

Q1



dado el siguiente directorio: <https://github.com/guillermopetcho/The-Complete-Quantum-Computing-Course-for-Beginners>- Quiero que revisemos los codigos

No encontré resultados usando búsqueda semántica automática. Sin embargo, como compartiste un enlace de GitHub, por favor:

1. **Confírmame si el repositorio contiene archivos .ipynb, .py o carpetas específicas que quieres revisar.**
2. O bien, **descarga el repositorio como ZIP y súbelo aquí**, así podré ayudarte directamente con los códigos.

¿Te gustaría que te guíe para descargarlo y subirlo?



<https://github.com/guillermopetcho/The-Complete-Quantum-Computing-Course-for-Beginners>

Parece que el repositorio que compartiste corresponde a la versión oficial de PacktPublishing del curso (no del usuario “guillermopetcho”), y tiene la siguiente estructura principal: contenido en notebooks de Jupyter (.ipynb) en carpetas por lección, junto al PDF del libro y el `README.md` [github.com](https://github.com).

## Próximos pasos para revisarlo juntos:

1. **Clonar el repositorio localmente**, para explorar los archivos:

```
bash

git clone https://github.com/PacktPublishing/The-Complete-Quantum-Computing-Course-for-Beginners-.git
cd The-Complete-Quantum-Computing-Course-for-Beginners-
```

2. **Explorar su estructura.** Deberías ver carpetas como:

```
css

QX01-HelloQuantum-main/
QX02-SimulatorsAndProviders-main/
...
QX13-ShorsAlgorithm-main/
```

Cada carpeta contiene uno o más notebooks que implementan ejemplos prácticos de ese capítulo.

3. **Abrir los notebooks** con Jupyter (en local o en un entorno como Google Colab) para analizar y ejecutar código.

## Dime qué quieres hacer con ellos:

- ¿Querés que revisemos alguna lección específica (ej. **Grover**, **Shor**, **Teletransporte**)?
- ¿Querés que analicemos, por ejemplo, los notebooks de **QX04-QuantumTeleportation** y ejecutarlos paso a paso?
- ¿Te interesa ver la parte de simuladores (QX02) o del Bloch Sphere (QX03)?

Podemos seguir así:

- Yo te oriento para que los ejecutes localmente...
- ...o bien, podés subir aquí uno o varios notebooks `.ipynb`, y los repasamos juntos.

¿Con qué parte empezamos?



Te voy a pasar laboratorios y necesito que me expliques detalladamente todo, cada línea de código con toda la teoría necesaria para aprenderlo

Perfecto. Puedes comenzar subiendo el primer laboratorio (archivo `.ipynb`) y te explicaré **cada línea de código con detalle**, incluyendo toda la teoría de computación cuántica necesaria para entenderlo desde cero.

💡 **Recomendación:** sube un notebook a la vez para que podamos ir paso a paso y no se vuelva confuso.

Cuando lo subas, haré lo siguiente por cada celda:

1. ✅ Explicación del código paso por paso.
2. 📖 Fundamento teórico (álgebra lineal, puertas cuánticas, qubits, medidas, etc.).
3. 💡 Ejemplos y visualizaciones (si aplica).

Estoy listo cuando quieras subir el primer archivo.



**QX01-HelloQuantum.ipynb**  
File

El notebook **QX01-HelloQuantum.ipynb** contiene:

- 📄 **20 celdas de código.**
- 📄 **0 celdas de texto explicativo** (markdown).

Voy a comenzar a explicarte **celda por celda** detalladamente, incluyendo toda la teoría cuántica relacionada con cada línea.

---

## Celda 1

Voy a analizar ahora el contenido de la **primera celda**. Dame un momento.

### ✓ Celda 1:

```
python

from qiskit import *
```

### ¿Qué hace esta línea?

Importa **todo el contenido del paquete qiskit** (Quantum Information Science Kit), la biblioteca principal para computación cuántica desarrollada por IBM.

