量子計算入門ハンズオン

このノートは計算物理春の学校2023の個別講義「量子計算入門ハンズオン」の講義資料です。

ノートを編集して実行するには、メニュのファイル (File) → ドライブにコピーを保存 (Save a copy in Drive) から自分のノートを開いてください。

QURI Partsの紹介

量子計算を記述するためのSDKは、各ハードウェアベンダーなどから様々提供されていますが、あるSDKで記述したアルゴリズムは、当然ながら別のSDKでは動きません。また目的の分野での量子アルゴリズムを記述するための、高度な機能がデフォルトで提供されているかどうかも、SDKによってまちまちです。QURI Partsはこうした SDK間の差異を吸収し、共通したインタフェースで、高度な量子アルゴリズムを比較的容易に記述、実行できることを目的として開発されています。

QURI PartsはQunaSysでの研究開発のために開発されていますが、一部はオープンソース化されており、pipから簡単にインストールできます。興味を持たれた方は、講義後もQURI Partsで量子計算を楽しんでみてください。

このレクチャーでは量子回路の実行のためのバックエンドとして、量子回路シミュレータQulacsと、実機実行も可能なSDKであるQiskitを使用しています。

(この講義資料は、QURI Parts TutorialおよびQuantum Native Dojoの内容を一部改変して作成されています。)

環境構築

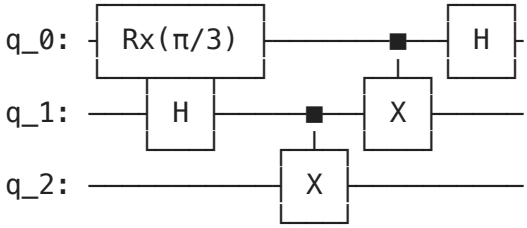
まずはQuri-Partsをインストールします。以下のセルを順に実行していきましょう。(Shift + Enterで実行できます。)

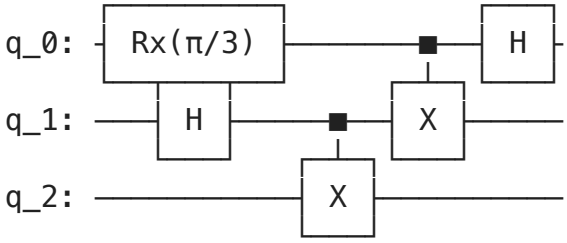
In [ ]:
```
!python -m pip install quri-parts[qulacs,qiskit] matplotlib
```

```
Successfully installed ...
```

インストールが終わって上のようなメッセージが表示されたら、最初の量子回路を作ってみましょう。

In [ ]:
```
from math import pi
from quri_parts.circuit import X, RX, CNOT, QuantumCircuit
from quri_parts.qiskit.circuit import convert_circuit

from quri_parts.circuit import QuantumCircuit

circuit = QuantumCircuit(3)
circuit.add_RX_gate(0, pi/3)
circuit.add_H_gate(1)
circuit.add_CNOT_gate(1, 2)
circuit.add_CNOT_gate(0, 1)
circuit.add_H_gate(0)

print(convert_circuit(circuit))
```





上のような回路が表示されたでしょうか? これで準備は終わりです。上手く行っていない場合は、手を上げて教えてください。

第1章 量子ゲートと量子回路

(共通講義資料では、(P4) 1.量子コンピュータの概要、(P15) 2.基本的な量子ゲート操作をご参照ください。)

現在の量子計算では、量子ビットに作用する量子ゲートを並べて量子回路を構成し、実行、測定することが必要です。

Q#やSilqのような高級言語も幾つか提案されていますが、量子ゲートは意識する必要があります。将来的にはより高級な表現で量子計算を記述することが一般的になる可能性もありますが、未来はまだわかりません。

主な量子ゲートについては、すでに共通講義 (P15 - P39) で紹介されていると思います。ここでは、QURI Partsで量子ゲートや量子回路をどのように扱うのか見て行きましょう。

QuantumGateオブジェクト

QURI Partsでは量子ゲートは `QuantumGate` オブジェクトとして表現されています。 `QuantumGate` オブジェクトは、ゲートの種類や、パラメータ、作用対象の量子ビットの情報などを持っています。ファクトリ関数を使って、量子ゲートを作ってみましょう。

In [ ]:
```
from quri_parts.circuit import X, RX, CNOT

gates = [
    # 量子ビット0に作用するXゲート
    X(0),
    # 量子ビット1に作用する角度pi/3の回転ゲート
    RX(1, pi/3),
    # 量子ビット2（制御）と量子ビット1（ターゲット）に作用するCNOTゲート
    CNOT(2, 1),
]

for gate in gates:
    print(gate)
```

```
QuantumGate(name='X', target_indices=(0,), control_indices=(), params=()
, pauli_ids=(), unitary_matrix=())
QuantumGate(name='RX', target_indices=(1,), control_indices=(), params=(
1.0471975511965976,), pauli_ids=(), unitary_matrix=())
QuantumGate(name='CNOT', target_indices=(1,), control_indices=(2,), para
ms=(), pauli_ids=(), unitary_matrix=())
```

ゲートオブジェクトのアトリビュートは参照できます。(設定はできません。)

In [ ]:
```
from quri_parts.circuit import PauliRotation

x_gate = X(0)
print(f"name: {x_gate.name}, target: {x_gate.target_indices}")

rx_gate = RX(1, pi/3)
print(f"name: {rx_gate.name}, target: {rx_gate.target_indices}, angle: {r

cnot_gate = CNOT(2, 1)
print(f"name: {cnot_gate.name}, control: {cnot_gate.control_indices}, tar

pauli_rot_gate = PauliRotation(target_indices=(0, 1, 2), pauli_ids=(1, 2,
print(f"name: {pauli_rot_gate.name}, target: {pauli_rot_gate.target_indic
```

```
name: X, target: (0,)
name: RX, target: (1,), angle: 1.0471975511965976
name: CNOT, control: (2,), target: (1,)
name: PauliRotation, target: (0, 1, 2), pauli_ids: (1, 2, 3), angle: 1.0
471975511965976
```

QuantumCircuitオブジェクト

QURI Partsでは、以下のように、回路で使用される量子ビット数を指定して、量子回路を作成します。

In [ ]:
```
from quri_parts.circuit import QuantumCircuit

# 3量子ビットの回路を作成
circuit = QuantumCircuit(3)
# QuantumGateオブジェクトを作成してから回路に追加
circuit.add_gate(X(0))
# ゲート追加用のメソッドを使用
circuit.add_X_gate(0)
circuit.add_RX_gate(1, pi/3)
circuit.add_CNOT_gate(2, 1)
circuit.add_PauliRotation_gate(target_qubits=(0, 1, 2), pauli_id_list=(1,
```

