Universidad Internacional de la Rioja (UNIR)

Escuela Superior de Ingeniería y Tecnología

Máster en Computación Cuántica

Computación cuántica

Actividad 2. Ejercicios con puertas lógicas cuánticas

Actividad de la asignatura

presentada por: Albert Nieto Morales, Javier Hernanz Zajara

Profesor: Francisco José Orts Gómez

Fecha: 29 de diciembre de 2023

December 29, 2023

1 Información del Documento

Este documento forma parte de los recursos del Máster en Computación Cuántica en la Universidad Internacional de La Rioja (UNIR).

1.1 Detalles del Cuaderno

- **Asignatura:** Computación Cuántica
- **Autores:** Albert Nieto Morales, Javier Hernanz Zajara
- **Fecha:** 2023-12-29

1.2 Disponibilidad

Este cuaderno y otros recursos relacionados están disponibles en el repositorio en GitHub.

1.3 Nota Importante

Este material educativo se enfoca en la Computación Cuántica y ha sido desarrollado como parte del programa de Máster en Computación Cuántica en UNIR. La información proporcionada en este documento puede estar sujeta a actualizaciones. Se recomienda verificar la fecha de la última actualización para asegurarse de contar con la información más reciente.

2 Enunciado y librerías

2.1 Enunciado

A través de esta actividad podrás adquirir las competencias necesarias para utilizar las puertas cuánticas básicas, así como la evolución del estado cuántico, los estados de Bell y la implementación de un incrementador.

2.2 Librerías

Los paquetes necesarios para realizar la actividad son:

- `qiskit`: La biblioteca principal de Qiskit.
- `qiskit_ibm_provider`: Proporciona acceso a los servicios en la nube de IBM Quantum.
- `qiskit-aer`: Proporciona acceso a simuladores cuánticos.

[1]:
```
%%capture
%pip install qiskit
```

```
%pip install qiskit_ibm_provider
%pip install qiskit-aer
```

[2]:
```
# Importing standard Qiskit libraries
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit,␣
 ↪QuantumCircuit, transpile, Aer
from qiskit_ibm_provider import IBMProvider
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from qiskit.circuit.library import C3XGate

# Importing matplotlib
import matplotlib.pyplot as plt

# Importing Numpy, Cmath and math
import numpy as np
import os, math, cmath
from numpy import pi

# Other imports
from IPython.display import display, Math, Latex
```

Cargamos y actualizamos el entorno con las variables de entorno guardadas en `config.env`.

[3]:
```
# Specify the path to your env file
env_file_path = 'config.env'

# Load environment variables from the file
os.environ.update(line.strip().split('=', 1) for line in open(env_file_path) if␣
 ↪'=' in line and not line.startswith('#'))

# Load IBM Provider API KEY
IBMP_API_KEY = os.environ.get('IBMP_API_KEY')
```

Luego, extraemos la clave de API de IBM Quantum Provider y se guarda en la variable `IBMP_API_KEY`.

[4]:
```
# Loading your IBM Quantum account(s)
IBMProvider.save_account(IBMP_API_KEY, overwrite=True)

# Run the quantum circuit on a statevector simulator backend
backend = Aer.get_backend('statevector_simulator')
```

3 Ejercicio 1

Implementar cuatro circuitos cuánticos de forma que cada uno de ellos haga evolucionar el estado del sistema a cada uno de los cuatro estados de Bell. Describe la evolución del sistema paso a paso

de dos formas: con notación de Dirac y en forma matricial. Finalmente, implementa los circuitos utilizando QISKIt Quantum Lab y verifica que los resultados son los esperados.

3.1 ¿Qué son los estados de Bell?

En computación cuántica, los Estados de Bell son un conjunto de cuatro estados cuánticos maximamente entrelazados. Estos estados, también conocidos como "pares EPR" o "qubits entrelazados", son generados mediante una serie de operaciones cuánticas específicas. Los cuatro Estados de Bell se denotan como:

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|0\rangle_A \otimes |0\rangle_B + |1\rangle_A \otimes |1\rangle_B) = \frac{|00\rangle + |11\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

$$|\Phi^-\rangle = \frac{1}{\sqrt{2}}(|0\rangle_A \otimes |0\rangle_B - |1\rangle_A \otimes |1\rangle_B) = \frac{|00\rangle - |11\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}|00\rangle - \frac{1}{\sqrt{2}}|11\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$$

$$|\Psi^+\rangle = \frac{1}{\sqrt{2}}(|0\rangle_A \otimes |1\rangle_B + |1\rangle_A \otimes |0\rangle_B) = \frac{|01\rangle + |10\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|10\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$

$$|\Psi^-\rangle = \frac{1}{\sqrt{2}}(|0\rangle_A \otimes |1\rangle_B - |1\rangle_A \otimes |0\rangle_B) = \frac{|01\rangle - |10\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}}|01\rangle - \frac{1}{\sqrt{2}}|10\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$$

3.2 Estado de Bell $|\Phi^+\rangle$

El objetivo es llegar al estado $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ partiendo del estado base $|00\rangle$

Desarrollo usando notación de Dirac

1. Iniciamos con dos qubits $|A\rangle$ y $|B\rangle$ inicializados a $|0\rangle$, representados también como $|0\rangle_A$ y $|0\rangle_B$.
2. Aplicamos una puerta Hadamard sobre $|0\rangle_A$ tal que:

   $$\langle H|A\rangle$$

   Lo cual es equivalente a

   $$\frac{1}{\sqrt{2}}\left(|0\rangle_A + |1\rangle_A\right)$$

3. Dado que tenemos 2 qubits en nuestro sistema, representamos el producto tensorial como:

   $$\langle H|A\rangle \otimes |B\rangle$$

Equivalente a

$$\frac{1}{\sqrt{2}}\left(|00\rangle + |10\rangle\right)$$

4. Finalmente, realizamos la operación CNOT en $|B\rangle$ controlado por $|A\rangle$:

$$\langle \text{CNOT}| \left(\langle H|A\rangle \otimes |B\rangle\right)\rangle$$

Resultando en

$$\frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right)$$

3.2.1 Desarrollo utilizando Notación Matricial

1. **Definición de Qubits y Puertas Cuánticas:**
   - Definimos los qubits $|0\rangle_A = |0\rangle_B = \begin{bmatrix}1\\0\end{bmatrix}$.
   - La puerta Hadamard se expresa como $H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$.
   - La puerta CNOT se representa como $\text{CNOT} = \begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{bmatrix}$.
2. **Aplicación de la Puerta Hadamard a $|A\rangle$:**
   - Aplicamos la puerta Hadamard a $|A\rangle$ de la siguiente manera:

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

3. **Cálculo del Estado del Sistema con $|B\rangle$:**
   - Calculamos el estado del sistema considerando $|B\rangle$ mediante el producto tensorial:

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

4. **Aplicación de la Puerta CNOT entre Qubits:**
   - Finalmente, aplicamos la puerta CNOT entre ambos qubits:

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

Este resultado corresponde a la representación matricial del estado deseado.

