

HHL 및 Qiskit 구현을 사용하여 선형 연립 방정식 풀기 {v*}

이 튜토리얼에서는 HHL 알고리즘, d를 소개합니다. 회로를 도출하고 Qiskit을 사용하여 구현합니다. 시뮬레이터와 {v*}에서 HHL을 실행하는 방법을 보여줍니다. 5-큐비트 장치.

목차

- 1. 소개 2. HHL 알고리즘 i. 몇 가지 수학적 배경 ii. HHL 설명 iii. HHL 내의 양자 위상 추정 (QPE) iv. 비정확한 QPE 3. 예제 1: 4-큐비트 HHL 4. Qiskit 구현 i. 시뮬레이터에서 HHL 실행: 일반적인 방법 ii. 실제 양자 장치에서 HHL 실행: 최적화된 예제 5. 문제

6. 참고 문헌

1. 소개

우리는 다양한 분야에서 많은 실제 응용 분야에서 선형 방정식 시스템을 봅니다. 그 예로는 편미분 방정식의 해, 금융 모델의 보정, 유체 시뮬레이션 또는 수치 필드 계산이 있습니다. 문제는 행렬 A 와 벡터 \vec{b} 가 주어졌을 때, 다음을 만족하는 \vec{x} 를 찾는 것으로 정의할 수 있습니다.

$$A \in \mathbb{C}^{N \times N} \quad \vec{b} \in \mathbb{C}^N \quad \vec{x} \in \mathbb{C}^N \quad A\vec{x} = \vec{b}$$

예를 들어, {v*}를 찾아봅시다.

$$A = \begin{pmatrix} 1 & -1/3 \\ -1/3 & 1 \end{pmatrix}, \quad \vec{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \quad \text{and} \quad \vec{b} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

그러면 문제는 다음과 같이 쓸 수도 있습니다. $\{x_1, x_2\} \in \mathbb{C}$ 을 만족하는 것을 찾으시오.

$$\begin{cases} x_1 - \frac{x_2}{3} = 1 \\ -\frac{x_1}{3} + x_2 = 0 \end{cases}$$

선형 방정식 시스템은 행 또는 열당 최대 $s\{v\}$ 개의 0이 아닌 항목을 갖는 경우 {v*}-희소라고 합니다. 고전적인 컴퓨터로 $s\{v\}$ 의 {v*}-희소 시스템을 푸는 데는 켈레 기울기 방법 1을 $O(N^2 \log(1/\epsilon))$ 시간이 필요합니다. 여기서 {v*}는 시스템의 조건수를 나타내며, {v*}는 근사치의 정확도를 나타냅니다.

HHL 알고리즘은 해의 함수를 추정하며 실행 시간 복잡도는 2입니다. 행렬은 에르미트 행렬 $O(\log(N)/\epsilon)$ 로딩, 해밀토니안 시뮬레이션, 해의 함수 계산을 위한 효율적인 오라클이 있다고 가정합니다. 이는 시스템 크기에 대한 지수적인 속도 향상이지만, HHL은 해 벡터의 함수만 근사할 수 있는 반면, 고전적인 알고리즘은 전체 해를 반환한다는 단점이 있습니다.

2. HHL 알고리즘

A. 몇 가지 수학적 배경

양자 컴퓨터로 선형 방정식 시스템을 풀기 위한 첫 번째 단계는 문제를 양자 언어로 인코딩하는 것입니다. 시스템을 재조정하여 A 와 b 가 정규화되었다고 가정하고 각각 양자 상태 $|x\rangle$ 와 $|b\rangle$ 로 매핑할 수 있습니다. 일반적으로 사용되는 매핑은 (resp.)의 성분이 양자 상태 $|u_j\rangle$ 와 기저 상태의 진폭에 해당하도록 하는 것입니다. 이제부터는 재조정된 문제에 집중하겠습니다.

$A|x\rangle = |b\rangle$

에르미트 행렬이므로 스펙트럼 분해를 가집니다.

$A = \sum_{j=0}^{N-1} \lambda_j |u_j\rangle \langle u_j|, \quad \lambda_j \in \mathbb{R}$

여기서 $|u_j\rangle$ 는 고유값 λ_j 에 해당하는 j 의 고유벡터입니다. 그러면,

$A^{-1} = \sum_{j=0}^{N-1} \lambda_j^{-1} |u_j\rangle \langle u_j|$

그리고 시스템의 우변은 $|b\rangle$ 의 고유 기저로 쓸 수 있습니다.

$|b\rangle = \sum_{j=0}^{N-1} b_j |u_j\rangle, \quad b_j \in \mathbb{C}$

HHL의 목표는 판독 레지스터를 다음 상태로 알고리즘에서 종료하는 것임을 명시하는 것이 유용합니다. $|v^*\rangle$

$|x\rangle = A^{-1}|b\rangle = \sum_{j=0}^{N-1} \lambda_j^{-1} b_j |u_j\rangle$

여기서 양자 상태에 대해 이야기하고 있으므로 이미 암묵적인 정규화 상수가 있음을 유의하십시오. $|v^*\rangle$

B. HHL 알고리즘 설명

알고리즘 U_{HHL} 세 개의 양자 레지스터는 알고리즘 시퀀스 U_{HHL} 로 설정됩니다. 아래 첨자 $0, 1, 2$ 로 표시할 레지스터는 의 n_0, n_1, n_2 비트 표현을 저장하는 데 사용됩니다. 두 번째 레지스터로 표시되며, 벡터 해를 포함하고, 지금부터는 $|b\rangle$ 입니다. 계산의 중간 단계를 위해 사용되는 보조 큐비트용 레지스터가 추가로 있습니다. 보조 i 는 무시할 수 있습니다. 설명에서 각 연산 시작 시점에 보관되고, 각 개별 연산 종료 시점에 복원되는 $|v^*\rangle$ 에 대한 설명입니다.