`QuantumCircuit`オブジェクトには幾つかのプロパティがあります。

In [ ]:
```
print("Qubit count:", circuit.qubit_count)
print("Circuit depth:", circuit.depth)

gates = circuit.gates # .gatesプロパティは回路に含まれるゲートのSequenceを返しま
print("# of gates in the circuit:", len(gates))
for gate in gates:
    print(gate)
```

```
Qubit count: 3
Circuit depth: 3
# of gates in the circuit: 5
QuantumGate(name='X', target_indices=(0,), control_indices=(), params=()
, pauli_ids=(), unitary_matrix=())
QuantumGate(name='X', target_indices=(0,), control_indices=(), params=()
, pauli_ids=(), unitary_matrix=())
QuantumGate(name='RX', target_indices=(1,), control_indices=(), params=(
1.0471975511965976,), pauli_ids=(), unitary_matrix=())
QuantumGate(name='CNOT', target_indices=(1,), control_indices=(2,), para
ms=(), pauli_ids=(), unitary_matrix=())
QuantumGate(name='PauliRotation', target_indices=(0, 1, 2), control_indi
ces=(), params=(1.0471975511965976,), pauli_ids=(1, 2, 3), unitary_matri
x=())
```

量子ビット数が同じ `QuantumCircuit` オブジェクト同士は連結することができます。

In [ ]:
```
circuit2 = QuantumCircuit(3)
circuit2.add_Y_gate(1)
circuit2.add_H_gate(2)

combined = circuit + circuit2 # 右の処理と同等: combined = circuit.combine(
print("Combined circuit:", combined.gates)

circuit2 += circuit # 右の処理と同等: circuit2.extend(circuit)
print("Extended circuit:", circuit2.gates)
```

```
Combined circuit: (QuantumGate(name='X', target_indices=(0,), control_in
dices=(), params=(), pauli_ids=(), unitary_matrix=()), QuantumGate(name=
'X', target_indices=(0,), control_indices=(), params=(), pauli_ids=(), u
nitary_matrix=()), QuantumGate(name='RX', target_indices=(1,), control_i
ndices=(), params=(1.0471975511965976,), pauli_ids=(), unitary_matrix=()
), QuantumGate(name='CNOT', target_indices=(1,), control_indices=(2,), p
arams=(), pauli_ids=(), unitary_matrix=()), QuantumGate(name='PauliRotat
ion', target_indices=(0, 1, 2), control_indices=(), params=(1.0471975511
965976,), pauli_ids=(1, 2, 3), unitary_matrix=()), QuantumGate(name='Y',
target_indices=(1,), control_indices=(), params=(), pauli_ids=(), unitar
y_matrix=()), QuantumGate(name='H', target_indices=(2,), control_indices
=(), params=(), pauli_ids=(), unitary_matrix=()))
Extended circuit: (QuantumGate(name='Y', target_indices=(1,), control_in
dices=(), params=(), pauli_ids=(), unitary_matrix=()), QuantumGate(name=
'H', target_indices=(2,), control_indices=(), params=(), pauli_ids=(), u
nitary_matrix=()), QuantumGate(name='X', target_indices=(0,), control_in
dices=(), params=(), pauli_ids=(), unitary_matrix=()), QuantumGate(name=
'X', target_indices=(0,), control_indices=(), params=(), pauli_ids=(), u
nitary_matrix=()), QuantumGate(name='RX', target_indices=(1,), control_i
ndices=(), params=(1.0471975511965976,), pauli_ids=(), unitary_matrix=()
), QuantumGate(name='CNOT', target_indices=(1,), control_indices=(2,), p
arams=(), pauli_ids=(), unitary_matrix=()), QuantumGate(name='PauliRotat
ion', target_indices=(0, 1, 2), control_indices=(), params=(1.0471975511
965976,), pauli_ids=(1, 2, 3), unitary_matrix=()))
```

回路の変換

QURI Partsで作成した回路は、バックエンドのシミュレータや実機の回路に変換することができます。例えばQulacsの回路を直接使用したい場合には、`quri_parts.qulacs.circuit.convert_circuit`関数が利用できます。

In [ ]:
```
from quri_parts.qulacs.circuit import convert_circuit
qulacs_circuit = convert_circuit(circuit)
print(qulacs_circuit)

import qulacs
qulacs_state = qulacs.QuantumState(3)
qulacs_circuit.update_quantum_state(qulacs_state)
print(qulacs_state)
```

```
*** Quantum Circuit Info ***
# of qubit: 3
# of step : 3
# of gate : 5
# of 1 qubit gate: 3
# of 2 qubit gate: 1
# of 3 qubit gate: 1
Clifford  : no
Gaussian  : no


 *** Quantum State ***
 * Qubit Count : 3
 * Dimension   : 8
 * State vector :
     (0.75,0)
     (0,0.25)
(0,-0.433013)
 (0.433013,0)
        (0,0)
        (0,0)
        (0,0)
        (0,0)
```

こうした変換用の関数はバックエンドごとに用意されており、基本的には`quri_parts.[SDK].circuit.convert_circuit`に置かれています。

Parametric回路

変分アルゴリズム等のいくつかの量子アルゴリズムでは、パラメータを持つ量子回路が重要な役割を果たしています。QURI Partsではそうした回路を扱う特別な方法を用意することで、アルゴリズムの処理を効率化しています。

Parameter

束縛されていないパラメータは、`quri_parts.circuit.Parameter`クラスで表現されています。`Parameter`オブジェクトはプレースホルダとして機能し、特定の値は保持しません。`Parameter`オブジェクトは、Pythonのオブジェクトの固有識別子によって区別されるため、同じ名前の`Parameter`オブジェクトであっても、別の変数として扱われます。

In [ ]:
```
from quri_parts.circuit import Parameter, CONST

phi = Parameter("phi")
psi1 = Parameter("psi")
psi2 = Parameter("psi2")

# CONSTは定数を意味する定義済みのパラメータ
print(phi, psi1, psi2, CONST)
print("phi == psi1:", phi == psi1)
print("psi1 == psi2:", psi1 == psi2)
print("phi == CONST:", phi == CONST)
```

```
Parameter(name=phi) Parameter(name=psi) Parameter(name=psi2) Parameter(n
ame=)
phi == psi1: False
psi1 == psi2: False
phi == CONST: False
```

`Parameter`オブジェクトは、parametric回路の中で自動的に処理されるため、通常はユーザが直接扱う必要はありません。

Unbound parametric回路

Unbound parametric回路には幾つかの種類がありますが、まずは共通の使用方法を説明します。

In [ ]:
```
# ここではunbound parametric回路を作成する1つの方法を示しています
# 詳細は後のセクションで解説されます
from quri_parts.circuit import UnboundParametricQuantumCircuit
parametric_circuit = UnboundParametricQuantumCircuit(2)
parametric_circuit.add_H_gate(0)
parametric_circuit.add_CNOT_gate(0, 1)
param1 = parametric_circuit.add_ParametricRX_gate(0)
param2 = parametric_circuit.add_ParametricRZ_gate(1)
```