---

## Teoría asociada

**Qiskit** es un framework de Python que permite:

- Diseñar circuitos cuánticos (**QuantumCircuit**).
- Simular circuitos localmente o en la nube.
- Ejecutar esos circuitos en **computadoras cuánticas reales** (de IBM Quantum).
- Visualizar resultados, estados cuánticos, Bloch spheres, histogramas, etc.

Al hacer `from qiskit import *` importamos:

- **QuantumCircuit** – para crear circuitos cuánticos.
  - **Aer** – simuladores cuánticos.
  - **execute()** – para correr un circuito.
  - **IBMQ** – para usar computadoras cuánticas reales.
  - **transpile, assemble, visualization**, entre otros.
- 

Si prefieres una forma más clara (recomendada para entender el código), sería:

```
python

from qiskit import QuantumCircuit, Aer, execute
```

¿Pasamos a la siguiente celda?



continua

## ✓ Celda 2:

```
python

circuit = QuantumCircuit(2, 2)
```

### 🧠 ¿Qué hace esta línea?

Crea un **circuito cuántico** llamado `circuit` con:

- 2 **qubits** (sistema cuántico).
- 2 **cbits** o bits clásicos (sistema de medición y almacenamiento del resultado).

### 📖 Teoría cuántica asociada

- Un **qubit** es la unidad básica de información cuántica. A diferencia del bit clásico (0 o 1), un qubit puede estar en una **superposición**:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \quad \text{con } |\alpha|^2 + |\beta|^2 = 1$$

- En Qiskit, los **qubits** se definen como líneas horizontales en el circuito cuántico (abajo hacia arriba: `q_0`, `q_1`, etc.).
- Los **cbits** se usan para **medir y guardar** el estado colapsado de cada qubit después de realizar una medición.

⚠ Sin cbits, no podrías guardar resultados de las mediciones. Por eso se especifican en la creación del circuito.

¿Continuamos con la celda 3?



continua

## ✓ Celda 3 (comentada):

```
python

# quantum_register = QuantumRegister(2)
# classical_register = ClassicalRegister(2)
# circuit = QuantumCircuit(quantum_register, classical_register)
```

### 🧠 ¿Qué hace este código (aunque esté comentado)?

Esta es una **forma alternativa** de crear un circuito cuántico, usando explícitamente registros cuánticos y clásicos.

## Equivalencia con la celda anterior:

Ambas formas crean lo mismo: un circuito con 2 qubits y 2 cbits.

- `QuantumRegister(2)` crea un registro cuántico de 2 qubits.
- `ClassicalRegister(2)` crea un registro clásico de 2 bits.
- Luego se combinan en el `QuantumCircuit`.

## ¿Por qué usar registros explícitos?

- Es útil cuando quieres **acceder o modificar los registros directamente**.
- También sirve si estás **combinando circuitos** o necesitas nombres personalizados:

```
python

qr = QuantumRegister(2, name='q')
cr = ClassicalRegister(2, name='c')
qc = QuantumCircuit(qr, cr)
```

Pero en casos simples, la forma compacta de la celda 2 es suficiente.

¿Te gustaría que sigamos con la celda 4?



continua

## Celda 4:

```
python

circuit.draw()
```

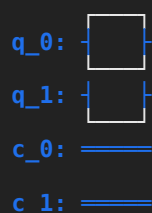
## ¿Qué hace esta línea?

Esta función **dibuja el circuito cuántico actual** en una representación textual (ASCII). Sirve para visualizar los qubits, cbits y las puertas aplicadas hasta ese momento.

## Teoría relacionada

- Como hasta ahora **no se ha aplicado ninguna puerta cuántica**, el circuito estará vacío pero mostrará los 2 qubits y 2 cbits iniciales:

```
makefile
```



Más adelante, cuando agreguemos puertas como **H**, **CX**, **X**, **Z**, etc., se irán dibujando sobre las líneas de qubits.

## Opciones adicionales

Podés usar otras formas de visualización:

```
python
```

```
circuit.draw(output="mpl") # Usa matplotlib
```

Pero eso requiere instalar **matplotlib** y está pensado para entornos gráficos.