다음은 ou입니다. 해당 회로의 개략적인 그림과 함께 HHL 알고리즘의 timeline을 나타냅니다. 설명을 간소화하기 위해 이어지는 설명에서는 모든 계산이 정확하다고 가정하며, 더 자세한 설명은 다음과 같습니다. 비정확한 경우의 $|v^*\rangle$ 섹션 2.D.에 제공됩니다. $|v^*\rangle$



1. 데이터 로드, 즉, 변환을 수행합니다.

$$|0\rangle_{n_b} \mapsto |b\rangle_{n_b}$$

2. 양자 위상 추정(QPE) 적용: $\{v^*\}$

$$U = e^{iAt} := \sum_{j=0}^{N-1} e^{i\lambda_j t} |u_j\rangle \langle u_j|$$

레지스터의 양자 상태는 의 고유 기저로 표현되며 다음과 같습니다. $\{v^*\}$

$$\sum_{j=0}^{N-1} b_j |\lambda_j\rangle_{n_l} |u_j\rangle_{n_b}$$

의 $|\lambda_j\rangle_{n_l}$ -비트 이진 표현은 n_l 에 있습니까?

$$\lambda$$

3. 보조 큐비트를 추가하고 $\{v^*\}$ 에 따라 회전을 적용합니다.

$$j$$

$$\sum_{j=0}^{N-1} b_j |\lambda_j\rangle_{n_l} |u_j\rangle_{n_b} \left(\sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle + \frac{C}{\lambda_j} |1\rangle \right)$$

여기서 C 는 정규화 상수이고, 현재 형태로 표현된 바와 같이, 크기가 가장 작은 고유값 보다 작아야 합니다. 즉, $\min_j |\lambda_j| > C$ 입니다. 4. QPE를 적용합니다. QPE로 인한 가능한 오류를 무시하면 다음과 같은 결과가 나타납니다.

$$\sum_{j=0}^{N-1} b_j |0\rangle_{n_l} |u_j\rangle_{n_b} \left(\sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle + \frac{C}{\lambda_j} |1\rangle \right)$$

5. 보조 큐비트를 계산 기저에서 측정합니다. 결과가 이면 레지스터는 측정 후 상태에 있습니다.

$$\left(\sqrt{\frac{1}{\sum_{j=0}^{N-1} |b_j|^2 / |\lambda_j|^2}} \right) \sum_{j=0}^{N-1} \frac{b_j}{\lambda_j} |0\rangle_{n_l} |u_j\rangle_{n_b}$$

정규화 인자를 고려하면 $\{v^*\}$ 에 해당하는 해입니다.

6. 관찰 가능한 $\{v^*\}$ 를 적용하여 $\{v^*\}$ 를 계산합니다. $x|M|x$

C. HHL 내 양자 위상 추정 (QPE) $\{v^*\}$

양자 위상 추정은 3장에서 더 자세히 설명되어 있습니다. 그러나 이 양자 절차가 HHL 알고리즘의 핵심이므로 여기에서 정의를 상기합니다. 대략적으로 말하면, 고유 벡터 $\{v^*\}$ 와 고유값 λ 를 갖는 유니타리 U 가 주어졌을 때 λ 를 찾는 양자 알고리즘입니다. 이를 다음과 같이 공식적으로 정의할 수 있습니다.

$$e^{2\pi i \theta^m}$$

$$\theta$$

정의: 가 유니타리 행렬이고 각각의 고유값에 대한 근사값을 나타내며, 아래 첨자는 자릿수로 잘못음을 나타낸다.

$$\text{QPE}(U,|0\rangle_n|\psi\rangle_m)=|\tilde{\theta}\rangle_n|\psi\rangle_m$$

HHL의 경우 QPE를 사용하며, 여기서 U 는 역행렬이 풀고자 하는 시스템과 관련된 행렬입니다. 이 경우,

$$e^{iAt}=\sum_{j=0}^{N-1}e^{i\lambda_jt}|u_j\rangle\langle u_j|$$

그러면, e^{iAt} 에 대해 $|\tilde{u}_j\rangle_{n_b}$ 를 곱해준 후 레지스터의 양자 상태를 다음과 같습니다. 이제, $|\tilde{\lambda}_j\rangle_{n_l}$ 는 n_l 비트를 나타냅니다. 이진 근사 $\{v^*\}$ 이 선택된 후, $\{v^*\}$ 를 $\{v^*\}$ 비트로 정확하게 표현할 수 있다면,

$$\text{QPE}(e^{iAt},\sum_{j=0}^{N-1}b_j|0\rangle_{n_l}|u_j\rangle_{n_b})=\sum_{j=0}^{N-1}b_j|\lambda_j\rangle_{n_l}|u_j\rangle_{n_b}$$

D. 비정확한 QPE $\{v^*\}$

실제로, 초기 상태에 QPE를 적용한 후 레지스터의 양자 상태는 다음과 같습니다.

$$\sum_{j=0}^{N-1}b_j\left(\sum_{l=0}^{2^{n_l}-1}\alpha_{l|j}|l\rangle_{n_l}\right)|u_j\rangle_{n_b}$$

어디에

$$\alpha_{l|j}=\frac{1}{2^{n_l}}\sum_{k=0}^{2^{n_l}-1}\left(e^{2\pi i\left(\frac{\lambda_jt}{2\pi}-\frac{l}{2^{n_l}}\right)}\right)^k$$

로 나타낼 수 있다. 최적의 n_l 비트 근사값은 $\lfloor \log_2 \frac{1}{\epsilon} \rfloor$ 입니다. 그런 다음 레지스터의 레이블을 다시 지정하여 λ_j 의 진폭을 나타내도록 할 수 있습니다. 이제, $|\tilde{\lambda}_j\rangle_{n_l}$ 는 n_l 비트를 나타냅니다.

$$\alpha_{l|j}:=\frac{1}{2^{n_l}}\sum_{k=0}^{2^{n_l}-1}\left(e^{2\pi i\left(\frac{\lambda_jt}{2\pi}-\frac{l+\tilde{\lambda}_j}{2^{n_l}}\right)}\right)^k$$

각각이 n_l 비트로 정확하게 표현될 수 있다면, $\frac{\lambda_jt}{2\pi}=\sum_{l=0}^{n_l-1}\frac{\tilde{\lambda}_l}{2^{n_l}}+\frac{\epsilon}{2^{n_l}}$ 입니다. 따라서 이 경우, $\alpha_{0|j}=1$, $\alpha_{l|j}=0$ for $l\neq 0$ 가 성립합니다. 이 경우에만 QPE 후 레지스터의 상태를 다음과 같이 쓸 수 있습니다.

$$\sum_{j=0}^{N-1}b_j|\lambda_j\rangle_{n_l}|u_j\rangle_{n_b}$$

그렇지 않으면, $\{v^*\}$ 가 크고 레지스터의 상태가 $\frac{\lambda_jt}{2\pi}=\sum_{l=0}^{n_l-1}\frac{\tilde{\lambda}_l}{2^{n_l}}+\frac{\epsilon}{2^{n_l}}$ 인 경우에만 그렇습니다.

$$\sum_{j=0}^{N-1} \sum_{l=0}^{2^{n_l}-1} \alpha_{l|j} b_j |l\rangle_{n_l} |u_j\rangle_{n_b}$$

3. 예시: 4-큐비트 HHL

알고리즘을 설명하기 위해 소개 부분에서 사용한 작은 예제를 살펴보겠습니다. 즉,

$$A = \begin{pmatrix} 1 & -1/3 \\ -1/3 & 1 \end{pmatrix} \quad \text{and} |b\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

우리는 {qubit}을 사용하여 $|b\rangle$ 를 나타내고, 나중에 $|u\rangle$ 를 나타냅니다. qubits은 고유값의 이진 표현을 저장하고, 보조 qubit은 조건부 회전을 저장하므로 알
고리즘은 $|b\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ 로 시작합니다. $n_l = 2$

