

Qiskit Document Tutorials 勉強会

アルゴリズム編 - 1

Kifumi Numata

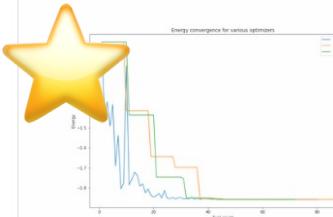
Quantum Tokyo



アルゴリズム編



Qiskitにおけるアルゴリズム入門



VQEの収束性モニタリング



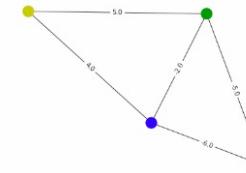
ノイズのあるAerシミュレーターでのVQE



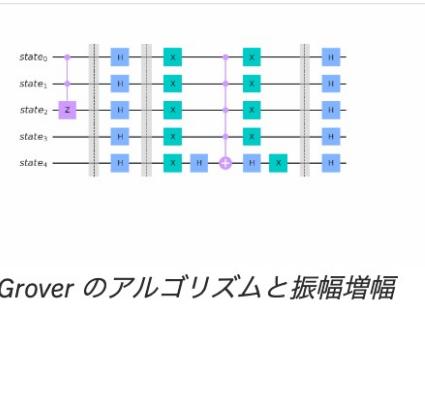
VQEの高度な使い方



IQPEとVQE出力状態の発展



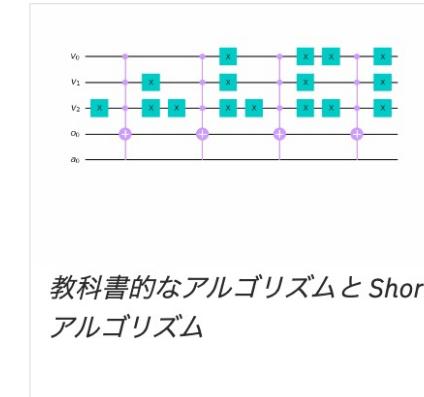
Quantum Approximate
Optimization Algorithm



Groverのアルゴリズムと振幅増幅



Groverのアルゴリズムの例

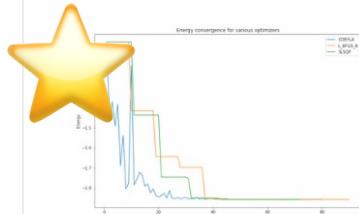


教科書的なアルゴリズムとShorのアルゴリズム

アルゴリズム編



Qiskitにおけるアルゴリズム入門



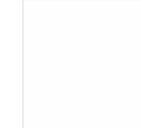
VQEの収束性モニタリング



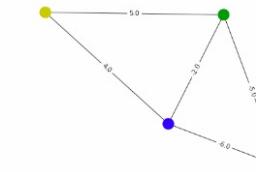
ノイズのあるAerシミュレーターでのVQE



VQEの高度な使い方



IQPEとVQE出力状態の発展



Quantum Approximate Optimization Algorithm

重要参考資料：

Qiskit Algorithms Migration Guide (Qiskitアルゴリズム移行ガイド)

https://qiskit.org/documentation/aqua_tutorials/Qiskit%20Algorithms%20Migration%20Guide.html

1. Qiskit におけるアルゴリズム入門



Qiskit におけるアルゴリズム入門

アルゴリズム・ライブラリーの構成

<https://qiskit.org/documentation/apidoc/algorithms.html>

Qiskitのアルゴリズムライブラリーは、タスクに応じてカテゴリーごとにグループ分けされています。
(一部は、今回の改編でFinance, MLなどの専用のアプリケーションモジュールに移動。)

カテゴリー一覧

- Amplitude Amplifiers (振幅増幅)
 - Grover, etc.
- Amplitude Estimators (振幅推定)
 - AmplitudeEstimation, etc.
- Eigensolvers (演算子の固有値を求める)
 - Eigensolver, NumPyEigensolver, etc.
- Factorizers (因数を求める)
 - Shor, etc.
- Linear Solvers (線形方程式の解を求める)
 - HHL, etc.
- Minimum Eigensolvers (最小固有値を求める)
 - VQE, QAOA, QPE, etc.
 - 化学系のハミルトニアンの基底エネルギー、イジングハミルトニアンの最適化問題など
- Optimizers (VQE用の古典最適化法)
- Phase Estimators (位相推定)
 - HamiltonianPhaseEstimation, etc.

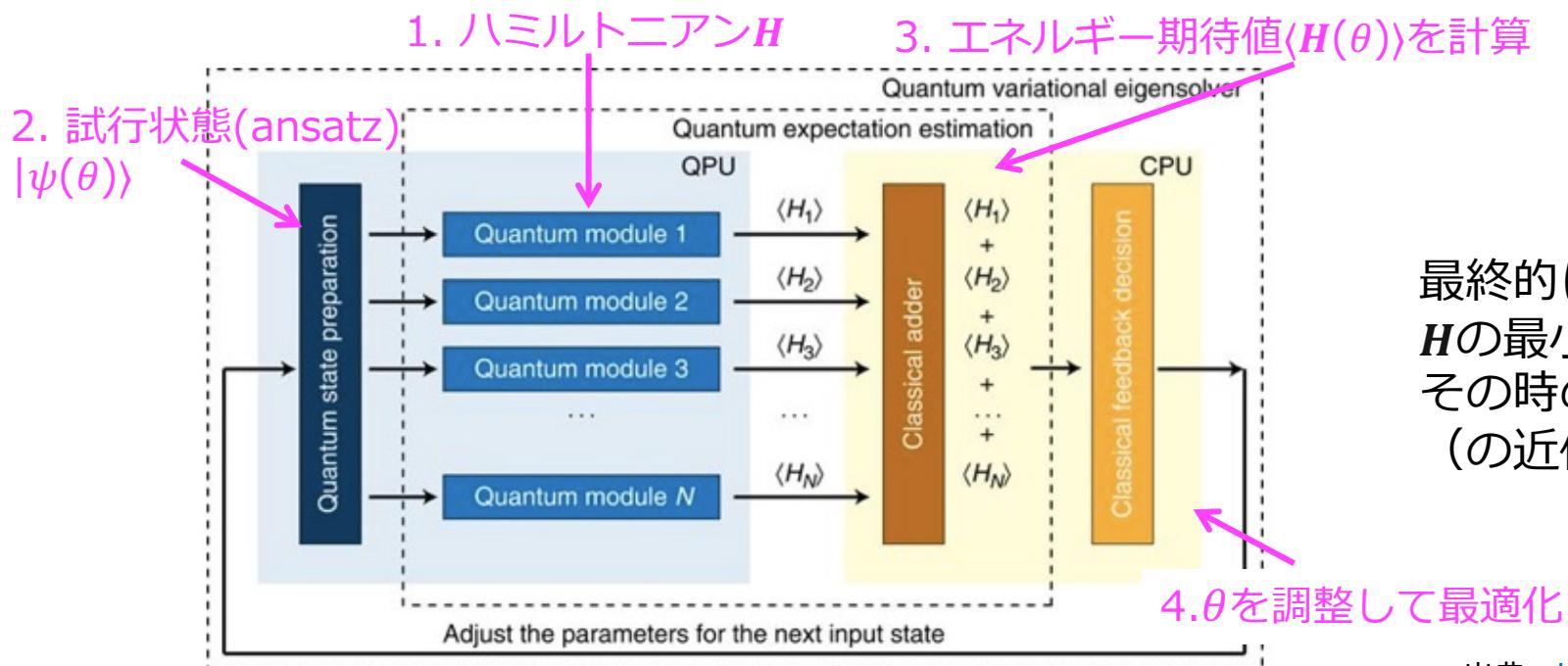
今回のチュートリアルは、

VQE (Variational Quantum Eigensolver)を例にして紹介されています。

ご参考 : Qiskit Textbook 4.1.2章 「VQE を利用しての分子シミュレーションを行う」

VQE (Variational Quantum Eigensolver) 変分量子固有値ソルバー アルゴリズム概要

1. 解きたい問題のハミルトニアン H を準備する。
2. パラメーター θ の量子回路 $U(\theta)$ を量子コンピューターに適用し、試行状態(ansatz) $|\psi(\theta)\rangle$ を作る。
3. $|\psi(\theta)\rangle$ に H 演算をして測定して、エネルギー期待値 $\langle H(\theta) \rangle$ を求める。
4. 古典コンピューターで θ を調整して 2.、3.を繰り返し、エネルギーが最小になるように最適化する。



ハミルトニアンとは？

物理の世界でエネルギーに相当する演算子をハミルトニアン H と呼びます。

シュレディンガー方程式（例：電子のエネルギーを求める式）

$$\left(-\frac{\hbar^2}{2m} \frac{d^2}{dx^2} + V \right) |\psi\rangle = E|\psi\rangle$$

の左側の演算子（行列）がハミルトニアン H です。

量子状態を $|\psi\rangle$ とすると、

エネルギーの期待値は、 $\langle \psi | H | \psi \rangle$ です。（期待値とは測定値の平均のことでした。）

自然界では、エネルギーが最小の状態で系が安定します。

よって、**エネルギーの期待値 $\langle \psi | H | \psi \rangle$ の最小値**を求めることで、自然をシミュレーションすることができます。

ハミルトニアンとは？

物理の世界：エネルギーに相当する演算子 H

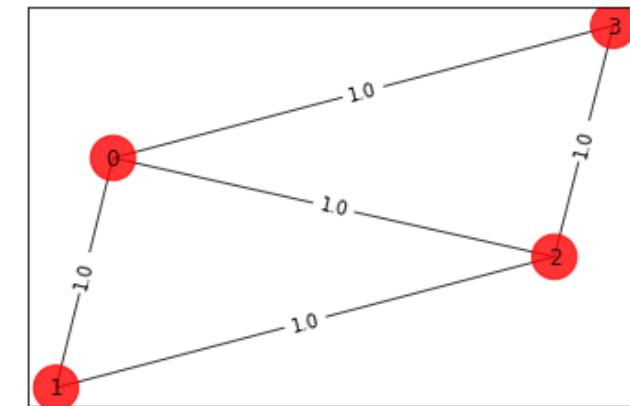
物理の世界以外：最適化問題のコスト関数

最小化したいものをハミルトニアン（コスト関数）で表し、その期待値を最小化問題として扱います。

例) Maxカットのハミルトニアン

$$\begin{aligned} H &= -\frac{1}{2} \sum_{i,j} C_{i,j} (1 - Z_i Z_j) \quad (\text{ここで, } Z_i = \{1, -1\}, C_{i,j} \text{ は重み}) \\ &= \frac{1}{2} (Z_0 Z_1 + Z_1 Z_2 + Z_2 Z_3 + Z_3 Z_1 + Z_0 Z_2) - \frac{5}{2} \end{aligned}$$

: 最小値とそのときの Z_i の値を求める



まとめ：解きたい問題の問題設定部分がハミルトニアン。

ハミルトニアンの期待値を求めたい

ハミルトニアン H の最小のエネルギー期待値（基底状態エネルギー）を求めるることは、
ハミルトニアン行列の最小の固有値を求めるることと同じ。

$$\underset{\text{行列}}{H} \underset{\text{数字}}{|\psi\rangle} = E \underset{\text{数字}}{|\psi\rangle}$$

数学的背景の復習：

行列 A の固有ベクトル $|\psi_i\rangle$ とその固有値 λ_i は、以下の関係にあります。

$$A |\psi_i\rangle = \lambda_i |\psi_i\rangle$$

行列 H がエルミート行列 $H = H^\dagger$ の場合、 H の固有値は実数 ($\lambda_i = \lambda_i^*$)。

実際に実験で測定できる量は実数である必要があるため、
量子系のハミルトニアンを記述するためにはエルミート行列が適切。

\dagger (ダガー) :
随伴行列。転置と要素の複素共役を同時に取る。

例) $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ のとき

$$A^\dagger = \begin{bmatrix} a^* & c^* \\ b^* & d^* \end{bmatrix}$$

ここで
 $a = x + iy$
 $a^* = x - iy$

ハミルトニアンの固有値・固有ベクトルを求めたい

$$H|\psi\rangle = E|\psi\rangle$$

水素分子のハミルトニアンの一例： $H = a_0I + a_1Z_0 + a_2Z_1 + a_3Z_0Z_1 + a_4X_0X_1$

(X, Z などはパウリ演算子)