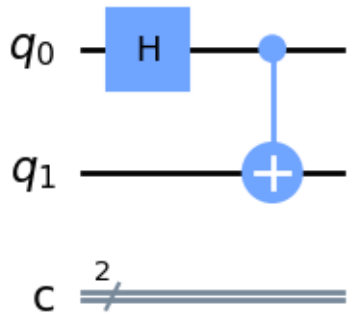
3.2.2 Implementación y comprobación

A continuación, se implementan los circuito descrito anteriormente y se verifica haciendo uso de Qiskit.

[5]:
```
qc_b1 = QuantumCircuit(2, 2)
qc_b1.h(0)
qc_b1.cx(0, 1)
qc_b1.draw(output='mpl', style="iqp")
```

```
/opt/conda/lib/python3.10/site-
packages/qiskit/visualization/circuit/matplotlib.py:282: UserWarning: Style JSON
file 'iqp.json' not found in any of these locations:
/opt/conda/lib/python3.10/site-
packages/qiskit/visualization/circuit/styles/iqp.json, iqp.json. Will use
default style.
  self._style, def_font_ratio = load_style(self._style)
```

[5]:



[6]:
```
sv = backend.run(qc_b1).result().get_statevector()
sv.draw(output='latex', prefix = "|\Phi^+\\rangle = ")
```

[6]:

$$|\Phi^+\rangle = \frac{\sqrt{2}}{2}|00\rangle + \frac{\sqrt{2}}{2}|11\rangle$$

3.3 Estado de Bell $|\Phi^-\rangle$

El objetivo es llegar al estado $|\Phi^-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$ partiendo del estado base $|00\rangle$

3.3.1 Desarrollo usando notación de Dirac

1. Partimos de dos qubits $|A\rangle$ y $|B\rangle$ inicializados a $|0\rangle$, ilustrados también como $|0\rangle_A$ y $|0\rangle_B$.
2. Aplicamos una puerta Pauli X (X) sobre $|0\rangle_A$ tal que:

$$\langle X|A\rangle$$

Equivalente a

$$|1\rangle_A$$

3. A continuación, aplicamos una puerta Hadamard (H) sobre $|0\rangle_A$ tal que:

$$\langle H|X|A\rangle$$

Equivalente a

$$\frac{1}{\sqrt{2}}(|0\rangle_A - |1\rangle_A)$$

4. Dado que tenemos 2 qubits en nuestro sistema, representamos el producto tensorial como:

$$\langle H|X|A\rangle \otimes |B\rangle$$

Equivalente a

$$\frac{1}{\sqrt{2}}(|00\rangle - |10\rangle)$$

5. Por último, necesitamos realizar la operación CNOT en $|B\rangle$ controlado por $|A\rangle$, por lo que tendríamos:

$$\langle \text{CNOT}|(\langle H|X|A\rangle \otimes |B\rangle)\rangle$$

Equivalente a

$$\frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$$

3.3.2 Desarrollo usando Notación Matricial

1. **Definición de Qubits y Puertas Cuánticas:**
   - Definimos los qubits $|0\rangle_A = |0\rangle_B = \begin{bmatrix}1\\0\end{bmatrix}$.
   - La puerta Hadamard se expresa como $H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$.
   - La puerta CNOT se representa como $\text{CNOT} = \begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{bmatrix}$.
   - La puerta X (Pauli X) se define como $X = \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$.

2. **Aplicación de Puertas X y Hadamard a $|A\rangle$:**
   - Aplicamos la puerta X y, a continuación, la Hadamard sobre $|A\rangle$ de la siguiente manera:

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

3. **Cálculo del Estado del Sistema con $|B\rangle$:**
   - Calculamos el estado del sistema considerando $|B\rangle$ mediante el producto tensorial:

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

4. **Aplicación de la Puerta CNOT entre Qubits:**
   - Finalmente, aplicamos la puerta CNOT entre ambos qubits:

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

Este resultado corresponde a la representación matricial del estado buscado.

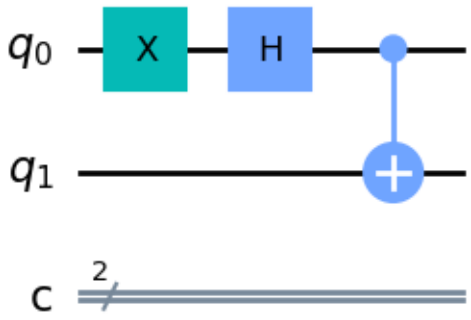
3.3.3 Implementación y comprobación

A continuación, se implementan los circuito descrito anteriormente y se verifica haciendo uso de Qiskit.

[7]:
```
qc_b2 = QuantumCircuit(2, 2)
qc_b2.x(0)
qc_b2.h(0)
qc_b2.cx(0, 1)
qc_b2.draw(output='mpl', style="iqp")
```

```
/opt/conda/lib/python3.10/site-
packages/qiskit/visualization/circuit/matplotlib.py:282: UserWarning: Style JSON
file 'iqp.json' not found in any of these locations:
/opt/conda/lib/python3.10/site-
packages/qiskit/visualization/circuit/styles/iqp.json, iqp.json. Will use
default style.
  self._style, def_font_ratio = load_style(self._style)
```

[7]:



[8]:
```
sv = backend.run(qc_b2).result().get_statevector()
sv.draw(output='latex', prefix = "|\Phi^-\\rangle = ")
```

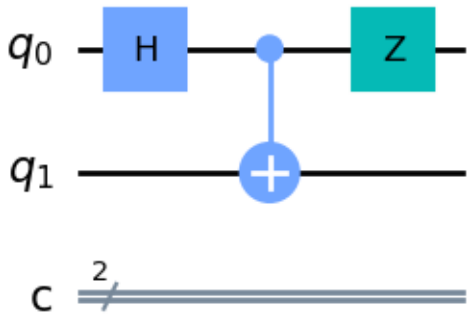
[8]:

$$|\Phi^-\rangle = \frac{\sqrt{2}}{2}|00\rangle - \frac{\sqrt{2}}{2}|11\rangle$$

De la misma manera, conseguimos el mismo estado con la combinación de las puertas H, CNOT y Z. Las comprobaciones con el desarrollo matricial y en notación de Dirac se puede encontrar en el apéndice.