알고리즘을 설명하기 위해 약간 속임수를 써서 λ 의 고유값을 계산하여 n -레지스터에서 스케일 조정된 고유값의 정확한 이진 표현을 얻을 수 있도록 n_l 를 선택할 것입니다. 그러나 HHL 알고리즘 구현에는 고유값에 대한 사전 지식이 필요하지 않다는 점을 명심하세요. 그렇긴 하지만, 간단한 계산을 통해 $\{v^*\}$ 를 구할 수 있습니다.

$$\lambda_1 = 2/3 \text{ and } \lambda_2 = 4/3$$

이전 섹션에서 QPE는 n -비트(이 경우 n_l -비트) 이진 근사값을 출력한다는 것을 상기하십시오. 따라서, 만약 우리가 $\{v^*\}$ 를 설정하면 $t = 2\pi \cdot \frac{\lambda_j}{2\pi}$

$$t = 2\pi \cdot \frac{3}{8}$$

QPE는 $\{v^*\}$ 에 대한 n_l -비트 이진 근사값을 제공합니다.

$$\frac{\lambda_1 t}{2\pi} = 1/4 \text{ and } \frac{\lambda_2 t}{2\pi} = 1/2$$

각각 $\{v^*\}$ 입니다.

$$|01\rangle_{n_l} \quad \text{and} \quad |10\rangle_{n_l}$$

고유 벡터는 각각 다음과 같습니다.

$$|u_1\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix} \quad \text{and} \quad |u_2\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

다시 말하지만, HHL 구현을 위해 고유 벡터를 계산할 필요는 없습니다. 실제로, 차원 $\{v^*\}$ 의 일반적인 에르미트 행렬은 최대 $\{v^*\}$ 개의 서로 다른 고유값을 가질 수 있으므로, 이를 계산하는 데 $\{v^*\}$ 시간이 걸리고 양자 이점이 사라
집니다. $O(N)$

그러면 n_l 의 고유기저에서 다음과 같이 쓸 수 있습니다.

$$|b\rangle_{n_b} = \sum_{j=1}^2 \frac{1}{\sqrt{2}} |u_j\rangle_{n_b}$$

이제 HHL 알고리즘의 다양한 단계를 살펴볼 준비가 되었습니다.

1. 이 예제에서는 상태 준비가 $\{v^*\}$ 이므로 간단합니다. **양자 위상 추정 알고리즘(QPE)**을 적용하면 다음과 같은 결과를 얻습니다.

$$\frac{1}{\sqrt{2}}|01\rangle|u_1\rangle + \frac{1}{\sqrt{2}}|10\rangle|u_2\rangle$$

3. 가의 가장 작은 (재조정된) 고유값보다 작은 조건부 회전. ϕ_1 에 상수는 의 가장 작은 (재조정된) 고유값보다 작으면서 보조 큐비트가 측정될 때 상태에 있을 확률이 커지도록 가능한 한 크게 선택해야 합니다. -

$$\frac{1}{\sqrt{2}}|01\rangle|u_1\rangle \left(\sqrt{1 - \frac{(1/8)^2}{(1/4)^2}}|0\rangle + \frac{1/8}{1/4}|1\rangle \right) + \frac{1}{\sqrt{2}}|10\rangle|u_2\rangle \left(\sqrt{1 - \frac{(1/8)^2}{(1/2)^2}}|0\rangle + \frac{1/8}{1/2}|1\rangle \right)$$

$$= \frac{1}{\sqrt{2}}|01\rangle|u_1\rangle \left(\sqrt{1 - \frac{1}{4}}|0\rangle + \frac{1}{2}|1\rangle \right) + \frac{1}{\sqrt{2}}|10\rangle|u_2\rangle \left(\sqrt{1 - \frac{1}{16}}|0\rangle + \frac{1}{4}|1\rangle \right)$$

4. QPE를 적용한 후 양자 컴퓨터는 다음 상태에 있습니다. $\{v^*\}$

$$\frac{1}{\sqrt{2}}|00\rangle|u_1\rangle \left(\sqrt{1 - \frac{1}{4}}|0\rangle + \frac{1}{2}|1\rangle \right) + \frac{1}{\sqrt{2}}|00\rangle|u_2\rangle \left(\sqrt{1 - \frac{1}{16}}|0\rangle + \frac{1}{4}|1\rangle \right)$$

5. 보조 큐비트를 측정할 때 결과에 따라 상태는 다음과 같습니다. $\{v^*\}$

$$\frac{\frac{1}{\sqrt{2}}|00\rangle|u_1\rangle\frac{1}{2}|1\rangle + \frac{1}{\sqrt{2}}|00\rangle|u_2\rangle\frac{1}{4}|1\rangle}{\sqrt{5/32}}$$

간단한 계산을 통해 다음을 알 수 있습니다.

$$\frac{\frac{1}{2\sqrt{2}}|u_1\rangle + \frac{1}{4\sqrt{2}}|u_2\rangle}{\sqrt{5/32}} = \frac{|x\rangle}{\|x\|}$$

6. 추가적인 게이트 없이, $w\{v^*\}$ e는 $\{v^*\}$ 의 노름을 계산할 수 있습니다. 이는 p에서 보조 큐비트에서 $\{v^*\}$ 를 측정할 확률입니다. 이전 단계.

$$P(|1\rangle) = \left(\frac{1}{2\sqrt{2}} \right)^2 + \left(\frac{1}{4\sqrt{2}} \right)^2 = \frac{5}{32} = \|x\|^2$$

4. Qiskit 구현

이제 예제에서 문제를 분석적으로 해결했으므로 양자 시뮬레이터와 실제 하드웨어에서 HHL을 실행하는 방법을 설명하는 데 사용하겠습니다. 다음은 이 저장소에서 찾을 수 있고 해당 파일에 설명된 대로 설치할 수 있는 Qiskit 기반 패키지인 를 사용합니다. 양자 시뮬레이터의 경우 이미 행렬 `quantum_linear_solvers` Readme `quantum_linear_solvers` 및 를 가장 간단한 예의 입력으로만 요구하는 HHL 알고리즘의 구현을 제공합니다. NumPy 배열로 일반적인 Hermitian 행렬과 임의의 초기 상태를 알고리즘에 제공할 수 있지만 이러한 경우 양자 알고리즘은 지수적 속도 향상을 달성하지 못합니다. 이는 기본 구현이 정확하고 따라서 큐비트 수에서 지수적이기 때문입니다. 다항식 알고리즘이 없습니다.