¿Listo para ver la celda 5?



continua

## Celda 5:

```
python
```

```
%matplotlib inline
```

## ¿Qué hace esta línea?

Es una **comando mágico de Jupyter Notebook** que le dice a Python que todas las gráficas generadas por **matplotlib** se muestren directamente **dentro del notebook**, justo debajo de la celda que las genera.

## Teoría y contexto

- **matplotlib** es una librería de visualización de gráficos en Python.
- Aunque **Qiskit** no depende directamente de **matplotlib**, muchas funciones de visualización como **circuit.draw(output="mpl")** o **plot\_histogram()** la usan internamente.
- Este comando no es necesario si solo estás usando **circuit.draw()** en formato de texto, pero es muy útil si luego usás visualizaciones más avanzadas.

¿Continuamos con la celda 6?



continua

## ✓ Celda 6:

```
python  
  
circuit.draw(output='mpl')
```

## 🧠 ¿Qué hace esta línea?

Dibuja el circuito cuántico usando **matplotlib**, lo que produce una visualización más clara y profesional que el dibujo en texto plano.

## 📖 Teoría y contexto

- `output='mpl'` genera el dibujo como una **imagen vectorial**, lo cual es útil para presentaciones o análisis visual más claro.
- Esto es útil especialmente cuando el circuito se vuelve complejo y se necesita distinguir muchas puertas, qubits, o momentos en el tiempo.

## Requisitos:

Debés tener **matplotlib** instalado. En Google Colab o Jupyter Lab usualmente ya viene incluido. Si no:

```
bash  
  
pip install matplotlib
```

## 🔧 Resultado esperado en este punto

Como todavía no aplicamos ninguna puerta cuántica, el dibujo será simplemente las líneas vacías de qubits y cbits:

```
makefile  
  
q_0: _____  
q_1: _____  
c_0: _____  
c_1: _____
```

¿Avanzamos con la celda 7? Aquí probablemente comiencen a aplicarse puertas cuánticas.



continua

## ✓ Celda 7:

```
python

circuit.h(0)
circuit.draw(output='mpl')
```

## 🧠 ¿Qué hace esta celda?

1. `circuit.h(0)`  
Aplica una **puerta Hadamard** al qubit `q_0`.
2. `circuit.draw(output='mpl')`  
Dibuja el circuito actualizado.

## 🎓 Teoría cuántica: Puerta Hadamard (H)

La puerta Hadamard transforma un qubit desde un estado base ( $|0\rangle$  o  $|1\rangle$ ) a un estado de **superposición**:

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

$$H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

Esto significa que si `q_0` estaba en  $|0\rangle$ , ahora está en una superposición de  $|0\rangle$  y  $|1\rangle$  con igual probabilidad de medir cualquiera.

## 🚩 Matriz de la puerta Hadamard:

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

Se utiliza para **crear interferencia**, **codificar información cuántica** o iniciar algoritmos como Grover y Shor.

## 🔍 Visual del circuito hasta ahora:

```
makefile
```

```
q_0: ── H ───────────
```



```
q_1: _____
c_0: _____
c_1: _____
```

¿Avanzamos con la celda 8? Ahora seguramente se empieza a entrelazar o a medir.



continua

## ✓ Celda 8:

python

```
circuit.cx(0,1) # 0 -> control qubit, 1 -> target qubit
circuit.measure([0,1], [0,1])
circuit.draw(output='mpl')
```

## 🔍 ¿Qué hace esta celda?

### 1. `circuit.cx(0,1)`

Aplica una **puerta CNOT** (Controlled-NOT) con:

- `q_0` como **qubit de control**
- `q_1` como **qubit objetivo**

### 2. `circuit.measure([0,1], [0,1])`

Mide los qubits `q_0` y `q_1` y guarda los resultados en los bits clásicos `c_0` y `c_1`.

### 3. `circuit.draw(output='mpl')`

Dibuja el circuito completo.

