qiskit:

```
qc = QuantumCircuit(2)
qc.h(0)
qc.x(1)
display(qc.draw())
# See Results:
qc.save_statevector()
qobj = assemble(qc)
final_state = svsim.run(qobj).result().get_statevector()
plot_bloch_multivector(final_state)
```

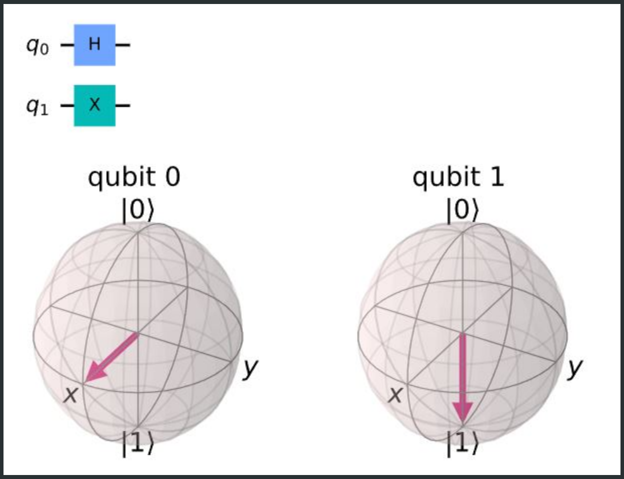


Figure 8: Output for $|1+\rangle$

```
qc = QuantumCircuit(2)
qc.h(0)
qc.x(1)
# Add Controlled-T
qc.cp(pi/4, 0, 1)
display(qc.draw())
# See Results:
qc.save_statevector()
qobj = assemble(qc)
final_state = svsim.run(qobj).result().get_statevector()
plot_bloch_multivector(final_state)
```

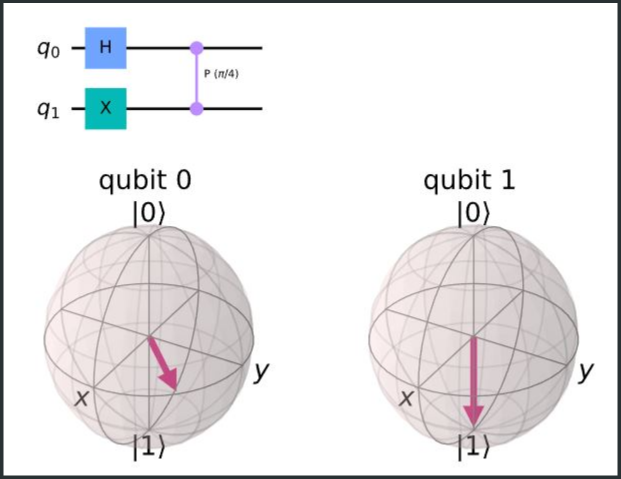
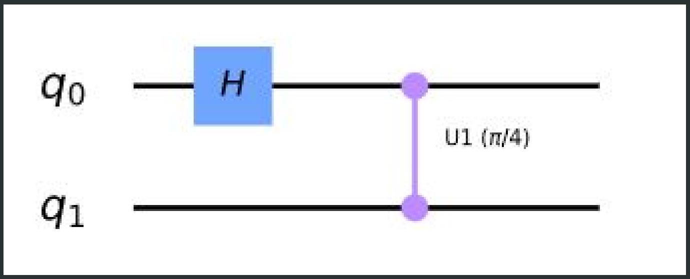


Figure 9: Output for Controlled-T$|1+\rangle$

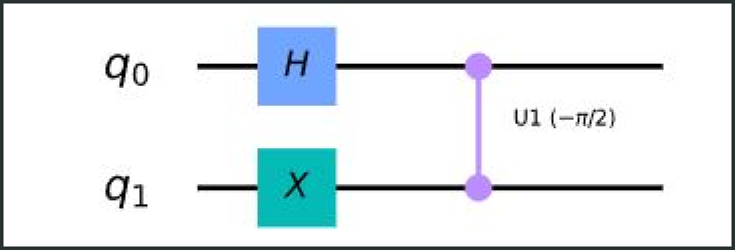
We see that the leftmost qubit has been rotated by $\pi/4$ about the Z-axis of the Bloch sphere, as expected. After further exploring this behavior, it may become clear why Qiskit draws the controlled Z-rotation gates in this symmetrical fasion (two controls instead of a control and a target). There is no clear control qubit for all cases.

1.2.3 Quick Excercises

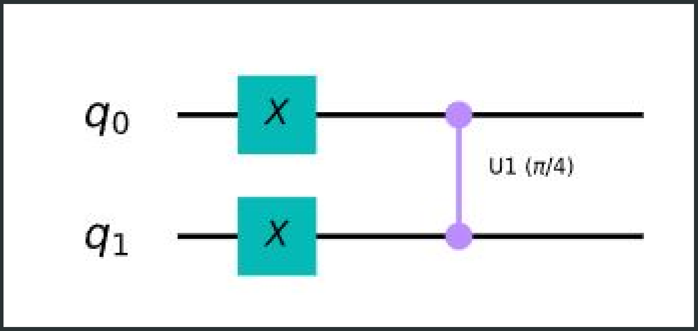
1. What would be the resulting state of the control qubit (q0) if the target qubit (q1) was in the state $|0\rangle$? (as shown in the circuit below)? Use Qiskit to check your answer.



2. What would happen to the control qubit (q0) if the target qubit (q1) was in the state $|1\rangle$, and the circuit used a controlled-S dagger gate instead of the controlled-T (as shown in the circuit below)? Recall that the S-gate is a phase gate with $\phi = \frac{\pi}{2}$.



3. What would happen to the control qubit (q0) 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)?



2 Basic Circuit Identities

2.1 Making a Controlled-Z from CNOT

The Controlled-Z gate (cz) is similar to the CNOT; just as CNOT applies X to the target qubit if the control is in $|1\rangle$, the controlled-Z applies Z if the control qubit is in $|1\rangle$. In Qiskit it's invoked directly with:

```
# a controlled-Z
qc.cz(c,t)
qc.draw()
```

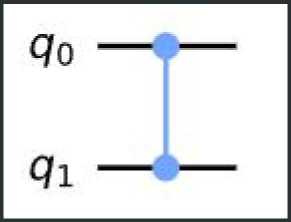


Figure 10: Controlled-Z circuit diagram.

where c and t are the control and target qubits, respectively. However, in IBM Q devices, the only two-qubit gate that can be invoked directly is the CNOT, so we need to be able to transform one into the other. Since we know the Hadamard transforms $|0\rangle \rightarrow |+\rangle \wedge |1\rangle \rightarrow |-\rangle$, and since the Z-gate on the states $|+\rangle$ and $|-\rangle$ is the same as X on the states $|0\rangle$ and $|1\rangle$, respectively, we find:

$$HXH = Z \qquad HZH = X$$

We can prove this mathematically:

$$HXH = \frac{1}{2}\begin{pmatrix}1 & 1\\1 & -1\end{pmatrix}\begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}\begin{pmatrix}1 & 1\\1 & -1\end{pmatrix}$$

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

$$= \frac{1}{2}\begin{pmatrix}2 & 0\\0 & -2\end{pmatrix} = \begin{pmatrix}1 & 0\\0 & -1\end{pmatrix} = Z$$

The same trick can be used to transform CNOT into controlled-Z: wrap the CNOT in Hadamards for only the target qubit, as this transforms the X into a Z.

```
qc = QuantumCircuit(2)
# also a controlled-Z
qc.h(t)
qc.cx(c,t)
qc.h(t)
qc.draw()
```

More generally, we transform CNOT into any controlled rotation about the Bloch sphere by an angle π, simply by preceeding and following with the proper gates. For example, the controlled-Y

$$\overrightarrow{S^{\dagger}XS} = Y$$

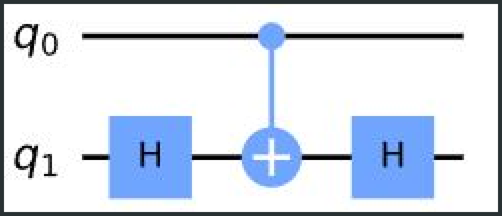


Figure 11: Controlled-Z circuit diagram.