[9]:
```
qc_b2 = QuantumCircuit(2, 2)
qc_b2.h(0)
qc_b2.cx(0, 1)
qc_b2.z(0)
qc_b2.draw(output='mpl', style="iqp")
```

[9]:



[10]:
```
sv = backend.run(qc_b2).result().get_statevector()
sv.draw(output='latex', prefix = "|\Phi^-\\rangle = ")
```

[10]:

$$|\Phi^-\rangle = \frac{\sqrt{2}}{2}|00\rangle - \frac{\sqrt{2}}{2}|11\rangle$$

3.4 Estado de Bell $|\Psi^+\rangle$

El objetivo es llegar al estado $|\Psi^+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$ partiendo del estado base $|00\rangle$

3.4.1 Desarrollo usando notación de Dirac

1. Partimos de dos qubits $|A\rangle$ y $|B\rangle$ inicializados a $|0\rangle$, ilustrados también como $|0\rangle_A$ y $|0\rangle_B$.
2. Aplicamos una puerta Pauli X (X) sobre $|0\rangle_B$ tal que:

   $$\langle X|B\rangle$$

   Equivalente a

   $$|1\rangle_B$$

3. En paralelo, aplicamos una puerta Hadamard (H) sobre $|0\rangle_A$ tal que:

   $$\langle H|A\rangle$$

   Equivalente a

   $$\frac{1}{\sqrt{2}}(|0\rangle_A + |1\rangle_A)$$

4. Dado que tenemos 2 qubits en nuestro sistema, representamos el producto tensorial como:

   $$\langle H|A\rangle \otimes \langle X|B\rangle$$

   Equivalente a

   $$\frac{1}{\sqrt{2}}(|01\rangle + |11\rangle)$$

5. Por último, necesitamos realizar la operación CNOT en $|B\rangle$ controlado por $|A\rangle$, por lo que tendríamos:

   $$\langle \text{CNOT}|(\langle H|A\rangle \otimes \langle X|B\rangle)\rangle$$

   Equivalente a

   $$\frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$

3.4.2 Desarrollo usando Notación Matricial

1. **Definición de Qubits y Puertas Cuánticas:**
   - Definimos los qubits $|0\rangle_A = |0\rangle_B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$.
   - La puerta Hadamard se expresa como $H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$.
   - La puerta CNOT se representa como CNOT $= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$.
   - La puerta X (Pauli X) se define como $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.
2. **Aplicación de la Puerta X sobre $|B\rangle$:**
   - Aplicamos la puerta X sobre $|B\rangle$ de la siguiente manera:

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

3. **Aplicación de la Puerta Hadamard sobre $|A\rangle$:**
   - Aplicamos la puerta Hadamard sobre $|A\rangle$ de la siguiente manera:

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

4. **Cálculo del Estado del Sistema con $|B\rangle$:**
   - Calculamos el estado del sistema considerando $|B\rangle$ mediante el producto tensorial:

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

5. **Aplicación de la Puerta CNOT entre Qubits:**
   - Finalmente, aplicamos la puerta CNOT entre ambos qubits:

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

   Este resultado corresponde a la representación matricial del estado buscado.

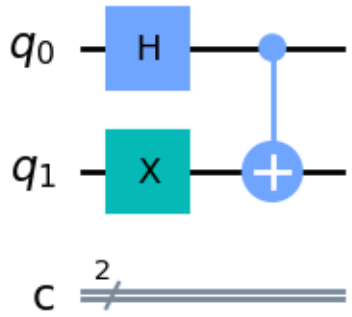
3.4.3 Implementación y comprobación

A continuación, se implementan los circuito descrito anteriormente y se verifica haciendo uso de Qiskit.

[11]:
```
qc_b3 = QuantumCircuit(2, 2)
qc_b3.x(1)
qc_b3.h(0)
```

```
qc_b3.cx(0, 1)
qc_b3.draw(output='mpl', style="iqp")
```

[11]:



[12]:
```
sv = backend.run(qc_b3).result().get_statevector()
sv.draw(output='latex', prefix = "|\Psi^+\\rangle = ")
```

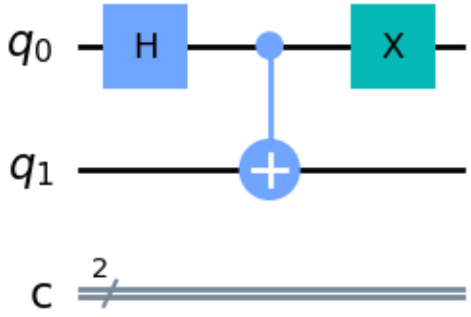
[12]:

$$|\Psi^+\rangle = \frac{\sqrt{2}}{2}|01\rangle + \frac{\sqrt{2}}{2}|10\rangle$$

De la misma manera, conseguimos el mismo estado con la combinación de las puertas H, CNOT y X. Las comprobaciones con el desarrollo matricial y en notación de Dirac se puede encontrar en el apéndice.

[13]:
```
qc_b3 = QuantumCircuit(2, 2)
qc_b3.h(0)
qc_b3.cx(0, 1)
qc_b3.x(0)
qc_b3.draw(output='mpl', style="iqp")
```

[13]:



[14]:
```
sv = backend.run(qc_b3).result().get_statevector()
sv.draw(output='latex', prefix = "|\Psi^+\\rangle = ")
```

[14]:

$$|\Psi^+\rangle = \frac{\sqrt{2}}{2}|01\rangle + \frac{\sqrt{2}}{2}|10\rangle$$

3.5 Estado de Bell $|\Psi^-\rangle$

El objetivo es llegar al estado $|\Psi^-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$ partiendo del estado base $|00\rangle$

3.5.1 Desarrollo usando notación de Dirac

1. Partimos de dos qubits $|A\rangle$ y $|B\rangle$ inicializados a $|0\rangle$, ilustrados también como $|0\rangle_A$ y $|0\rangle_B$.
2. Aplicamos una puerta Pauli X (X) sobre $|0\rangle_B$ tal que:

   $$\langle X|B\rangle$$

   Equivalente a

   $$|1\rangle_B$$

3. Aplicamos una puerta Pauli X (X) sobre $|0\rangle_A$ tal que:

   $$\langle X|A\rangle$$

   Equivalente a $|1\rangle_A$
4. A continuación, aplicamos una puerta Hadamard (H) en $|A\rangle$ tal que:

   $$\langle H|X|A\rangle$$

   Equivalente a

   $$\frac{1}{\sqrt{2}}(|0\rangle_A - |1\rangle_A)$$

5. Dado que tenemos 2 qubits en nuestro sistema, representamos el producto tensorial como:

   $$\langle H|X|A\rangle \otimes \langle X|B\rangle$$

   Equivalente a

   $$\frac{1}{\sqrt{2}}(|01\rangle - |11\rangle)$$

6. Por último, necesitamos realizar la operación CNOT en $|B\rangle$ controlado por $|A\rangle$, por lo que tendríamos:

$$\langle \text{CNOT}|(\langle H|X|A\rangle \otimes \langle X|B\rangle)\rangle$$

Equivalente a

$$\frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$$

3.5.2 Desarrollo usando Notación Matricial

1. **Definición de Qubits y Puertas Cuánticas:**
   - Definimos los qubits $|0\rangle_A = |0\rangle_B = \begin{bmatrix}1\\0\end{bmatrix}$.
   - La puerta Hadamard se expresa como $H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\1 & -1\end{bmatrix}$.
   - La puerta CNOT se representa como $\text{CNOT} = \begin{bmatrix}1 & 0 & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 0 & 1\\0 & 0 & 1 & 0\end{bmatrix}$.
   - La puerta X (Pauli X) se define como $X = \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$.
2. **Aplicación de la Puerta X sobre $|B\rangle$:**
   - Aplicamos la puerta X sobre $|B\rangle$ de la siguiente manera:

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

3. **Aplicación de Puertas X y Hadamard, en ese orden, sobre $|A\rangle$:**
   - Aplicamos las puertas X y Hadamard, en ese orden, sobre $|A\rangle$ de la siguiente manera:

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

4. **Cálculo del Estado del Sistema con $|B\rangle$:**
   - Calculamos el estado del sistema considerando $|B\rangle$ mediante el producto tensorial:

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

5. **Aplicación de la Puerta CNOT entre Qubits:**
   - Finalmente, aplicamos la puerta CNOT entre ambos qubits:

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

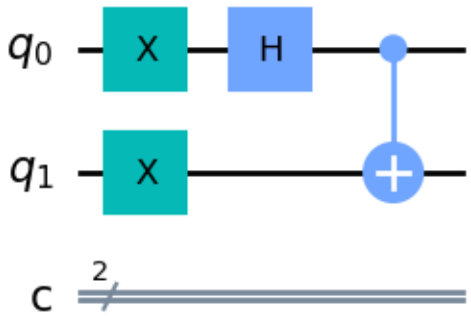
Este resultado corresponde a la representación matricial del estado buscado.

3.5.3 Implementación y comprobación

A continuación, se implementan los circuito descrito anteriormente y se verifica haciendo uso de Qiskit.

[15]:
```
qc_b4 = QuantumCircuit(2, 2)
qc_b4.x(0)
qc_b4.h(0)
qc_b4.x(1)
qc_b4.cx(0, 1)
qc_b4.draw(output='mpl', style="iqp")
```

[15]:



[16]:
```
sv = backend.run(qc_b4).result().get_statevector()
sv.draw(output='latex', prefix = "|\Psi^-\\rangle = ")
```

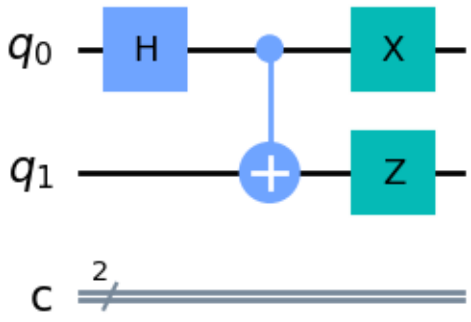
[16]:

$$|\Psi^-\rangle = -\frac{\sqrt{2}}{2}|01\rangle + \frac{\sqrt{2}}{2}|10\rangle$$

De la misma manera, conseguimos el mismo estado con la combinación de las puertas H, CNOT X y Z. Las comprobaciones con el desarrollo matricial y en notación de Dirac se puede encontrar en el apéndice.

[17]:
```
qc_b4 = QuantumCircuit(2, 2)
qc_b4.h(0)
qc_b4.cx(0, 1)
qc_b4.x(0)
qc_b4.z(1)
qc_b4.draw(output='mpl', style="iqp")
```

[17]:



[18]:
```
sv = backend.run(qc_b4).result().get_statevector()
sv.draw(output='latex', prefix = "|\Psi^-\\rangle = ")
```

[18]:

$$|\Psi^-\rangle = \frac{\sqrt{2}}{2}|01\rangle - \frac{\sqrt{2}}{2}|10\rangle$$

4 Ejercicio 2

Implementar los circuitos de cuatro cúbits que suman al registro cuántico los valores 1, 2, 3, 4, 5, 6, 7 y 8 y verificar que funcionan correctamente. Utilizar QISKit Quantum lab.

[19]:
```
def sv_latex_from_qc(qc, backend):
    sv = backend.run(qc).result().get_statevector()
    return sv.draw(output='latex')
```

Creamos un circuito cuántico de 4 qubits

[20]:
```
qc_ej2 = QuantumCircuit(4, 4)
qc_ej2.x(0)
sv_latex_from_qc(qc_ej2, backend)
```

[20]:

$$|0001\rangle$$

[21]:
```
qc_ej2 = QuantumCircuit(4, 4)
qc_ej2.x(1)
sv_latex_from_qc(qc_ej2, backend)
```

[21]:

$$|0010\rangle$$

[22]:
```
qc_ej2 = QuantumCircuit(4, 4)
qc_ej2.x(0)
qc_ej2.x(1)
sv_latex_from_qc(qc_ej2, backend)
```

[22]:

$$|0011\rangle$$

[23]:
```
qc_ej2 = QuantumCircuit(4, 4)
qc_ej2.x(2)
sv_latex_from_qc(qc_ej2, backend)
```

[23]:

$$|0100\rangle$$

[24]:
```
qc_ej2 = QuantumCircuit(4, 4)
qc_ej2.x(0)
qc_ej2.x(2)
sv_latex_from_qc(qc_ej2, backend)
```

[24]:

$$|0101\rangle$$

[25]:
```
qc_ej2 = QuantumCircuit(4, 4)
qc_ej2.x(0)
qc_ej2.x(2)
sv_latex_from_qc(qc_ej2, backend)
```

[25]:

$$|0101\rangle$$

[26]:
```
qc_ej2 = QuantumCircuit(4, 4)
qc_ej2.x(0)
qc_ej2.x(1)
qc_ej2.x(2)
sv_latex_from_qc(qc_ej2, backend)
```

[26]:

$$|0111\rangle$$

[27]:
```
qc_ej2 = QuantumCircuit(4, 4)
qc_ej2.x(3)
sv_latex_from_qc(qc_ej2, backend)
```

[27]:

$$|1000\rangle$$

Puesto que la generación de circuitos de adición binaria es conceptualmente sencilla de automatizar, en lugar de crear los circuitos de suma pedidos en el enunciado vamos a definir una función que los cree en base a un parámetro (el valor a sumar). Veamos la implementación de esta función:

4.0.1 `circuit_adder` Function:

Purpose: The `circuit_adder` function is designed to create a quantum circuit to add a number to an existing 4-qubit circuit.

Attributes: - `num` (int): A number to add to the circuit. It must be between 0 and 8.