Unbound parametric回路オブジェクトは通常の回路と共通の幾つかのプロパティを持っています。

In [ ]:
```
print("Qubit count:", parametric_circuit.qubit_count)
print("Circuit depth:", parametric_circuit.depth)

# Parametric回路は.gatesプロパティを持ちません

print("Parameter count:", parametric_circuit.parameter_count)

# QuantumCircuitと同じように通常のゲートを追加することもできます
parametric_circuit.add_X_gate(1)
print("Circuit depth:", parametric_circuit.depth)
```

```
Qubit count: 2
Circuit depth: 3
Parameter count: 2
Circuit depth: 4
```

Unbound parametric回路も凍結したり、複製したりできます。

In [ ]:
```
frozen_parametric_circuit = parametric_circuit.freeze()
copied_parametric_circuit = parametric_circuit.get_mutable_copy()
```

`.bind_parameters`メソッドを使うことで、特定の値をparametric回路に束縛できます。このメソッドはオリジナルの回路を変更することはなく、新しく作成された回路オブジェクトを返します。

In [ ]:
```
bound_circuit = parametric_circuit.bind_parameters([0.2, 0.3])
# bound_circuitは通常の回路の不変なバージョンで、.gatesプロパティを持ちます
for gate in bound_circuit.gates:
    print(gate)
```

```
QuantumGate(name='H', target_indices=(0,), control_indices=(), params=()
, pauli_ids=(), unitary_matrix=())
QuantumGate(name='CNOT', target_indices=(1,), control_indices=(0,), para
ms=(), pauli_ids=(), unitary_matrix=())
QuantumGate(name='RX', target_indices=(0,), control_indices=(), params=(
0.2,), pauli_ids=(), unitary_matrix=())
QuantumGate(name='RZ', target_indices=(1,), control_indices=(), params=(
0.3,), pauli_ids=(), unitary_matrix=())
QuantumGate(name='X', target_indices=(1,), control_indices=(), params=()
, pauli_ids=(), unitary_matrix=())
```

UnboundParametricQuantumCircuitと LinearMappedUnboundParametricQuantumCircuit

いまのところ、QURI Partsは2種類のparametric回路: `UnboundParametricQuantumCircuit` と `LinearMappedUnboundParametricCircuit` を提供しています。

`UnboundParametricQuantumCircuit` は、各パラメータが独立に変化する回路を表現しています。以下のようにパラメータゲートを追加することができます。

In [ ]:
```
from quri_parts.circuit import UnboundParametricQuantumCircuit
parametric_circuit = UnboundParametricQuantumCircuit(2)
parametric_circuit.add_H_gate(0)
parametric_circuit.add_CNOT_gate(0, 1)
param1 = parametric_circuit.add_ParametricRX_gate(0)
param2 = parametric_circuit.add_ParametricRZ_gate(1)
print("param1 == param2:", param1 == param2)
```

```
param1 == param2: False
```

パラメータゲートは、 `.add_Parametric{}_gate` というメソッドで追加でき、(ここでは、{}には具体的なゲート名が入ります。) 新しく作成されたパラメータが返ります。ここでは、$[H_0, CNOT_{0,1}, RX(\theta)_0, RZ(\phi)_1]$という回路が作成されています。$\theta$と$\phi$は独立に変化するパラメータで、独立に束縛することができます。

In [ ]:
```
bound_circuit = parametric_circuit.bind_parameters([0.2, 0.3])
for gate in bound_circuit.gates:
    print(gate)
```

```
QuantumGate(name='H', target_indices=(0,), control_indices=(), params=()
, pauli_ids=(), unitary_matrix=())
QuantumGate(name='CNOT', target_indices=(1,), control_indices=(0,), para
ms=(), pauli_ids=(), unitary_matrix=())
QuantumGate(name='RX', target_indices=(0,), control_indices=(), params=(
0.2,), pauli_ids=(), unitary_matrix=())
QuantumGate(name='RZ', target_indices=(1,), control_indices=(), params=(
0.3,), pauli_ids=(), unitary_matrix=())
```

一方で、複数のパラメータゲートに対して、同一のパラメータから計算された値を適用したい場合がよくあります。よくありま
す。 `LinearMappedUnboundParametricQuantumCircuit` はこうしたケースをサポートしています。(ただし線型の変換のみに対応しています。) 例えば、以下のような独立パラメータθ、ϕを持つ回路を考えてみましょう。

$$[H_0, CNOT_{0,1}, RX(\theta/2 + \phi/3 + \pi/2)_0, RZ(\theta/3 - \phi/2 - \pi/2)_1]$$

この回路は以下のように作成できます。

In [ ]:
```
from math import pi
from quri_parts.circuit import LinearMappedUnboundParametricQuantumCircui

linear_param_circuit = LinearMappedUnboundParametricQuantumCircuit(2)
linear_param_circuit.add_H_gate(0)
linear_param_circuit.add_CNOT_gate(0, 1)

theta, phi = linear_param_circuit.add_parameters("theta", "phi")
linear_param_circuit.add_ParametricRX_gate(0, {theta: 1/2, phi: 1/3, CONS
linear_param_circuit.add_ParametricRZ_gate(1, {theta: 1/3, phi: -1/2, CON
```

`LinearMappedUnboundParametricQuantumCircuit`にパラメータゲートを追加するには、まず`.add_parameters`メソッドで独立パラメータを追加します。次に各パラメータゲートの追加時に独立パラメータがキー、その係数が値となった辞書を渡します。定数項は`quri_parts.circuit.CONST`を使って表現できます。

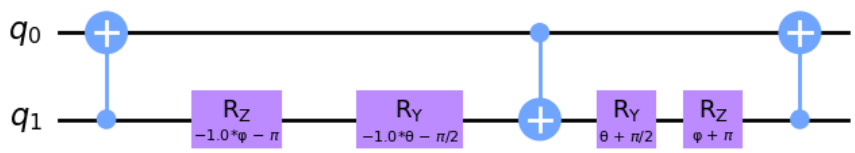
こうして作成された回路に対しては、2つのパラメータに対して2つの値を束縛できます。

In [ ]:
```
bound_linear_circuit = linear_param_circuit.bind_parameters([0.2, 0.3])
for gate in bound_linear_circuit.gates:
    print(gate)
```

```
QuantumGate(name='H', target_indices=(0,), control_indices=(), params=()
, pauli_ids=(), unitary_matrix=())
QuantumGate(name='CNOT', target_indices=(1,), control_indices=(0,), para
ms=(), pauli_ids=(), unitary_matrix=())
QuantumGate(name='RX', target_indices=(0,), control_indices=(), params=(
1.7707963267948965,), pauli_ids=(), unitary_matrix=())
QuantumGate(name='RZ', target_indices=(1,), control_indices=(), params=(
-1.6541296601282298,), pauli_ids=(), unitary_matrix=())
```