리소스 {v*} 정확한 임의의 양자 상태를 준비할 수 있거나, 일반적인 에르미트 행렬 에 대해 정확한 연산을 수행할 수 있는 큐비트 회로입니다. 만약 우리가 {v*}에 대한 효율적인 구현을 알고 있다면 특정 문제 행렬 및/또는 벡터는 객체로 제공될 수 있습니다. 또는 삼중대각 Toeplitz 행렬에 대한 효율적인 구현이 이미 있으며 앞으로 더 많은 구현이 있을 수 있습니다. QuantumCir cuit

하지만 이 글을 쓰는 시점에서 기존 양자 컴퓨터는 노이즈가 심하고 작은 회로만 실행할 수 있습니다. 따라서 4.B절에서는 우리 예제가 속한 문제 클래스에 사용할 수 있는 최적화된 회로를 살펴보고 양자 컴퓨터의 노이즈를 처리하기 위한 기존 절차를 언급하겠습니다.

A. 시뮬레이터에서 HHL 실행하기: 일반적인 방법

이 페이지의 코드를 실행하려면 선형 솔버 패키지를 설치해야 합니다. 다음 명령을 통해 설치할 수 있습니다.

```
pip install git+https://github.com/anedumla/quantum_linear_solvers
```

선형 시스템 문제를 해결하는 모든 알고리즘의 인터페이스는 입니다. 해결할 문제는 메서드가 호출될 때만 지정됩니다. LinearSolver solve()

```
LinearSolver(...).solve(matrix, vector)
```

가장 간단한 구현은 행렬과 벡터를 NumPy 배열로 취합니다. 아래에서는 솔루션을 검증하기 위해 (고전적인 알고리즘)도 만듭니다. NumPyLinearSolver

```
numpy를 np로 가져오기
# pylint: disable=line-too-long
선형_솔버에서 NumPyLinearSolver, HHL 가져오기
행렬 = np.array([[1, -1/3], [-1/3, 1]]) 벡터 = np.array([1, 0]) naive_hhl_
solution = HHL().solve(matrix, vector)
```

클래식카의 경우1 솔버에서 HHL 내의 양자 상태로 인코딩되면 발생하는 재정규화를 고려하기 위해 우변(즉,)의 스케일을 조정해야 합니다. vector / np.linalg.norm(vector) vector

```
classical_solution = NumPyLinearSolver().solve(matrix,
vector/np.linalg.norm(vector))
```

패키지 co 특정 유형의 행렬에 대한 효율적인 구현을 위한 자리 표시자 역할을 하는 폴더가 있습니다. 현재 작성 시점에서 포함된 유일하게 진정으로 효율적인 구현은 {v*}입니다(예: 복잡성). 스케일링 다항식 {v*} 수에서) 클래스입니다. 삼중대각 토폴리츠 대칭 실수 행렬은 다음 형식을 갖습니다. linear_solvers matrices TridiagonalToeplitz y

$$A = \begin{pmatrix} a & b & 0 & 0 \\ b & a & b & 0 \\ 0 & b & a & b \\ 0 & 0 & b & a \end{pmatrix}, a, b \in \mathbb{R}$$

(이 설정에서는 HHL 알고리즘이 입력 행렬이 에르미트 행렬이라고 가정하므로 비대칭 행렬은 고려하지 않습니다).

행렬이 우리 예제와 같은 형태이므로 인스턴스를 생성하고 배열을 입력으로 사용하여 시스템을 푸는 결과와 비교할 수 있습니다. TridiagonalToeplitz(num_qubits, a, b)

```
from linear_solvers.matrices.tridiagonal_toeplitz import TridiagonalToeplitz
tridi_matrix = TridiagonalToeplitz(1, 1, -1 / 3)
tridi_solution = HHL().solve(tridi_matrix, vector)
```

{v*}를 상기하는 HHL 알고리즘은 고전적인 알고리즘보다 시스템 크기에 따라 지수적으로 더 빠르게 해를 찾을 수 있습니다(예: 다항식 복잡도 대신 로그 복잡도). 그러나 이 지수적인 {v*} speedup은 다음과 같은 한계를 얻지는 못합니다.

대신, 벡터 {v*}를 나타내는 양자 상태를 얻고 이 벡터의 모든 구성 요소를 학습하는 데 차원에서 선형 시간이 걸리므로 양자 알고리즘으로 얻은 속도 향상이 줄어듭니다.

따라서 해에 대한 정보를 학습하기 위해 (소위 관측가능한) 함수에서만 함수를 계산할 수 있습니다. 이는 다음 속성을 포함하는 에 의해 반환된 객체에 반영됩니다. LinearSolverResult solve()

- state : 해를 준비하는 회로 또는 벡터로서의 해 euclidean_norm : 알고리즘이 계산
- 방법을 아는 경우의 유클리드 노름 observable : 계산된 관측값(목록) circuit_result
- s : 회로(목록)에서 얻은 관측값 결과
-

다음과 같이 회로 결과를 얻은 솔루션을 확인합니다. observable circuit_ 결과

먼저, 고전적인 알고리즘의 결과였으므로 호출하면 배열을 반환합니다. classical_solution .state

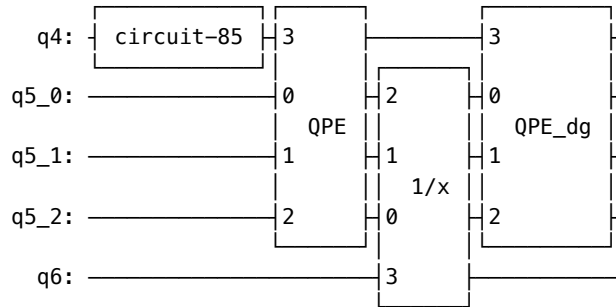
```
print('고전적 상태:', classical_solution.state)
```

```
classical state: [1.125 0.375]
```

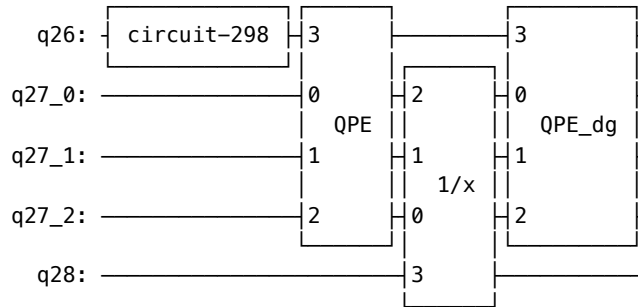
다른 두 예시는 양자 알고리즘이었으므로 양자 상태에만 접근할 수 있습니다. 이는 솔루션 상태를 준비하는 양자 회로를 반환하여 달성됩니다.

```
print('naive state:')
print(naive_hhl_solution.state)
print('tridiagonal state:')
print(tridi_solution.state)
```


순진한 상태:



삼중대각 상태:



벡터의 유클리드 놈은 $\mathbf{x}=(x_1,\dots,x_N)$ 로 정의됩니다. 따라서 섹션 B의 단계에서 $\frac{\|\mathbf{x}\|_2}{\sqrt{N}}$ 에 $\frac{1}{\sqrt{N}}$ 를 곱한 값은 의 제곱 놈입니다. 이는 HHL 알고리즘이 항상 해의 유클리드 놈을 계산할 수 있으며 결과의 정확도를 비교할 수 있음을 의미합니다.

```
print('고전적인 유클리드 놈:', classical_solution.euclidean_norm) print('단순한 유클리드 놈:', naive_hhl_solution.euclidean_norm) print('삼중대각 유클리드 놈:', tridi_solution.euclidean_norm)
```

고전적인 유클리드 놈: 1.1858541225631423 순진한 유클리드
놈: 1.185854122563138 삼중대각 유클리드 놈: 1.1858541225
631365

해결 벡터를 성분별로 비교하는 것은 더 까다로운데, 이는 양자 알고리즘에서 전체 해결 벡터를 얻을 수 없다는 생각을 다시 반영합니다. 그러나 교육적인 목적으로 얻은 다양한 해결 벡터가 벡터 성분 수준에서도 좋은 근사값인지 확인할 수 있습니다.

그렇게 하려면 `Qiskit`은 패키지에서 가져와 올바른 벡터 구성 요소를 추출해야 합니다. 즉, 보조 큐비트(회로의 맨 아래)에 해당하는 구성 요소는 `Statevector` `quantum_info` 이고 작업 큐트(가운데 두 회로에서) 입니다. 따라서 우리는 해 벡터의 첫 번째 및 두 번째 구성 요소에 해당하는 상태 `10000` `10001` 에 관심이 있습니다.

```
from qiskit.quantum_info import Statevector
```

```
naive_sv = Statevector(naive_hhl_solution.state).data
tridi_sv = Statevector(tridi_solution.state).data
```

```
# 벡터 컴포넌트 추출; 10000(bin) == 16 & 10001(bin) == 17
naive_full_vector = np.array([naive_sv[16], naive_sv[17]])
tridi_full_vector = np.array([tridi_sv[16], tridi_sv[17]])
```

```
print('naive raw solution vector:', naive_full_vector)
print('tridi raw solution vector:', tridi_full_vector)
```

순진한 raw 해 벡터: [0.75+3.52055626e-16j 0.25+1.00756137e-16j]
삼중대각 raw 해 벡터: [0.75-9.01576529e-17j 0.25+3.74736911e-16j]

언뜻 보기에 이것이 잘못된 것처럼 보일 수도 있습니다. 왜냐하면 구성 요소가 실수가 아닌 복소수이기 때문입니다. 그러나 허수 부분은 매우 작고 컴퓨터 정확도 때문일 가능성이 높다는 점에 유의하십시오. $\{v^*\}$ 에서 무시됩니다. `.real` 경우 (배열의 속성을 사용하여 실수 부분을 가져옵니다).

다음으로, 회로의 각 부분에서 오는 상수를 억제하기 위해 벡터를 각 노름으로 나눕니다. 그런 다음 위에서 계산한 각 유클리드 노름을 이러한 정규화된 벡터에 곱하여 전체 솔루션 벡터 $\{v^*\}$ 를 복구할 수 있습니다.

```
def get_solution_vector(solution):
    """LinearSolverResult에서 시뮬레이션된 상태 벡터를 추출하고 정규화합니다."""
    solution_vector = Statevector(solution.state).data[16:18].real
    norm = solution.euclidean_norm
    return norm * solution_vector / np.linalg.norm(solution_vector)
```

```
print('완전한 순진한 해 벡터:', get_solution_vector(naive_hhl_solution))
print('완전한 삼중대각 해 벡터:', get_solution_vector(tridi_solution))
print('고전적 상태:', classical_solution.state)
```

전체 순진한 해 벡터: [1.125 0.375]
전체 삼중대각 행렬 해 벡터: [1.125 0.375]
고전적 상태: [1.125 0.375]

정확한 이유는 사용된 모든 기본 방법이 정확하기 때문이라는 것은 놀라운 일이 아닙니다. 그러나 `naive_hhl_solution` `tridi_solution` 시스템 크기 경우에만 정확합니다. 더 큰 행렬의 경우 아래의 약간 더 큰 예에서 볼 수 있듯이 근사값이 됩니다. 2×2

```
from scipy.sparse import diags
```

```
NUM_QUBITS = 2
```

```
MATRIX_SIZE = 2 ** NUM_QUBITS
```

```
# 삼중대각 퇴플리츠 대칭 행렬의 항목
```

```
# pylint: disable=invalid-name
```

```
a = 1
```

```
b = -1/3
```

```
행렬 = diags([b, a, b], [-1, 0, 1],
```

```
              shape=(MATRIX_SIZE, MATRIX_SIZE)).toarray()
```

```
벡터 = np.array([1] + [0]*(MATRIX_SIZE - 1))
```

```
# 알고리즘 실행
```

```
classical_solution = NumPyLinearSolver(
```

```
    ).solve(matrix, vector / np.linalg.norm(vector))
```

```
naive_hhl_solution = HHL().solve(matrix, vector)
```

```
tridi_matrix = TridiagonalToeplitz(NUM_QUBITS, a, b)
```

```
tridi_solution = HHL().solve(tridi_matrix, vector)
```

```
print('고전적인 유클리드 노름:', classical_solution.euclidean_norm) print('단순한 유클리드 노름:', naive_hhl_solution.euclidean_norm) print('삼중대각 유클리드 노름:', tridi_solution.euclidean_norm)
```

고전적인 유클리드 노름: 1.237833351044751 순진한 유클리드 노
름: 1.209980623111888 삼중대각 유클리드 노름: 1.209457721870
5271

We can also compare the difference in resources from the exact method and the efficient implementation. The 2×2 system size is again special in that the exact algorithm requires less resources, but as we increase the system size, we can see that indeed the exact method scales exponentially in the number of qubits while is polynomial. `TridiagonalToeplitz`

```
from qiskit import transpile
```

```
MAX_QUBITS = 4
```

```
a = 1
```

```
b = -1/3
```

```
나 = 1
```

```
# 양자 비트 수에 따른 회로 깊이를 계산하여 사용량 비교
```

```
리소스 수 (경고: 실행하는 데 시간이 오래 걸립니다) naive_depths = [] tridi_depths =
```

```
[]
```

```
for n_qubits in range(1, MAX_QUBITS+1): matrix = diags([b, a, b], [-1, 0, 1], shape=(2**n
_qubits, 2**n_qubits)).toarray() vector = np.array([1] + [0]*(2**n_qubits - 1))
```

```
naive_hhl_solution = HHL().solve(matrix, vector) tridi_matrix = TridiagonalT
```

```
oeplitz(n_qubits, a, b) tridi_solution = HHL().solve(tridi_matrix, vector)
```

```
naive_qc = 트랜스파일(naive_hhl_solution.state,
```

```
basis_gates=['id', 'rz', 'sx', 'x', 'cx'])
```

```
tridi_qc = 트랜스파일(tridi_solution.state,
```

```
basis_gates=['id', 'rz', 'sx', 'x', 'cx'])
```

```
naive_depths.append(naive_qc.depth())tridi_depths.append(tr
```

```
idi_qc.depth())i +=1
```

```
크기] = [f"{2**n_qubits}×{2**n_qubits}" for n_qubits in range(1, MAX_QU
```

```
BITS+1)] 열 = ['시스템 크기', '양자 솔루션 깊이', '삼중대각 솔루션 깊이']
```

```
데이터 = np.array([크기, naive_depths, tridi_depths]) ROW_FORMAT="{:
```

```
>23}" * (len(columns) + 2) 팀과 행을 zip하여 열과 데이터를 반복합니다. p
```

```
rint(ROW_FORMAT.format(team, *row))
```