Methods: None

Example Usage: "'python qc_adder_7 = circuit_adder(7) qc_final = qc_already_exist.compose(qc_adder_7) # Concatenate both quantum circuits to add 7 to the already existing qc_already_exist quantum circuit

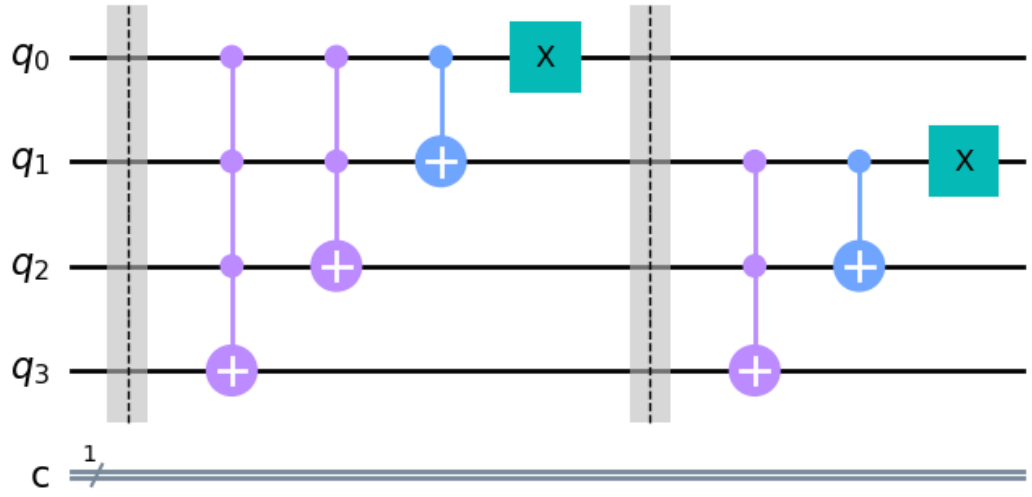
[28]:
```
def circuit_adder (num):
    if num<1 or num>8:
        raise ValueError("Out of range")  ## El enunciado limita el sumador a␣
↪los valores entre 1 y 8. Quitar esta restricción sería directo.
    # Definición del circuito base que vamos a construir
    qreg_q = QuantumRegister(4, 'q')
    creg_c = ClassicalRegister(1, 'c')
    circuit = QuantumCircuit(qreg_q, creg_c)

    qbit_position = 0
    for element in reversed(np.binary_repr(num)):
        if (element=='1'):
            circuit.barrier()
            match qbit_position:
                case 0: # +1
                    circuit.append(C3XGate(), [qreg_q[0], qreg_q[1], qreg_q[2],␣
↪qreg_q[3]])
                    circuit.ccx(qreg_q[0], qreg_q[1], qreg_q[2])
                    circuit.cx(qreg_q[0], qreg_q[1])
                    circuit.x(qreg_q[0])
                case 1: # +2
                    circuit.ccx(qreg_q[1], qreg_q[2], qreg_q[3])
                    circuit.cx(qreg_q[1], qreg_q[2])
                    circuit.x(qreg_q[1])
                case 2: # +4
                    circuit.cx(qreg_q[2], qreg_q[3])
                    circuit.x(qreg_q[2])
                case 3: # +8
                    circuit.x(qreg_q[3])
        qbit_position+=1
    return circuit
```

Probamos la función generadora de circuitos generando un sumador para el número binario 3

[29]:
```
add_3 = circuit_adder(3)
add_3.draw(output='mpl', style="iqp")
```

[29]:

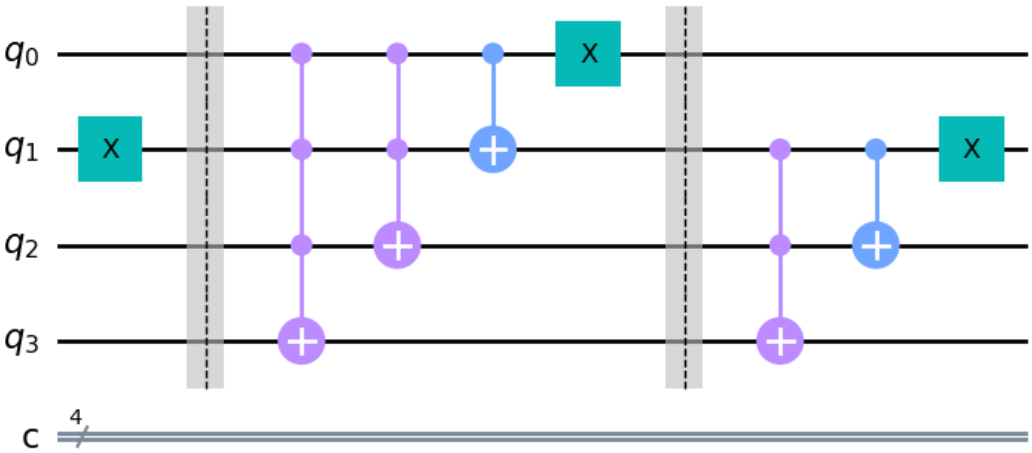


A continuación, reutilizaremos el circuito anterior (+3) y creamos un circuito cuántico que represente el número 2 para validar que genera un valor equivalente a 5.

[30]:
```
qc_test_2 = QuantumCircuit(4, 4)
qc_test_2.x(1)

qc_test_2_plus_3 = qc_test_2.compose(add_3)
qc_test_2_plus_3.draw(output='mpl', style="iqp")
```

[30]:



Y validamos su resultado:

[31]:
```
sv_latex_from_qc(qc_test_2_plus_3, backend)
```

[31]:

$$|0101\rangle$$

Realizamos otra prueba sumando 8 a un estado previo de 7:

[32]:
```
qc_test_7 = QuantumCircuit(4, 4)
qc_test_7.x(0)
qc_test_7.x(1)
qc_test_7.x(2)

qc_test_7_plus_8 = qc_test_7.compose(circuit_adder(8))
sv_latex_from_qc(qc_test_7_plus_8, backend)
#qc_test_7_plus_8.draw()
```

[32]:

$$|1111\rangle$$

Con lo que validamos que se realizan todas las sumas correctamente, incluyendo aquellas que requieren de acarreo.

5 Ejercicio 3

Implementar en Python el algoritmo de teleportación cuántica para teleportar el estado $|\Psi\rangle$ del cúbit de Alice al cúbit de Bob. El estado $|\Psi\rangle$ viene determinado por los ángulos $\theta = 37.5$ grados y $\phi = 13.4$ grados, de la esfera de Bloch.

En primer lugar convertiremos los ángulos proporcionados por el enunciado a radianes, que nos permitan operar correctamente en el sistema:

- $\theta = 37.5$ grados $= 6.544985$ radianes
- $\phi = 13.4$ grados $= 2.338741$ radianes

Utilizaremos θ como el ángulo de rotación en el eje X del qubit, mientras que ϕ representará la rotación en el eje Z.

[33]:
```
theta = 6.544985
phi = 2.338741
lmbda = 0
```

Para realizar este algoritmo, Alice dispone de dos qubits y Bob dispone de uno. En el circuito, los dos primeros qubits serán los de Alice y el tercero será de Bob

[34]:
```
alice_1 = 0
alice_2 = 1
bob_1 = 2
```

Creamos un circuito cuántico con 3 cúbits, los dos de Alice y uno de Bob.