Exercise 1

以下の回路は、第4章で扱うVQEというアルゴリズムで使われる、Symmetry Preserving Ansatzと呼ばれるパラメータ付きの状態準備回路の一部です。この回路をQURI Partsで作成してみてください。図中のθとϕはパラメータです。



In [ ]:

第2章 演算子と期待値

(共通講義資料では、(P84) 6-5.期待値測定はどうやるの? をご参照ください。)

量子力学では物理量はエルミート演算子$\hat{O}$で表され、オブザーバブル (観測量) と呼ばれます。ある状態 $|\psi\rangle$ について、$\langle\psi|\hat{O}|\psi\rangle$ によって期待値が計算できます。

エルミート演算子はパウリ演算子の積の和で表現できるので、

$$\hat{O} = \sum_i a_i \hat{P}_i$$

量子計算では各パウリ演算子の積 $\hat{P}_i$ に対して測定を行い、係数 a_i を掛けて足し合わせることで、目的の演算子の期待値を得ます。

$$\hat{O} = \langle\psi|\hat{O}|\psi\rangle = \sum_i a_i \langle\psi|\hat{P}_i|\psi\rangle$$

この章ではQURI Partsを使って、演算子を表現し、期待値を推定する方法を紹介していきたいと思います。まずはパウリ演算子の積を指定するための、パウリラベルという表現を見ていきましょう。

パウリラベル

パウリラベル (パウリ文字列) は、各量子ビットに作用する幾つかのパウリ行列の積を表現します。例えば、$X_0Y_2Z_4$は、量子ビット0に作用するX、量子ビット2に作用するY、量子ビット4に作用するZを意味しており、以下のように定義できます。

In [ ]:
```
from quri_parts.core.operator import pauli_label
label = pauli_label("X0 Y2 Z4")
# パウリ名と量子ビット番号の間にスペースを入れることもできます
label = pauli_label("X 0 Y 2 Z 4")
print(label)
```

```
X0 Y2 Z4
```

`PAULI_IDENTITY`は0個のパウリ行列の作用を表現するPauliラベルです。

In [ ]:
```
from quri_parts.core.operator import PAULI_IDENTITY
print(PAULI_IDENTITY)
```

```
I
```

パウリラベルは不変でハッシュ化可能なオブジェクトで、量子ビット番号と`SinglePauli`列挙型のペアからなり、イテレータで取り出すことができます。

In [ ]:
```
for pair in label:
    print(pair)
```

```
(0, <SinglePauli.X: 1>)
(4, <SinglePauli.Z: 3>)
(2, <SinglePauli.Y: 2>)
```

In [ ]:
```
for index, matrix in label:
    print(f"qubit index: {index}, Pauli matrix: {matrix}")
```

```
qubit index: 0, Pauli matrix: 1
qubit index: 4, Pauli matrix: 3
qubit index: 2, Pauli matrix: 2
```

`SinglePauli` は `IntEnum` なので、`SinglePauli.X`、`SinglePauli.Y`、`SinglePauli.Z` の代わりに、1、2、3を使うこともできます。

In [ ]:
```
from quri_parts.core.operator import SinglePauli
print(SinglePauli.X == 1)
print(SinglePauli.Y == 2)
print(SinglePauli.Z == 3)
```

```
True
True
True
```

演算子

演算子はパウリラベルとその係数で定義されます。例えば、
$(0.5+0.5i)X_0Y_1 + 0.2iZ_0Z_2 + 0.3 + 0.4i$は、以下のように定義されます。

In [ ]:
```
from quri_parts.core.operator import Operator
op = Operator({
    pauli_label("X0 Y1"): 0.5 + 0.5j,
    pauli_label("Z0 Z2"): 0.2j,
    PAULI_IDENTITY: 0.3 + 0.4j,
})
print(op)
```

```
(0.5+0.5j)*X0 Y1 + 0.2j*Z0 Z2 + (0.3+0.4j)*I
```

オペレータは、順番にパウリ項を追加することで作成することもできます。

In [ ]:
```
op = Operator()
op.add_term(pauli_label("X0 Y1"), 0.5 + 0.5j)
op.add_term(pauli_label("Z0 Z2"), 0.2j)
op.constant = 0.3 + 0.4j
print(op)
print(f"Number of terms: {op.n_terms}")

# 既存の項に追加した場合、係数は加算されます
op.add_term(pauli_label("X0 Y1"), 0.5)
print(op)
print(f"Number of terms: {op.n_terms}")

# 係数が0になった場合、項は削除されます
op.add_term(pauli_label("X0 Y1"), -1.0 - 0.5j)
print(op)
print(f"Number of terms: {op.n_terms}")
```

```
(0.5+0.5j)*X0 Y1 + 0.2j*Z0 Z2 + (0.3+0.4j)*I
Number of terms: 3
(1+0.5j)*X0 Y1 + 0.2j*Z0 Z2 + (0.3+0.4j)*I
Number of terms: 3
0.2j*Z0 Z2 + (0.3+0.4j)*I
Number of terms: 2
```

エルミート共役なオペレータは以下のメソッドで得られます。

In [ ]:
```
conj = op.hermitian_conjugated()
print(conj)
```

```
-0.2j*Z0 Z2 + (0.3-0.4j)*I
```

内部的には `Operator` は `dict` なので、辞書として操作することもできます。

In [ ]:
```
p = pauli_label("Z0 Z2")
coef = op[p]
print(f"Coefficient of {p} = {coef}")

op[p] = 0.4
coef = op[p]
print(f"Coefficient of {p} = {coef}")

for label, coef in op.items():
    print(f"Coefficient of {label} = {coef}")
```

```
Coefficient of Z0 Z2 = 0.2j
Coefficient of Z0 Z2 = 0.4
Coefficient of Z0 Z2 = 0.4
Coefficient of I = (0.3+0.4j)
```

量子状態

QURI Partsでは幾つかの種類の量子状態準備が可能です。まずは、最も基本的な計算基底について紹介しましょう。

計算基底状態は、各量子ビットが各量子ビットが0またはまたは1の固有状態であるような、量子状態です。５量子ビットの計算基底状態を作ってみましょう。

In [ ]:
```
from quri_parts.core.state import ComputationalBasisState
state1 = ComputationalBasisState(5, bits=0b10100)
print(state1)
```

```
ComputationalBasisState(qubit_count=5, bits=0b10100, phase=0π/2)
```

ここでは `bits=0b10100` は、量子ビット0が$|0\rangle$、量子ビット1が$|0\rangle$、量子ビット2が$|1\rangle$、量子ビット3が$|0\rangle$、量子ビット4が$|1\rangle$の状態を意味します。インデックスは０始まりで、下位ビット側から上位ビット側の順で割り振られています。