Arrow to show order of operations from left to right

Remember: matricies do not commute AND the first gate to be applied is the rightmost gate when calculating the unitary (this is because while they don't commute, they do associate).

$$SXS^\dagger$$
$$= \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{2}} \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & e^{-\frac{i\pi}{2}} \end{pmatrix}$$
$$= \begin{pmatrix} 0 & 1 \\ e^{\frac{i\pi}{2}} & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & e^{-\frac{i\pi}{2}} \end{pmatrix}$$
$$= \begin{pmatrix} 0 & e^{-\frac{i\pi}{2}} \\ e^{\frac{i\pi}{2}} & 0 \end{pmatrix}$$
$$= \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = \sigma_y$$

```python
qc = QuantumCircuit(2)
# a controlled-Y
qc.sdg(t)
qc.cx(c,t)
qc.s(t)
qc.draw()
```

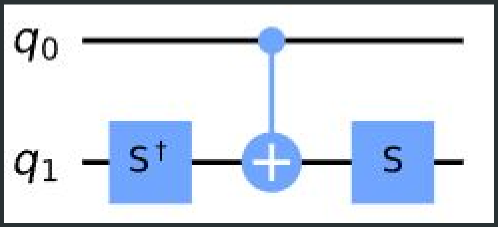


Figure 12: Controlled-Y circuit diagram.

and a controlled-H:

```python
qc = QuantumCircuit(2)
# a controlled-H
qc.ry(pi/4,t)
qc.cx(c,t)
qc.ry(-pi/4,t)
qc.draw()
```

We will see more use of these R-gates later.

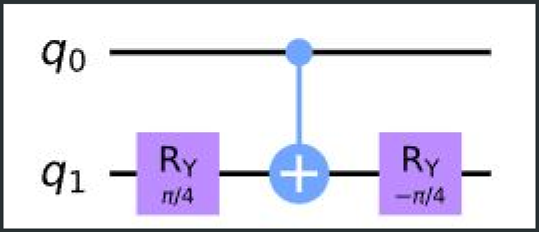


Figure 13: Controlled-H circuit diagram.

2.2 Swapping Qubits

Sometimes we need to move information around in a quantum computer. Sometimes this can be done by physically moving the qubits, but often it's done by moving the state between two qubits. This is done by the SWAP gate.

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

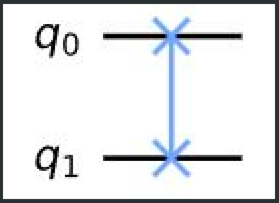


Figure 14: SWAP gate circuit diagram.

Although the command above directly invokes this gate, we should learn how to construct it from standard gates. We begin by considering some examples: First, consider the cases of a qubit a being in the state $|a\rangle = |1\rangle$, and a qubit b is in the state $|b\rangle = |0\rangle$.

```
qc = QuantumCircuit(2)
# 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
qc.draw()
```

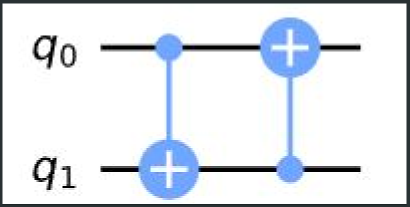


Figure 15: Circuit diagram.

This has the effect of putting qubit b in state $|1\rangle$ and qubit a in state $|0\rangle$:

$$|a\rangle = |1\rangle \qquad |b\rangle = |0\rangle$$

$$\Rightarrow \text{CNOT}(a,b)\text{CNOT}(b,a)(|0\rangle\otimes|1\rangle) = \begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}\begin{pmatrix}1&0&0&0\\0&0&0&1\\0&0&1&0\\0&1&0&0\end{pmatrix}\begin{pmatrix}0\\1\\0\\0\end{pmatrix}$$
$$= \begin{pmatrix}1&0&0&0\\0&0&0&1\\0&1&0&0\\0&0&1&0\end{pmatrix}\begin{pmatrix}0\\1\\0\\0\end{pmatrix}$$
$$= \begin{pmatrix}0\\0\\1\\0\end{pmatrix}$$
$$= |10\rangle = |1\rangle\otimes|0\rangle$$

To SWAP this state back to the original, we take the inverse:

$$\text{CNOT}(b,a)\text{CNOT}(a,b)|10\rangle = \begin{pmatrix}1&0&0&0\\0&0&0&1\\0&0&1&0\\0&1&0&0\end{pmatrix}\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}\begin{pmatrix}0\\0\\1\\0\end{pmatrix}$$
$$= \begin{pmatrix}1&0&0&0\\0&0&1&0\\0&0&0&1\\0&1&0&0\end{pmatrix}\begin{pmatrix}0\\0\\1\\0\end{pmatrix}$$
$$= \begin{pmatrix}0\\1\\0\\0\end{pmatrix} = |01\rangle = |0\rangle\otimes|1\rangle$$

Notice 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(b,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(b,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 canc ombine the two processes by adding an ineffective gate from one onto the other.

```
qc = QuantumCircuit(2)
qc.cx(b,a)
qc.cx(a,b)
qc.cx(b,a)
qc.draw()
```

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 $|1\rangle$ 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.

A more intuitive way to think about this that I came up with is: for a set of basis vectors on $|q_1\rangle = |b\rangle$ and $|q_2\rangle = |a\rangle$, CNOT only acts on the circuit if the control is $|1\rangle$. If $|b\rangle = |1\rangle$, then b is

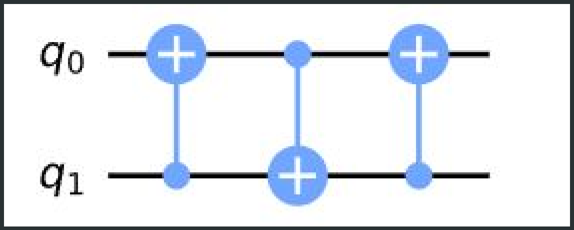


Figure 16: Circuit diagram.

the first control, so the $|1\rangle$ gets swapped to a by the second gate. The third gate then has $|b\rangle$ as the control again, but b is not $|1\rangle$, so it acts trivially on the state. Vice versa is also true. It also has the correct effect on the $|00\rangle$ state. This is symmetric, so swapping the states should have no effect. Since CNOT have no effect with control qubits $|0\rangle$, this holds. The $|11\rangle$ state is also symmetric, and so needs a trivial effect from the swap. In this case, the CNOT gate in the process above will cause the second to have no effect, and the third undoes the first. Therefore, the effect is trivial. We have thus found a way to decompose SWAP gates into our standard gate set of single-qubit rotations and CNOT gates.

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

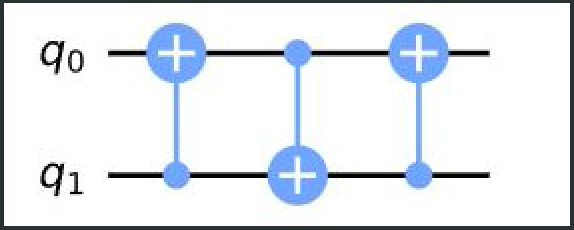


Figure 17: Decomposed SWAP gate circuit diagram

It works for the states $|00\rangle, |01\rangle, |10\rangle, |11\rangle$, and if it works for all states in the computational basis, it must work for all states generally. This circuit therefore swaps all possible two-qubit states. Notably, the same effect is achieved if the order of the gates is reversed. The derivation used here was very much based on the Z-basis states, but it could 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 equivelant to the ones here.