1) 行列計算で固有値を求める。

水素分子のような小さいものではなく、一般の分子のハミルトニアンの固有値を求めるには、巨大な行列計算が必要。古典コンピューターでは計算能力が足りない。

2) 量子コンピューターで、位相推定アルゴリズムから固有値が求められる。

ただし、まだ精度が足りない。

$$U|\Psi\rangle = \exp(-iHt)|\Psi\rangle = \exp(-iEt)|\Psi\rangle$$

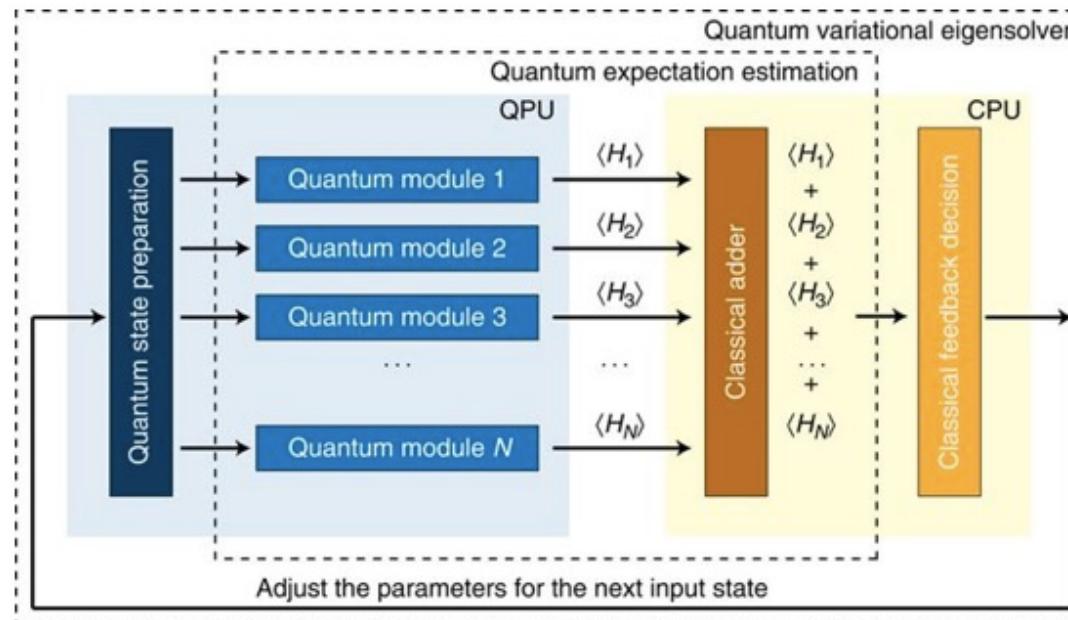
3) 量子コンピューターと古典コンピューターと合わせ技で固有値を求める。 · · · VQE

VQE (Variational Quantum Eigensolver) 変分量子固有値ソルバーとは？

ハミルトニアン H の最小のエネルギー期待値（基底状態エネルギー） λ_{\min} の近似解 λ_θ を求める変分アルゴリズム。

$$\lambda_{\min} \leq \lambda_\theta \equiv \langle \psi(\theta) | H | \psi(\theta) \rangle \quad (\text{ここで、 } |\psi(\theta)\rangle \text{は固有値 } \lambda_\theta \text{に対応する固有状態。})$$

量子コンピューター：
 $\langle \psi(\theta) | H | \psi(\theta) \rangle$ の部分の
設定と個別の測定。



古典コンピューター：
測定結果の合算と θ の調整。

$|\psi(\theta)\rangle$ の最適なパラメータ θ の値を、期待値 $\langle \psi(\theta) | H | \psi(\theta) \rangle$ が最小になるように繰り返し計算しながら求める。

ハミルトニアンの測定

ハミルトニアン H は、エルミート行列（物理量なので固有値は実数）であり、エルミート行列は、パウリ行列と単位行列のテンソル積の線形和で表せます。

例) 水素分子のハミルトニアン : $H = a_0 I + a_1 Z_0 + a_2 Z_1 + a_3 Z_0 Z_1 + a_4 X_0 X_1$

つまり、 $\langle \psi | Z | \psi \rangle$ 、 $\langle \psi | X | \psi \rangle$ 、 $\langle \psi | X_0 X_1 | \psi \rangle$ などを求めれて係数をかけて足し合わせれば、ハミルトニアンの期待値 $\langle \psi | H | \psi \rangle$ が求まります。

物理量Zの期待値

物理量Z（パウリ演算子Z）の期待値、 $\langle \psi | Z | \psi \rangle$ を求めるには、

量子ビット $|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$ のとき、 $|\psi\rangle$ を計算基底で測定する（Z測定する）と、

$Z |0\rangle = |0\rangle$, $Z |1\rangle = -|1\rangle$ より、確率 $|\alpha|^2$ で1、確率 $|\beta|^2$ で-1を得るので、

$$\langle \psi | Z | \psi \rangle = \alpha^* \alpha \langle 0 | Z | 0 \rangle + \beta^* \beta \langle 1 | Z | 1 \rangle = |\alpha|^2 - |\beta|^2 \quad : \text{物理量Zの期待値}$$

同様に

- Xの期待値： $X = HZH$ より、

$\langle \psi | HZH | \psi \rangle = \langle H\psi | Z | H\psi \rangle \quad : H(\text{アダマール})\text{演算を掛けてから}Z\text{測定する。}$

- ZZの期待値

2量子ビット $|\psi\rangle = a_{00} |00\rangle + a_{01} |01\rangle + a_{10} |10\rangle + a_{11} |11\rangle$ とすると

$$\begin{aligned} \langle \psi | (Z \otimes Z) | \psi \rangle &= (\alpha_{00}^* \langle 00 | + \alpha_{01}^* \langle 01 | + \alpha_{10}^* \langle 10 | + \alpha_{11}^* \langle 11 |)(Z \otimes Z) \\ &\quad |(a_{00} |00\rangle + a_{01} |01\rangle + a_{10} |10\rangle + a_{11} |11\rangle) = |\alpha_{00}|^2 + |\alpha_{11}|^2 - |\alpha_{01}|^2 - |\alpha_{10}|^2 \end{aligned}$$

- XXの期待値

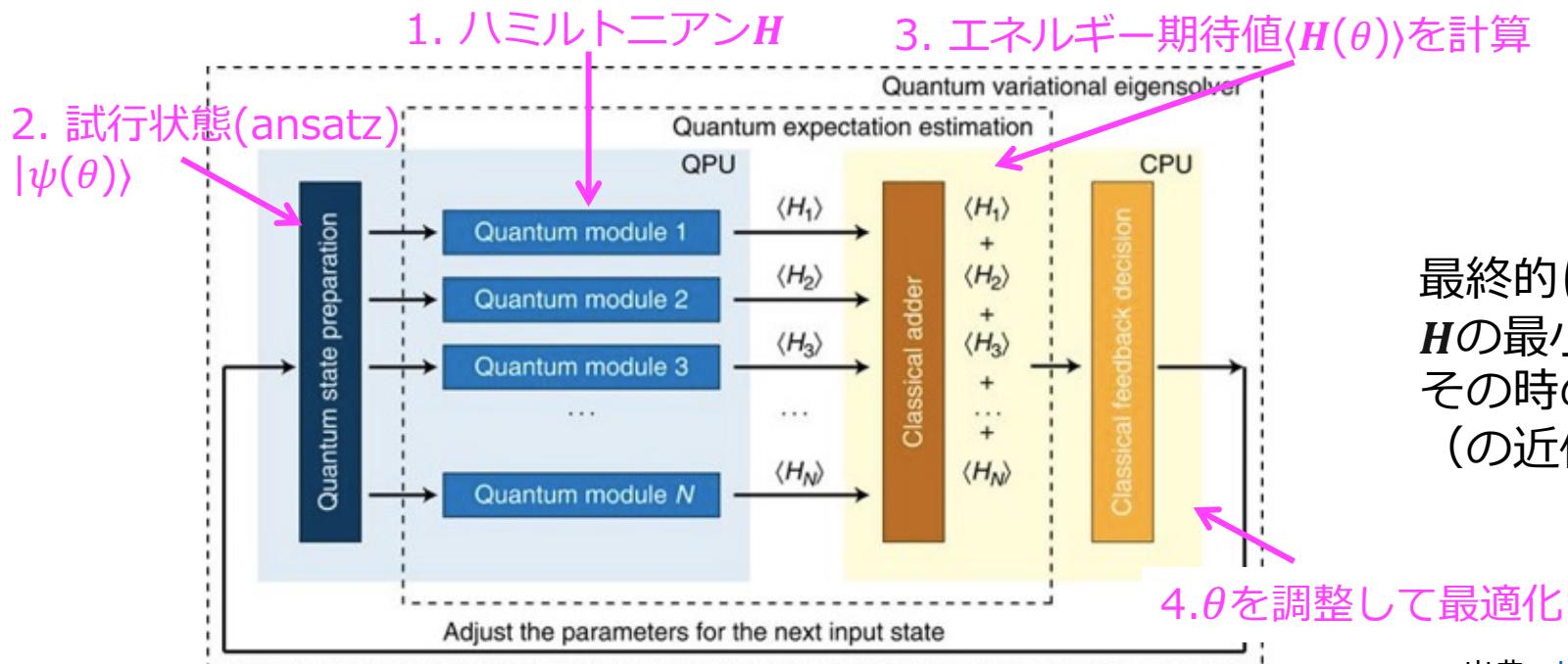
$$\langle \psi | (X \otimes X) | \psi \rangle = \langle \psi | (HZH) \otimes (HZH) | \psi \rangle = \langle \psi | (H \otimes H)(Z \otimes Z)(H \otimes H) | \psi \rangle = \langle (H \otimes H)\psi | Z \otimes Z | (H \otimes H)\psi \rangle$$

: 各量子ビットにH演算を掛けてからZ測定する。

これらより、ハミルトニアンの期待値 $\langle \psi | H | \psi \rangle$ が計算できます。

VQE (Variational Quantum Eigensolver) 変分量子固有値ソルバー アルゴリズム概要

1. 解きたい問題のハミルトニアン H を準備する。
2. パラメーター θ の量子回路 $U(\theta)$ を量子コンピューターに適用し、試行状態(ansatz) $|\psi(\theta)\rangle$ を作る。
3. $|\psi(\theta)\rangle$ に H 演算をして測定して、エネルギー期待値 $\langle H(\theta) \rangle$ を求める。
4. 古典コンピューターで θ を調整して 2.、3.を繰り返し、エネルギーが最小になるように最適化する。



VQE (Variational Quantum Eigensolver) インスタンス

注) AquaがDeprecatedになりTutorialと異なる部分は 新コード と記載しました。

変分形式（試行状態, ansatz, $|\psi(\theta)\rangle$ の部分）は、 QuantumCircuit の形で受け取る（Circuit Libraryから）。古典オプティマイザーなどの構成要素も受け取る。

```
# from qiskit.aqua.algorithms import VQE # deprecated
from qiskit.algorithms import VQE  新コード
# from qiskit.aqua.components.optimizers import SLSQP # deprecated
from qiskit.algorithms.optimizers import SLSQP  新コード
from qiskit.circuit.library import TwoLocal

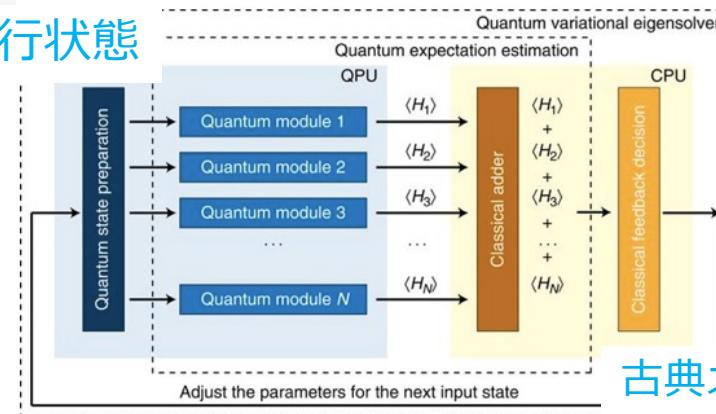
num_qubits = 2
ansatz = TwoLocal(num_qubits, 'ry', 'cz')
opt = SLSQP(maxiter=1000)
# vqe = VQE(var_form=ansatz, optimizer=opt) # deprecated
vqe = VQE(ansatz=ansatz, optimizer=opt)  新コード
```

試行状態 : TwoLocal

新コード

古典オプティマイザー :
SLSQP (Sequential Least Squares (最小二乗) Programming optimizer)
TextbookではCOBYLAを使ってます。