[35]:
```
qr_alice = QuantumRegister(2, 'Alice')
qr_bob = QuantumRegister(1, 'Bob')
cr = ClassicalRegister(3, 'c')
qc_ej3 = QuantumCircuit(qr_alice, qr_bob, cr)
```

Inicializamos el qubit 1 de Alice con los ángulos proporcionados, donde theta, phi, y lmbda son los 3 ángulos de Euler [1].

[36]:
```
qc_ej3.barrier(label='1')
qc_ej3.u(theta, phi, lmbda, alice_1);
```

En el algoritmo de teletransportación cuántica, la creación de un par entrelazado se realiza típicamente utilizando un estado Bell específico. La elección estándar es el estado Bell maximalmente entrelazado $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$ [2].

[37]:
```
qc_ej3.barrier(label='2')
qc_ej3.h(alice_2)
qc_ej3.cx(alice_2, bob_1);
```

Alice entrelaza su qubit con el estado cuántico [3].

[38]:
```
qc_ej3.barrier(label='3')
qc_ej3.cx(alice_1, alice_2)
qc_ej3.h(alice_1);
```

Alice mide sus qubits y envía los resultados a Bob a través de un canal de comunicación clásico [4].

[39]:
```
qc_ej3.barrier(label='4')
qc_ej3.measure([alice_1, alice_2], [alice_1, alice_2]);
```

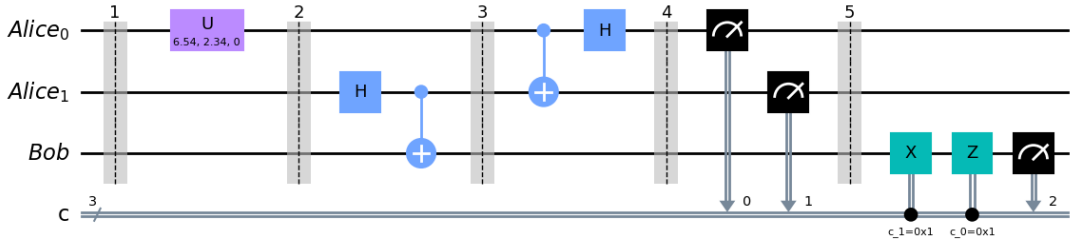
Bob corrige su qubit basado en las mediciones de Alice y mide el resultado. Aplica la puerta Pauli-X si el valor del qubit 1 de Alice ha sido 1 y/o aplica la puerta Pauli-Z si el valor del qubit 2 de Alice ha sido 1 [5].

[40]:
```
qc_ej3.barrier(label='5')
qc_ej3.x(bob_1).c_if(alice_2, 1)
qc_ej3.z(bob_1).c_if(alice_1, 1)
qc_ej3.measure(bob_1, bob_1);
```

El circuito resultante sería el siguiente. Las barreras indican las diferentes partes del algoritmo y se referencian con su número en los previos pasos.

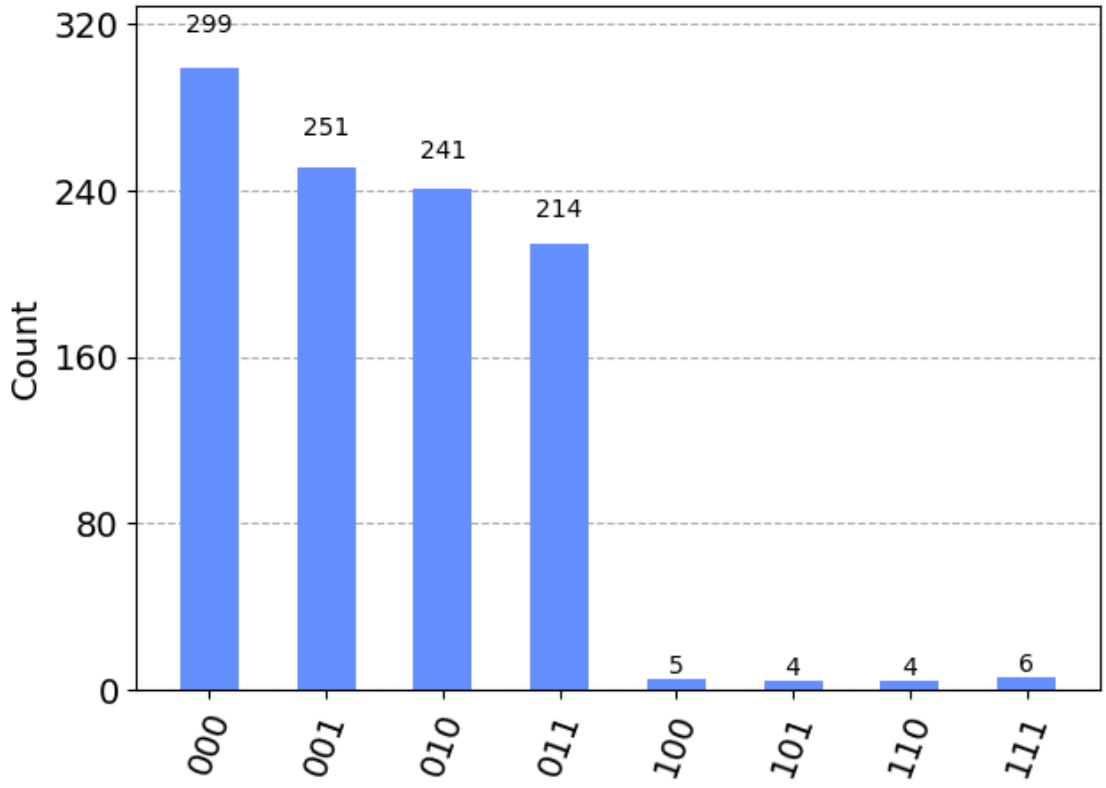
[41]:
```
qc_ej3.draw(output='mpl', style="iqp")
```

[41]:



Simulamos el circuito cuántico y obtenemos el histograma de las mediciones finales. De los 3 bits de medición mostrados, el primero bit mide el qubit de Bob, el segundo bit mide el segundo qubit de Alice y el tercer bit es el primer qubit de Alice.

[42]:
```
result = backend.run(qc_ej3, shots=1024).result()
counts = result.get_counts()
plot_histogram(counts)
```

[42]:


De esta manera verificamos que el estado $|\Psi\rangle$ de Alice se ha podido reproducir en el qubit de Bob.

6 Apéndice

6.1 Comprobación de los estados de Bell con arrays

Para realizar este ejercicio, definamos los vectores de estado |0> y |1> con su forma matricial en NumPy.