I have devised an even better way to realize this intuitively. Notice the SWAP gate is made of two CNOTs sandwiching a third, acting on the qubits in the opposite manner. We show the unitaries of the operations are equivelant:

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

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

Now consider how these matrices act on a 4 dimensional statevector. They have 1's in the top left and bottom right, and so will act trivially on the top and bottom amplitudes of the statevector, but the one is in the third column of the second row, and the second column of the third row, effectively swapping these amplitudes:

$$\begin{gathered}
\text{SWAP}|\psi\rangle\\
= \begin{pmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{pmatrix}\begin{pmatrix}\alpha\\\beta\\\gamma\\\delta\end{pmatrix}\\
= \begin{pmatrix}(1\cdot\alpha)+(0\cdot\beta)+(0\cdot\gamma)+(0\cdot\delta)\\(0\cdot\alpha)+(0\cdot\beta)+(1\cdot\gamma)+(0\cdot\delta)\\(0\cdot\alpha)+(1\cdot\beta)+(0\cdot\gamma)+(0\cdot\delta)\\(0\cdot\alpha)+(0\cdot\beta)+(0\cdot\gamma)+(1\cdot\delta)\end{pmatrix}\\
= \begin{pmatrix}\alpha\\\gamma\\\beta\\\delta\end{pmatrix}
\end{gathered}$$

Now that we notice the unitary swaps the probability amplitudes β and γ, let's analyze our statevector to see how this swaps the quantum states. Consider a 4-dimensional statevector:

$$|\psi_\alpha\rangle = \begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix}$$
$$|\psi_\beta\rangle = \begin{pmatrix}\beta_0\\ \beta_1\end{pmatrix}$$
$$|\psi\rangle = |\psi_\alpha\rangle \otimes |\psi_\beta\rangle = \begin{pmatrix}\alpha_0\\ \alpha_1\end{pmatrix} \otimes \begin{pmatrix}\beta_0\\ \beta_1\end{pmatrix}$$
$$= \alpha_0\beta_0|00\rangle + \alpha_0\beta_1|01\rangle + \alpha_1\beta_0|10\rangle + \alpha_1\beta_1|11\rangle$$
$$\text{SWAP}|\psi\rangle = \alpha_0\beta_0|00\rangle + \alpha_1\beta_0|01\rangle + \alpha_0\beta_1|10\rangle + \alpha_1\beta_1|11\rangle$$

Any probability representing the state $|00\rangle$ or $|11\rangle$ represents a symmetric state - if the system is in this state, then each qubit must be in the same state. Thus, swapping the states between the qubits is equivelant to acting trivially on the system. If the system is in a superposition, then swapping the symmetric states has no change in the probability amplitudes, since there was no change in the state itself. It's only the non-symmetric states that must be swapped, and we see that the unitary derived above swaps the probability amplitudes of each state, thus performing a SWAP operation.
Quick Exercise: Find a different circuit that swaps qubits in the states $|+\rangle$ and $|-\rangle$, and show that this is equivelant to the circuit shown above. will solve later

2.3 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, consider arbitrary rotations about the y axis. Specifically, consider the following sequence of gates:

```
qc = QuantumCircuit(2)
theta = pi # theta can be anything (pi chosen arbitrarily)
qc.ry(theta/2,t)
qc.cx(c,t)
qc.ry(-theta/2,t)
qc.cx(c,t)
qc.draw()
```

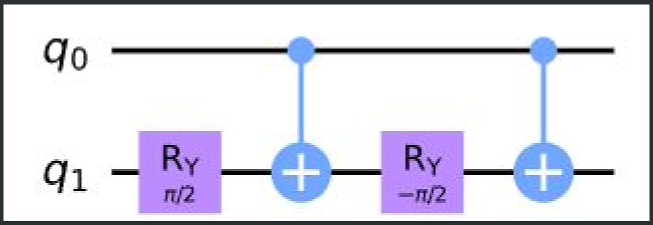


Figure 18: Circuit Diagram

Note: R-gates of the form $R_a(\theta)$ give a rotation of θ degrees about axis a. They are arbitrary for this section and do not need to be well-defined.
If the control qubit is in state $|0\rangle$, all we have here is a $R_y\left(\frac{\theta}{2}\right)$ immediately followed by its inverse $R_y\left(\frac{\theta}{2}\right)$. The end effect is obviously trivial.
If the control qybit is in state $|1\rangle$, $R_y\left(\frac{\theta}{2}\right)$ is effectively preceeded and followed by an X-gate. This

has the effect of flipping the direction of the y rotation and giving a second $R_y\left(\frac{\theta}{2}\right)$. The net effect in this case is therefore to make a controlled $R_y(\theta)$.

This method works because the x and y axes are orthogonal, which causes the x gates to flip the direction of 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, V. For this we simply need to find three rotations A, B, C and a phase α such that

$$ABC = \mathbb{I} \wedge e^{i\alpha} AZBZC = V$$

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 in 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.

```
A = Gate('A', 1, [])
B = Gate('B', 1, [])
C = Gate('C', 1, [])
alpha = 1 # arbitrarily define alpha to allow drawing of circuit
```

```
qc = QuantumCircuit(2)
qc.append(C, [t])
qc.cz(c,t)
qc.append(B, [t])
qc.cz(c,t)
qc.append(A, [t])
qc.p(alpha,c)
qc.draw()
```

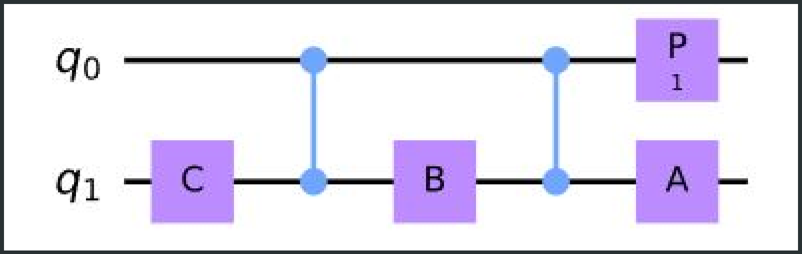


Figure 19: Diagram of arbitrary rotation

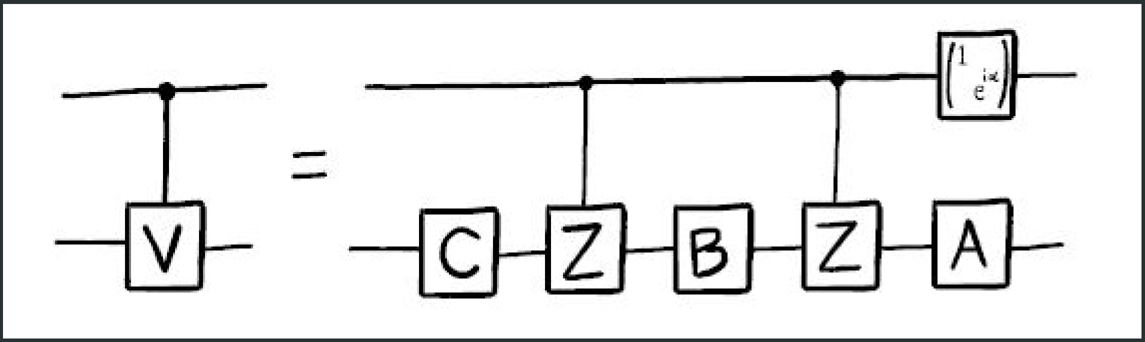


Figure 20: Well defined diagram of arbitrary rotation

Here, A, B, and C are the gates that implement A, B, and C. We can visualize what this actually means if we accept A, B, C, and V as arbitrary operations that can be derived at a later date, and instead trace the values of the qubits.

1. Consider what happens if the control qubit a is in state $|0\rangle$. The controlled-Z gates will not apply, and the phase gate will not apply. The target qubit will undergo $\overrightarrow{ABC}$ (arrow depicts direction of time, opposite of multiplication), and by definition,

   $$\text{CBA} := \mathbb{I}$$

   It follows that

   $$\forall |v\rangle \in \mathscr{H}\ (\mathbb{I}|v\rangle \equiv |v\rangle) \Longrightarrow \forall |v\rangle \in \mathscr{H}\ (\text{CBA}|v\rangle = |v\rangle)$$

   Thus, there is no effect on the system.

2. Consider what happens if the control qubit a is in state $|1\rangle$. The controlled-Z gates **will apply**, and so will the phase gate. The control qubit undergoes a phase change $R_z(2\alpha)$, and the target qubit undergoes $\overrightarrow{AZBZC}$. By definition, CZBZA := V, so the operation V is applied to the vector.