試行状態

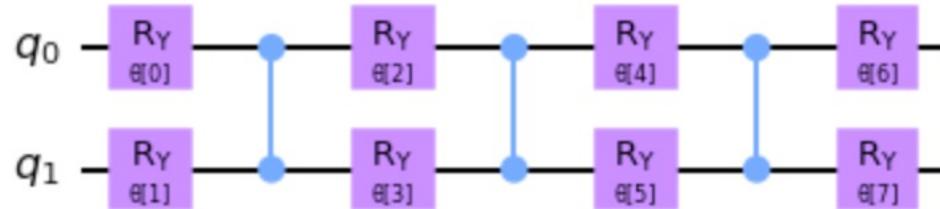


古典オプティマイザー

変分形式（試行状態, ansatz）を表示してみる

[2]: `ansatz.draw()`

[2]:



今回は、 Circuit Library にある TwoLocalを使っている。

この8個のパラメーター θ を変更して状態を変える。

補足：試行状態の作り方は色々あり、量子化学の分野では、対象とするハミルトニアンをもとに量子化学の理論や知識に基づいた手法で求めた量子状態を試行状態として用いることが多い。

チュートリアルの Chemistry(Nature)編-3 「基底状態ソルバー」では、量子化学における標準的な試行状態である、ユニタリー結合クラスター (UCC) を使用。

QuantumInstanceの設定

QuantumInstance で、回路の処理や実行に関する設定をします。
実行のためにバックエンド（シミュレーターや実デバイス）を設定。

```
[4]: # from qiskit.aqua import QuantumInstance # depricated  
from qiskit.utils import QuantumInstance
```

← 新コード

```
backend = BasicAer.get_backend('qasm_simulator')  
quantum_instance = QuantumInstance(backend=backend, shots=800, seed_simulator=99)
```

ショット数、ノイズモデルを使うかどうか、回路のトランスペイル周りのオプションなど

今回は、毎回全く同じ結果を再現したいので乱数のシード値を設定

VQE は、 MinimumEigensolver であるので、 compute_minimum_eigenvalue() メソッドを使います。

ハミルトニアン演算子の設定

```
[5]: # from qiskit.aqua.operators import X, Z, I # deprecated  
from qiskit.opflow import X, Z, I
```

```
H2_op = (-1.052373245772859 * I ^ I) + \  
         (0.39793742484318045 * I ^ Z) + \  
         (-0.39793742484318045 * Z ^ I) + \  
         (-0.01128010425623538 * Z ^ Z) + \  
         (0.18093119978423156 * X ^ X)
```

◀ 新コード

Qiskit Natureによって、原子間距離 0.735Å の水素 (H_2) 分子のハミルトニアンとして作成されたもの。

チュートリアルのChemistry(Nature)編-1「電子構造」に詳細な説明あり。

水素分子のハミルトニアン： $H = a_0I + a_1Z_0 + a_2Z_1 + a_3Z_0Z_1 + a_4X_0X_1$

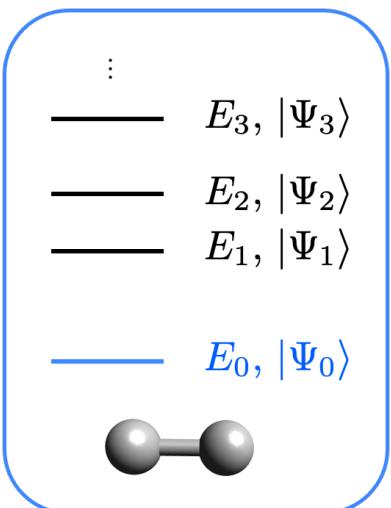
もともとの水素分子のハミルトニアン（電子と原子の運動エネルギーと位置エネルギー）を以下のように変形（近似）させていくと上記のハミルトニアンになる。（かなり大雑把ですみません。）

- 1) 原子核は動かないものとして、電子のみに近似。(Born-Oppenheimer近似)
- 2) 電子1個ずつの運動に近似。(Hartree-Fock法。STO-3G基底。)
- 3) 電子の存在を量子化。（第二量子化：生成消滅演算子に変換。）
- 4) パウリ演算子に変換。(Jordan-Wigner変換)
- 5) 対称性を考慮し、パリティ・マッピングを活用し、演算数を削減。
- 5) 係数を古典コンピューターで積分して計算。

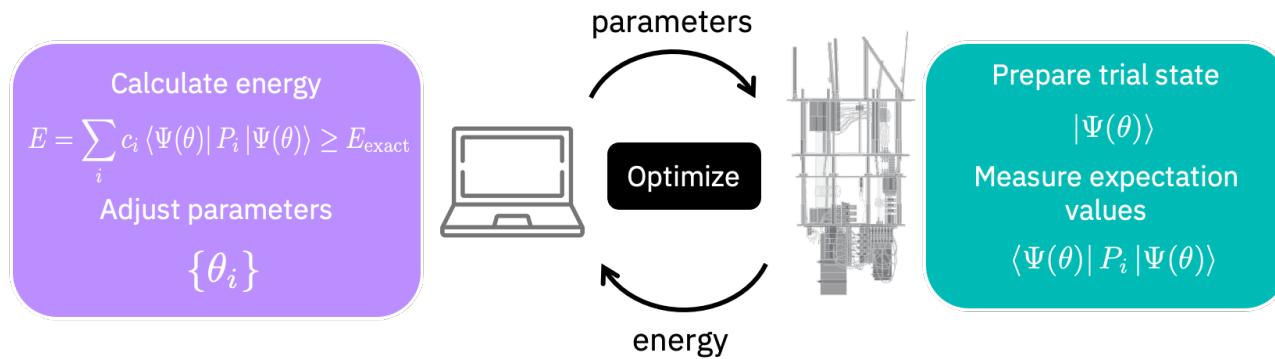
ハミルトニアン演算子の設定

```
[5]: # from qiskit.aqua.operators import X, Z, I # deprecated  
from qiskit.opflow import X, Z, I ← 新コード
```

```
H2_op = (-1.052373245772859 * I ^ I) + \  
         (0.39793742484318045 * I ^ Z) + \  
         (-0.39793742484318045 * Z ^ I) + \  
         (-0.01128010425623538 * Z ^ Z) + \  
         (0.18093119978423156 * X ^ X)
```



チュートリアルのChemistry(Nature)編-3「基底状態ソルバー」より



これまでの設定を合わせてVQEを実行

```
from qiskit import BasicAer
from qiskit.algorithms import VQE ← 新コード
from qiskit.algorithms.optimizers import SLSQP ← 新コード
from qiskit.circuit.library import TwoLocal
from qiskit.utils import QuantumInstance ← 新コード
```

```
from qiskit.opflow import X, Z, I ← 新コード
H2_op = (-1.052373245772859 * I ^ I) + \
         (0.39793742484318045 * I ^ Z) + \
         (-0.39793742484318045 * Z ^ I) + \
         (-0.01128010425623538 * Z ^ Z) + \
         (0.18093119978423156 * X ^ X)
```

```
# from qiskit.aqua import aqua_globals # depricated
```

```
from qiskit.utils import algorithm_globals ← 新コード
```

```
seed = 50
```

```
algorithm_globals.random_seed = seed
```

```
qi = QuantumInstance(BasicAer.get_backend('statevector_simulator'), seed_transpiler=seed, seed_simulator=seed)
```

```
ansatz = TwoLocal(rotation_blocks='ry', entanglement_blocks='cz')
```

```
slsqp = SLSQP(maxiter=1000)
```

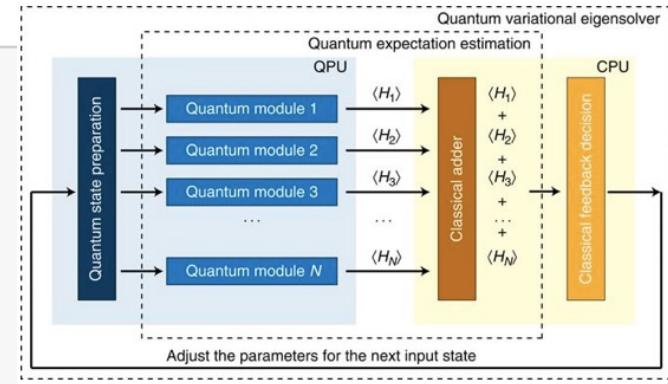
```
# vqe = VQE(operator=H2_op, var_form=ansatz, optimizer=slsqp, quantum_instance=qi) # depricated
```

```
vqe = VQE(ansatz=ansatz, optimizer=slsqp, quantum_instance=qi) ← 新コード
```

```
# result = vqe.run() # depricated
```

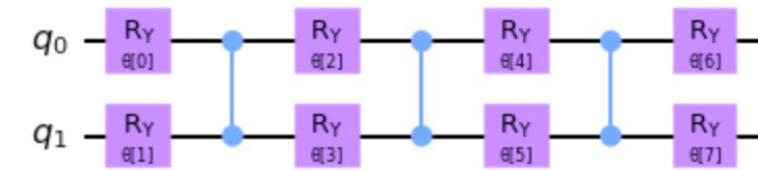
```
result = vqe.compute_minimum_eigenvalue(H2_op) ← 新コード ← 実行
```

```
print(result)
```



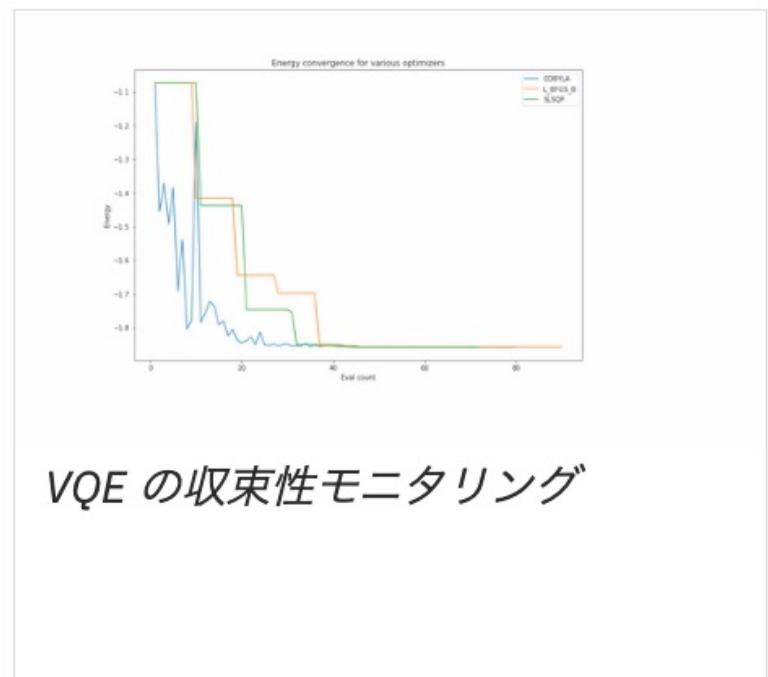
結果

```
{  'aux_operator_eigenvalues': None, 補助演算子の固有値（今回設定なし）
  'cost_function_evals': 65, 繰り返し回数
  'eigenstate': array([ 9.54998859e-05-1.19134723e-16j, -9.93766275e-01-2.10873478e-16j, 固有状態|ψ(θ)⟩
    1.11483549e-01+2.26453286e-17j, 1.77240156e-05+6.62878347e-17j]), 
  'eigenvalue': -1.8572750175659285, 最小固有値（エネルギー）
  'optimal_parameters': {  ParameterVectorElement(θ [7]): 0.36020722970466373,
    最適化されたパラメーター ParameterVectorElement(θ [1]): 4.426962084521693,
    ParameterVectorElement(θ [6]): -4.717618122820399,
    ParameterVectorElement(θ [5]): 1.5683259885445988,
    ParameterVectorElement(θ [0]): 4.296520309146017,
    ParameterVectorElement(θ [4]): -2.5983258651155454,
    ParameterVectorElement(θ [3]): 6.092947776896387,
    ParameterVectorElement(θ [2]): 0.5470753589336304},
  'optimal_point': array([ 4.29652031, 4.42696208, 0.54707536, 6.09294778, -2.59832587,
    1.56832599, -4.71761812, 0.36020723]), 最適化されたパラメーター(0～7の順に上記を並べ替えたもの)
  'optimal_value': -1.8572750175659285, 最小（最適）固有値（エネルギー）
  'optimizer_evals': 65, 最適化された時の繰り返し回数
  'optimizer_time': 0.2173597812652588} かかった時間
```



上記の結果から、与えられた H_2 分子の基底エネルギーである -1.85727 という最小固有値（の近似値）を得ました。最適な値を得るまで、オプティマイザーが 65 回のパラメーター評価をしたことがわかります。