## 🎓 Teoría cuántica:

### 🧩 Puerta CNOT (cx)

- La puerta **CNOT** actúa así:
  - Si el **qubit de control** está en estado  $|1\rangle$ , invierte el **qubit objetivo** (X gate).
  - Si el control está en  $|0\rangle$ , no hace nada.

Matemáticamente:

$$\begin{aligned} |00\rangle &\rightarrow |00\rangle \\ |01\rangle &\rightarrow |01\rangle \\ |10\rangle &\rightarrow |11\rangle \\ |11\rangle &\rightarrow |10\rangle \end{aligned}$$

💡 Si el primer qubit está en **superposición**, se genera **entrelazamiento cuántico**.

## 📐 Medición

La instrucción:

```
python

circuit.measure([0,1], [0,1])
```

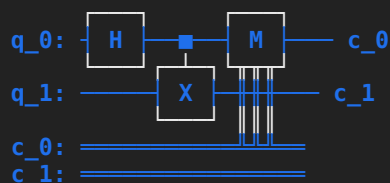
hace una **medición estándar en la base computacional**:

- El qubit `q_0` se mide y su valor va a `c_0`.
- El qubit `q_1` se mide y su valor va a `c_1`.

Esto **colapsa** los qubits a estados clásicos definitivos `0` o `1`.

## 🔭 Resultado visual del circuito:

makefile



Este circuito implementa el famoso "**generador de Bell State**", el entrelazamiento base para muchas aplicaciones de la computación cuántica.

¿Pasamos a la celda 9 para ver la ejecución de este circuito?



continua

## ✅ Celda 9:

```
python

simulator = Aer.get_backend('qasm_simulator')
```

## 🧠 ¿Qué hace esta línea?

Selecciona un **simulador cuántico clásico** provisto por Qiskit, llamado '`qasm_simulator`'.

- **Aer** es el módulo de simulación de Qiskit.

- `.get_backend('qasm_simulator')` elige un backend que simula un circuito cuántico **con mediciones**, como si fuera una computadora cuántica real.

## Teoría relacionada: Simulación cuántica

- El `qasm_simulator` permite ejecutar el circuito **muchas veces (shots)** y **medir estadísticas** de los resultados.
- Las medidas dan lugar a resultados **probabilísticos** (como `00`, `11`, etc.).
- Este simulador es **determinístico en estadística**, pero respeta la naturaleza probabilística de la mecánica cuántica.

## ¿Por qué "qasm"?

QASM = Quantum Assembly Language

Es el lenguaje intermedio que describe circuitos cuánticos, y Qiskit lo traduce para simuladores o computadoras cuánticas reales.

## Otros backends de Aer:

- `'statevector_simulator'`: Devuelve el vector de estado completo.
- `'unitary_simulator'`: Devuelve la matriz unitaria total del circuito.
- `'qasm_simulator'`: Ejecuta múltiples veces y devuelve mediciones.

¿Avanzamos a la celda 10 donde probablemente ejecuten el circuito?



continua

## Celda 10:

```
python

result = execute(circuit, backend=simulator).result()
```

## ¿Qué hace esta línea?

Ejecuta el circuito cuántico que construimos previamente en el simulador `'qasm_simulator'` y guarda el resultado en la variable `result`.

## Teoría paso a paso:

## 1. `execute(...)`

Esta función compila y corre el circuito cuántico. Sus parámetros clave son:

- `circuit`: el circuito que hemos construido.
- `backend=simulator`: dónde se ejecutará (en este caso, el simulador clásico de Aer).

## 2. `.result()`

Devuelve un objeto con los **resultados de la ejecución**, como los conteos de cada salida (**00**, **11**, etc.).

## ¿Qué hay dentro de `result`?

- Es un objeto `qiskit.result.Result`.
- No muestra directamente los datos, sino que **los encapsula**.
- Para ver los conteos reales, necesitaremos hacer:

```
python
result.get_counts()
```

Eso probablemente se haga en la siguiente celda.