[43]:
```
sv_0 = np.array([1, 0])
sv_1 = np.array([0, 1])
```

6.1.1 Funciones auxiliares

Para simplificar los pasos, se ha creado una función principal llamada `array_to_dirac_and_matrix_latex`. Esta función toma una matriz de NumPy que representa el estado del sistema y genera código LaTeX para visualizar tanto la representación matricial como la notación de Dirac del estado cuántico. Además, utiliza tres funciones auxiliares para realizar estas tareas:

- `array_to_matrix_representation`: Convierte un array unidimensional en una representación matricial en columna.
- `array_to_dirac_notation`: Convierte un array complejo que representa un estado cuántico en superposición a la notación de Dirac.
- `find_symbolic_representation`: Verifica si un valor numérico corresponde a una constante simbólica dentro de una tolerancia especificada. Si se encuentra una correspondencia, devuelve la representación simbólica como cadena (con un signo '-' si el valor es negativo), de lo contrario, devuelve el valor original.

`is_symbolic_constant` **Function:** **Purpose:** The `is_symbolic_constant` function checks if an amplitude corresponds to a symbolic constant within a specified tolerance.

Attributes: - `amplitude` (float): The amplitude to check. - `symbolic_constants` (dict): A dictionary mapping numerical values to their symbolic representations. - `tolerance` (float): Tolerance for comparing amplitudes with symbolic constants.

Methods: None

Example Usage:

`symbol = is_symbolic_constant(my_amplitude)`

[44]:
```
def find_symbolic_representation(value, symbolic_constants={1/np.sqrt(2): '1/
↪2'}, tolerance=1e-10):
    """
    Check if the given numerical value corresponds to a symbolic constant within␣
↪a specified tolerance.

    Parameters:
    - value (float): The numerical value to check.
    - symbolic_constants (dict): A dictionary mapping numerical values to their␣
↪symbolic representations.
```

```
                                Defaults to {1/np.sqrt(2): '1/2'}.
    - tolerance (float): Tolerance for comparing values with symbolic constants.␣
↪Defaults to 1e-10.

    Returns:
    str or float: If a match is found, returns the symbolic representation as a␣
↪string
                  (prefixed with '-' if the value is negative); otherwise,␣
↪returns the original value.
    """
    for constant, symbol in symbolic_constants.items():
        if np.isclose(abs(value), constant, atol=tolerance):
            return symbol if value >= 0 else '-' + symbol
    return value
```

`array_to_dirac_notation` **Function:** **Purpose:** The `array_to_dirac_notation` function is designed to convert a complex-valued array representing a quantum state in superposition to Dirac notation.

Attributes: - `array` (numpy.ndarray): The complex-valued array representing the quantum state in superposition. - `tolerance` (float): Tolerance for considering amplitudes as negligible.

Methods: None

Example Usage: “‘python dirac_notation = array_to_dirac_notation(my_quantum_state)

[45]:
```
def array_to_dirac_notation(array, tolerance=1e-10):
    """
    Convert a complex-valued array representing a quantum state in superposition
    to Dirac notation.

    Parameters:
    - array (numpy.ndarray): The complex-valued array representing
      the quantum state in superposition.
    - tolerance (float): Tolerance for considering amplitudes as negligible.

    Returns:
    str: The Dirac notation representation of the quantum state.
    """
    # Ensure the statevector is normalized
    array = array / np.linalg.norm(array)

    # Get the number of qubits
    num_qubits = int(np.log2(len(array)))

    # Find indices where amplitude is not negligible
    non_zero_indices = np.where(np.abs(array) > tolerance)[0]
```

```
    # Generate Dirac notation terms
    terms = [
        (find_symbolic_representation(array[i]), format(i, f"0{num_qubits}b"))
        for i in non_zero_indices
    ]

    # Format Dirac notation
    dirac_notation = " + ".join([f"{amplitude}|{binary_rep}⟩" for amplitude,␣
↪binary_rep in terms])

    return dirac_notation
```

`array_to_matrix_representation` **Function: Purpose:** The `array_to_matrix_representation` function is designed to convert a one-dimensional array to a column matrix representation.

Attributes: - `array` (numpy.ndarray): The one-dimensional array to be converted.

Methods: None

Example Usage: “‘python matrix_rep = array_to_matrix_representation(my_array)

[46]:
```
def array_to_matrix_representation(array):
    """
    Convert a one-dimensional array to a column matrix representation.

    Parameters:
    - array (numpy.ndarray): The one-dimensional array to be converted.

    Returns:
    numpy.ndarray: The column matrix representation of the input array.
    """
    # Replace symbolic constants with their representations
    matrix_representation = np.array([find_symbolic_representation(value) or␣
↪value for value in array])

    # Return the column matrix representation
    return matrix_representation.reshape((len(matrix_representation), 1))
```

`array_to_dirac_and_matrix_latex` **Function: Purpose:** The `array_to_dirac_and_matrix_latex` function is designed to generate LaTeX code for displaying both the matrix representation and Dirac notation of a quantum state.

Attributes: - `array` (numpy.ndarray): The complex-valued array representing the quantum state.

Methods: - `array_to_matrix_representation(array)`: Converts a one-dimensional array to a column matrix representation. - `array_to_dirac_notation(array, tolerance=1e-10)`: Converts a complex-valued array representing a quantum state in superposition to Dirac notation. - `is_symbolic_constant(amplitude, symbolic_constants={1/np.sqrt(2): '1/2'},`

`tolerance=1e-10`): Checks if an amplitude corresponds to a symbolic constant within a specified tolerance.

Example Usage: “‘python result = array_to_dirac_and_matrix_latex(my_quantum_state)

[47]:
```
def array_to_dirac_and_matrix_latex(array):
    """
    Generate LaTeX code for displaying both the matrix representation and Dirac␣
↪notation
    of a quantum state.

    Parameters:
    - array (numpy.ndarray): The complex-valued array representing the quantum␣
↪state.

    Returns:

    Latex: A Latex object containing LaTeX code for displaying both␣
↪representations.
    """
    matrix_representation = array_to_matrix_representation(array)
    latex = "Matrix representation\n\\begin{bmatrix}\n" + \
            "\\\\\n".join(map(str, matrix_representation.flatten())) + \
            "\n\\end{bmatrix}\n"
    latex += f'Dirac Notation:\n{array_to_dirac_notation(array)}'
    return Latex(latex)
```

6.1.2 Evolución del estado cuántico del sistema de $|\Phi^+\rangle$

Si queremos ver cuál sería la evolució del estado cuántico del sistema, vamos a simularlo haciendo operaciones a los vectores en forma de Numpy Array.