2. VQE の収束性モニタリング



VQE や QAOA といった Qiskit の変分アルゴリズムでは、最適化の進捗状況をモニタリングすることができます。

進捗状況を見るコールバックメソッドは、オプティマイザーによる関数の計算ごとに呼び出され、現在のパラメーターや関数値、反復回数などを返します。
オプティマイザーによっては、これが各反復（ステップ）ごとでない場合があります。

今回は、VQEでの計算過程をモニタリングします。

1. 3種の最適化手法での最適化の過程をプロット
2. 古典的な厳密解との差で最適化の過程をプロット
3. 勾配フレームワークでの最適化の過程をプロット

VQEの計算に必要なライブラリーをインポートします。

```
import numpy as np
import pylab

from qiskit import BasicAer
# from qiskit.aqua.operators import X, Z, I # depricated
from qiskit.opflow import X, Z, I ← 新コード
# from qiskit.aqua import QuantumInstance, aqua_globals # depricated
from qiskit.utils import QuantumInstance, algorithm_globals ← 新コード
# from qiskit.aqua.algorithms import VQE, NumPyMinimumEigensolver # depricated
from qiskit.algorithms import VQE, NumPyMinimumEigensolver ← 新コード
# from qiskit.aqua.components.optimizers import COBYLA, L_BFGS_B, SLSQP # depricated
from qiskit.algorithms.optimizers import COBYLA, L_BFGS_B, SLSQP ← 新コード ← 3種の最適化手法
from qiskit.circuit.library import TwoLocal
```

ハミルトニアン演算子は、先ほどと同じ水素分子を使います。

```
H2_op = (-1.052373245772859 * I ^ I) + \
         (0.39793742484318045 * I ^ Z) + \
         (-0.39793742484318045 * Z ^ I) + \
         (-0.01128010425623538 * Z ^ Z) + \
         (0.18093119978423156 * X ^ X)
```

最適化メソッドを3種使って、最適化の過程を比較

```
optimizers = [COBYLA(maxiter=80), L_BFGS_B(maxiter=60), SLSQP(maxiter=60)] 3種のオプティマイザー
converge_cnts = np.empty([len(optimizers)], dtype=object) 繰り返し回数
converge_vals = np.empty([len(optimizers)], dtype=object) 計算結果

水素のハミルトニアンの最小エネルギーは非常に
容易に見つけることができるので、maxiters (最
大繰り返し回数) を小さな値に設定できます。

for i, optimizer in enumerate(optimizers):
    print('\rOptimizer: {}          '.format(type(optimizer).__name__), end=' ')
    # aqua_globals.random_seed = 50
    algorithm_globals.random_seed = 50 ← 新コード
    var_form = TwoLocal(rotation_blocks='ry', entanglement_blocks='cz')
        变分形式は先ほどと同じTwoLocal
    counts = []
    values = []
    def store_intermediate_result(eval_count, parameters, mean, std): 途中結果を入れる関数定義（4つの変数が必要）
        counts.append(eval_count)
        values.append(mean)

    # vqe = VQE(H2_op, var_form, optimizer, callback=store_intermediate_result, # depricated
    #             # quantum_instance=QuantumInstance(backend=BasicAer.get_backend('statevector_simulator')) # depricated
    vqe = VQE(ansatz=var_form, optimizer=optimizer, callback=store_intermediate_result, ← 新コード
              quantum_instance=QuantumInstance(backend=BasicAer.get_backend('statevector_simulator')))
    result = vqe.compute_minimum_eigenvalue(operator=H2_op)
    converge_cnts[i] = np.asarray(counts)
    converge_vals[i] = np.asarray(values)
print('\rOptimization complete      '');
```

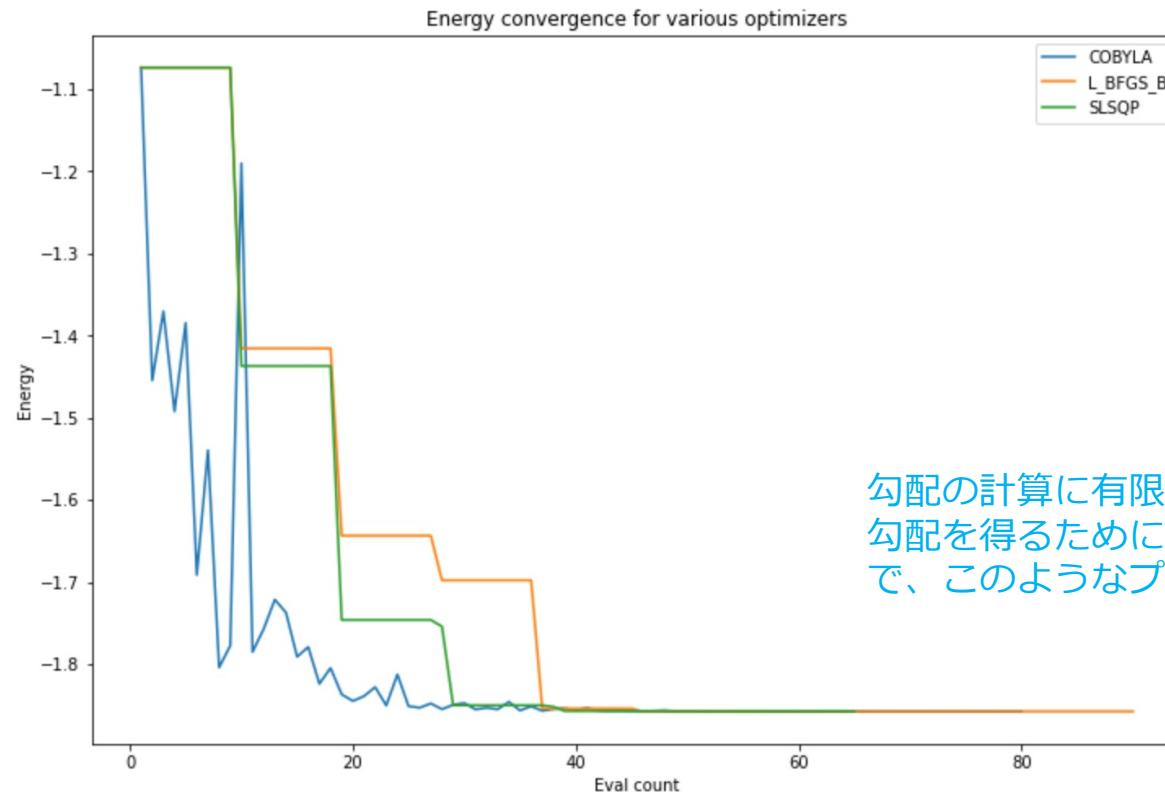
Optimization complete

結果表示：

保存されたコールバック・データから、各オプティマイザーが行う目的関数の呼び出しごとのエネルギー値をプロットします。

```
pylab.rcParams['figure.figsize'] = (12, 8)
for i, optimizer in enumerate(optimizers):
    pylab.plot(converge_cnts[i], converge_vals[i], label=type(optimizer).__name__)
pylab.xlabel('Eval count')
pylab.ylabel('Energy')
pylab.title('Energy convergence for various optimizers')
pylab.legend(loc='upper right');
```

- COBYLA : 線形近似による制約付き最適化
(Qiskit Textbookによるとノイズがない場合おすすめ)
- BFGS : ブロイデン・フレッチャー・ゴールドファーブ・シャンノ法
- SLSQP : シーケンシャル最小二乗プログラミング最適化



厳密解との差分のプロット

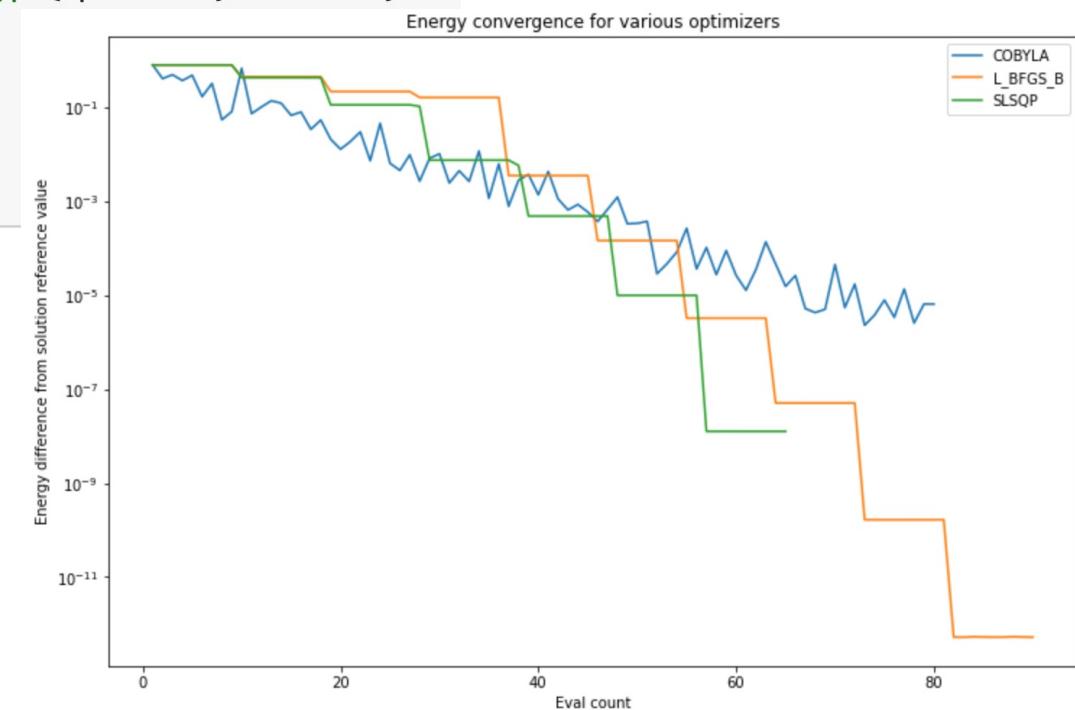
水素分子の問題は古典的な手法で解くことができる、NumPyMinimumEigensolverを用いて厳密解を計算して、厳密解との差をプロットします。

厳密解である最小値へ収束していく過程を見ることができます。

```
npme = NumPyMinimumEigensolver()
result = npme.compute_minimum_eigenvalue(operator=H2_op)
ref_value = result.eigenvalue.real
print(f'Reference value: {ref_value:.5f}')
```

Reference value: -1.85728

```
pylab.rcParams['figure.figsize'] = (12, 8)
for i, optimizer in enumerate(optimizers):
    pylab.plot(converge_cnts[i], abs(ref_value - converge_vals[i]), label=type(optimizer).__name__)
pylab.xlabel('Eval count')
pylab.ylabel('Energy difference from solution reference value')
pylab.title('Energy convergence for various optimizers')
pylab.yscale('log')
pylab.legend(loc='upper right');
```



- COBYLA : 線形近似による制約付き最適化
(Qiskit Textbookによるとノイズがない場合おすすめ)
- BFGS : ブロイデン・フレッチャー・ゴールドファーブ・シャンノ法
- SLSQP : シーケンシャル最小二乗プログラミング最適化

勾配フレームワークでのプロット

Qiskitには、Operator機能の一部としてGradientフレームワークがあります。
オプティマイザー用に計算された勾配によって、最適化ステップそのものを可視化することができます。

```
# from qiskit.aqua.operators.gradients import Gradient # deprecated
from qiskit.opflow.gradients import Gradient ← 新コード
```

```
algorithm_globals.random_seed = 50
var_form = TwoLocal(rotation_blocks='ry', entanglement_blocks='cz')
```