また、2つの状態の重ね合わせを作成することもできます。結果として得られる状態は、計算基底状態ではなくなることに注意してください。 `comp_basis_superposition()` メソッドは4つの引数を取ります。最初の2つは重ね合わされる状態です。3つ目の引数θは重ね合わせの重みで、4つ目の引数ϕは重ね合わせの位相因子を意味します。2つの状態の係数はそれぞれ、$\cos\theta$、$e^{i\phi}\sin\theta$になります。

In [ ]:
```
import math
from quri_parts.core.state import comp_basis_superposition
state2 = ComputationalBasisState(5, bits=0b01011)
sp_state = comp_basis_superposition(state1, state2, math.pi/2, math.pi/4)
print(sp_state)
```

```
GeneralCircuitQuantumState(n_qubits=5, circuit=<quri_parts.circuit.circuit.ImmutableQuantumCircuit object at 0x7f3014f63d30>)
```

Estimator

あたえられた状態に対して演算子の期待値を計算するには、 `QuantumEstimator` を使います。 `QuantumEstimator` (`quri_parts.core.estimator` パッケージにあります。) 自体は抽象的なインタフェースなので、使用する際は適宜、具体的なインスタンスが必要です。 `QuantumEstimator` のインタフェースは、状態ベクトルのシミュレーション、サンプリングのシミュレーション、実デバイスでのサンプリングなど様々な方法での期待値推定に共通して使用できます。

In [ ]:
```python
from quri_parts.qulacs.estimator import create_qulacs_vector_estimator
# Estimatorを作成し
estimator = create_qulacs_vector_estimator()
# 演算子と状態を渡します
estimate = estimator(op, sp_state)
# 戻り値には推定された期待値と、誤差が含まれています
print(f"Estimated expectation value: {estimate.value}")
# (状態ベクトルのシミュレーションを行なった場合、誤差は0になります)
print(f"Estimation error: {estimate.error}")
```

```
Estimated expectation value: (-0.10000000000000003+0.4j)
Estimation error: 0.0
```

`QuantumEstimator` の戻り値には、期待値 (`.value`) と誤差 (`.error`) が含まれています。

並列実行のためのEstimator (Skip)

また、並列実行のための `ConcurrentQuantumEstimator` インタフェースも用意されており、以下のように、複数の演算子や複数の状態を1度に計算できます。

- 1つの演算子、複数の状態
- 複数の演算子、1つの状態
- 同じ数の、複数の演算子と複数の状態

例えばQulacsを使用する場合は、 `concurrent.futures.Executor` (デフォルトでは `None` で、並列化を行わない。) と並列数 (デフォルトでは1) を指定して、 `ConcurrentQuantumEstimator` を作成します。Qulacsはそれ自体が並列処理をサポートしているため、サポートしているため、 `ThreadPoolExecutor` や `ProcessPoolExecutor` の使用では、パフォーマンスが改善されないかもしれません。

In [ ]:
```
from concurrent.futures import ThreadPoolExecutor
from quri_parts.qulacs.estimator import create_qulacs_vector_concurrent_e
# Executorを作成します（オプション）
executor = ThreadPoolExecutor(max_workers=4)
# 並列処理されるEstimatorを作成します
estimator = create_qulacs_vector_concurrent_estimator(executor, concurren

estimates = estimator(op, [state1, state2])
for i, est in enumerate(estimates):
    print(f"State {i}: value={est.value}, error={est.error}")

# EstimatorにはPauliLabelを渡すこともできます
estimates = estimator([label, op], [sp_state])
for i, est in enumerate(estimates):
    print(f"Operator {i}: value={est.value}, error={est.error}")

estimates = estimator([label, op], [state1, state2])
for i, est in enumerate(estimates):
    print(f"Operator {i}, state {i}: value={est.value}, error={est.error}
```

```
State 0: value=(-1+0j), error=0.0
State 1: value=(1+0j), error=0.0
Operator 0: value=(1+0j), error=0.0
Operator 1: value=(-0.10000000000000003+0.4j), error=0.0
Operator 0, state 0: value=(1+0j), error=0.0
Operator 1, state 1: value=(-0.10000000000000003+0.4j), error=0.0
```

パラメトリック状態と、演算子の期待値推定

パラメータ回路に対しても、パラメータを束縛し、`GeneralCircuitQuantumState`を作成し、Estimatorを使用して、その状態に対する演算子の期待値を推定することができます。

In [ ]:
```
from quri_parts.core.state import GeneralCircuitQuantumState

circuit = parametric_circuit.bind_parameters([0.2, 0.3])
circuit_state = GeneralCircuitQuantumState(2, circuit)

from quri_parts.core.operator import Operator, pauli_label
op = Operator({
    pauli_label("X0 Y1"): 0.5 + 0.5j,
    pauli_label("Z0 X1"): 0.2,
})

from quri_parts.qulacs.estimator import create_qulacs_vector_estimator
estimator = create_qulacs_vector_estimator()

estimate = estimator(op, circuit_state)
print(f"Estimated expectation value: {estimate.value}")
```

```
Estimated expectation value: (0.15950226366943507+0.14776010333066977j)
```

しかし、より直接的に、`ParametricCircuitQuantumState`を使うこともできます。これには幾つかの利点があります。

- 状態がパラメータに依存していることが明確になり、パラメータ付きの状態に対する問題を扱えるようになる (例えば、パラメータに対する期待値の勾配)
- Qulacs等のシミュレータでの処理性能の向上

Parametric状態に対する演算子の期待値は、`ParametricQuantumEstimator`を使用して、以下のように記述できます。

In [ ]:
```
from quri_parts.core.state import ParametricCircuitQuantumState

parametric_state = ParametricCircuitQuantumState(2, parametric_circuit)

# Qulacsを使用したparametric estimatorを作成します。このestimatorはParametricQ
from quri_parts.qulacs.estimator import create_qulacs_vector_parametric_e
parametric_estimator = create_qulacs_vector_parametric_estimator()

estimate = parametric_estimator(op, parametric_state, [0.2, 0.3])
print(f"Estimated expectation value: {estimate.value}")
```

```
Estimated expectation value: (0.15950226366943507+0.14776010333066977j)
```

並列実行のための`ConcurrentParametricQuantumEstimator`も用意されており、特にQulacsのバックエンドを使用したものでは、毎回パラメータを束縛するよりも性能が向上しています。

In [ ]:
```
from quri_parts.qulacs.estimator import create_qulacs_vector_concurrent_p
# ここではexecutorとconcurrency引数を指定していませんが、パフォーマンスは改善されて
concurrent_parametric_estimator = create_qulacs_vector_concurrent_paramet

estimates = concurrent_parametric_estimator(op, parametric_state, [[0.2,
for i, est in enumerate(estimates):
    print(f"Parameters {i}: value={est.value}")
```

```
Parameters 0: value=(0.15950226366943507+0.14776010333066977j)
Parameters 1: value=(0.2770521890028376+0.23971276930210147j)
```