2.4 The Toffoli

The Toffoli gate is a three-qubit gate with two controls and one target. It performs an X on the target if and only if both controls have state $|1\rangle$. The final state of the target is then equal to either then AND or NAND of the controls, depending on whether the initial state of the target was $|0\rangle$ or $|1\rangle$. A Toffoli can be thought of as a controlled-controlled-NOT, and is sometimes known as the CCX gate.

```
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()
```

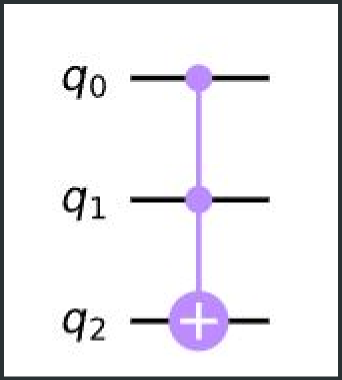


Figure 21: Toffoli diagram

To see how to build it from single- and two-qubit gates, it's more helpful to define how to build an arbitrary controlled-controlled-U for any single-qubit rotation U. For this we need to redfine controlled versions of $V = \sqrt{U}$ and $V^\dagger$. In the code below, we use `cp(theta,c,t)` and `cp(-theta,c,t)` in place of the undefined subroutines `cv` and `cvdg`, respectively. The controls are qubits a and b, and the target is qubit t.

```
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()
```

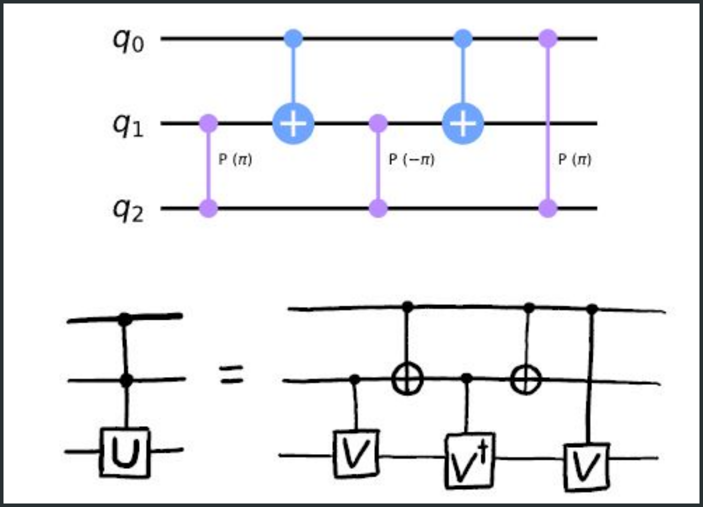


Figure 22: Toffoli diagram

Let's visualize the values of the qubits.

1. The starting value of the target qubit t is arbitrary, we only want to perform the U operation on it. Suppose we have both control qubits a, b in state $|0\rangle$. None of the controlled V gates will be applied, since the first V and $V^\dagger$ require qubit b to be in the state $|1\rangle$. The CNOT operations with target qubit b and control qubit a will not happen, since they require a begin with state $|1\rangle$. The final V gate will not be applied either, since it requires control qubit a to be in state $|1\rangle$. We deduce that for a starting set with all qubits in state $|0\rangle$, the operation will not be applied.

2. Suppose we have qubit a in state $|1\rangle$, and b in state $|0\rangle$. The first V gate has no effect, since b is in state $|0\rangle$. The first CNOT applies since control qubit a is in state $|1\rangle$, changing b to $|1\rangle$, where it applies the $V^\dagger$ gate. The second CNOT gate then applies for the same reasons, changing b back to $|0\rangle$, leaving its state the same as when we started. a then undoes the $V^\dagger$ performed by b by performing a controlled V on the target. All qubits are in the same state as when they started, so the operation will not be applied.

3. Suppose we have a qubit a in state $|0\rangle$, and b in state $|1\rangle$. The first V applies to the target qubit t, since b is in state $|1\rangle$. The first CNOT has no effect since a is in state $|0\rangle$. $V^\dagger$ is then applied by b still being in state $|1\rangle$, undoing the first V. a then does not perform the second CNOT or the V gate. No qubits change states.

4. Suppose both controls are in state $|1\rangle$. The first V is applied, then CNOT is applied to turn b into $|0\rangle$. The $V^\dagger$ is then skipped, since b was in $|0\rangle$. The second CNOT brings b back to its original state of $|1\rangle$, and then a performs V on the target. The target qubit undergoes 2 V transformations, effectively undergoing a U transformation.

This tracing allows us to realize that the only way the U transformation is applied is if both controls are $|1\rangle$. Using the ideas we have already described, we can now implement each controlled-V gate to arrive at some circuit for the doubly-controlled-U gate. It turns out the minimum number of CNOTs needed to implement a Toffoli is 6.

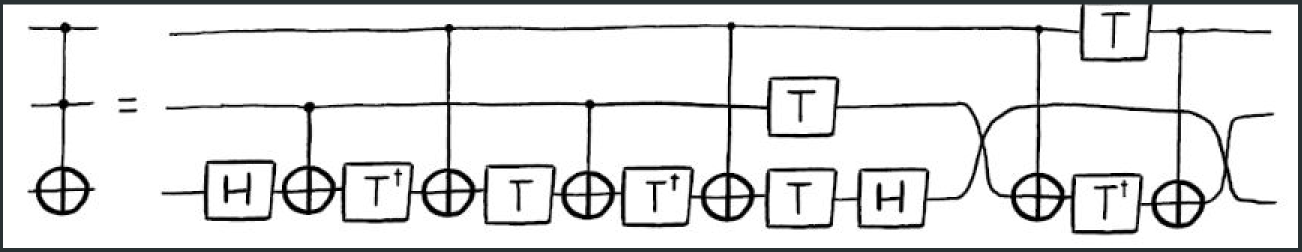


Figure 23: Toffoli diagram, fully decomposed

This is a Toffoli with 3 qubits, q_0, q_1, q_2 respectively. In this example, q_0 is connected with q_2, but q_0 is not connected with q_1.
There are other ways to implement AND in quantum computing with less CNOTs, but these induce relative phases. For example, one can be constructed with the controlled-Hadamard and controlled-Z gates:

```
qc = QuantumCircuit(3)
qc.ch(a,t)
qc.cz(b,t)
qc.ch(a,t)
qc.draw()
```

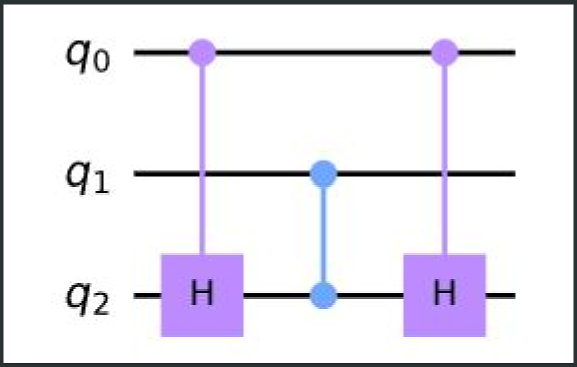


Figure 24: AND gate implementation

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 preceeded and followed by an H gate, with a net effect of an X gate. For states $|01\rangle$ and $|10\rangle$, the target experiences either two hadamards, which cancel each other out, or just the Z, which induces relative phase. This therefore replicates AND with just 3 CNOTs.

2.5 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, and operations 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 noise. This is done by making gates harder to do by mistake, or implementing them wrong.

This is unfortunate for the single-qubit rotations $R_x(\theta), R_y(\theta), R_z(\theta)$. It's impossible to implement an angle θ with perfect accuracy, such that you don't accidentally implement something like $\theta + 0.00000001$. There is always a limit to the accuracy achievable, and it's often larger than 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 more deliberate manner.
Fault tolerant schemes typically perform these rotations using multiple applications of just two gates: H and T. The T gate in Qiskit is `.t()`:

```
qc = QuantumCircuit(1)
qc.t(0) # T gate on qubit 0
qc.draw()
```

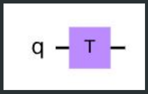


Figure 25: T-gate

It's a rotation about the z-axis by $\theta = \frac{\pi}{4}$, and is expressed mathematically as $R_z\left(\frac{\pi}{4}\right) = e^{\frac{i\pi}{4}Z}$.
In the following we assume 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 last chapter, we can use the T gate to create a similar rotation about the x axis.