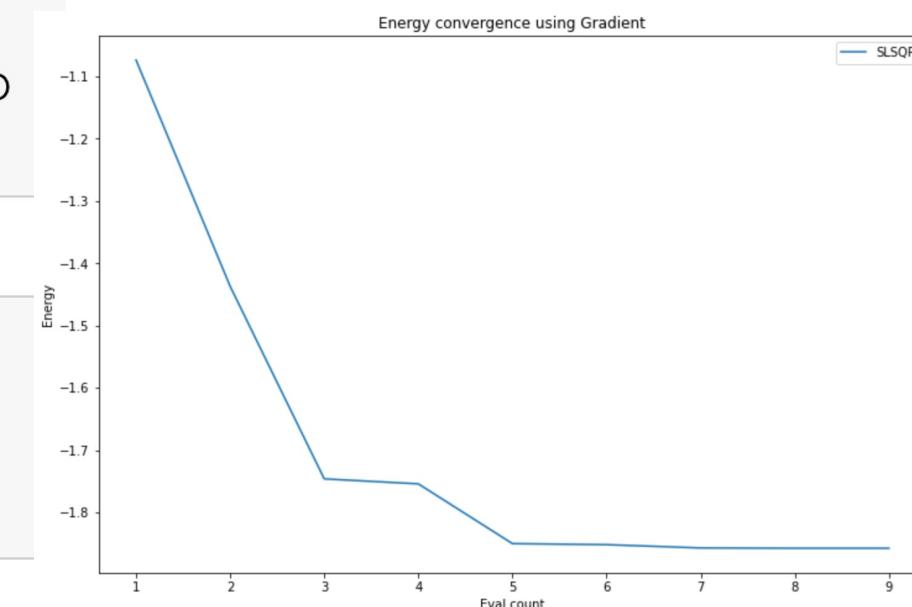
```
optimizer = SLSQP(maxiter=60) OptimizerはSLSQPのとき
```

```
counts = []
values = []
def store_intermediate_result(eval_count, parameters, mean, std):
    counts.append(eval_count)
    values.append(mean)
```

```
vqe = VQE(ansatz=var_form, optimizer=optimizer, callback=store_intermediate_result,
           gradient=Gradient(grad_method='fin_diff'),
           quantum_instance=QuantumInstance(backend=BasicAer.get_backend('statevector_simulator')))
result = vqe.compute_minimum_eigenvalue(operator=H2_op)
print(f'Value using Gradient: {result.eigenvalue.real:.5f}')
```

```
Value using Gradient: -1.85728
```

```
pylab.rcParams['figure.figsize'] = (12, 8)
pylab.plot(counts, values, label=type(optimizer).__name__)
pylab.xlabel('Eval count')
pylab.ylabel('Energy')
pylab.title('Energy convergence using Gradient')
pylab.legend(loc='upper right');
```



3. ノイズのある Aer シミュレーターでのVQE



ノイズのあるAer シミュレーター
でのVQE

Aer シミュレーターでノイズのある状況でVQEを計算

Qasm_simulatorを使って、ノイズ有り無しシミュレーションを行います。

今回は、VQEを例に計算しますが、この手法はQiskit の他の量子アルゴリズムすべてに適用可能です。

VQEの計算に必要なライブラリーをインポートします。

```
import numpy as np
import pylab

from qiskit import Aer
# from qiskit.aqua import QuantumInstance, aqua_globals # depricated
from qiskit.utils import QuantumInstance, algorithm_globals ← 新コード
# from qiskit.aqua.algorithms import VQE, NumPyMinimumEigensolver # depricated
from qiskit.algorithms import VQE, NumPyMinimumEigensolver ← 新コード
# from qiskit.aqua.components.optimizers import SPSA # depricated
from qiskit.algorithms.optimizers import SPSA ← 新コード
from qiskit.circuit.library import TwoLocal
# from qiskit.aqua.operators import I, X, Z # depricated
from qiskit.opflow import X, Z, I ← 新コード
```

ハミルトニアン演算子は、先ほどと同じ水素分子を使います。

```
H2_op = (-1.052373245772859 * I ^ I) + \
(0.39793742484318045 * I ^ Z) + \
(-0.39793742484318045 * Z ^ I) + \
(-0.01128010425623538 * Z ^ Z) + \
(0.18093119978423156 * X ^ X)

print(f'Number of qubits: {H2_op.num_qubits}')
```

Number of qubits: 2

この問題は古典手法によって簡単に解けるので、後で結果を比較できるように、
NumPyMinimumEigensolver で参照値を計算しておきます。

```
npme = NumPyMinimumEigensolver()
result = npme.compute_minimum_eigenvalue(operator=H2_op)
ref_value = result.eigenvalue.real
print(f'Reference value: {ref_value:.5f}')
```

Reference value: -1.85728

ノイズがない場合

変分形式はTwoLocalで、OptimizerはSPSA。
BackendはQasm_simulator。

```
seed = 170
iterations = 125
# aqua_globals.random_seed = seed # depricated
algorithm_globals.random_seed = seed ← 新コード
backend = Aer.get_backend('qasm_simulator')
qi = QuantumInstance(backend=backend, seed_simulator=seed, seed_transpiler=seed)

counts = []
values = []
def store_intermediate_result(eval_count, parameters, mean, std): 途中経過も保存
    counts.append(eval_count)
    values.append(mean)

var_form = TwoLocal(rotation_blocks='ry', entanglement_blocks='cz')
spsa = SPSA(maxiter=iterations) OptimizerはSPSA (同時摂動確率的近似オプティマイザー)
# vqe = VQE(var_form=var_form, optimizer=spsa, callback=store_intermediate_result, quantum_instance=qi) # depricated
vqe = VQE(ansatz=var_form, optimizer=spsa, callback=store_intermediate_result, quantum_instance=qi) ← 新コード
result = vqe.compute_minimum_eigenvalue(operator=H2_op)
print(f'VQE on Aer qasm simulator (no noise): {result.eigenvalue.real:.5f}')
print(f'Delta from reference energy value is {(result.eigenvalue.real - ref_value):.5f}')
```

VQE on Aer qasm simulator (no noise): -1.85332 最小固有値
Delta from reference energy value is 0.00395 厳密解との差

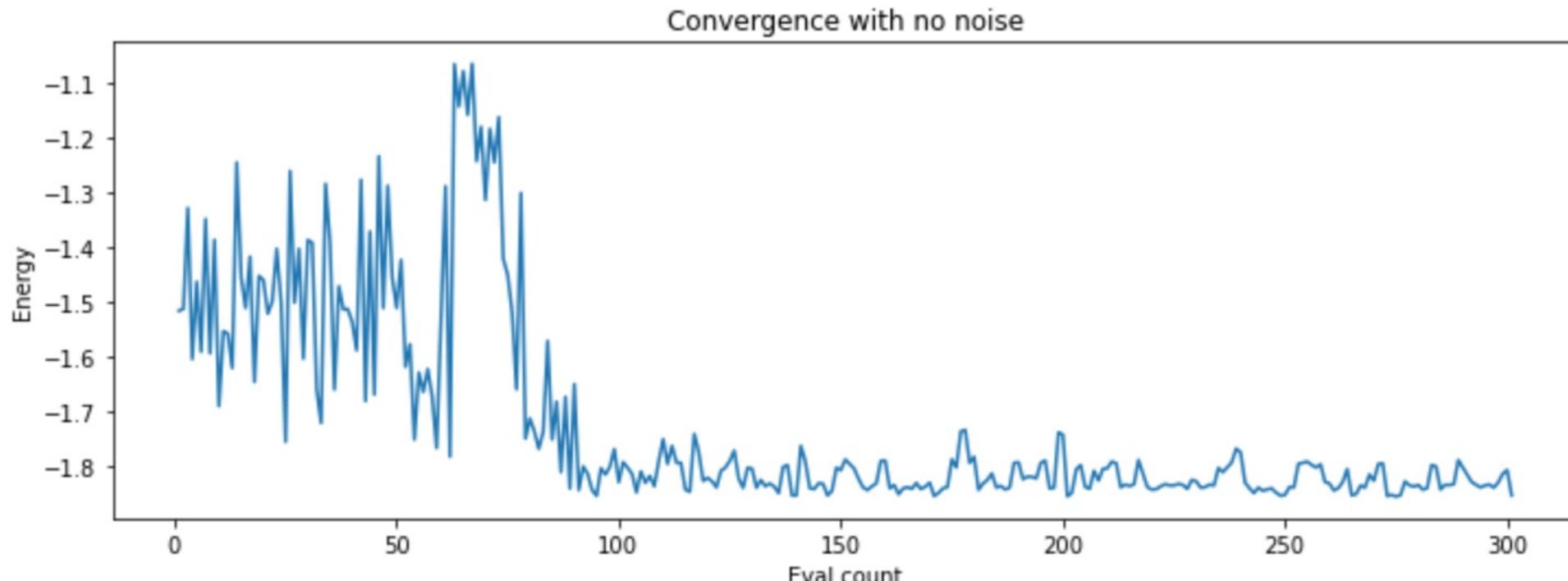
Qiskit TextbookによるとSPSA
は、ノイズのある目的関数を最適化する適切なオプティマイザー。

ノイズがない場合の結果

エネルギーの収束過程

```
: pylab.rcParams['figure.figsize'] = (12, 4)
pylab.plot(counts, values)
pylab.xlabel('Eval count')
pylab.ylabel('Energy')
pylab.title('Convergence with no noise')

: Text(0.5, 1.0, 'Convergence with no noise')
```



ノイズがある場合

実際のデバイスのバックエンドと同じように、結合マップとノイズモデルを設定します。

(デフォルトの全結合マップのままにしておく設定もできます。)

応用編として、変分形式のエンタングルメント・マップとして、この結合マップを用いることもできます。

注: ノイズを伴うシミュレーションは、ノイズなしより計算時間がかかります。

チュートリアルでは、**ibmq_vigo**デバイスのノイズモデルを使っていましたが、このデバイスが引退していたので **ibmq_belem**デバイスのノイズモデルにしました。

ibmq_belem

System status

● Online

Processor type

Falcon r4

5 Qubits

16 Quantum volume

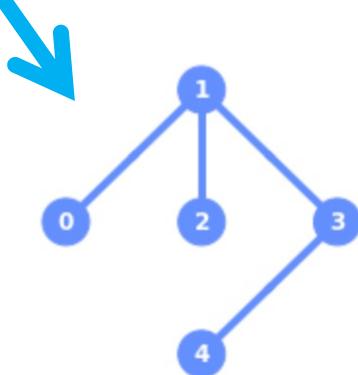


```
from qiskit.tools.jupyter import *
from qiskit import IBMQ
provider = IBMQ.load_account()
```

```
backend = provider.get_backend('ibmq_belem')
backend
```

実行結果は次ページ。

ibmq_belem

Configuration	Qubit Properties	Multi-Qubit Gates	Error Map	Job History	
Property	Value	結合マップ			
n_qubits	5				
quantum_volume	16				
operational	True				
status_msg	active				
pending_jobs	30				
backend_version	1.0.8				
基礎ゲート		sx ゲート (2回かけるとXゲート)			
basis_gates	['id', 'rz', 'sx', 'x', 'cx', 'reset']	$\sqrt{X} = \frac{1}{2} \begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix}$			
max_shots	8192				
max_experiments	75				
qubit_lo_range		[[4.590211543324174e+18, 5.590211543324174e+18], [4.74528082521587e+18], [4.860977555741272e+18, 5.860977			

ibmq_belem

Configuration	Qubit Properties	Multi-Qubit Gates	Error Map	Job History		
last_update_date: Thu 29 April 2021 at 14:20 JST						
Frequency	T1	T2	RZ	SX	X	Readout error
Q0 GHz	107.26654 us	142.96676 us	0	0.00019	0.00019	0.0353
Q1 GHz	97.65472 us	103.19902 us	0	0.0003	0.0003	0.0187
Q2 GHz	68.28878 us	92.99157 us	0	0.00024	0.00024	0.0205
Q3 GHz	116.22069 us	143.09591 us	0	0.00036	0.00036	0.0139
Q4 GHz	82.14631 us	126.13369 us	0	0.00028	0.00028	0.0199

頻繁に更正しているので値はその時々で変わります

ibmq_belem



Configuration Qubit Properties **Multi-Qubit Gates** Error Map Job History

last_update_date: 2021-04-29 14:20:34+09:00

Type	Gate error	Type	Gate error	Type	Gate error
------	------------	------	------------	------	------------

cx4_3	cx	0.01116	cx1_3	cx	0.00982
-------	----	---------	-------	----	---------

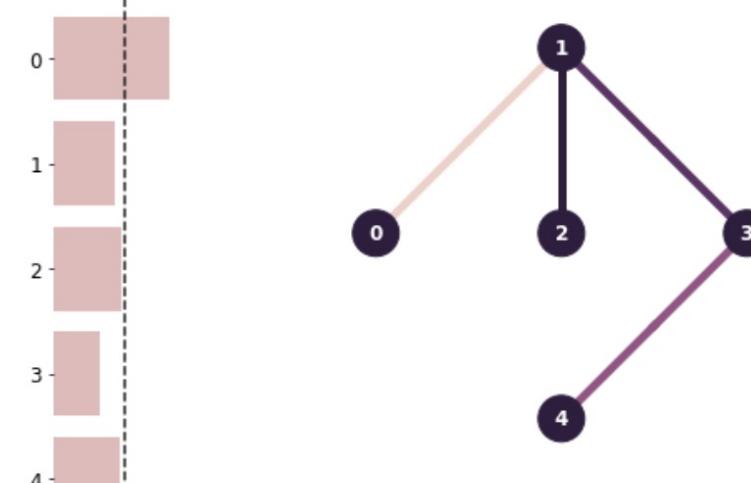
cx3_4	cx	0.01116	cx2_1	cx	0.00873
-------	----	---------	-------	----	---------

cx3_1	cx	0.00982	cx1_2	cx	0.00873
-------	----	---------	-------	----	---------

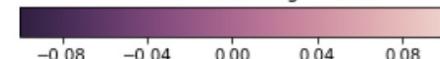
ibmq_belem

Configuration Qubit Properties Multi-Qubit Gates **Error Map** Job History

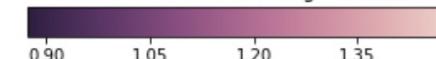
Readout Error (%)