Sampling シミュレーション

量子計算機で演算子の期待値を推定するには、サンプリング測定が必要です。サンプリング測定では、量子回路の実行と量子ビットの測定が複数回繰り返され、複数回の測定結果の統計値から、演算子の期待値が推定されます。

回路の準備

まずはサンプリングを行うための回路を作成しましょう。

In [ ]:
```
from math import pi
from quri_parts.circuit import QuantumCircuit
# 4量子ビットの回路
circuit = QuantumCircuit(4)
circuit.add_X_gate(0)
circuit.add_H_gate(1)
circuit.add_Y_gate(2)
circuit.add_CNOT_gate(1, 2)
circuit.add_RX_gate(3, pi/4)
```

Sampler

サンプリング測定には、`Sampler`を使用します。`Sampler` (quri_parts.core.samplingで定義されています。) 自体は抽象的なインタフェースなので、実際にサンプリングを行うには、適宜、具体的なインスタンスが必要です。`Sampler`にはシミュレータを使用するものや、実機を使用するものなど幾つかの実装があります。

ここでは、Qulacsの状態ベクトルシミュレーションを利用したSamplerを使ってみましょう。

In [ ]:
```
from quri_parts.qulacs.sampler import create_qulacs_vector_sampler
# Samplerを作成
sampler = create_qulacs_vector_sampler()
sampling_result = sampler(circuit, shots=1000)
print(sampling_result)
```

```
Counter({3: 429, 5: 429, 11: 82, 13: 60})
```

Samplerはサンプリング対象の回路と、サンプリング回数 (shots) の2つの引数を取ります。戻り値は以下のようなKeyとValueからなる辞書になっています。

- **Keys** 測定結果を表現するint型のビット列。各量子ビットの測定結果は下位ビット側から上位ビット側の順に並んでいる。例えば量子ビット0と量子ビット2が$|1\rangle$で、それ以外が$|0\rangle$の場合、ビット列は`0b0101`となる。
- **Values** 各ビット列が測定された回数。すべての回数を足し合わせると`shots`に等しくなる。

In [ ]:
```
for bits, count in sampling_result.items():
    print(f"A bitstring '{bin(bits)}' is measured {count} times")
print(f"Total count is {sum(sampling_result.values())}")
```

```
A bitstring '0b11' is measured 429 times
A bitstring '0b101' is measured 429 times
A bitstring '0b1011' is measured 82 times
A bitstring '0b1101' is measured 60 times
Total count is 1000
```

上の例では、量子ビット0はXゲートのみが作用しており、常に$|1\rangle$として測定されています。一方で、量子ビット1、2、3は0または1になりますが、量子ビット1と2はCNOTゲートによってもつれた状態にあるため、測定結果のパターンは4通りになっています。

サンプリング推定 (Skip)

パウリグルーピング

演算子の期待値推定には演算子と状態が必要ですが、Samplerはそれらを直接扱えません。解こうとしている問題は、回路として表現されSamplerに渡される必要があります。

まずは推定される演算子を定義してみましょう。

In [ ]:
```
from quri_parts.core.operator import Operator, pauli_label, PAULI_IDENTIT
op = Operator({
    pauli_label("Z0"): 0.25,
    pauli_label("Z1 Z2"): 2.0,
    pauli_label("X1 X2"): 0.5 + 0.25j,
    pauli_label("Z1 Y3"): 1.0j,
    pauli_label("Z2 Y3"): 1.5 + 0.5j,
    pauli_label("X1 Y3"): 2.0j,
    PAULI_IDENTITY: 3.0,
})
print(op)
```

```
0.25*Z0 + 2.0*Z1 Z2 + (0.5+0.25j)*X1 X2 + 1j*Z1 Y3 + (1.5+0.5j)*Z2 Y3 +
2j*X1 Y3 + 3.0*I
```

演算子はパウリ演算子の和として表現されます。そうした演算子の期待値を推定する1つの方法は、それぞれのパウリ項の期待値を推定し足し上げることです。

パウリ項が交換可能な場合、複数のパウリ項を1度に測定することができます。そのためにまずはパウリ項を交換可能なパウリ項ごとにグループ化します。パウリグルーピングは、演算子の推定において1つの重要な研究分野になっています。

最もシンプルな方法の1つはビットごとの交換可能性に基づいたグループ化 (*bitwise grouping*) です。これは以下のように実行できます。

In [ ]:
```
from quri_parts.core.operator.grouping import bitwise_pauli_grouping
pauli_sets = bitwise_pauli_grouping(op)
print(pauli_sets)
```

```
frozenset({frozenset({PauliLabel({(1, <SinglePauli.X: 1>), (3, <SinglePa
uli.Y: 2>)})}), frozenset({PauliLabel()}), frozenset({PauliLabel({(1, <S
inglePauli.X: 1>), (2, <SinglePauli.X: 1>)})}), frozenset({PauliLabel({(
0, <SinglePauli.Z: 3>)}), PauliLabel({(2, <SinglePauli.Z: 3>), (1, <Sing
lePauli.Z: 3>)})}), frozenset({PauliLabel({(2, <SinglePauli.Z: 3>), (3,
<SinglePauli.Y: 2>)}), PauliLabel({(3, <SinglePauli.Y: 2>), (1, <SingleP
auli.Z: 3>)})})})
```

グループ化関数はパウリラベルのfrozensetのfrozensetを返すため、やや複雑な見た目になっています。

In [ ]:
```
print(f"Number of groups: {len(pauli_sets)}")
for i, pauli_set in enumerate(pauli_sets):
    labels = ", ".join([str(pauli) for pauli in pauli_set])
    print(f"Group {i} contains: {labels}")
```

```
Number of groups: 5
Group 0 contains: X1 Y3
Group 1 contains: I
Group 2 contains: X1 X2
Group 3 contains: Z0, Z1 Z2
Group 4 contains: Z2 Y3, Z1 Y3
```

測定回路

交換可能なパウリセットを測定するには、測定の前に適用される回路を作成する必要があります。bitwise groupingを使用する場合、以下のように回路を構築できます。

In [ ]:
```
from quri_parts.core.measurement import bitwise_commuting_pauli_measureme
pauli_set = {pauli_label("Z2 Y3"), pauli_label("Z1 Y3")}
measurement_circuit = bitwise_commuting_pauli_measurement_circuit(pauli_s
print(measurement_circuit)
```

```
(QuantumGate(name='Sdag', target_indices=(3,), control_indices=(), param
s=(), pauli_ids=(), unitary_matrix=()), QuantumGate(name='H', target_ind
ices=(3,), control_indices=(), params=(), pauli_ids=(), unitary_matrix=(
)))
```

状態に対するサンプリング

次に、以下のような手順で状態に対するサンプリングを行います。

- 状態準備のための回路を作成する
- パウリセットの測定のための回路を、状態準備の回路の後ろに連結する
- 連結された回路に対してサンプリングを行う

ここでは簡単のために `ComputationalBasisState` を使って初期状態を準備していますが、任意の `CircuitQuantumState` を使用することができます。

