응용물리연구실심화실습3 (PHY3075)

2024년 9월 4주차

응용물리학과 2022006971 이민성

번역

- Qiskit 예제

  이 섹션에서는 이번 강의에서 소개된 개념들의 Qiskit 구현 예제를 소개합니다.

  - 파이썬에서 벡터와 행렬

    Qiskit은 파이썬 프로그래밍 언어를 사용하므로, Qiskit을 본격적으로 논의하기 전에 파이썬에서의 행렬 및 벡터 계산에 대해 간단히 다루는 것이 유용할 수 있습니다. 파이썬에서는 `NumPy` 라이브러리의 `array` 클래스를 사용하여 행렬 및 벡터 계산을 수행할 수 있습니다(이 라이브러리에는 수치 계산을 위한 다양한 추가 컴포넌트가 포함되어 있습니다).

    다음은 두 벡터 `ket0`와 `ket1`을 정의하고, 그들의 평균을 출력하는 코드 셀의 예시입니다. 이 벡터들은 각각 큐비트 상태 벡터 $|0\rangle$와 $|1\rangle$에 해당합니다.

```
from numpy import array

ket0 = array([1, 0])
ket1 = array([0, 1])

display(ket0 / 2 + ket1 / 2)
```

Run

Output:

```
array([0.5, 0.5])
```

이 연산의 결과를 보기 위해 `display` 명령어를 명시적으로 사용할 필요는 없습니다. 코드 셀의 마지막 줄에 관심 있는 표현식을 단순히 쓰면, 그것이 출력으로 반환됩니다.

```
ket0 / 2 + ket1 / 2
```

Run

Output:

```
array([0.5, 0.5])
```

이 코드 셀은 또한 이 교재의 특정 페이지에서 코드 셀을 순차적으로 실행할 때 누적 효과가 있음을 보여줍니다. 따라서 `array` 클래스를 다시 로드하거나 `ket0`과 `ket1`을 다시 정의할 필요는 없습니다. 그러나 페이지를 새로 고치거나 다른 페이지로 전환하면 모든 것이 초기 상태로 재설정됩니다.

일반적으로, 이 강좌의 각 하위 섹션 내의 코드 셀은 순차적으로 실행되도록 설계되었습니다. 따라서 코드 셀 실행 중 오류가 발생하면 해당 코드 셀이 나타나는 하위 섹션 내의 모든 이전 코드 셀을 먼저 실행해야 합니다.

우리는 또한 `array`를 사용하여 연산을 나타내는 행렬을 만들 수 있습니다.

```
M1 = array([[1, 1], [0, 0]])
M2 = array([[1, 1], [1, 0]])

M1 / 2 + M2 / 2
```

Run

Output:

```
array([[1. , 1. ],
       [0.5, 0. ]])
```

행렬 곱셈(특별한 경우로서 행렬-벡터 곱셈을 포함)은 **NumPy**의 `matmul` 함수를 사용하여 수행할 수 있습니다:

```
from numpy import matmul

display(matmul(M1, ket1))
display(matmul(M1, M2))
display(matmul(M2, M1))
```

Run

Output:

```
array([1, 0])


array([[2, 1],
       [0, 0]])


array([[1, 1],
       [1, 1]])
```

- 상태, 측정 및 연산

  Qiskit에는 상태, 측정 및 연산을 쉽게 생성하고 조작할 수 있는 여러 클래스를 포함하고 있습니다. 따라서 처음부터 시작해서 양자 상태, 측정 및 연산을 시뮬레이션하는 데 필요한 모든 것을 Python에서 프로그래밍할 필요는 없습니다. 시작하는 데 필요한 몇 가지 예가 아래에 포함되어 있습니다.