size of the system	2×2	4×4	8×8	16×16
quantum_solution depth	334	2593	34008	403899
tridi_solution depth	565	5107	14756	46552

구현 이유

여전히 지수적인 리소스가 필요한 이유는 현재 조건부 회전 구현(섹션 2.B의 3단계)이 정확하기 때문입니다(즉, {v*}에서 지수적인 리소스가 필요함). 대신에 우리는

계산 방법 {v*}

Tridiagonal과 비교하여 기본 구현에 필요한 리소스가 더 많은 이유는 계산 방식의 차이 때문입니다. {v*}

n

```
print('초과:',
      [naive_depths[i] - tridi_depths[i] for i in range(0, len(naive_depths))])

초과: [-231, -2514, 19252, 357347]
```

가까운 미래에는 조건부 회전의 다항식 구현을 얻기 위해 통합할 계획입니다. `qiskit.circuit.library.arithmetics.PiecewiseChebyshev`

이제 관측 가능량 주제로 돌아가서 `observable` 및 `circuit_results`에 어떤 속성이 포함되어 있는지 알아볼 수 있습니다.

해 벡터의 함수를 계산하는 방법은 메서드에 `a`를 입력으로 제공하는 것입니다. 입력으로 제공할 수 있는 두 가지 유형이 있습니다. `.solve()` `LinearSystemObservable` `LinearSystemObservable`

선형_솔버.관찰_가능물에서 절대평균, 행렬함수 가져오기

벡터 $\mathbf{x}=(x_1,...,x_N)$ 에 대해 관찰 가능한 값은 $Average\{v^*\}$ 를 계산합니다.
$$\frac{1}{N}\sum_{i=1}^N x_i$$

```
NUM_QUBITS = 1
MATRIX_SIZE = 2 ** NUM_QUBITS
# 삼중대각 토플리츠 대칭 행렬의 항목
a = 1
b = -1/3

행렬 = diags([b, a, b], [-1, 0, 1], shape=(MATRIX_SIZE, MATRIX_SIZE)).toarray()
```

```
벡터 = np.array([1] + [0]*(MATRIX_SIZE - 1))
삼중_행렬 = 삼중대각Toeplitz(1, a, b)
```

```
average_solution = HHL().solve(tridi_matrix,
벡터, AbsoluteAverage())
classical_average = NumPyLinearSolver().solve(matrix, 벡터 / np.linalg.norm(vector), AbsoluteAverage())
```

```
print("양자 평균:", average_solution.observable)
print('고전적 평균:', classical_average.observable)
print("양자 회로 결과:", average_solution.circuit_results)
```

양자 평균: 0.74999999999999962
고전적 평균: 0.75
양자 회로 결과: (0.49999999999999952+0j)

관찰 가능한 것은 `MatrixFunctional` `fo`를 계산합니다. `Matrix` 벡터와 `Observable` 대칭 Toeplitz 행렬 `{v*}`입니다. 이 클래스는 주 대각선과 오프 대각선을 취합니다. 생성자 메서드를 위한 행렬의 `{v*}` 값입니다.

관찰 가능한 = `MatrixFunctional(1, 1 / 2)`

```
functional_solution = HHL().solve(tridi_matrix, vector, observable)
classical_functional = NumPyLinearSolver().solve(matrix, vector / np.linalg.norm(vector), observable)
```

```
print("양자 함수:", functional_solution.observable)
print('고전적 함수형:', classical_functional.observable)
print("양자 회로 결과:", functional_solution.circuit_results)
```

양자 함수: 1.82812499999999818
고전적 함수형: 1.828125
양자 회로 결과: [(0.62499999999999941+0j), (0.49999999999999952+0j), (0.12499999999999988+0j)]

따라서 `observable` 에 대한 함수의 최종 값을 포함하고, `circuit_results` `observable` 의 결과를 처리하는 데 사용되는 회로에서 얻은 원시 값을 포함합니다.

이 '결과 처리 방법'은 어떤 인수를 취하는지 살펴보는 것이 더 좋습니다. 이 메서드는 최대 5개의 인수를 허용합니다. `.solve()` `solve()`

```
def solve(self, matrix: Union[np.ndarray, QuantumCircuit], vector: Union[np.ndarray, QuantumCircuit], observable: Optional[Union[LinearSystemObservable, BaseOperator, List[BaseOperator]]] = None, post_rotation: Optional[Union[QuantumCircuit, List[QuantumCircuit]]] = None, post_processing: Optional[Callable[[Union[float, List[float]]], Union[float, List[float]]]] = None) -> LinearSolverResult:
```

처음 두 개는 선형 시스템을 정의하는 행렬과 방정식의 우변 벡터이며, 이는 이미 다룬 내용입니다. 나머지 매개변수는 해 벡터에서 계산될 관측 가능한 항목(목록)에 관한 것이며, 두 가지 다른 방식으로 지정할 수 있습니다. 한 가지 옵션은 세 번째이자 마지막 매개변수로 (목록) (`s`)를 제공하는 것입니다. 또는, `및` 의 자체 구현을 제공할 수 있습니다. 여기서 `LinearSystemObservable` `observable` `post_rotation` `post_processing`

- `observable` 은 관측 가능한 값의 기대값을 계산하는 연산자이며, 예를 들어 `PauliSumOp` 일 수 있습니다. `post_rotation`
- `ion` 은 추가 게이트가 필요한 경우 정보를 추출하기 위해 해법에 적용될 회로입니다. `post_processing` 은 계산된 확률로부터 관측 가능한 값의 값을 계산하는 함수입니다.

다시 말해, 회로는 circuit_results post_rotation post_processing circuit_results observable 만큼 많을 것이며, 인쇄할 때 보이는 값을 사용하여 인쇄할 때 보이는 값을 얻는 방법을 알고리즘에 알려줍니다.