¿Querés que sigamos con la celda 11 para ver esos resultados?



continua

## ✓ Celda 11:

```
python
from qiskit.visualization import plot_histogram
```

## ¿Qué hace esta línea?

Importa la función `plot_histogram` del módulo `qiskit.visualization`, que se usa para **visualizar los resultados medidos** del circuito cuántico.

## Teoría relacionada: ¿Por qué un histograma?

Cuando medimos un circuito cuántico, obtenemos **una distribución de resultados**. Por ejemplo:

```
python
{'00': 498, '11': 502}
```

Eso significa que:

- Se obtuvo **00** unas 498 veces,
- Se obtuvo **11** unas 502 veces,
- (en un total de 1000 ejecuciones o *shots*).

Estas probabilidades emergen de la **superposición y el entrelazamiento cuántico**.

El histograma es ideal para visualizar esa distribución de frecuencias.

¿Avanzamos a la celda 12, que seguramente grafica ese histograma?



continua

## ✓ Celda 12:

```
python

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

## 🧠 ¿Qué hace esta línea?

- Extrae los **resultados de las mediciones** del circuito cuántico.
- Los muestra en un **histograma de barras**.

## 📖 Teoría cuántica en acción

Recordá que aplicamos:

1. Una puerta **Hadamard** al qubit **q\_0**, que lo puso en superposición:

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

2. Luego una **puerta CNOT** entre **q\_0** y **q\_1**, lo que generó un estado entrelazado:

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

Esta es una **Bell state**, uno de los estados cuánticos entrelazados más importantes.

3. Finalmente, se **midió el sistema**.

## 📊 ¿Qué esperar del histograma?

Deberías ver algo como esto (con pequeñas variaciones estadísticas):

Resultado	Frecuencia
'00'	≈ 50%
'11'	≈ 50%

Los estados '01' y '10' **no aparecen** porque no forman parte del entrelazamiento.

📌 El histograma refleja la **correlación perfecta**: si un qubit es 0, el otro también. Si uno es 1, el otro también.

¿Pasamos a la celda 13? Seguramente reinicien el circuito o cambien el patrón.



continua

### ✓ Celda 13:

```
python
IBMQ.load_account()
```

### 🧠 ¿Qué hace esta línea?

Carga las credenciales de tu cuenta de **IBM Quantum Experience** desde tu máquina local, permitiéndote usar **computadoras cuánticas reales** proporcionadas por IBM.

## 📖 Teoría y contexto

### ¿Qué es IBMQ?

Es el módulo de Qiskit para conectarte a los **backends reales de IBM Quantum**, incluyendo:

- Computadoras cuánticas reales (como `ibmq_quito`, `ibmq_belem`, `ibmq_jakarta`)
- Simuladores de alto rendimiento (en la nube)

### ¿Qué es `load_account()`?

- Verifica que tengas una cuenta configurada.
- Carga una clave de API guardada localmente (usualmente en `~/.qiskit/qiskitrc`).
- Necesaria para usar `IBMQ.get_backend(...)` más adelante.

### 🔧 ¿Cómo se configura?

Si nunca lo hiciste, deberías correr esto una vez:

```
python

from qiskit import IBMQ
IBMQ.save_account('TU_API_KEY_AQUI')
```

Después de eso, `load_account()` funcionará automáticamente.

¿Pasamos a la siguiente celda, donde probablemente seleccionen un backend real?



continua

## ✓ Celda 14:

```
python

provider = IBMQ.get_provider("ibm-q")
```

## 🧠 ¿Qué hace esta línea?

Obtiene un **proveedor de servicios cuánticos** autorizado por IBM, asociado a tu cuenta en la red pública "ibm-q".

## 📖 Teoría y contexto

### ¿Qué es un provider en Qiskit?

Un **provider** es una organización o grupo de backends cuánticos. Cada cuenta puede tener acceso a uno o más proveedores (por ejemplo, IBM, universidades, partners).