  - 상태 벡터 정의 및 표시

    Qiskit의 **Statevector** 클래스는 양자 상태 벡터를 정의하고 조작하는 기능을 제공합니다. 아래의 코드 셀은 **Statevector** 클래스를 가져오고 몇 가지 벡터를 정의합니다. (참고로 벡터 `u`의 제곱근을 계산하기 위해 **NumPy** 라이브러리의 `sqrt` 함수를 사용합니다.)

```python
from qiskit.quantum_info import Statevector
from numpy import sqrt

u = Statevector([1 / sqrt(2), 1 / sqrt(2)])
v = Statevector([(1 + 2.0j) / 3, -2 / 3])
w = Statevector([1 / 3, 2 / 3])

print("State vectors u, v, and w have been defined.")
```

Run

Output:

```
State vectors u, v, and w have been defined.
```

Statevector 클래스는 상태 벡터를 시각적으로 표시하는 `draw` 메서드를 제공합니다. 이 메서드는 다양한 시각화를 위해 `latex` 및 `text` 옵션을 포함합니다. 아래 코드 셀이 이를 보여줍니다:

```python
display(u.draw("latex"))
display(v.draw("text"))
```

Run

Output:

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

```
[ 0.33333333+0.66666667j,-0.66666667+0.j        ]
```

Statevector 클래스에는 **is_valid** 메서드도 포함되어 있으며, 이는 주어진 벡터가 유효한 양자 상태 벡터인지(즉, 유클리드 노름이 1인지)를 확인합니다.

```python
display(u.is_valid())
display(w.is_valid())
```

Run

Output:

```
True

False
```

- Statevector를 사용한 측정 시뮬레이션

  다음으로 우리는 **Statevector** 클래스의 **measure** 메서드를 사용하여 Qiskit에서 양자 상태의 측정을 시뮬레이션하는 방법을 살펴보겠습니다.

  먼저, 큐비트 상태 벡터 **v**를 생성한 후 표시합니다.

```
v = Statevector([(1 + 2.0j) / 3, -2 / 3])
v.draw("latex")
```

Run

Output:

$(\frac{1}{3} + \frac{2i}{3})|0\rangle - \frac{2}{3}|1\rangle$

코드 셀은 수정할 수 있으므로, 벡터의 사양을 변경하고 싶다면 변경하십시오.

다음으로 **measure** 메서드를 실행하면 표준 기저 측정을 시뮬레이션합니다. 이는 해당 측정의 결과와 그 측정 후 시스템의 새로운 양자 상태를 반환합니다.

```
v.measure()
```

Run

Output:

```
('1',
 Statevector([ 0.+0.j, -1.+0.j],
             dims=(2,)))
```

측정 결과는 확률적이므로 같은 메서드가 다른 결과를 반환할 수 있습니다. 몇 번 실행해 보면서 이를 확인해 보세요.

위에서 정의한 벡터 **v**의 경우, **measure** 메서드는 측정이 수행된 후 양자 상태 벡터를 다음 중 하나로 정의합니다:

$$\frac{1+2i}{\sqrt{5}}|0\rangle$$

(기본적으로 $|0\rangle$) 혹은

$-|1\rangle$

(기본적으로 $|1\rangle$), 측정 결과에 따라 다릅니다. 두 경우 모두, 이러한 대안들은 사실 **동등**하며, 단지 전역 위상(global phase)에 의해 다르다고 말합니다. 하나는 단위 원에서 다른 복소수로 곱해진 값과 동일하기 때문입니다. 이 문제는 3강에서 더 자세히 설명되며, 지금은 무시해도 좋습니다.

덧붙여서, **Statevector**는 유효하지 않은 양자 상태 벡터에 **measure** 메서드를 적용하면 오류를 발생시킵니다. 오류가 어떻게 보이는지 궁금하면 시도해 보세요.

Statevector에는 시스템에서 여러 측정을 시뮬레이션할 수 있는 **sample_counts** 메서드도 있습니다. 예를 들어, 다음 셀은 벡터 **v**를 1000번 측정한 결과를 보여줍니다. 이는 높은 확률로 결과 **0**가 약 9번 중 5번(또는 1000번 시도 중 약 556번) 나오고, 결과 **1**이 약 9번 중 4번(또는 1000번 시도 중 약