In [ ]:
```
from quri_parts.core.state import ComputationalBasisState
initial_state = ComputationalBasisState(4, bits=0b0101)
# 状態準備のための回路
state_prep_circuit = initial_state.circuit
# 測定回路を連結します
sampled_circuit = state_prep_circuit + measurement_circuit
# サンプリングを行います
sampling_result = sampler(sampled_circuit, shots=1000)
print({bin(bits): count for bits, count in sampling_result.items()})
```

```
{'0b101': 494, '0b1101': 506}
```

サンプリング結果からのパウリ項の再構築

パウリ項の値はサンプリング結果から再構築する必要があります。上の例ではZ_2Y_3とZ_1Y_3がサンプリング測定され、`0b1101`と`0b0101`の2つのビットパターンが得られました。
bitwise groupingの場合、パウリ演算子の値は以下のように再構築できます。

In [ ]:
```
from quri_parts.core.measurement import bitwise_pauli_reconstructor_facto
# Z2 Y3のためのreconstructorを作成します
reconstructor = bitwise_pauli_reconstructor_factory(pauli_label("Z2 Y3"))
# サンプル結果のビットパターン0b1101からZ2 Y3の値を再構築します
pauli_value = reconstructor(0b1101)
print(pauli_value)
# 0b0101から再構築します
pauli_value = reconstructor(0b0101)
print(pauli_value)
```

```
1
-1
```

Z_2Y_3の期待値は以下のように計算できます。

In [ ]:
```
pauli_exp = (
    reconstructor(0b1101) * sampling_result[0b1101] +
    reconstructor(0b0101) * sampling_result[0b0101]
) / 1000
print(pauli_exp)

# 上の例と等価です
pauli_exp = sum(
    reconstructor(bits) * count for bits, count in sampling_result.items(
) / sum(sampling_result.values())
print(pauli_exp)

# より便利な方法もあります
from quri_parts.core.estimator.sampling import trivial_pauli_expectation_
pauli_exp = trivial_pauli_expectation_estimator(sampling_result, pauli_la
print(pauli_exp)
```

```
0.012
0.012
0.012
```

ここではbitwise groupingを使用しているため、`trivial_pauli_expectation_estimator`を使用しています。より一般的な場合には、`general_pauli_expectation_estimator`を`PauliReconstuctorFactory`とともに使用してください。

In [ ]:
```
from quri_parts.core.estimator.sampling import general_pauli_expectation_
pauli_exp = general_pauli_expectation_estimator(
    sampling_result, pauli_label("Z2 Y3"), bitwise_pauli_reconstructor_fa
)
print(pauli_exp)
```

```
0.012
```

オリジナルの演算子の期待値を、パウリ項の推定値から推定する

最後に、各パウリ項を足し合わせることで、オリジナルの演算子の期待値を推定します。Z_2Y_3の寄与は以下のように計算できます。

In [ ]:
```
# opに含まれるZ2 Y3の係数を取得します
coef = op[pauli_label("Z2 Y3")]
pauli_contrib = coef * pauli_exp
print(pauli_contrib)
```

```
(0.018000000000000002+0.006j)
```

各パウリ項についてこの手順を繰り返すことで、オリジナルの演算子の期待値が推定できます。

サンプリング推定のショートカットメソッド

上の手順は少し複雑なので、ショートカットできるメソッドが用意されています。まずは`CommutablePauliSetMeasurement`オブジェクトについて紹介します。このデータ構造は、以下のような要素を保持しています。

- `pauli_set`: 一緒に測定される、交換可能なパウリ演算子のセット
- `measurement_circuit`: あたえられた`pauli_set`を測定するための回路
- `pauli_reconstructor_factory`: サンプリング結果からパウリ演算子の値を再構築するためのファクトリ関数

`CommutablePauliSetMeasurement`を構築するには、特定の測定スキームを選択する必要があります。例えば、bitwise groupingを使用する場合は以下のようにします。

In [ ]:
```
from quri_parts.core.measurement import bitwise_commuting_pauli_measureme
measurements = bitwise_commuting_pauli_measurement(op)
print(f"Number of CommutablePauliSetMeasurement: {len(measurements)}")
measurement = measurements[0]
print(measurement.pauli_set)
print(measurement.measurement_circuit)
print(measurement.pauli_reconstructor_factory)
```

```
Number of CommutablePauliSetMeasurement: 5
frozenset({PauliLabel({(1, <SinglePauli.X: 1>), (3, <SinglePauli.Y: 2>)}
)})
(QuantumGate(name='H', target_indices=(1,), control_indices=(), params=(
), pauli_ids=(), unitary_matrix=()), QuantumGate(name='Sdag', target_ind
ices=(3,), control_indices=(), params=(), pauli_ids=(), unitary_matrix=(
)), QuantumGate(name='H', target_indices=(3,), control_indices=(), param
s=(), pauli_ids=(), unitary_matrix=()))
<function bitwise_pauli_reconstructor_factory at 0x7f2fe3686700>
```

推定のために必要なもう1つの入力は`PauliSamplingShotsAllocator`です。このオブジェクトはサンプリングの全体のショット数が各パウリセットに対してどのように割り振られるかを指定します。幾つかのアロケータが用意されています。

In [ ]:
```
from quri_parts.core.sampling.shots_allocator import (
    create_equipartition_shots_allocator,
    create_proportional_shots_allocator,
    create_weighted_random_shots_allocator,
)
# ショット数をパウリセットに均等に分配する
allocator = create_equipartition_shots_allocator()
# ショット数を演算子のパウリ係数に比例するように割り当てる
allocator = create_proportional_shots_allocator()
# ショット数をランダムな重みで分配する
allocator = create_weighted_random_shots_allocator(seed=777)
```

これらの入力値を使用して、以下のようにサンプリング推定を行うことができます。

In [ ]:
```
from quri_parts.qulacs.sampler import create_qulacs_vector_concurrent_sam
from quri_parts.core.estimator.sampling import sampling_estimate
concurrent_sampler = create_qulacs_vector_concurrent_sampler()
estimate = sampling_estimate(
    op,            # 期待値を推定する演算子
    initial_state, # 初期（回路）状態
    5000,          # 合計サンプリングショット数
    concurrent_sampler, # ConcurrentSampler
    bitwise_commuting_pauli_measurement, # CommutablePauliSetMeasurement0
    allocator,     # PauliSamplingShotsAllocator
)
print(f"Estimated expectation value: {estimate.value}")
print(f"Standard error of estimation: {estimate.error}")
```

```
Estimated expectation value: (0.7501670146137788-0.09211214014306741j)
Standard error of estimation: 0.07067193262543597
```