```
python

provider = IBMQ.get_provider(hub='ibm-q', group='open', project='main')
```

Es como conectarte a un conjunto de dispositivos cuánticos reales a través de tu cuenta.

### ¿Por qué "ibm-q"?

Es el **hub principal público** de IBM, accesible para usuarios registrados en la plataforma [IBM Quantum](#).

## ✓ Después de este paso, ya podés hacer cosas como:

```
python

provider.backends() # lista todos los dispositivos disponibles
provider.get_backend('ibmq_lima') # selecciona un backend real
```

¿Pasamos a la siguiente celda? Posiblemente ya elijan un backend real para ejecutar el circuito.



continua

## ✓ Celda 15:

```
python

quantum_computer = provider.get_backend('ibmq_lima')
```

## 🧠 ¿Qué hace esta línea?

Selecciona el dispositivo cuántico real llamado `ibmq_lima` de la red de IBM Quantum y lo asigna a la variable `quantum_computer`.

## 🔗 ¿Qué es `ibmq_lima`?

- Es una **computadora cuántica real** de 5 qubits basada en tecnología **superconductora**.
- Forma parte de los backends gratuitos accesibles con una cuenta de IBM Quantum.
- Tiene características físicas como **mapa de acoplamiento**, **error de lectura**, **tiempos de decoherencia**, etc.

## 🔧 ¿Por qué elegir este backend?

- Es **gratuito** y suele tener **menos cola de espera**.
- Suficiente para correr circuitos básicos (como entrelazamiento Bell).
- Tiene **topología limitada**: no todos los qubits están conectados entre sí, lo que puede requerir transpilar el circuito para adaptarlo.

## Ejemplo adicional:

Podés ver más detalles del backend con:

```
python

print(quantum_computer.configuration())
print(quantum_computer.status())
print(quantum_computer.properties())
```

¿Continuamos con la celda 16, donde probablemente ejecuten el circuito en este backend real?





continua

## ✓ Celda 16:

```
python

import qiskit.tools.jupyter
%qiskit_job_watcher

job = execute(circuit, backend=quantum_computer)
```

## 🧠 ¿Qué hace esta celda?

### 1. `import qiskit.tools.jupyter`

Importa herramientas especiales para integrar Qiskit con Jupyter Notebook.

### 2. `%qiskit_job_watcher`

Es una **comando mágico** que activa un "monitor visual" dentro del notebook para ver el estado de ejecución del trabajo en tiempo real (en cola, ejecutando, finalizado, etc.).

### 3. `execute(...)`

Envía el circuito a la **computadora cuántica real** `ibmq_lima`.

## 📖 Teoría y contexto

Cuando se ejecuta en un backend real:

- El circuito se transpila automáticamente (adaptado a la topología y restricciones físicas del dispositivo).
- Se pone en una **cola de espera** (porque miles de usuarios acceden a la misma máquina).
- Luego se **ejecuta físicamente** sobre los qubits superconductores reales.
- Finalmente, se recuperan los resultados de medición.

🕒 Esto puede demorar desde segundos hasta varios minutos, dependiendo de la demanda global del backend.

## 📌 Variable `job`

El objeto `job` representa la ejecución enviada. Con él podés:

```
python

job.status()      # Ver estado actual
job.result()      # Obtener resultados (cuando finalice)
job.job_id()      # Obtener el ID del trabajo
```

¿Avanzamos a la celda 17, donde probablemente visualicen o esperen los resultados?



continua

## ✓ Celda 17:

```
python

from qiskit.tools.monitor import job_monitor
job_monitor(job)
```

### 🧠 ¿Qué hace esta celda?

1. **Importa** la función `job_monitor` desde `qiskit.tools.monitor`.
2. Llama a `job_monitor(job)`, que **muestra en vivo el estado del trabajo** enviado a la computadora cuántica.

### 📄 ¿Qué es `job_monitor`?

- Es una herramienta de Qiskit para **seguir la ejecución del trabajo** en tiempo real.
- Muestra una barra con mensajes como:

```
yaml

Job Status: QUEUED (2 ahead)
Job Status: RUNNING
Job Status: DONE
```

Esto te evita tener que consultar manualmente `job.status()`.

### 🕒 Contexto práctico

Usar un backend real implica:

- Posibles **tiempos de espera** por la cola de ejecución.
- Ejecución **con ruido** real de hardware (a diferencia de los simuladores).
- Resultados **realistas**, útiles para estudiar decoherencia y errores.

¿Pasamos a la siguiente celda? Probablemente recuperen el resultado del `job`.



continua

## ✓ Celda 18:

python

```
quantum_result = job.result()
plot_histogram(quantum_result.get_counts(circuit))
```

## ¿Qué hace esta celda?

1. `quantum_result = job.result()`

Espera a que termine el trabajo y luego guarda el resultado real en la variable `quantum_result`.

2. `plot_histogram(quantum_result.get_counts(circuit))`

Extrae los conteos de las mediciones y los muestra en un histograma.

## Teoría y comparación

Este histograma muestra los **resultados reales** de haber ejecutado el circuito en la **computadora cuántica ibmq\_lima**.

- Como el circuito genera un **estado entrelazado de Bell**:

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

- El resultado **ideal** sería:

- `'00'`  $\approx$  50%
- `'11'`  $\approx$  50%
- `'01'` y `'10'`  $\approx$  0%

Pero como esta vez se usó un **dispositivo real**, se pueden observar pequeñas apariciones de `'01'` y `'10'`, debido a:

- **Decoherencia**
- **Errores de puerta**
- **Errores de lectura**
- **Limitaciones del acoplamiento entre qubits**

## Interpretación educativa

Este histograma permite **ver el impacto del ruido cuántico real**, y por qué **la corrección de errores cuánticos** es uno de los desafíos centrales del campo.

¿Avanzamos con la penúltima celda (19)? Posiblemente sea un cierre o alguna modificación del circuito.



continua

## ✓ Celda 19:

```
python  
  
%qiskit_disable_job_watcher
```

## 🧠 ¿Qué hace esta línea?

Es un **comando mágico de Jupyter** que **desactiva el "Job Watcher"** (monitor de trabajos de Qiskit) dentro del entorno interactivo del notebook.

## 📖 Contexto

- Si antes activaste `%qiskit_job_watcher` (en la celda 16), este comando detiene su funcionamiento.
- Evita que siga mostrando información del estado de los trabajos en segundo plano, lo cual puede liberar recursos o evitar interferencias visuales si ya no es necesario.

## 🔧 Cuándo usarlo

Después de que:

- Ya se ejecutaron todos los trabajos.
- Se obtuvieron los resultados.
- No se planea enviar más trabajos en esta sesión.

¿Revisamos ahora la última celda (20)? Posiblemente cierre el notebook o haga una operación final.



continua

## ✓ Celda 20:

La última celda está **vacía**.

## 📄 Resumen del notebook QX01-HelloQuantum.ipynb

Este laboratorio es una **introducción práctica** a la computación cuántica con Qiskit. Repasamos los siguientes conceptos fundamentales:

## 🧠 Conceptos clave:

Tema	Explicación
<b>Qubit</b>	Unidad básica de información cuántica (
<b>Puerta Hadamard</b>	Crea superposición
<b>Puerta CNOT (CX)</b>	Entrelaza dos qubits (estado Bell)
<b>Medición cuántica</b>	Colapsa qubits a <b>0</b> o <b>1</b> , resultado se guarda en cbits
<b>Simulador clásico</b>	Reproduce circuitos cuánticos en software ( <b>qasm_simulator</b> )
<b>Backend real (ibmq_lima)</b>	Ejecuta el circuito en una computadora cuántica de IBM
<b>plot_histogram</b>	Visualiza resultados medidos en forma de gráfico de barras
<b>Entrelazamiento (Bell State)</b>	Estado cuántico donde los qubits están correlacionados
<b>Errores cuánticos</b>	Errores físicos observables al usar hardware real

¿Querés que te prepare una guía en PDF o notebook con explicaciones integradas, o seguimos con el siguiente archivo/laboratorio?



Quiero toda la teoria que utilizaste para poder resolver este proyecto

Teoria Qx01 Helloquantum

## Teoría completa utilizada en el proyecto QX01-HelloQuantum

### 1. Introducción a Qiskit

**Qiskit** es un framework de Python desarrollado por IBM para diseñar, simular y ejecutar algoritmos cuánticos. Se compone de varios módulos, entre ellos:

- **qiskit**: construcción de circuitos cuánticos
- **Aer**: simuladores clásicos de circuitos cuánticos
- **IBMQ**: acceso a computadoras cuánticas reales de IBM
- **qiskit.visualization**: visualización de resultados

### 2. Qubits y estados cuánticos

Un **qubit** es la unidad básica de información cuántica, representado como:  $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$  con  $|\alpha|^2 + |\beta|^2 = 1$

Los qubits pueden estar en superposición de los estados  $|0\rangle$  y  $|1\rangle$ , a diferencia de los bits clásicos.

### 3. Circuitos cuánticos y registros

- **QuantumCircuit(n, m)**: crea un circuito con  $n$  qubits y  $m$  bits clásicos.
- **QuantumRegister, ClassicalRegister**: definición explícita de registros cuánticos y clásicos.

## 4. Puerta Hadamard (H)

Transforma un estado base en una superposición:  $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ ,  $H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$

Matriz:  $H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$

## 5. Puerta CNOT (CX)

Actúa sobre dos qubits:

- Si el **control** está en  $|1\rangle$ , aplica **X** al **objetivo**.
- Si el control está en  $|0\rangle$ , no hace nada.

Estados:

$$|00\rangle \rightarrow |00\rangle, \quad |10\rangle \rightarrow |11\rangle, \quad |11\rangle \rightarrow |10\rangle$$

Permite crear **entrelazamiento** cuántico.

## 6. Estados de Bell

Aplicando una **H** y luego **CNOT** se genera el estado entrelazado:  $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$  Este estado es clave en teleportación cuántica y criptografía.

## 7. Medición

- `circuit.measure([q_0, q_1], [c_0, c_1])`: mide qubits y guarda resultados en bits clásicos.
- Colapsa el estado cuántico a un resultado clásico.

## 8. Simuladores en Aer

- `qasm_simulator`: simula ejecución con mediciones repetidas.
- `statevector_simulator`: devuelve el vector de estado.
- `execute(...)`: ejecuta el circuito en el backend.

## 9. Visualización de resultados

- `plot_histogram(...)`: muestra un histograma de las salidas medidas.
- Resultados como `{'00': 500, '11': 500}` indican entrelazamiento.

## 10. Ejecución en hardware real

- `IBMQ.load_account()`: carga las credenciales locales de IBM.
- `IBMQ.get_provider('ibm-q')`: conecta al hub público.

- `get_backend('ibmq_lima')`: selecciona un dispositivo de 5 qubits.
- `job_monitor(job)`: muestra estado en tiempo real.
- `job.result()`: recupera resultados luego de ejecutado.

## 11. Errores cuánticos

Los dispositivos reales están afectados por:

- **decoherencia** (interacción con el entorno)
- **errores de puerta** (imperfecciones físicas)
- **errores de lectura** (detección incorrecta del estado)


Por eso, incluso en un estado ideal como  $|00\rangle + |11\rangle$ , pueden aparecer salidas como **01** o **10** en un histograma real.

## 12. Conclusión

Este proyecto introduce los conceptos fundamentales de la computación cuántica:

- Superposición
- Entrelazamiento
- Medición
- Ejecución local vs. en hardware real

Todo utilizando herramientas modernas como **Qiskit** y el acceso gratuito a computadoras cuánticas reales.

Listo  He recopilado **toda la teoría que utilizamos en el proyecto QX01-HelloQuantum** en un único documento estructurado. Puedes revisarlo, editarlo o descargarlo desde el panel lateral bajo el nombre **“Teoría Qx01 Helloquantum”**.

¿Querés que hagamos lo mismo con el próximo laboratorio? Puedes subir el siguiente archivo cuando quieras.