마지막으로, 클래스는 생성자 메서드에서 다음 매개변수를 허용합니다. HHL

- 오차 허용 범위: 해의 근사 정확도, 기본값은 1e-2 기대값: 기대값을 평가하는 방법, 기본값은 PauliExpectation 양자 인스턴스: 또는 백엔드, 기본값은 시뮬레이션 QuantumInstance Statevector
-

```
from qiskit import Aer

백엔드 = Aer.get_backend('aer_simulator') hhl = HHL(1e-3, quantum_instance=backend)

정확한_해 = hhl.solve(matrix, vector) 고전적인_해 = NumPyLinearSolver(matrix, vector / np.linalg.norm(vector))

print(accurate_solution.euclidean_norm) print(classical_solution.euclidean_norm)

1. 185854122563138 1.18
58541225631423
```

B. 실제 양자 장치에서 HHL 실행: 최적화된 예제

이전 섹션에서는 Qiskit에서 제공하는 표준 알고리즘을 실행하여 큐비트를 사용하고, 깊이가 ~ 게이트 100,545 CNOT 게이트가 필요하다는 것을 확인했습니다. 이러한 수치는 현재 사용 가능한 하드웨어에는 실현 불가능하므로 이러한 양을 줄여야 합니다. 특히, 단일 큐비트 게이트보다 충실도가 떨어지므로 CNOT 수를 배로 줄이는 것이 목표입니다. 또한, 문제의 원래 설명과 같이 큐비트 수를 로 줄일 수 있습니다. Qiskit 방법은 일반적인 문제를 위해 작성되었기 때문에 추가적인 보조 큐비트가 필요합니다.

그러나, 오로지 {q}와 큐비트 수를 줄이는 것은 실제 하드웨어에서 솔루션에 대한 좋은 근사값을 제공하지 않습니다. 이는 오류의 원인이 두 가지이기 때문입니다. t 실행 중에 발생하는 오류와 {v*} 회로를 읽습니다. 오류가 발생했습니다.

Qiskit은 모든 기본 상태를 개별적으로 준비하고 측정하여 판독 오류를 완화하는 모듈을 제공합니다. 이 주제에 대한 자세한 내용은 Dewes 등의 논문3에서 확인할 수 있습니다. 오류를 완화하기 위해 Richardson 외삽법을 사용하여 회로를 세 번 실행하여 오류를 0으로 제한할 수 있습니다. 각 회로는 각 CNOT 게이트를 , 및 CNOT로 대체합니다4. 아이디어는 이론적으로 세 회로가 동일한 결과를 생성해야 하지만 실제 하드웨어에서 CNOT를 추가하면 오류가 증폭된다는 것입니다. 증폭된 오류로 결과를 얻었고 각 경우에 오류가 얼마나 증폭되었는지 추정할 수 있으므로 수량을 재결합하여 분석 솔루션에 더 가까운 새로운 결과를 얻을 수 있습니다.

아래는 {v*} 형태의 모든 문제에 사용할 수 있는 최적화된 회로입니다.

$$A = \begin{pmatrix} a & b \\ b & a \end{pmatrix} \quad \text{and} |b\rangle = \begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix}, a, b, \theta \in \mathbb{R}$$

The following optimisation was extracted from a work on the HHL for tridiagonal symmetric matrices^[5], this particular circuit was derived with the aid of the UniversalQCompiler software^[6].


```
from qiskit import QuantumRegister, QuantumCircuit import numpy as np
```

t = 2 # 이것은 최적이지 않습니다. 연습 삼아, 이것을 최상의 결과를 얻을 수 있는 값으로 설정하십시오. 해법은 8절을 참조하십시오.

NUM_QUBITS = 4 # 총 큐비트 수 nb = 1 # 해를 나타내는 큐비트 수 nl = 2 # 고유값을 나타내는 큐비트 수

theta = 0 # |b>를 정의하는 각도

a = 1 # 행렬 대각선 b = -1/3 # 행렬 비대각선

양자 및 고전 레지스터 초기화 qr = QuantumRegister(NUM_QUBITS)

양자 회로 qc = QuantumCircuit(qr)
생성

qrb = qr[0:nb] qrl = qr[nb:nb+nl]
qra = qr[nb+nl:nb+nl+1]

상태 준비. qc.ry(2*theta, qrb[0])

qrl에서 qu에 대한 $e^{iA}t$
를 사용한 QPE: qc.h(qu)

qc.p(a*t, qrl[0]) qc.p(a*t*2,
qrl[1])

qc.u(b*t, -np.pi/2, np.pi/2, qrb[0])

$|\lambda_1\rangle$ 에 대한 제어된 $e^{iA}t$: params=b*t

qc.p(np.pi/2,qrb[0]) qc.cx(qrl[0],
qrb[0]) qc.ry(params,qrb[0]) qc.c
x(qrl[0],qrb[0]) qc.ry(-params,q
rb[0]) qc.p(3*np.pi/2,qrb[0])

```
# \lambda_{2} 상의 제어된 e^{2iAt}: params = b*t*2
```

```
qc.p(np.pi/2,qrb[0]) qc.cx(qrl[1],
qrb[0]) qc.ry(params,qrb[0]) qc.c
x(qrl[1],qrb[0]) qc.ry(-params,qr
b[0]) qc.p(3*np.pi/2,qrb[0])
```

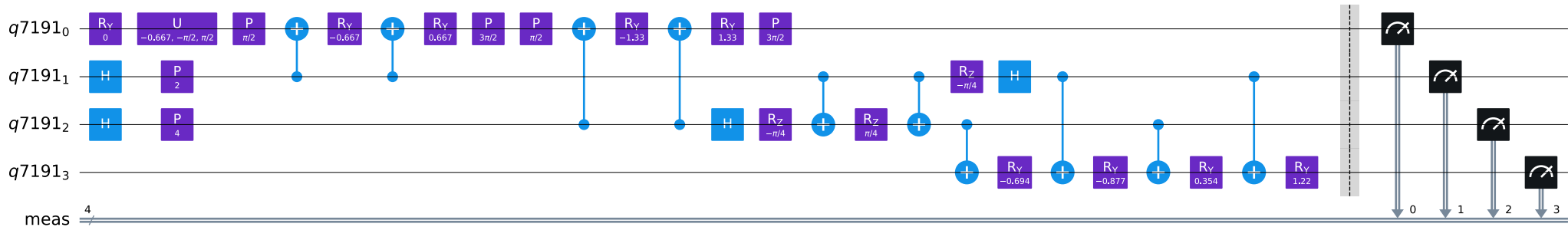
```
# 역 QFT qc.h(qrl[1]) qc.rz(-np.
pi/4,qrl[1]) qc.cx(qrl[0],qrl[1]) q
c.rz(np.pi/4,qrl[1]) qc.cx(qrl[0],q
rl[1]) qc.rz(-np.pi/4,qrl[0]) qc.h(
qrl[0])
```

```
# 고유값 회전 t1=(-np.pi +np.pi/3 - 2*np.arcsin(1/3))/4 t2
=(-np.pi -np.pi/3 + 2*np.arcsin(1/3))/4 t3=( np.pi -np.pi/3 -
2*np.arcsin(1/3))/4 t4=( np.pi +np.pi/3 + 2*np.arcsin(1/3))/
4
```

```
qc.cx(qrl[1],qra[0]) qc.ry(t1,q
ra[0]) qc.cx(qrl[0],qra[0]) qc.r
y(t2,qra[0]) qc.cx(qrl[1],qra[0
]) qc.ry(t3,qra[0]) qc.cx(qrl[0]
,qra[0]) qc.ry(t4,qra[0]) qc.me
asure_all()
```

```
print(f"깊이: {qc.depth()}") print(f"CNOTS: {qc.count_ops()['cx']}") qc.draw(fold=-1)
```