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

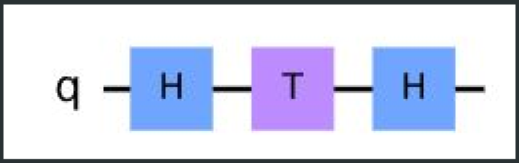


Figure 26: Rotation about the z axis by $\frac{\pi}{4}$

We verify this mathematically rather trivially:

$$\text{HTH} = \frac{1}{2}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\begin{pmatrix}1 & 0\\ 0 & e^{\frac{i\pi}{4}}\end{pmatrix}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$$

$$= \frac{1}{2}\begin{pmatrix}1 & e^{\frac{i\pi}{4}}\\ 1 & -e^{\frac{i\pi}{4}}\end{pmatrix}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$$

$$= \frac{1}{2}\begin{pmatrix}1+e^{\frac{i\pi}{4}} & 1-e^{\frac{i\pi}{4}}\\ 1-e^{\frac{i\pi}{4}} & 1+e^{\frac{i\pi}{4}}\end{pmatrix}$$

This is an interesting state, so we show that it's a valid quantum operation by taking HTH$|\psi\rangle$ for some arbitrary $|\psi\rangle$. For simplicity's sake, $|0\rangle$ has been chosen. I could be more rigorous but thats effort.

$$\text{HTH}|0\rangle$$
$$= \frac{1}{2}\begin{pmatrix}1+e^{\frac{i\pi}{4}} & 1-e^{\frac{i\pi}{4}}\\ 1-e^{\frac{i\pi}{4}} & 1+e^{\frac{i\pi}{4}}\end{pmatrix}\begin{pmatrix}1\\0\end{pmatrix}$$
$$= \left(1+e^{\frac{i\pi}{4}}\right)|0\rangle + \left(1-e^{\frac{i\pi}{4}}\right)|1\rangle$$
$$\left|\frac{1+e^{\frac{i\pi}{4}}}{2}\right|^2 = \left|\frac{1+\frac{1+i}{\sqrt{2}}}{2}\right|^2 = \left(\frac{1+\frac{1}{\sqrt{2}}}{2}\right)^2 + \left(\frac{1}{2\sqrt{2}}\right)^2$$
$$= \frac{2+\sqrt{2}}{4}$$
$$\left|\frac{1-e^{\frac{i\pi}{4}}}{2}\right|^2 = \left|\frac{1-\frac{1+i}{\sqrt{2}}}{2}\right|^2 = \left(\frac{1-\frac{1}{\sqrt{2}}}{2}\right)^2 + \left(\frac{1}{2\sqrt{2}}\right)^2$$
$$= \frac{2-\sqrt{2}}{4}$$
$$\frac{2+\sqrt{2}}{4} + \frac{2-\sqrt{2}}{4} = 1$$

Now let's put the two together. Let's make the gate $R_z(\pi/4)q; R_x(\pi/4)$.