H error rate (%) [Avg. = 0.0]



CNOT error rate (%) [Avg. = 1.115]



ノイズがある場合

実デバイスを模したノイズモデル「FakeBelem」

```
import os
from qiskit.providers.aer import QasmSimulator
from qiskit.providers.aer.noise import NoiseModel
from qiskit.test.mock import FakeBelem
device_backend = FakeBelem()

backend = Aer.get_backend('qasm_simulator')
counts1 = []
values1 = []
noise_model = None
os.environ['QISKIT_IN_PARALLEL'] = 'TRUE'
device = QasmSimulator.from_backend(device_backend)
coupling_map = device.configuration().coupling_map
noise_model = NoiseModel.from_backend(device)
basis_gates = noise_model.basis_gates

print(noise_model)
print()

# aqua_globals.random_seed = seed # depricated
algorithm_globals.random_seed = seed ← 新コード
qi = QuantumInstance(backend=backend, seed_simulator=seed, seed_transpiler=seed,
                     coupling_map=coupling_map, noise_model=noise_model,) 結合マップとノイズモデルをセット

def store_intermediate_result1(eval_count, parameters, mean, std):
    counts1.append(eval_count)
    values1.append(mean)

var_form = TwoLocal(rotation_blocks='ry', entanglement_blocks='cz')
spsa = SPSA(maxiter=iterations)
# vqe = VQE(var_form=var_form, optimizer=spsa, callback=store_intermediate_result1, quantum_instance=qi) # depricated
vqe = VQE(ansatz=var_form, optimizer=spsa, callback=store_intermediate_result1, quantum_instance=qi) ← 新コード
result1 = vqe.compute_minimum_eigenvalue(operator=H2_op)
print(f'VQE on Aer qasm simulator (with noise): {result1.eigenvalue.real:.5f}')
print(f'Delta from reference energy value is {(result1.eigenvalue.real - ref_value):.5f}')
```

ノイズがある場合の結果

NoiseModel: ノイズモデルの特性

Basis gates: ['cx', 'id', 'kraus', 'reset', 'roerror', 'rz', 'save_amplitudes', 'save_amplitudes_sq', 'save_density_matrix', 'save_expval', 'save_expval_var', 'save_probabilities', 'save_probabilities_dict', 'save_stabilizer', 'save_state', 'save_statevector', 'save_statevector_dict', 'set_density_matrix', 'set_stabilizer', 'set_statevector', 'snapshot', 'sx', 'x'] 基本ゲート一覧

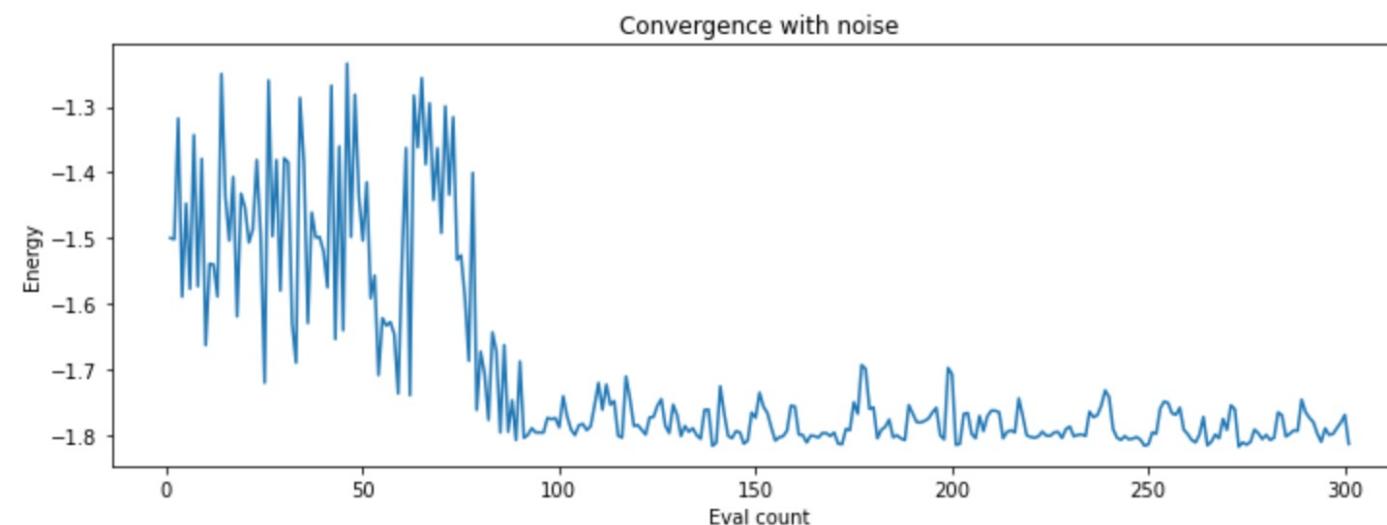
Instructions with noise: ['reset', 'x', 'cx', 'measure', 'sx', 'id'] ノイズの種類

Qubits with noise: [0, 1, 2, 3, 4] 量子ビット一覧

Specific qubit errors: [('id', [0]), ('id', [1]), ('id', [2]), ('id', [3]), ('id', [4]), ('sx', [0]), ('sx', [1]), ('sx', [2]), ('sx', [3]), ('sx', [4]), ('x', [0]), ('x', [1]), ('x', [2]), ('x', [3]), ('x', [4]), ('cx', [4, 3]), ('cx', [3, 4]), ('cx', [3, 1]), ('cx', [1, 3]), ('cx', [2, 1]), ('cx', [1, 2]), ('cx', [1, 0]), ('cx', [0, 1]), ('reset', [0]), ('reset', [1]), ('reset', [2]), ('reset', [3]), ('reset', [4]), ('measure', [0]), ('measure', [1]), ('measure', [2]), ('measure', [3]), ('measure', [4])] 量子ビットごとのエラー一覧

VQE on Aer qasm simulator (with noise): -1.81342 結果：最小固有値

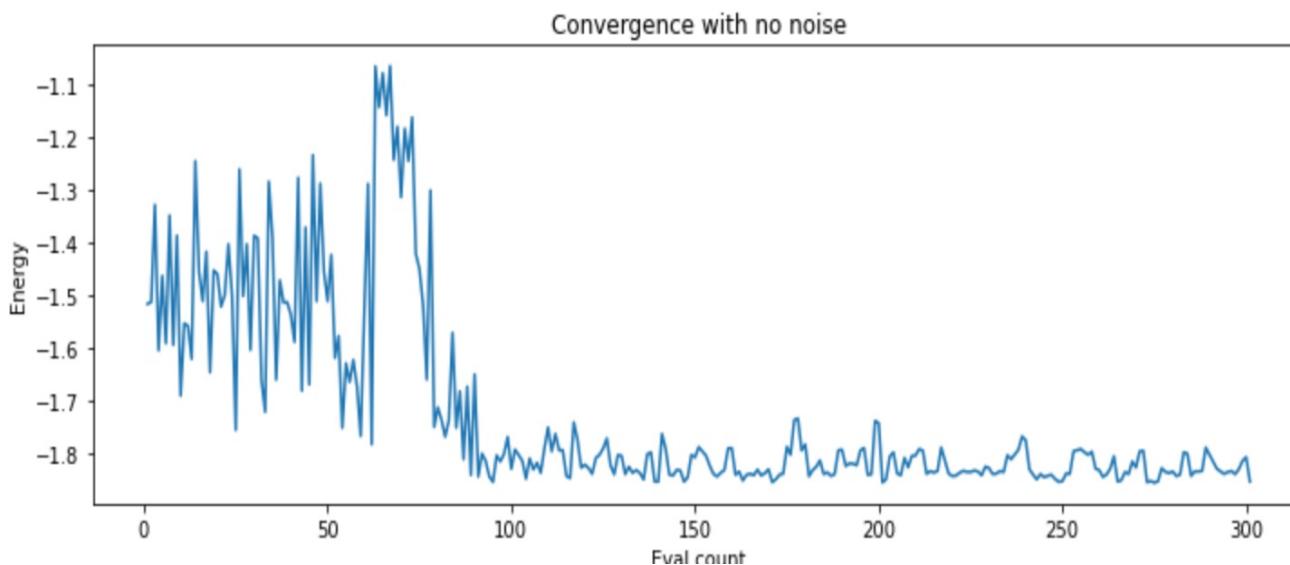
Delta from reference energy value is 0.04386 結果：厳密解との差



ノイズ有無の比較

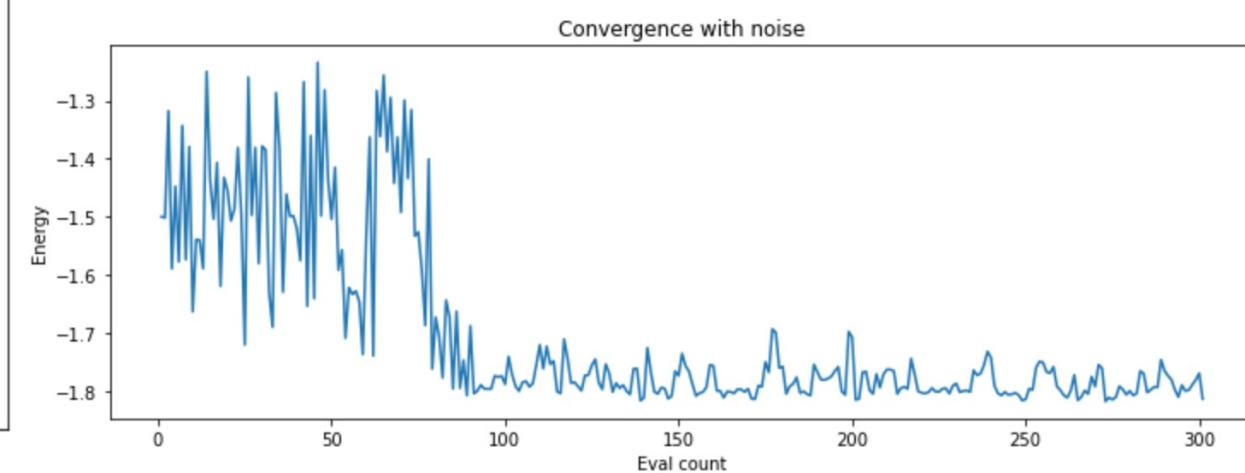
ノイズなし

VQE on Aer qasm simulator (no noise): -1.85332 **最小固有値**
Delta from reference energy value is **0.00395** **厳密解との差**



ノイズあり

VQE on Aer qasm simulator (with noise): -1.81342 **最小固有値**
Delta from reference energy value is **0.04386** **厳密解との差**



