Q12

QX12-ShorQuantumManual.ipynb
File

necesito que me expliques detalladamente cada celda del siguiente proyecto

A continuación encontrarás un desglose paso a paso de cada celda del notebook **QX12-ShorQuantumManual.ipynb**, con explicaciones detalladas de qué hace el código y la teoría subyacente.

Celda 1

```
python

from qiskit import *
import numpy as np
from qiskit.tools.visualization import plot_histogram
```

Explicación detallada:

1. **`from qiskit import *`**
   Importa todas las clases y funciones principales de Qiskit (QuantumCircuit, Aer, execute, etc.). Qiskit es el framework de IBM para programar y simular circuitos cuánticos.
2. **`import numpy as np`**
   Importa NumPy, la biblioteca estándar de Python para cálculo numérico. Aquí se usa, por ejemplo, para constantes como π y operaciones matriciales si hiciera falta.
3. **`from qiskit.tools.visualization import plot_histogram`**
   Trae la función **`plot_histogram`**, que permite visualizar los conteos de resultados de una simulación cuántica en forma de histograma de barras.

Celda 2

```
python

def c_amod15(a, power):
    """Controlled multiplication by a mod 15"""
    if a not in [2,7,8,11,13]:
        raise ValueError("'a' must be 2,7,8,11 or 13")
    U = QuantumCircuit(4)
    for iteration in range(power):
        if a in [2,13]:
            U.swap(0,1)
            U.swap(1,2)
            U.swap(2,3)
```

```
        if a in [7,8]:
            U.swap(2,3)
            U.swap(1,2)
            U.swap(0,1)
        if a == 11:
            U.swap(1,3)
            U.swap(0,2)
        if a in [7,11,13]:
            for q in range(4):
                U.x(q)
    U = U.to_gate()
    U.name = "%i^%i mod 15" % (a, power)
    c_U = U.control()
    return c_U
```

Explicación detallada:

- **Propósito**: construye el *gate* cuántico que realiza la operación

$$|x\rangle \mapsto |a \cdot x \bmod 15\rangle$$

  de forma **controlada**, listo para usarse sobre 4 qubits.
- **Validación de `a`**: solo son válidos los valores `a` que sean coprimos con 15 (2,7,8,11,13), requisito para que el algoritmo de Shor encuentre un periodo.
- **Construcción de `U`**:
  - Se crea un circuito de 4 qubits: en él se implementa la permutación de estados que corresponde a multiplicar por `a` módulo 15.
  - Según el valor de `a`, se usan secuencias de **SWAP** y **X** para permutar las cuatro bases $|0\rangle, |1\rangle, |2\rangle, \ldots, |14\rangle$ de manera equivalente a multiplicar por `a`.
  - El bucle `for iteration in range(power)` aplica la misma operación `power` veces, logrando así $a^{\text{power}} \bmod 15$.
- **Conversión a puerta y control**:
  - `U.to_gate()` transforma el circuito en un *gate* reutilizable.
  - `U.control()` crea la versión controlada, de modo que la operación solo se aplique cuando el qubit de control (un contador) esté en $|1\rangle$.
- **Salida**: devuelve `c_U`, el *gate* controlado $(\cdot) \bmod 15$.

Celda 3

```
python

n_count = 8
a = 7
```

Explicación detallada:

- `n_count = 8`: define el número de qubits del registro contador.
  En Shor, se usa un registro de fase de tamaño suficiente (aquí 8 qubits) para estimar el periodo r con la precisión necesaria.

- `a = 7`: el entero aleatorio elegido (2 ≤ a < N=15) y coprimo con 15.
  Shor escoge un `a` al azar; aquí se fija en 7, cuyo periodo módulo 15 es $r = 4$.

Celda 4

```
python

def qft_dagger(n):
    qc = QuantumCircuit(n)
    for qubit in range(n//2):
        qc.swap(qubit, n-qubit-1)
    for j in range(n):
        for m in range(j):
            qc.cu1(-np.pi/float(2**(j-m)), m, j)
        qc.h(j)
    qc.name = "QFT Dagger"
    return qc
```

Explicación detallada:

- Implementa la **Transformada de Fourier Cuántica inversa** (QFT†) sobre `n` qubits.
- **Swaps iniciales**: revierte el orden de los qubits (bit-reversal), requisito de la QFT.
- **Rondas de fase**:
  - Para cada par (m, j) con $m < j$, aplica una puerta controlada de fase `cu1` con ángulo $-\pi/2^{j-m}$.
- **Hadamard**: una capa de puertas H en cada qubit.
- Se nombra `"QFT Dagger"` para identificarse en los esquemas.
- **Uso**: después de la etapa de estimación de fase, aplicaremos QFT† antes de medir el registro contador.