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

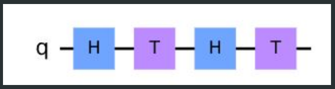


Figure 27: Arbitrary rotation gate

Since this is a single-qubit gate, we can think of it as a rotation about the Bloch sphere. That means that it's 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's an irrational multiple of π. You can prove this yourself with a bunch of math, but you can also see the 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 modulos 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 n times into 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 $n_2 - n_1$ 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 guarenteed to eb 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 = QuantumCircuit(1)
qc.t(0)
qc.h(0)
qc.t(0)
qc.h(0)
qc.draw()
```

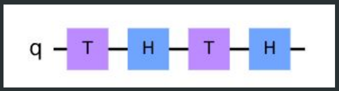


Figure 28: Arbitrary rotation gate

The axis that corresponds to this rotation is not always the same as that for the gate considered previously. We therefore now have arbitrary rotation around twoa xes, 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's 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 T gates can be used this way, and does not represent the most efficient method we know.

3 Proving Universality

3.1 Introduction

What can any given computer do? What are the limits of what is deemed computable, in general? These were questions tackled by Alan Turing before we even had a good idea of what a computer was, or how to build one.

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. But first we'll need to brush up on a few techniques.

3.2 Fun With Matrices

3.2.1 Matrices as Outer Products

In previous sections we have 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 yields a matrix by putting them in the opposite order. Recall this is called an outer product.

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

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

This also means we can write any matrix purely in terms of outer products. In the above example, 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.

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

This property also extends for any number n of qubits. We simply use the outer products of n dimensional strings.

3.2.2 Unitary and Hermitian Matrices

The Hermitian conjugate on a matrix $M^\dagger$ is equal to the complex conjugate of its transpose. This defines two families of matrices that are extremely important for quantum computing. One is the family of unitary matrices,

$$UU^\dagger = U^\dagger U = \mathbb{I}$$

This means the Hermitian conjugate of a unitary is its inverse - $U^\dagger$ has the power to undo U. All gates in quantum computing, except measurement and reset gates, can be represented by unitaries. Another consequence of unitary matrices is that it preserves the inner product between two arbitrary states. Specifically, for some $|\psi_0\rangle$ and $|\psi_1\rangle$, we have the inner product $\langle\psi_0 \,|\, \psi_1\rangle$. Applying the same unitary to both, we have

$$\left(\langle\psi_0|U^\dagger\right)\left(U|\psi_1\rangle\right) = \langle\psi_0|U^\dagger U|\psi_1\rangle = \langle\psi_0|\mathbb{I}|\psi_1\rangle = \langle\psi_0 \,|\, \psi_1\rangle$$

This property provides us with a useful way of thinking about these gates. It means that for any set of states $\{|\psi_j\rangle\}$ that provides an orthonormal basis for the system, the set of states $\{|\phi_j\rangle = U|\psi_j\rangle\}$ will also be an orthonormal basis. The unitary can then be thought of as a rotation between these bases, and can be defined as

$$U = \sum_j |\phi_j\rangle\langle\psi_j|$$

This is basically the quantum equivalent of 'truth tables' that describe classical Boolean gates.
The other important family of matrices are Hermitian matrices. These are matrices equal to their Hermitian conjugate:

$$H = H^\dagger$$

The matrices X, Y, Z, and H are all Hermitian (and coincidentally, unitary).
All Hermitian and unitary matrices have the property of being diagonalizable, meaning they can be written of the form

$$M = \sum_j \lambda_j |h_j\rangle\langle h_j|$$

where λ_j are the eigenvalues of the matrix and $|h_j\rangle$ are the corresponding eigenvectors.
For unitaries, $UU^\dagger = \mathbb{I}$ implies $\lambda_j\lambda_j^* = 1$. The eigenvalues are always complex values with amplitude 1, and can be expressed as e^{ih_j} for some real h_j:

$$\lambda_j \in \mathbb{C} \wedge |\lambda_j| = 1 \therefore \exists h_j \in \mathbb{R} \left(\lambda_j = e^{ih_j}\right)$$

For Hermitian matrices, the condition $H = H^\dagger$ implies $\lambda_j = \lambda_j^*$, thus $\lambda_j \in \mathbb{R}$.
These two types of matrices therefore only differ 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 by reusing the same eigenstates, and use the h_j from the e^{ih_j} as the corresponding eigenvalue.
Similarly, for every Hermitian matrix, there exists a unitary. We simply reuse eigenstates, and exponentiate h_j to e^{ih_j} to use as the eigenvalue. This can be expressed as

$$U = e^{iH}$$

Here we have used the standard definition of exponentiating a matrix, which has exactly the properties we require: preserving the eigenstates and exponentiating the eigenvalues.

$$e^H|v\rangle = e^\lambda|v\rangle$$

Thus we see by removing the vector and adding a multiple of i, we exponentiate the scalar quantities on the matrix but preserve the eigenvectors, since the matrix is still given of the same form by a Taylor series.

3.2.3 Pauli Decomposition

As we saw above, it's possible to write matrices completely in terms of outer products. Now we will see it's also possible to write them completely in terms of Pauli operators. For this, the key thing to note is that

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

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 $X|0\rangle = |1\rangle$, we have

$$|0\rangle\langle 1| = |0\rangle\langle 0|\sigma_x = \frac{\mathbb{I}+\sigma_z}{2}\sigma_x = \frac{\sigma_x + i\sigma_y}{2}$$

$$|1\rangle\langle 0|\sigma_x|0\rangle\langle 0| = \sigma_x\frac{\mathbb{I}+\sigma_z}{2} = \frac{\sigma_x - i\sigma_y}{2}$$

This makes use of the interesting properties:

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

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

Since we have all the outer products, we can now use this to write matrices in terms of Pauli matrices.

$$M = \begin{pmatrix}m_{00} & m_{01}\\ m_{10} & m_{11}\end{pmatrix} = m_{00}|0\rangle\langle 0| + m_{01}|1\rangle\langle 0| + m_{10}|0\rangle\langle 1| + m_{11}|1\rangle\langle 1|$$
$$M = m_{00}\frac{\mathbb{I}+\sigma_z}{2} + m_{01}\frac{\sigma_x + i\sigma_y}{2} + m_{10}\frac{\sigma_x - i\sigma_y}{2} + m_{11}\frac{1-\sigma_z}{2}$$
$$= \frac{m_{00}+m_{11}}{2}\mathbb{I} + \frac{m_{01}+m_{10}}{2}\sigma_x + \frac{m_{01}-m_{10}}{2}\sigma_y + \frac{m_{00}-m_{11}}{2}\sigma_z$$

This shows Pauli decomposition for a single-qubit operator, but this can be genralized to an n qubit system. We start from the observation that

$$\left(\frac{\mathbb{I}+\sigma_z}{2}\right)\otimes\left(\frac{\mathbb{I}+\sigma_z}{2}\right)\otimes\cdots\otimes\left(\frac{\mathbb{I}+\sigma_z}{2}\right) = |00\cdots 0\rangle\langle 00\cdots 0|$$

and can then proceed in the same manner as above with the same derivations. In the end it's generalized that any matrix can be expressed as the tensor products of Pauli matrices:

$$M = \sum_{P_{n-1},\cdots,P_0\in\{\mathbb{I},\sigma_x,\sigma_y,\sigma_z\}} C_{P_{n-1}\cdots P_0}\; P_{n-1}\otimes P_{n-2}\otimes\cdots\otimes P_0$$

For Hermitian matrices, the coefficients $C_{P_{n-1}\cdots P_0}$ will all be real.
This is hella confusing so I added a bit of context to make it understandable. The weird C thing is just a coefficient whos identity is dependent on the matrix being decomposed, and the weird tensor products are being summed for all 3 Pauli gates, effectively mimicking the above decomposition for a single qubit state but for an n qubit state.

3.3 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. The effect of this is to rotate from the input state to the output state.
One possible special case of this is that the input and output states describe bit strings expressed in quantum form. The mapping of each x to its output $f(x)$ can be described by some (reversible) classical computation. Any such computation could thus be expressed as some unitary

$$U = \sum_j |f(x)\rangle\langle x|$$

If we were able to implement any unitary, it would therefore mean we can achieve universality in the sense of standard digital computers.
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 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 means the ability to achieve any desired unitary on any desired number of qubits. If we have this, we can reproduce anything a classical system can do, simulate any quantum system, and do everything else that is possible of a quantum computer.

3.4 Basic Gate Sets

Whether or not we can build any unitary from a set of basic starting gates depends greatly on what gates we have access to. For every possible realization of fault-tolerant quantum computing, there is a set of operations that is the 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.

3.4.1 Clifford Gates

To understand Clifford gates, consider one we are familiar with: the Hadamard.

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

This gate has been expressed as outer products. While in this form, its famous effect becomes more obvious: it rotates $|0\rangle \rightarrow |+\rangle$, vice versa, and $|1\rangle \rightarrow |-\rangle$. More generally, it rotates the Z-basis states $\{|0\rangle, |1\rangle\}$ into the X-basis states $\{|+\rangle, |-\rangle\}$, and vice versa.
In this way, the Hadamard simply moves information around a qubit - it swaps information that would be measured by the Z-basis to information that would be measured by the X-basis.
The Hadamard can be combined with other gates to perform different operations, such as

$$HXH = Z \qquad HZH = X$$

By performing a Hadamard before and after an X, we transfer the action it previously applied to the Z basis states to the X states instead. The combined effect is that of a Z. Similarly, we create X from Hadamards about a Z. Similar behavior can be seen for the S gate and its Hermitian conjugate:

$$SXS^\dagger = Y \quad SYS^\dagger = -X \quad SZS^\dagger = Z$$

This has a similar affect to the Hadamard; it serves to transform the X basis into the Y basis, rather than Z. In combination with the Hadamard, we could then create a gate that shifts Y and Z.
This property of transforming Paulis into Paulis is the defining feature of Clifford gates. Stated explicitly for the single-qubit case: if U is a Clifford and P is a Pauli, $UPU^\dagger$ will also be a Pauli. For Hermitian gates, we take UPU.
Further examples of Clifford gates are the Paulis themselves. These do not transform the Pauli they act on, rather they assign a phase of -1 to the two they anti-communicate with. For example,

$$ZXZ = -Z \quad ZYZ = -Y \quad ZZZ = Z$$

Notably, a similar phase shift arose with the application of S gates. By combining this with a Pauli, we can create a composite gate swap that would cancel this phase, and swap X and Y more similarly to how the Hadamard swaps X and Z.
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 gate is the CNOT. The property of this that we make use of this chapter is

$$CX_{jk}(X \otimes \mathbb{I})CX_{jk} = X \otimes X$$

This effectively 'copies' an X from the control qubit over to the target qubit.
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 have the same eigenvalues.

3.4.2 Non-Clifford Gates

The Clifford gates are important, but not powerful on their own. In order to do any quantum computation, we need gates other than Cliffords. Three important examples are arbitrary rotations about the 3 axes, $R_x(\theta), R_y(\theta), R_z(\theta)$.
Consider $R_x(\theta)$. We saw any unitary can be expressed as an exponential form using a Hermitian matrix. For this gate, we have

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

The last section showed the unitary and its corresponding Hermitian matrix have equivalent eigenstates. We've also seen conjugation by a unitary changes 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 with a Clifford, we can therefore transform it into the same rotation about another axis. So even if we cannot directly apply $R_y(\theta), R_z(\theta)$, we can do so by combining $R_x(\theta)$ with Clifford gates. This technique of boosting the power of non-Clifford gates with Clifford gates is one we make great use of in quantum computing.

3.4.3 Expanding the Gate Set

As another example of combining $R_x(\theta)$ with Cliffords, let's conjugate it with CNOT.

$$CX_{jk}(R_x(\theta)\otimes\mathbb{I})CX_{jk} = CX_{jk}e^{i\frac{\theta}{2}(X\otimes\mathbb{I})}CX_{jk} = e^{i\frac{\theta}{2}CX_{jk}(X\otimes\mathbb{I})CX_{jk}} = e^{i\frac{\theta}{2}X\otimes X}$$

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

$$SXS^\dagger = Y$$
$$\Rightarrow (\mathbb{I}\otimes S)e^{i\frac{\theta}{2}\ X\otimes X}\left(\mathbb{I}\otimes S^\dagger\right) = e^{i\frac{\theta}{2}\ X\otimes Y}$$

With these techniques, we can make complex entangling operations that act on any arbitrary number of qubits, of the form

$$U = e^{i\frac{\theta}{2}P_{n-1}\otimes P_{n-2}\otimes\cdots\otimes P_0}, \quad P_j\in\{\mathbb{I},X,Y,Z\}$$

This all goes to show that combining the single and two-qubit Clifford gates with rotations about the x axis gives us a powerful set of possibilities. What's left to demonstrate is that we can use them to do anything.

3.5 Proving Universality

As for classical computers, we will need to split this big job up into manageable chunks. We'll need to find a set of basic gates that allow for universality. As we'll see, the single- and two-qubit gates of the last section will suffice.
Suppose we wish to implement the unitary

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

but the only gates we have are $R_x(\theta) = e^{i\frac{\theta}{2}X}$ and $R_z(\theta) = e^{i\frac{\theta}{2}Z}$. The best way to solve this problem would be to use Euler angles, but let's consider a different method instead.
The Hermitian matrix in the exponential U is simply the sum of those for the $R_x(\theta)$ and $R_z(\theta)$ rotations. This suggests a naive approach to solving our problem; we could apply $R_z(2b) = e^{ibZ}$ followed by $R_x(2a) = e^{iaX}$. Unfortunately since we're exponentiating matrices that don't commute

$$e^{iaX}e^{ibZ} \neq e^{i(aX+bZ)}$$

However, we could use the following modified version

$$U = lim_{n\to\infty}\left(e^{iaX/n}e^{ibZ/n}\right)^n$$

Here we split U up into n small slices. For each slice, it's a good approximation to say that

$$e^{iaX/n}e^{ibZ/n} = e^{i(aX+bZ)/n}$$

The error in this approximation scales as $1/n^2$. When we combine n slices, we get an approximation of our target unitary whos error scales as $1/n$. So by simply increasing the number of slices, we can get as close to U 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 more 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 $e^{i\frac{\theta}{2}X\otimes X\otimes X}$ from a single qubit $R_x\theta$ and two CNOTs.