分かりにくいですが、ノイズ有りの方が一桁精度が悪い

ノイズありで測定ノイズを軽減させた場合

測定誤差を軽減するために CompleteMeasFitter を選択します。
較正行列は 30 分ごとに自動更新されるようにします（デフォルトが30分）。

```
from qiskit.ignis.mitigation.measurement import CompleteMeasFitter

counts2 = []
values2 = []
if noise_model is not None:
    # aqua_globals.random_seed = seed # deprecated
    algorithm_globals.random_seed = seed
    qi = QuantumInstance(backend=backend, seed_simulator=seed, seed_transpiler=seed,
                         coupling_map=coupling_map, noise_model=noise_model,
                         measurement_error_mitigation_cls=CompleteMeasFitter, 測定ノイズを軽減
                         cals_matrix_refresh_period=30)

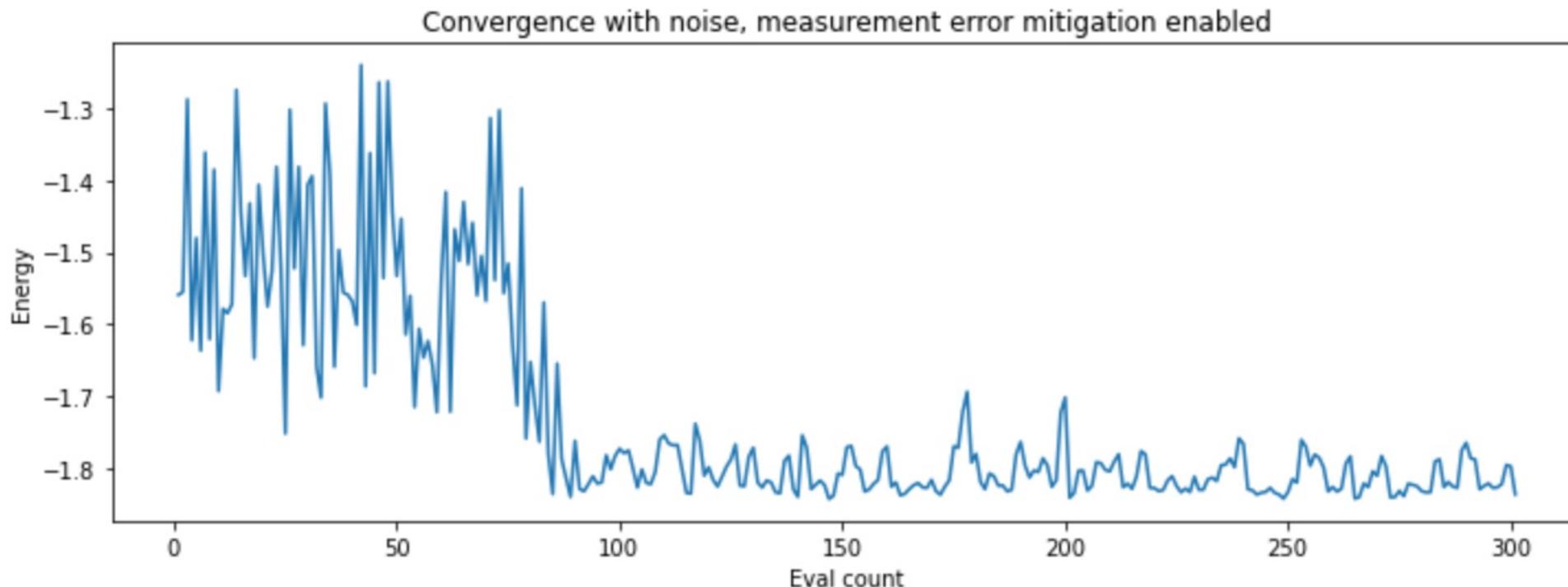
def store_intermediate_result2(eval_count, parameters, mean, std):
    counts2.append(eval_count)
    values2.append(mean)

var_form = TwoLocal(rotation_blocks='ry', entanglement_blocks='cz')
spsa = SPSA(maxiter=iterations)
# vqe = VQE(var_form=var_form, optimizer=spsa, callback=store_intermediate_result2, quantum_instance=qi)
vqe = VQE(ansatz=var_form, optimizer=spsa, callback=store_intermediate_result2, quantum_instance=qi)
result2 = vqe.compute_minimum_eigenvalue(operator=H2_op)
print(f'VQE on Aer qasm simulator (with noise and measurement error mitigation): {result2.eigenvalue.real:.5f}')
print(f'Delta from reference energy value is {(result2.eigenvalue.real - ref_value):.5f}')
```

ノイズありで測定ノイズを軽減させた場合

VQE on Aer qasm simulator (with noise and measurement error mitigation): -1.83574 最小固有値
Delta from reference energy value is 0.02154 厳密解との差

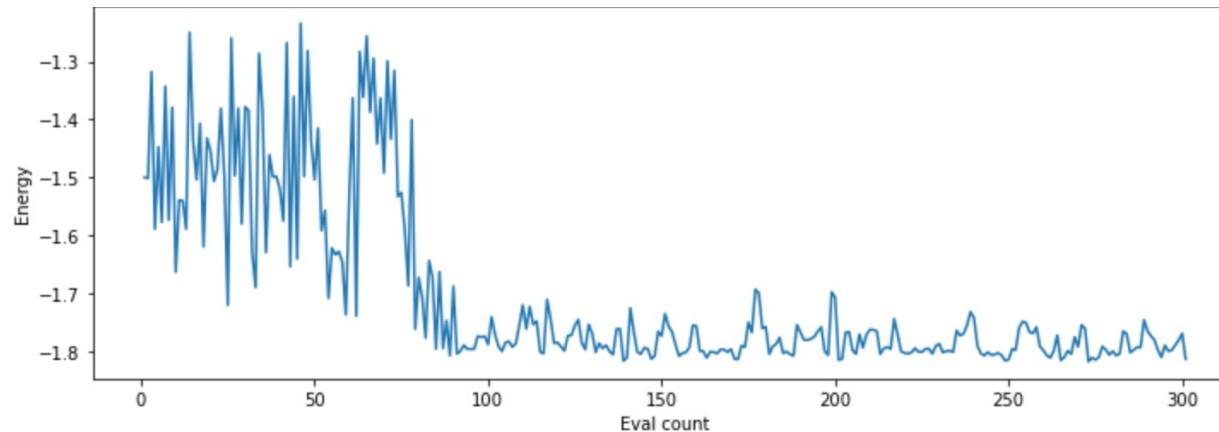
```
if counts2 or values2:  
    pylab.rcParams['figure.figsize'] = (12, 4)  
    pylab.plot(counts2, values2)  
    pylab.xlabel('Eval count')  
    pylab.ylabel('Energy')  
    pylab.title('Convergence with noise, measurement error mitigation enabled')
```



ノイズありとノイズありで測定ノイズを軽減させた場合の比較

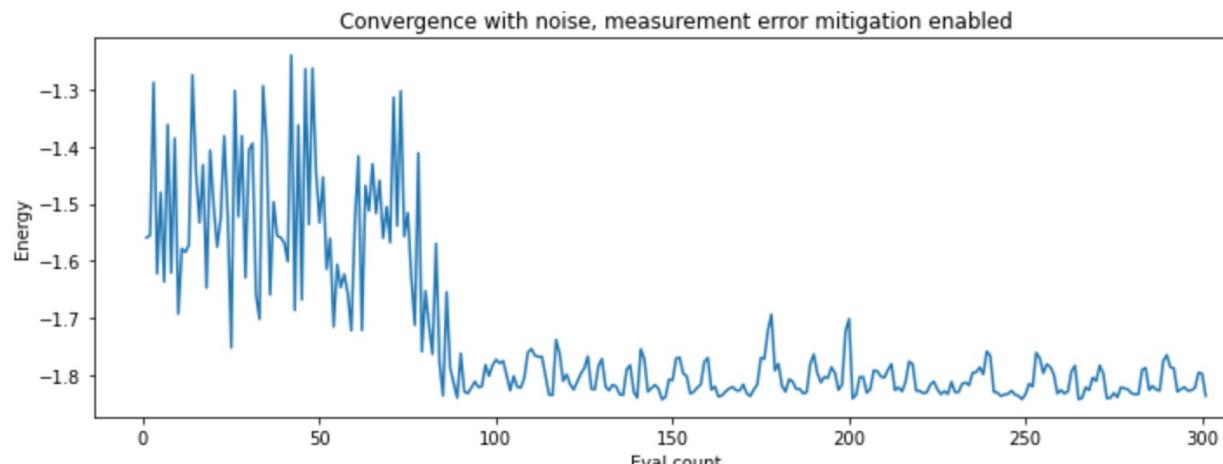
VQE on Aer qasm simulator (with noise): -1.81342

Delta from reference energy value is 0.04386



VQE on Aer qasm simulator (with noise and measurement error mitigation): -1.83574

Delta from reference energy value is 0.02154



ノイズのありシミュレーターでのVQE まとめ

1. 古典計算で参考値を求めました。
 2. Qasm_simulator で VQE を実行。
 - ・ シミュレーションは、理想的(ノイズがない)な計算だが、サンプリングに起因するショットノイズがある。
 - ・ ショット数を増やすとより誤差は減少する。今回の実験ではショット数は 1024 のままなので、結果に少し影響があり。
-
1. 実際のデバイスを模したノイズモデルでノイズを加え、結果が影響を受けたことを確認。
 2. ノイズモデルに、測定ノイズ緩和を加えた結果を確認。

```
print(f'Reference value: {ref_value:.5f}')
print(f'VQE on Aer qasm simulator (no noise): {result.eigenvalue.real:.5f}')
print(f'VQE on Aer qasm simulator (with noise): {result1.eigenvalue.real:.5f}')
print(f'VQE on Aer qasm simulator (with noise and measurement error mitigation): {result2.eigenvalue.real:.5f}')
```

```
Reference value: -1.85728
VQE on Aer qasm simulator (no noise): -1.85332
VQE on Aer qasm simulator (with noise): -1.81342
VQE on Aer qasm simulator (with noise and measurement error mitigation): -1.83574
```

4. VQE の高度な使い方



VQE の高度な使い方

4. VQE の高度な使い方

- 初期パラメーター (initial_point) の設定
- 期待値 (Expectations) オブジェクトの設定
- 勾配 (Gradient)
 - : チュートリアル Operator編-2 「Qiskit Gradientフレームワーク」
- Quantum Instance と高度なシミュレーション
 - : チュートリアル シミュレーター編-7 「行列積状態を用いたシミュレーション」

今回も同じ水素のハミルトニアンを例に使います。

```
H2_op = (-1.052373245772859 * I ^ I) + \
(0.39793742484318045 * I ^ Z) + \
(-0.39793742484318045 * Z ^ I) + \
(-0.01128010425623538 * Z ^ Z) + \
(0.18093119978423156 * X ^ X)
```

初期パラメーター(Initial point)の設定

本日の最初のチュートリアルの例を使って、最小固有値・固有ベクトルを求め、それを次に初期パラメーターとして使ってみます。

```
# from qiskit.aqua import aqua_globals
seed = 50
# aqua_globals.random_seed = seed
algorithm_globals.random_seed = seed
qi = QuantumInstance(BasicAer.get_backend('statevector_simulator'), seed_transpiler=seed, seed_simulator=seed)

ansatz = TwoLocal(rotation_blocks='ry', entanglement_blocks='cz')
slsqp = SLSQP(maxiter=1000)
# vqe = VQE(operator=H2_op, var_form=ansatz, optimizer=slsqp, quantum_instance=qi)
vqe = VQE(ansatz=ansatz, optimizer=slsqp, quantum_instance=qi)
# result = vqe.run()
result = vqe.compute_minimum_eigenvalue(operator=H2_op)

print(result)
```

{ 'aux_operator_eigenvalues': None,
'cost_function_evals': 65,
'eigenstate': array([9.54998859e-05-1.19134723e-16j, -9.93766275e-01-2.10873478e-16j,
1.11483549e-01+2.26453286e-17j, 1.77240156e-05+6.62878347e-17j]),
'eigenvalue': -1.8572750175659285,
'optimal_parameters': { ParameterVectorElement(θ [4]): -2.5983258651155454,
ParameterVectorElement(θ [3]): 6.092947776896387,
ParameterVectorElement(θ [0]): 4.296520309146017,
ParameterVectorElement(θ [7]): 0.36020722970466373,
ParameterVectorElement(θ [6]): -4.717618122820399,
ParameterVectorElement(θ [5]): 1.5683259885445988,
ParameterVectorElement(θ [2]): 0.5470753589336304,
ParameterVectorElement(θ [1]): 4.426962084521693},
'optimal_point': array([4.29652031, 4.42696208, 0.54707536, 6.09294778, -2.59832587,
1.56832599, -4.71761812, 0.36020723]),
'optimal_value': -1.8572750175659285,
'optimizer_evals': 65,
'optimizer_time': 0.2021770477294922}

optimal_point を次の計算の initial_point
として用います。

```

initial_pt = result.optimal_point    先ほどの計算のoptimal_point を今回の計算の
                                    initial_point として使います。
# aqua_globals.random_seed = seed
algorithm_globals.random_seed = seed
qi = QuantumInstance(BasicAer.get_backend('statevector_simulator'), seed_transpiler=seed, seed_simulator=seed)

ansatz = TwoLocal(rotation_blocks='ry', entanglement_blocks='cz')
slsqp = SLSQP(maxiter=1000)
# vqe = VQE(operator=H2_op, var_form=ansatz, optimizer=slsqp, initial_point=initial_pt, quantum_instance=qi)
vqe = VQE(ansatz=ansatz, optimizer=slsqp, initial_point=initial_pt, quantum_instance=qi)
# result1 = vqe.run()
result1 = vqe.compute_minimum_eigenvalue(operator=H2_op)

print(result1)