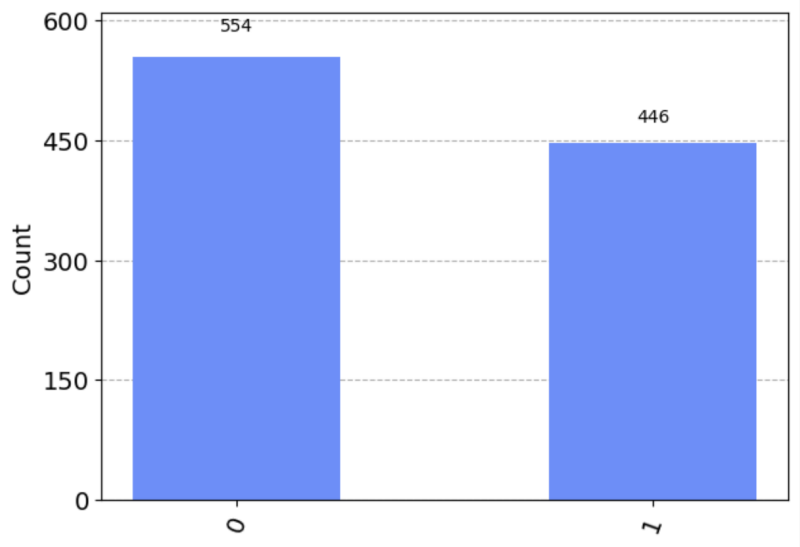
444번) 나옵니다. 셀에서는 결과를 시각화하는 **plot_histogram** 함수를 사용한 것도 볼 수 있습니다.

```
from qiskit.visualization import plot_histogram

statistics = v.sample_counts(1000)
display(statistics)
plot_histogram(statistics)
```

Run

Output:

```
{'0': 554, '1': 446}
```



셀을 여러 번 실행하고 1000개 대신 다른 수의 샘플을 시도해 보면 시도 횟수가 예상 확률에 어떤 영향을 미치는지 직관적으로 파악하는 데 도움이 될 수 있습니다.

- Qiskit의 `Operator`와 `Statevector` 사용하여 연산 수행

  유니터리 연산은 Qiskit에서 `Operator` 클래스를 사용하여 정의되고 상태 벡터에 대해 수행될 수 있습니다. 아래 예시에서처럼 말이죠.

```python
from qiskit.quantum_info import Operator

X = Operator([[0, 1], [1, 0]])
Y = Operator([[0, -1.0j], [1.0j, 0]])
Z = Operator([[1, 0], [0, -1]])
H = Operator([[1 / sqrt(2), 1 / sqrt(2)], [1 / sqrt(2), -1 / sqrt(
S = Operator([[1, 0], [0, 1.0j]])
T = Operator([[1, 0], [0, (1 + 1.0j) / sqrt(2)]])

v = Statevector([1, 0])

v = v.evolve(H)
v = v.evolve(T)
v = v.evolve(H)
v = v.evolve(T)
v = v.evolve(Z)

v.draw("text")
```

Run

Output:

```
[ 0.85355339+0.35355339j,-0.35355339+0.14644661j]
```

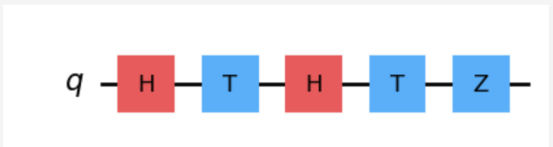
- 양자 회로로 나아가기

  양자 회로는 3강에서 정식으로 소개되지만, Qiskit의 `QuantumCircuit` 클래스를 사용하여 큐비트 유니터리 연산을 구성하는 실험을 미리 해볼 수 있습니다. 특히, 우리는 단일 큐비트에 수행되는 유니터리 연산의 순서로 구성된 양자 회로를 정의할 수 있습니다. 다음과 같이 양자 회로를 정의합니다:

```python
from qiskit import QuantumCircuit

circuit = QuantumCircuit(1)

circuit.h(0)
circuit.t(0)
circuit.h(0)
circuit.t(0)
circuit.z(0)

circuit.draw()
```

Run

Output:



연산은 그림의 왼쪽에서 시작하여 오른쪽으로 순차적으로 적용됩니다. 먼저 시작 양자 상태 벡터를 초기화한 후, 연산의 순서에 따라 그 상태를 진화시킵니다.

```
ket0 = Statevector([1, 0])
v = ket0.evolve(circuit)
v.draw("text")
```

Run

Output:

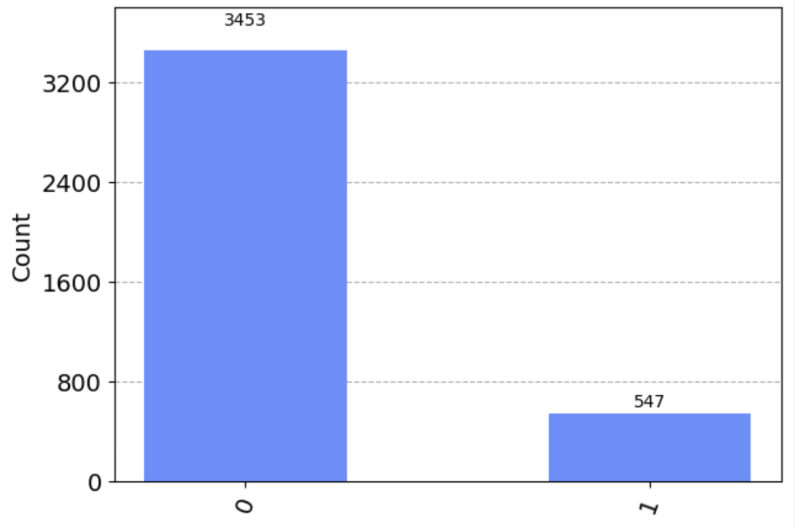
```
[ 0.85355339+0.35355339j,-0.35355339+0.14644661j]
```

마지막으로 이 실험을 실행한 결과(즉, $|0\rangle$ 상태를 준비하고 $|0\rangle$, 회로로 표현되는 연산 순서를 적용하고 측정)을 4000회 시뮬레이션해 보겠습니다.

```
statistics = v.sample_counts(4000)
plot_histogram(statistics)
```

Run

Output:



정리

1. **벡터 및 행렬 계산 (Python에서의 벡터 및 행렬)**:
   - Qiskit은 Python의 `NumPy` 라이브러리를 사용하여 벡터 및 행렬 연산을 수행합니다. 예제로 `array` 클래스를 사용하여 큐비트 상태 벡터인 `ket0`와 `ket1`를 정의하고 평균 값을 계산하는 코드를 소개했습니다.
2. **행렬 곱셈**:
   - 행렬과 벡터 곱셈은 `numpy`의 `matmul` 함수를 사용하여 처리할 수 있으며, 이를 통해 행렬 연산 및 벡터 변환을 구현합니다.
3. **Statevector 클래스**:

- `Statevector` 클래스는 양자 상태 벡터를 정의하고, 이를 조작할 수 있는 기능을 제공합니다. 벡터를 정의하고 `draw` 메서드를 사용하여 이를 시각화하는 방법이 소개되었습니다.
- `is_valid` 메서드를 통해 벡터가 유효한 양자 상태인지 (즉, 유클리드 노름이 1인지) 확인할 수 있습니다.

4. **측정 시뮬레이션**:
   - `Statevector` 클래스의 `measure` 메서드를 사용하여 양자 상태 측정을 시뮬레이션할 수 있습니다. 측정 결과는 확률적이며, 반복 측정 시 서로 다른 결과가 반환될 수 있습니다.
   - `sample_counts` 메서드는 특정 벡터의 측정 결과를 여러 번 반복해서 시뮬레이션하여 결과 분포를 볼 수 있습니다.
5. **연산 수행 (Operator 및 Statevector 사용)**:
   - `Operator` 클래스를 사용하여 상태 벡터에 유니터리 연산을 적용할 수 있습니다. 예제에서는 Hadamard 연산, 위상 연산(T 연산), Pauli 연산 등을 큐비트 상태 벡터에 적용하여 변화를 관찰하는 방법을 다루었습니다.
6. **양자 회로 (Quantum circuits)**:
   - `QuantumCircuit` 클래스를 사용하여 양자 회로를 구성할 수 있습니다. 큐비트에 대한 일련의 유니터리 연산을 정의하고, 회로의 시각화 기능을 사용하여 연산의 흐름을 확인할 수 있습니다.

Python에서의 벡터 및 행렬 연산부터 양자 상태의 측정과 연산에 이르기까지의 Qiskit 예제를 공부하였습니다.