```
from qiskit import QuantumCircuit
from qiskit.circuit import Parameter
theta = Parameter('theta')

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()
```

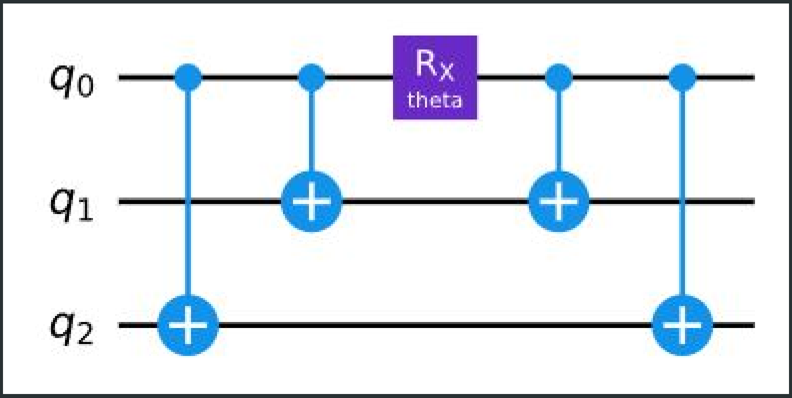


Figure 29: Circuit Diagram

With a few Hadamards, we can do the same for $e^{i\frac{\theta}{2}Z\otimes Z\otimes Z}$.

```
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()
```

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

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

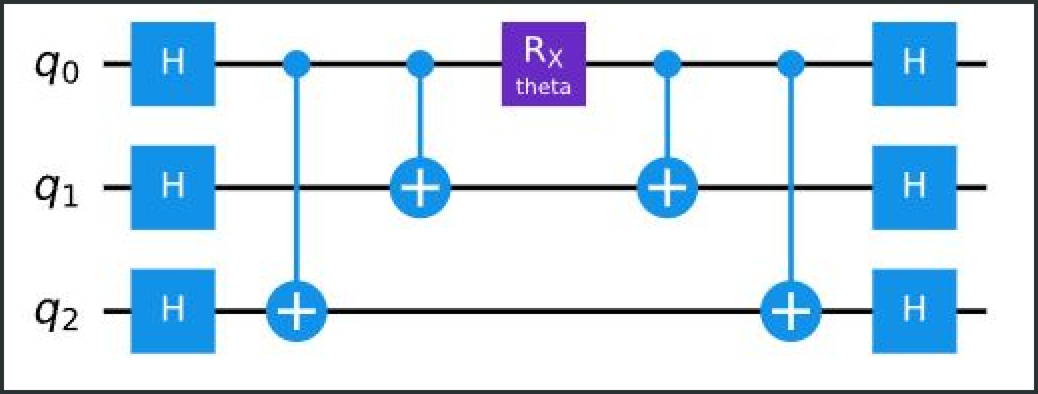


Figure 30: Circuit Diagram

As before, we can then continue to combine the slices together to get an arbitrarily accurate approximation of U.
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 $U = e^{iH}$ 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!
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. These other models often leverage other quantum mechanical properties in order to perform their computation. While these other forms of quantum computation exist, it is important to note that the benefits of each concern only physical and hardware problems. Since a universal model of quantum computation can perform any quantum algorithm, we need only stick with the circuit model and can ignore other universal models for our discussion.
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).

4 Classical Computation on a Quantum Computer

4.1 Introduction

One consequence of having a universal set of quantum gates is the ability to reproduce any classical computation. We simply need to compile the classical computation down into the Boolean logic gates that we saw in The Atoms of Computation, and then reproduce these on a quantum computer.

This demonstrates an important fact about quantum computers: they can do anything that a classical computer can do, and they can do so with at least the same computational complexity. Though it is not the aim to use quantum computers for tasks at which classical computers already excel, this is nevertheless a good demonstration that quantum computers can solve a general range of problems.

Furthermore, problems that require quantum solutions often involve components that can be tackled using classical algorithms. In some cases, these classical parts can be done on classical hardware. However, 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.

4.2 Consulting an Oracle

Many quantum algorithms are based around the analysis of some $f(x)$. Often these algorithms assume the existence of some 'black box' implementation of f, which we can give an input x and recieve 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.

In order to understand how an oracle works within a quantum algorithm, we need to be specific about how they are defined. One of the main forms that oracles take is *Boolean oracles*. These are described using the following unitary evolution,

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

Here, $|x,\bar{0}\rangle = |x\rangle \otimes |\bar{0}\rangle$ is used to represent a multi-qubit state consisting of two registers. The first register is in state $|x\rangle$, where x is a binary representation of the input to our function. The number of qubits in this register is the number of bits required to represent the inputs.

The job of the second register is to similarly encode the output. Specifically, the state of this register after applying U_f will be a binary representation of the output $|f(x)\rangle$, and this register will consist of as many qubits as are required for this. This initial state $|\bar{0}\rangle$ for this register represents the state for which all qubits are $|0\rangle$. For other initial states, applying U_f will lead to different results. The specific results that arise will depend on how we define the unitary U_f.

Another form of oracle is the *phase oracle*, which is defined as

$$P_f|x\rangle = (-1)^{f(x)}|x\rangle$$

where the output $f(x)$ is typically a simple bit value of 0 or 1.

Though it seems much different in form from the Boolean oracle, it's very much another expression of the same basic idea. In fact, it can be realized using the same 'phase kickback' mechanism as described previously.

To see this, consider the Boolean oracle U_f that would correspond to the same function. This can be implemented as something that is essentially a generalized form of the controlled-NOT. It's controlled on the input register, such that it leaves the output in a bit state $|0\rangle$ for $f(x) = 0$, and applies an X to flip it to $|1\rangle$ if $f(x) = 1$, since U_f is basically mapping $f(x)$ onto the vectors. If the initial state of the output register were $|-\rangle$ rather than $|0\rangle$, the effect of U_f would then be to induce exactly the phase of $(-1)^{f(x)}$ required.

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

Since the $|-\rangle$ is left unchanged by the XOR mapping of our U_f, it can be safely ignored. The end effect is that the phase oracle is simply implemented by the corresponding Boolean oracle.

4.3 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 it must be implemented instead with quantum gates. However, this is not as simple as taking the Boolean gates that can implement the classical algorithm, and replacing them with their quantum counterparts.
One issue that we must take care of is that of reversibility. A unitary of the form $U = \sum_x |f(x)\rangle\langle x|$ is only possible if every unique input x results in a unique output $f(x)$, which is not true in general. However, we can force it to be true by simply including a copy of the input in the output. It's this that leads us to the form for Boolean oracles we saw earlier

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

With the computation written as a unitary, we're able to consider the effects of applying it to a superposition. For example, let's take the superposition over all possible inputs x (unnormalized for simplicity). This will result in a superposition of all possible input/output pairs,

$$U_f \sum_x |x, 0\rangle = \sum_x |x, f(x)\rangle$$

When adapting classical algorithms, we also need to take care that these superpositions behave as we need them to. 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 be easily recovered by deleting them. From a quantum perspective, however, things aren't so easy.
For example, consider the case that a classical algorithm performs the following process,

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

Here we see a third register, which is used as a 'scratchpad' for the classical algorithm. We will refer to information that is left in this register at the end of the computation as the 'garbage', $g(x)$. Let's use V_f to denote a unitary that implements the above.
Quantum algorithms are typically built on 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 is, of course, trivial. However, we must ensure that our quantum computer is at least able to do such trivial things.
For example, suppose some process within our quantum computation has given us the superposition state $\sum_x |x, f(x)\rangle$, and we are required to return this to a state $\sum_x |x, 0\rangle$. For this we could simply

apply $U_f^\dagger$. The ability to apply this follows directly from knowing a circuit that would apply U_f, since we would simply need to replace each gate in the circuit with its inverse and reverse the order. However, suppose we don't know how to apply U_f, but instead know how to apply V_f. This means we can't apply $U_f^\dagger$ here, but we can use $V_f^\dagger$. Unfortunately, the presence of the garbage means that it won't have the same effect.
For an explicit example of this we can take a very simple case. We'll restrict x, $f(x)$, and $g(x)$ to all consist of just a single bit. We'll also use $f(x) = x$ and $g(x) = x$, each of which can be achieved with just a single `cx` gate controlled on the input register.
Specifically, the circuit to implement U_f is just the following single `cx` between the single bit of the input and output registers.