```

```

{
    'aux_operator_eigenvalues': None,
    'cost_function_evals': 9,
    'eigenstate': array([ 9.54998859e-05-1.19134723e-16j, -9.93766275e-01-2.10873478e-16j,
        1.11483549e-01+2.26453286e-17j,  1.77240156e-05+6.62878347e-17j]),
    'eigenvalue': -1.8572750175659285,
    'optimal_parameters': { ParameterVectorElement( $\theta$  [2]): 0.5470753589336304,
                            ParameterVectorElement( $\theta$  [3]): 6.092947776896387,
                            ParameterVectorElement( $\theta$  [0]): 4.296520309146017,
                            ParameterVectorElement( $\theta$  [1]): 4.426962084521693,
                            ParameterVectorElement( $\theta$  [4]): -2.5983258651155454,
                            ParameterVectorElement( $\theta$  [5]): 1.5683259885445988,
                            ParameterVectorElement( $\theta$  [7]): 0.36020722970466373,
                            ParameterVectorElement( $\theta$  [6]): -4.717618122820399},
    'optimal_point': array([ 4.29652031,  4.42696208,  0.54707536,  6.09294778, -2.59832587,
        1.56832599, -4.71761812,  0.36020723]),
    'optimal_value': -1.8572750175659285,
    'optimizer_evals': 9,
    'optimizer_time': 0.03960299491882324}

```

initial_point を与えた場合は、繰り返し回数が9となっていて、結果がより早く得られているのがわかります。

初期パラメーター(Initial point) を使うポイント

- 問題にとって合理的な出発点が推測される場合や、過去の実験から得られた情報がある場合は使った方が良い。
- 似た問題の初期値としても使えます。
- 例) 化学の問題で、分子の原子間距離を変えて、解離プロファイルをプロットする場合、距離の変化が小さい場合には、解がその前の解の近くにあることが予測されます。

期待値 (Expectations) オブジェクトの設定

```
from qiskit import Aer

# aqua_globals.random_seed = seed
algorithm_globals.random_seed = seed
qi = QuantumInstance(Aer.get_backend('qasm_simulator'), seed_transpiler=seed, seed_simulator=seed)

ansatz = TwoLocal(rotation_blocks='ry', entanglement_blocks='cz')
slsqp = SLSQP(maxiter=1000)
# vqe = VQE(operator=H2_op, var_form=ansatz, optimizer=slsqp, quantum_instance=qi, include_custom=True)
vqe = VQE(ansatz=ansatz, optimizer=slsqp, quantum_instance=qi, include_custom=True)
# result = vqe.run()
result = vqe.compute_minimum_eigenvalue(operator=H2_op)

print(result)
```

qasm_simulatorでinclude_custom=Trueとすると
スナップショット命令を使ったAerPauliExpectation が返され
statevector_simulatorと同じように理想的な計算結果になります。
(qasm_simulator はサンプリング・ノイズがある。)

statevector_simulator を使うよりも
実行がより速くなる可能性があります。

```
{   'aux_operator_eigenvalues': None,
    'cost_function_evals': 65,
    'eigenstate': {'01': 0.9921567416492215, '10': 0.125},
    'eigenvalue': -1.8572750175807682,
    'optimal_parameters': {   ParameterVectorElement(θ [7]): 0.3602071559531031,
                             ParameterVectorElement(θ [1]): 4.426962159645716,
                             ParameterVectorElement(θ [4]): -2.598325866938012,
                             ParameterVectorElement(θ [2]): 0.5470752986013949,
                             ParameterVectorElement(θ [0]): 4.296520300933687,
                             ParameterVectorElement(θ [6]): -4.717618259450455,
                             ParameterVectorElement(θ [3]): 6.092947713510392,
                             ParameterVectorElement(θ [5]): 1.5683258132970883},
    'optimal_point': array([ 4.2965203 ,  4.42696216,  0.5470753 ,  6.09294771, -2.59832587,
                           1.56832581, -4.71761826,  0.36020716]),
    'optimal_value': -1.8572750175807682,
    'optimizer_evals': 65,
    'optimizer_time': 0.23534297943115234}
```

qasm_simulator で (include_customがデフォルトのFalseのまま) SLSQPオプティマイザーの場合

```
# aqua_globals.random_seed = seed
algorithm_globals.random_seed = seed
qi = QuantumInstance(Aer.get_backend('qasm_simulator'), seed_transpiler=seed, seed_simulator=seed)

ansatz = TwoLocal(rotation_blocks='ry', entanglement_blocks='cz')
slsqp = SLSQP(maxiter=1000)
# vqe = VQE(operator=H2_op, var_form=ansatz, optimizer=slsqp, quantum_instance=qi)
vqe = VQE(ansatz=ansatz, optimizer=slsqp, quantum_instance=qi)
# result = vqe.run()
result = vqe.compute_minimum_eigenvalue(operator=H2_op)

print(result)
```

{ 'aux_operator_eigenvalues': None,
 'cost_function_evals': 9,
 'eigenstate': { '00': 0.7781187248742958,
 '01': 0.4881406047441659,
 '10': 0.39404750665370286,
 '11': 0.03125},
 'eigenvalue': -1.0741921795698932,
 'optimal_parameters': { ParameterVectorElement(θ [7]): 0.6984088030463615,
 ParameterVectorElement(θ [0]): 3.611860069224077,
 ParameterVectorElement(θ [5]): 1.8462931831829383,
 ParameterVectorElement(θ [6]): -5.466043598406607,
 ParameterVectorElement(θ [2]): 0.6019852007557844,
 ParameterVectorElement(θ [4]): -3.3070470445355764,
 ParameterVectorElement(θ [3]): 5.949536809130025,
 ParameterVectorElement(θ [1]): 4.19301252102391},
 'optimal_point': array([3.61186007, 4.19301252, 0.6019852 , 5.94953681, -3.30704704,
 1.84629318, -5.4660436 , 0.6984088]),
 'optimal_value': -1.0741921795698932,
 'optimizer_evals': 9,
 'optimizer_time': 0.07676887512207031}

SLSQP オプティマイザーでは、ショット・ノイズがあるため、最適化は突然終了。
最適値も正解の -1.857 に対して -1.074 とかなり異なる。

(SLSQP オプティマイザーは、Qiskit Textbookによるとノイズなしの場合に有効)

qasm_simulator で (include_customがデフォルトのFalseのまま) SPSAオプティマイザーの場合

```
# from qiskit.aqua.components.optimizers import SPSA
from qiskit.algorithms.optimizers import SPSA

# aqua_globals.random_seed = seed
algorithm_globals.random_seed = seed
qi = QuantumInstance(Aer.get_backend('qasm_simulator'), seed_transpiler=seed, seed_simulator=seed)

ansatz = TwoLocal(rotation_blocks='ry', entanglement_blocks='cz')
spsa = SPSA(maxiter=100)
# vqe = VQE(operator=H2_op, var_form=ansatz, optimizer=spsa, quantum_instance=qi)
vqe = VQE(ansatz=ansatz, optimizer=spsa, quantum_instance=qi)
# result = vqe.run()
result = vqe.compute_minimum_eigenvalue(operator=H2_op)

print(result)
```

```
{   'aux_operator_eigenvalues': None,
    'cost_function_evals': 200,
    'eigenstate': {'01': 0.9916644782889019, '10': 0.1288470508005519},
    'eigenvalue': -1.8627691667457413,
    'optimal_parameters': {   ParameterVectorElement(θ [0]): 3.7024534584167186,
                             ParameterVectorElement(θ [4]): -2.365851315862917,
                             ParameterVectorElement(θ [1]): 3.4131129686241786,
                             ParameterVectorElement(θ [3]): 6.659266627719731,
                             ParameterVectorElement(θ [5]): 2.909946635659971,
                             ParameterVectorElement(θ [2]): -0.27407648179456484,
                             ParameterVectorElement(θ [6]): -4.750881193409766,
                             ParameterVectorElement(θ [7]): 0.1703383787015398},
    'optimal_point': array([ 3.70245346,  3.41311297, -0.27407648,  6.65926663, -2.36585132,
                           2.90994664, -4.75088119,  0.17033838]),
    'optimal_value': -1.8627691667457413,
    'optimizer_evals': 200,
    'optimizer_time': 1.708824872970581}
```

オプティマイザーをノイズの多い環境で動作するよう設計された SPSA に変更すると、より良い結果が得られます。（正解は -1.857。）ただし、ノイズが結果に影響を与えていため、正確性は劣ります。

4. VQE の高度な使い方

- 初期パラメーター (initial_point) の設定
- 期待値 (Expectations) オブジェクトの設定
- 勾配 (Gradient)
 - : チュートリアル Operator編-2 「Qiskit Gradientフレームワーク」
- Quantum Instance と高度なシミュレーション
 - : チュートリアル シミュレーター編-7 「行列積状態を用いたシミュレーション」

まとめ： QiskitチュートリアルAlgorithm編からVQEを紹介

- ・ 自然界で実現しているような多くの系であれば（自然界で実現している量子力学の系なので）、量子力学と同じ原理で動いている量子コンピューターでその状態を効率よく生成できるのでは、と期待されている。
- ・ 古典コンピューターでは作り出すことのできない量子状態を試行関数とできるところに量子コンピューターを利用する強みがあると考えられている。
 - ・ 例）量子化学の標準的な試行状態、ユニタリー結合クラスター法(UCC)は、古典コンピューターでは効率的に生成することができない状態。

IBM Quantum Challenge 2021

May 20 at 11:00 PM (local time) – May 27 at 10:00 AM (local time)

We invite you to celebrate the fifth anniversary of IBM Quantum with five quantum programming exercises.

Five years ago, our team made history by launching the IBM Quantum Experience, putting the first quantum processor on the IBM Cloud so that anyone could run their own quantum computing experiments.

Today, we have over 320,000 users on the IBM Quantum platform, including 140+ client partners in the IBM Quantum Network. Thousands of developers run two billion quantum circuits on IBM Quantum systems daily with the help of the open source Qiskit software development kit. Together, our users have published more than 500 papers based on research conducted on IBM Quantum systems.

We believe the future of quantum computing is in open source software and access to real quantum hardware. Let's keep building that future together.

Join us on May 20th at 10 am EDT!

Challenge starts in:

14 : 12 : 52
days hours minutes

Interested in joining the challenge?

Please [create an IBMid account](#) if you are new to IBM Quantum.

量子プログラミングコンテスト
5/20(木)23:00pm
~5/27(金)10:00am

<https://challenges.quantum-computing.ibm.com/iqc21>

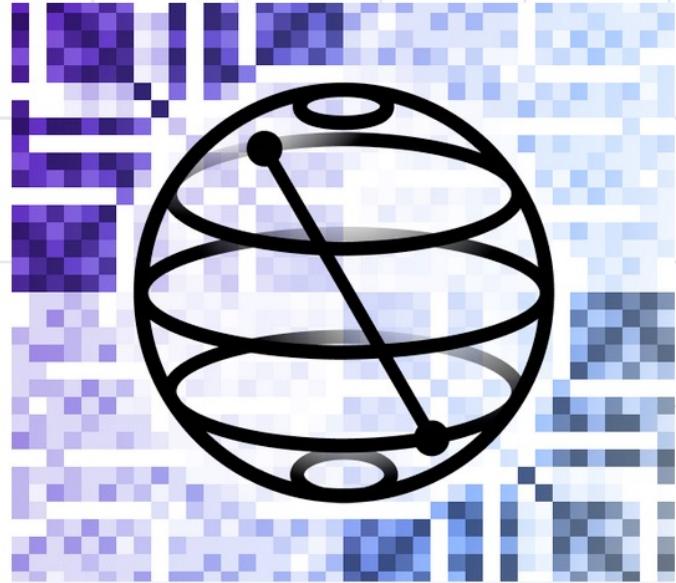
Qiskit Global Summer School 2021: Quantum Machine Learning

The Qiskit Global Summer School 2021 is a two-week intensive summer school designed to empower the next generation of quantum researchers and developers with the skills and know-how to explore quantum applications on their own. This second-annual course, made up of twenty lectures, five applied lab exercises, hands-on mentorship, and live Q&A sessions, focuses on developing hands-on experience and understanding of quantum machine learning.

Registration will open on May 26, 2021 at 12:00 PM EST. Please follow [Qiskit Twitter](#) for more details and updates. For any questions, please check out our FAQ or submit an enquiry using the form below!

量子コンピューター夏学校
～今年は量子機械学習です！
開催期間：7/12(月)～7/23(金)
申し込み開始：5/27(木)午前1:00～

<https://qiskit.org/events/summer-school/>



Qiskit Global Summer School 2021: Quantum Machine Learning

The Qiskit Global Summer School returns as a two-week intensive course focused on Quantum Machine Learning and more!

Online

July 12 - 23, 2021

[Learn more](#)