Celda 5

```
python

# Crear QuantumCircuit con n_count qubits de conteo y 4 qubits auxiliares
qc = QuantumCircuit(n_count + 4, n_count)

# 1) Inicializar registro de conteo en |+>
for q in range(n_count):
    qc.h(q)

# 2) Inicializar registro auxiliar en |1> (valor semilla)
qc.x(3 + n_count)

# 3) Aplicar las operaciones controladas U^(2^q)
for q in range(n_count):
    qc.append(c_amod15(a, 2**q),
              [q] + [i + n_count for i in range(4)])

# 4) Inversa de QFT sobre registro de conteo
qc.append(qft_dagger(n_count), range(n_count))

# 5) Medir registro de conteo
qc.measure(range(n_count), range(n_count))
```

```
# Mostrar diagrama en matplotlib
qc.draw(output='mpl')
```

Explicación detallada:

1. **Circuito total**:
   - `n_count` **qubits** para el registro de fase (donde se pone en superposición de exponentes).
   - **4 qubits** para el registro de trabajo que almacena el valor de $a^x \mod 15$.
   - `n_count` **bits clásicos** donde se almacenarán las mediciones.
2. **(1) Preparación**:
   - Cada qubit de conteo recibe H $\rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, preparando superposición de todos los exponentes x.
3. **(2) Registro auxiliar**:
   - Se prepara en $|1\rangle$ con una X en el qubit más significativo del registro auxiliar.
4. **(3) ModMultiplicación controlada**:
   - Para cada qubit de conteo q, se aplica `c_amod15(a, 2**q)` controlado por ese qubit.
   - Logra la transformación $|x\rangle|1\rangle \mapsto |x\rangle|a^x \mod 15\rangle$.
5. **(4) QFT†**:
   - Tras la fase de kickback de fase cuántica, invertimos la QFT para decodificar la información de r en la base computacional.
6. **(5) Medición**:
   - Medimos los `n_count` qubits de fase; los patrones de salida nos darán un múltiplo cercano a $2^{n\_count} \cdot k/r$.
7. **Visualización**:
   - `.draw(output='mpl')` genera la imagen del circuito.

Celda 6

```
python


backend = Aer.get_backend('qasm_simulator')
results = execute(qc, backend, shots=1024).result()
counts = results.get_counts()
plot_histogram([counts])
```

Explicación detallada:

1. **Simulador**:
   - Usa el simulador clásico de Qiskit `qasm_simulator` para emular la ejecución de mediciones.
2. **Ejecución**:

- `execute(qc, backend, shots=1024)` corre el circuito 1 024 veces para obtener estadísticas de conteo.

3. **Conteos**:
   - `results.get_counts()` devuelve un diccionario `{estado: frecuencia}`.
4. **Histograma**:
   - `plot_histogram([counts])` dibuja las barras de frecuencia de cada resultado medido.

Celda 7

```
python

# r = 4 => a^r/2 -1 = 7^2 - 1 = 48 (gcd(15,48) = 3)
# r = 4 => a^r/2 + 1 = 7^2 + 1 = 50 (gcd (15,50) = 5)
```

Explicación detallada:

- **Lectura de `r`**: de los picos del histograma se infiere que el periodo de 7 mod 15 es $r = 4$.
- **Factores de 15**:
  1. Calcula $\gcd(a^{r/2} - 1,\ 15) = \gcd(7^2 - 1, 15) = \gcd(48, 15) = 3$.
  2. Calcula $\gcd(a^{r/2} + 1,\ 15) = \gcd(7^2 + 1, 15) = \gcd(50, 15) = 5$.
- Así obtenemos los factores no triviales 3 y 5 de 15.

Celdas 8, 9 y 10

Estas celdas aparecen vacías en el notebook (no contienen código ni comentarios). Probablemente están reservadas para:

- Notas adicionales
- Gráficos
- Extensiones futuras

Se pueden ignorar o usar como plantilla si se quiere añadir más análisis o pruebas.

Resumen conceptual

Este notebook implementa **Shor's Algorithm** para factorizar el entero $N = 15$:

1. **Preparación** de un registro de fase en superposición y un registro de trabajo en $|1\rangle$.
2. **Aplicación** de puertas controladas que implementan multiplicación modular, generando interferencia de fase dependiente del periodo r.
3. **Transformada de Fourier Cuántica inversa** para extraer información sobre r.
4. **Medición y post-procesamiento clásico** (gcd) para obtener los factores de N.

Cada celda corresponde a una fase de este flujo y combina tanto lógica cuántica (Qiskit) como el paso final clásico de factorización.

Si te surge alguna duda concreta sobre algún fragmento de código, teoría matemática o detalles de la implementación, ¡dímelo y profundizamos aún más!