깊이: 26 CN
OT 게이트
수: 10



아래 코드는 다음을 수행하는 회로, 실제 하드웨어 백엔드 및 사용하려는 큐비트 세트를 사용하고 지정된 장치에서 실행할 수 있는 인스턴스를 반환합니다. {v*} 및 CNOT로 회로를 만드는 것은 3 5
t를 호출하는 것과 동일하지만 양자 회로를 사용하여 트랜스파일 메서드를 실행합니다.

실제 하드웨어 장치는 정기적으로 재보정해야 하며, 특정 큐비트 또는 게이트의 충실도는 시간이 지남에 따라 변경될 수 있습니다. 또한, 칩마다 큐비트 연결성이 다릅니다. 지정된 장치에서 연결되지 않은 두 큐비트 간에 2 큐비트 게이트를 수행하는 회로를 실행하려고 하면 트랜스파일러가 SWAP 게이트를 추가합니다. 따라서 다음 코드를 실행하기 전에 IBM Quantum Experience 웹페이지[7]를 확인하고 주어진 시간에 올바른 연결성과 가장 낮은 오류율을 가진 큐비트 세트를 선택하는 것이 좋습니다. {v*}

```
from qiskit import IBMQ, transpile from qiskit.utils.mitigation import complete
_meas_cal

공급자 = IBMQ.load_account()

백엔드 = provider.get_backend('ibmq_quito') # 실제 하드웨어 레이아웃을 사용하여 보정 = [2,3,0,4]

칩 큐비트 = 5

# 실제 하드웨어를 위한 트랜스파일된 회로
qc_qa_cx = transpile(qc, backend=backend, initial_layout=layout)
```

다음 단계는 판독 오류를 완화하는 데 사용되는 추가 회로를 만드는 것입니다[3].

```
meas_cals, state_labels = complete_meas_cal(qubit_list=layout, qr=QuantumRegister(chip_qubits))

qcs = meas_cals + [qc_qa_cx]

작업 = backend.run(qcs, shots=10)
```

다음 p lot[5]는 위 회로를 실제 하드웨어에서 다양한 초기 상태로 실행한 결과를 보여줍니다. -축은 각 경우의 초기 상태를 나타내는 각도를 나타냅니다. 결과는 다음에서 얻었습니다. 10 5
화 후 편집됨 판독 오류를 보정한 다음, , 및 CNOT를 사용하여 회로 결과로부터 회로 실행 중에 발생하는 오류를 추정합니다. 완



Compare to the results without error mitigation nor extrapolation from the CNOTs⁵.



8. 문제

실제 하드웨어:

1. 최적화된 예제에 대한 시간 매개변수를 설정합니다. {v*}
- ▶ 해결책 (클릭하여 펼치기)
2. 주어진 회로 "에서 및 CNOT에 의해 트랜스파일된 회로를 생성합니다. 회로를 생성할 때 배리어를 추가하여 이러한 연속적인 CNOT 게이트가 사용 시 취소되지 않도록 해야 합니다. 그 함수. qc transpile()
3. 실제 하드웨어에서 회로를 실행하고 결과에 이차 적합을 적용하여 외삽 값을 얻습니다. {v*}

9. 참고 문헌

1. J. R . Shewchuk. 고통 없이 배우는 켈레 기울기법 소개 (An Introduction to the Conjugate Gradient Method Without the Agonizing Pain). 기술 보고서 CMU-CS-94-125, 컴퓨터 과학 대학, 카네기 멜론 대학교, 피츠버그, 펜 펜실베이니아, 1994년 3월.
2. A. W. Harrow, A. Hassidim, and S. Lloyd, “선형 방정식의 양자 알고리즘,” Phys. Rev. Lett. 103.15 (2009), p. 150502. 3. A. Dewes, F. R. Ong, V. Schmitt, R. Lauro, N. Boulant, P. Bertet, D. Vion, and D. Esteve, “개별 단일 샷 큐비트 판독을 갖춘 2-트랜스몬 프로세서의 특성화,” Phys. Rev. Lett. 108, 057002 (2012). 4. N. Stamatopoulos, D. J. Egger, Y. Sun, C. Zoufal, R. Iten, N. Shen, and S. Woerner, “양자 컴퓨터를 사용한 옵션 가격 책정,” arXiv:1905.02666 . 5. A. Carrera Vazquez, A. Frisch, D. Steenken, H. S. Barowski, R. Hiptmair, and S. Woerner, “Richardson 외삽법을 이용한 양자 선형 시스템 알고리즘 개선,” ACM Trans. Quantum Comput. 3 (2022). 6. R. Iten, O. Reardon-Smith, L. Mondada , E. Redmond, R. Singh Kohli, R. Colbeck, “UniversalQCompiler 소개,” arXiv:1904.01072 . 7. <https://quantum-computing.ibm.com/> .
8. D. Bucher, J. Mueggenburg, G. Kus, I. Haide, S. Deutschle, H. Barowski, D. Steenken, A. Frisch, "Qiskit Aqua: HHL 알고리즘으로 선형 방정식 시스템 풀기" https://github.com/Qiskit/qiskit-tutorials/blob/master/legacy_tutorials/aqua/선형_방정식_시스템.ipynb it-

```
# pylint: disable=unused-import
import qiskit.tools.jupyter
%pyplot auto
%matplotlib inline
%qiskit_version_table
```

버전 정보

Qiskit Software	Version
qiskit-terra	0.21.0
qiskit-aer	0.10.4
qiskit-ibmq-provider	0.19.2
qiskit	0.37.2
qiskit-nature	0.4.1
qiskit-finance	0.3.2

qiskit-optimization	0.4.0
qiskit-machine-learning	0.4.0
System information	
Python version	3.8.13
Python compiler	Clang 13.1.6 (clang-1316.0.21.2.5)
Python build	default, Aug 29 2022 05:17:23
OS	Darwin
CPUs	8
Memory (Gb)	32.0
Wed Sep 21 15:26:35 2022 BST	