[48]:
```
sv_b1 = np.kron(sv_0, sv_0)
array_to_dirac_and_matrix_latex(sv_b1)
```

[48]:
Matrix representation $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$ Dirac Notation: $1.0|00\rangle$

Añadimos una puerta Hadamard al primer qubit:

[49]:
```
sv_b1 = (np.kron(sv_0, sv_0) + np.kron(sv_0, sv_1)) / np.sqrt(2)
array_to_dirac_and_matrix_latex(sv_b1)
```

[49]:
Matrix representation $\begin{bmatrix}1/2\\1/2\\0.0\\0.0\end{bmatrix}$ Dirac Notation: $1/2|00\rangle + 1/2|01\rangle$

Finalmente añadimos una puerta CNOT, donde el qubit de control es el primer qubit, y el qubit objetivo es el segundo:

[50]:
```
sv_b1 = (np.kron(sv_0, sv_0) + np.kron(sv_1, sv_1)) / np.sqrt(2)
array_to_dirac_and_matrix_latex(sv_b1)
```

[50]:
Matrix representation $\begin{bmatrix} 1/2 \\ 0.0 \\ 0.0 \\ 1/2 \end{bmatrix}$ Dirac Notation: $1/2|00\rangle + 1/2|11\rangle$

6.1.3 Evolución del estado cuántico del sistema de $|\Phi^-\rangle$

Si queremos ver cuál sería la evolució del estado cuántico del sistema, vamos a simularlo haciendo operaciones a los vectores en forma de Numpy Array.

[51]:
```
sv_b2 = np.kron(sv_0, sv_0)
array_to_dirac_and_matrix_latex(sv_b2)
```

[51]:
Matrix representation $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ Dirac Notation: $1.0|00\rangle$

Añadimos una puerta Hadamard al primer qubit:

[52]:
```
sv_b2 = (np.kron(sv_0, sv_0) + np.kron(sv_0, sv_1)) / np.sqrt(2)
array_to_dirac_and_matrix_latex(sv_b2)
```

[52]:
Matrix representation $\begin{bmatrix} 1/2 \\ 1/2 \\ 0.0 \\ 0.0 \end{bmatrix}$ Dirac Notation: $1/2|00\rangle + 1/2|01\rangle$

Añadimos una puerta CNOT, donde el qubit de control es el primer qubit, y el qubit objetivo es el segundo:

[53]:
```
sv_b2 = (np.kron(sv_0, sv_0) + np.kron(sv_1, sv_1)) / np.sqrt(2)
array_to_dirac_and_matrix_latex(sv_b2)
```

[53]:
Matrix representation $\begin{bmatrix} 1/2 \\ 0.0 \\ 0.0 \\ 1/2 \end{bmatrix}$ Dirac Notation: $1/2|00\rangle + 1/2|11\rangle$

Finalmente, añadimos una puerta Pauli-Z en el qubit 0 para pasar de $|+\rangle$ a $|-\rangle$.

[54]:
```
sv_b2 = (np.kron(sv_0, sv_0) - np.kron(sv_1, sv_1)) / np.sqrt(2)
array_to_dirac_and_matrix_latex(sv_b2)
```

[54]:

Matrix representation $\begin{bmatrix} 1/2 \\ 0.0 \\ 0.0 \\ -1/2 \end{bmatrix}$ Dirac Notation: $1/2|00\rangle + -1/2|11\rangle$

6.1.4 Evolución del estado cuántico del sistema de $|\Psi^+\rangle$

Si queremos ver cuál sería la evolució del estado cuántico del sistema, vamos a simularlo haciendo operaciones a los vectores en forma de Numpy Array.

[55]:
```
sv_b3 = np.kron(sv_0, sv_0)
array_to_dirac_and_matrix_latex(sv_b3)
```

[55]:
Matrix representation $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ Dirac Notation: $1.0|00\rangle$

Añadimos una puerta Pauli-X al primer qubit:

[56]:
```
sv_b3 = np.kron(sv_0, sv_1)
array_to_dirac_and_matrix_latex(sv_b3)
```

[56]:
Matrix representation $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ Dirac Notation: $1.0|01\rangle$

Añadimos una puerta Hadamard al primer qubit:

[57]:
```
sv_b3 = (np.kron(sv_0, sv_0) - np.kron(sv_0, sv_1)) / np.sqrt(2)
array_to_dirac_and_matrix_latex(sv_b3)
```

[57]:
Matrix representation $\begin{bmatrix} 1/2 \\ -1/2 \\ 0.0 \\ 0.0 \end{bmatrix}$ Dirac Notation: $1/2|00\rangle + -1/2|01\rangle$

Añadimos una puerta CNOT, donde el qubit de control es el primer qubit, y el qubit objetivo es el segundo:

[58]:
```
sv_b3 = (np.kron(sv_0, sv_1) - np.kron(sv_1, sv_0)) / np.sqrt(2)
array_to_dirac_and_matrix_latex(sv_b3)
```

[58]:
Matrix representation $\begin{bmatrix} 0.0 \\ 1/2 \\ -1/2 \\ 0.0 \end{bmatrix}$ Dirac Notation: $1/2|01\rangle + -1/2|10\rangle$

Finalmente, añadimos una puerta Pauli-Z en el qubit 0 para pasar de $|+\rangle$ a $|-\rangle$.

[59]:
```
sv_b3 = (np.kron(sv_0, sv_1) + np.kron(sv_1, sv_0)) / np.sqrt(2)
array_to_dirac_and_matrix_latex(sv_b3)
```

[59]:

Matrix representation $\begin{bmatrix} 0.0 \\ 1/2 \\ 1/2 \\ 0.0 \end{bmatrix}$ Dirac Notation: $1/2|01\rangle + 1/2|10\rangle$

6.1.5 Evolución del estado cuántico del sistema de $|\Psi^-\rangle$

Si queremos ver cuál sería la evolució del estado cuántico del sistema, vamos a simularlo haciendo operaciones a los vectores en forma de Numpy Array.

[60]:
```
sv_b4 = np.kron(sv_0, sv_0)
array_to_dirac_and_matrix_latex(sv_b4)
```

[60]:

Matrix representation $\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ Dirac Notation: $1.0|00\rangle$

Añadimos una puerta Pauli-X al primer qubit:

[61]:
```
sv_b4 = np.kron(sv_0, sv_1)
array_to_dirac_and_matrix_latex(sv_b4)
```

[61]:

Matrix representation $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ Dirac Notation: $1.0|01\rangle$

Añadimos una puerta Hadamard al primer qubit:

[62]:
```
sv_b4 = (np.kron(sv_0, sv_0) - np.kron(sv_0, sv_1)) / np.sqrt(2)
array_to_dirac_and_matrix_latex(sv_b4)
```

[62]:

Matrix representation $\begin{bmatrix} 1/2 \\ -1/2 \\ 0.0 \\ 0.0 \end{bmatrix}$ Dirac Notation: $1/2|00\rangle$ + -1/2$|01\rangle$

Añadimos una puerta CNOT, donde el qubit de control es el primer qubit, y el qubit objetivo es el segundo:

[63]:
```
sv_b4 = (np.kron(sv_0, sv_1) - np.kron(sv_1, sv_0)) / np.sqrt(2)
array_to_dirac_and_matrix_latex(sv_b4)
```

[63]:

Matrix representation $\begin{bmatrix} 0.0 \\ 1/2 \\ -1/2 \\ 0.0 \end{bmatrix}$ Dirac Notation: $1/2|01\rangle$ + -1/2$|10\rangle$