```
from qiskit import QuantumCircuit, QuantumRegister
input_bit = QuantumRegister(1, 'input')
output_bit = QuantumRegister(1, 'output')
garbage_bit = QuantumRegister(1, 'garbage')

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

Uf.draw()
```

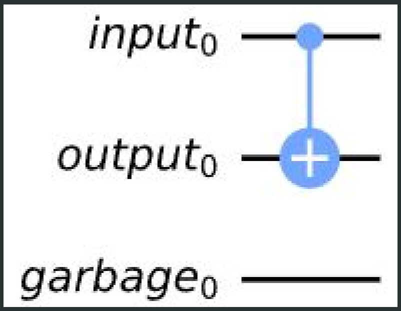


Figure 31: U_f acting on the circuit

For V_f, where we also need to make a copy of the input for the garbage, we can use the following two `cx` gates.

```
Vf = QuantumCircuit(input_bit, output_bit, garbage_bit)
Vf.cx(input_bit[0], garbage_bit[0])
Vf.cx(input_bit[0], output_bit[0])
Vf.draw()
```

Now we can look at the effect of first applying U_f, and then applying $V_f^\dagger$. The net effect is the following circuit:

```
qc = Uf.compose(Vf.inverse())
qc.draw()
```

This circuit begins with two identical `cx` gates, whose effects cancel each other out. All that remains is the final `cx` between the input and garbage registers. Mathematically, this means

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

Here we see that the action of $V_f^\dagger$ does not simply return us to the initial state, but leaves the first qubit entangled with unwanted garbage. Any subsequent steps in the algorithm will therefore not

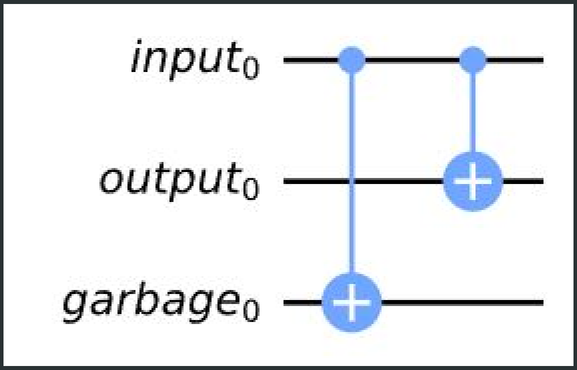


Figure 32: V_f acting on the circuit

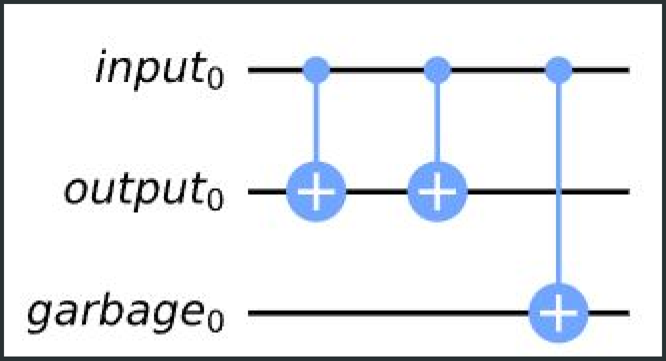


Figure 33: $V_f^\dagger$ acting on the circuit

run as expected, since this state is not the one we need.
For this reason we need a way of removing classical garbage from our quantum algorithms. This can be done by a method known as 'uncomputation'. We simply need to take another blank variable and apply V_f.

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

Then we apply a set of controlled-NOT gates, each controlled on one of the qubits used to encode the output, and targeted on the corresponding qubit in the extra blank variable.
Here's the circuit to do this for our example using single qubit registers.

```
final_output_bit = QuantumRegister(1, 'final-output')

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

copy.draw()
```

The effect of this is to copy the informatino over. Specifically, it transforms the state in the following way.

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

Finally we apply $V_f^\dagger$, which undoes the original computation.

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

The copied output nevertheless remains. The net effect is to perform the computation without garbage, and hence achieves our desired U_f.

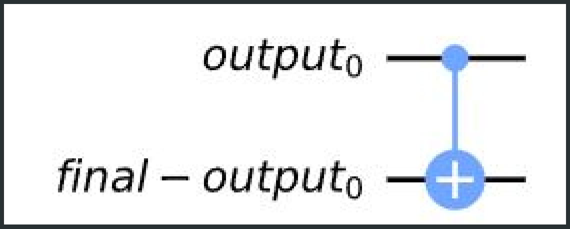


Figure 34: CX acting on the circuit

For example using single qubit registers, and for which $f(x) = x$, the whole process corresponds to the following circuit.

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

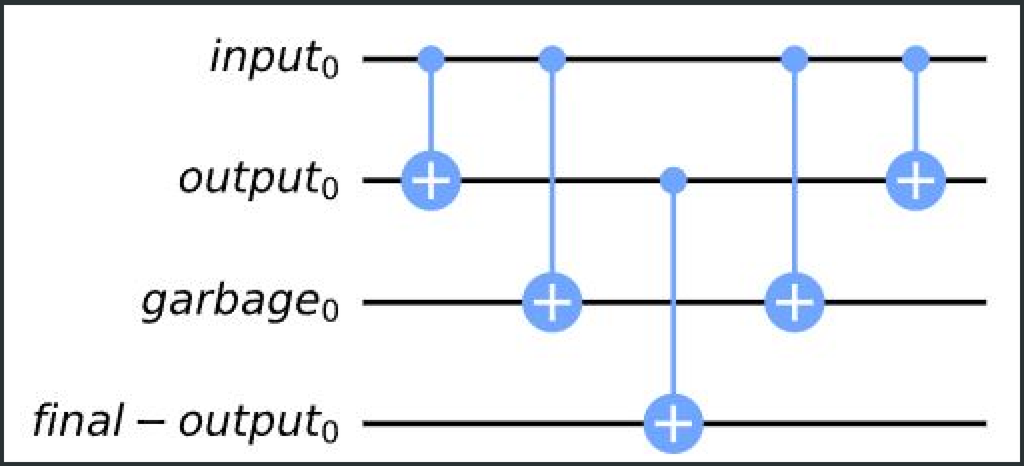


Figure 35: U_f acting on the circuit through V_f

Using what you know so far of how the `cx` gates work, you should be able to see that the two applied to the garbage register will cancel each other out. We have therefore successfully removed the garbage.
With this method, and all others covered in this chapter, we now have the tools we need to create quantum algorithms. Now we can move on to seeing those algorithms in action.

4.4 Quick Exercises

1. Show that the output is correctly written to the 'final input' 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 $|1\rangle$.