また、`QuantumEstimator`を使用してサンプリング推定を行うこともできます。

In [ ]:
```
from quri_parts.core.estimator.sampling import create_sampling_estimator
estimator = create_sampling_estimator(
    5000,          # 合計サンプリングショット数
    concurrent_sampler, # ConcurrentSampler
    bitwise_commuting_pauli_measurement, # CommutablePauliSetMeasurement
    allocator,     # PauliSamplingShotsAllocator
)
estimate = estimator(op, initial_state)
print(f"Estimated expectation value: {estimate.value}")
print(f"Standard error of estimation: {estimate.error}")
```

```
Estimated expectation value: (0.7637250737168275+0.029019686833330872j)
Standard error of estimation: 0.07036359239115449
```

Exercise 2

$-(X_0X_1 + Y_0Y_1 + Z_0Z_1)$という演算子を定義し、$|11\rangle$という状態に対して期待値推定を行ってみてください。`Estimator`は`create_qulacs_vector_estimator()`で作成してください。

In [ ]:
```

```

第3章 本物の量子計算機を使ってみよう

お知らせの通り、この章は別のノートを用意してあります。そちらをご参照ください。

第4章 変分アルゴリズム

(共通講義資料では、(P76) 6-4.Variational Quantum Eigensolver (VQE) をご参照ください。)

パラメータを持つ量子回路に対してパラメータの最適値を探索する、変分量子アルゴリズムと呼ばれる一群のアルゴリズムが、近年では活発に研究されています。この章ではまず演算子の期待値の勾配を計算し、続いてこの勾配を用いて変分量子固有値ソルバ (VQE) と呼ばれるアルゴリズムを構成する方法を説明したいと思います。また、量子回路学習 (QCL) と呼ばれる、量子回路を用いた機械学習アルゴリズムも紹介します。

演算子の期待値の勾配

変分アルゴリズムではしばしば特定のコスト関数の最小化が目的とされますが、こうしたコスト関数は特定のパラメータ付き量子回路の演算子の期待値として定義されます。
$f(\theta) = \langle O \rangle_\theta = \langle \psi(\theta) | O | \psi(\theta) \rangle$ ただし O は演算子、$\psi(\theta)$ は $\theta = \theta_0, \ldots, \theta_{m-1}$ によるパラメータ状態です。こうした最小化では、以下のようなコスト関数の勾配がしばしば用いられます。

$$\nabla_\theta f(\theta) = \left(\frac{\partial \langle O \rangle_\theta}{\partial \theta_0}, \ldots, \frac{\partial \langle O \rangle_\theta}{\partial \theta_{m-1}} \right)$$

以下ではこうした勾配を計算するための、数値勾配とパラメータシフトルールという2つの方法を見ていきましょう。

まずは対象の演算子とパラメータ状態を準備します。

In [ ]:
```
from quri_parts.core.operator import Operator, pauli_label

op = Operator({
    pauli_label("X0 Y1"): 0.5 + 0.5j,
    pauli_label("Z0 X1"): 0.2,
})

from math import pi
from quri_parts.circuit import LinearMappedUnboundParametricQuantumCircui

param_circuit = LinearMappedUnboundParametricQuantumCircuit(2)
param_circuit.add_H_gate(0)
param_circuit.add_CNOT_gate(0, 1)

theta, phi = param_circuit.add_parameters("theta", "phi")
param_circuit.add_ParametricRX_gate(0, {theta: 1/2, phi: 1/3, CONST: pi/2
param_circuit.add_ParametricRZ_gate(1, {theta: 1/3, phi: -1/2, CONST: -pi

from quri_parts.core.state import ParametricCircuitQuantumState

param_state = ParametricCircuitQuantumState(2, param_circuit)
```

数値勾配

QURI Partsでは勾配推定のため
に `quri_parts.core.estimator.GradientEstimator` インタフェースが用意されています。数値微分を使用したシンプルな勾配推定は以下のように行います。

In [ ]:
```
from quri_parts.core.estimator.gradient import create_numerical_gradient_
from quri_parts.qulacs.estimator import create_qulacs_vector_concurrent_p

qulacs_concurrent_parametric_estimator = create_qulacs_vector_concurrent_
gradient_estimator = create_numerical_gradient_estimator(
    qulacs_concurrent_parametric_estimator,
    delta=1e-4,
)

gradient = gradient_estimator(op, param_state, [0.2, 0.3])
print("Estimated gradient:", gradient.values)
```

```
Estimated gradient: [(0.0004866565750383245-0.013872819366045341j), (0.0
42165661391369014+0.020809229047680233j)]
```

勾配推定器の作成時には2つの引数をあたえます。1つ目は `ConcurrentParametricQuantumEstimator` でわずかにシフトした変数に対する期待値推定を行うために使用されます。2つ目は `delta` で数値微分の刻み幅を指定します。勾配推定器は演算子、パラメータ状態、勾配を評価するパラメータ値を用いて実行され、勾配の推定値を返します。

パラメータシフトルールを用いた勾配評価 (Skip)

パラメータシフトルールは、パラメータ回路によって生成された状態に対する演算子の期待値の勾配を評価する方法です[1]。 QURI Partsではパラメータシフトルールによって勾配を推定する方法も提供しています。ここではパラメータ回路に含まれるパラメータゲートはすべて $\exp\left(-i\theta P/2\right)$ ただし P はパウリ積という状態を仮定しています。QURI Partsによって定義されるパラメータゲートはすべてこの条件を満たしています。また、ゲートのパラメータは回路のパラメータに線型に依存していることも仮定しています。

[1]: Mitarai, K. and Negoro, M. and Kitagawa, M. and Fujii, K., Phys. Rev. A **98**, 032309 (2018). arXiv:1803.00745.

> ここで使用されているAPIは実験的なものであり、将来的に変更される可能性があります。また、以下の例は現状では `LinearMappedUnboundParametricQuantumCircuit` のみで機能します。

パラメータシフトルールで勾配を評価する際は、各パラメータゲートのパラメータは、もし同じ回路パラメータに依存していたとしても、独立にシフトする必要があります。また、各ゲートパラメータは回路パラメータに対して微分する必要があるため、微分の連鎖律を使用します。ゆえに以下の要素が必要になります。

- 各ゲートのパラメータが独立に扱われる、パラメータ回路 (QURI Partsでは `UnboundParametricQuantumCircuit`)
- パラメータは、各回路パラメータに対する各ゲートパラメータに対してシフトする
- 微分係数は各パラメータシフトに対応する

これは、以下のような関数で計算できます。

In [ ]:
```
from quri_parts.circuit.parameter_shift import ShiftedParameters
from quri_parts.core.state import ParametricCircuitQuantumState

def get_raw_param_state_and_shifted_parameters(state, params):
    param_mapping, raw_circuit = state.parametric_circuit.mapping_and_raw
    parameter_shift = ShiftedParameters(param_mapping)
    derivatives = parameter_shift.get_derivatives()
    shifted_parameters = [
        d.get_shifted_parameters_and_coef(params) for d in derivatives
    ]

    raw_param_state = ParametricCircuitQuantumState(state.qubit_count, ra

    return raw_param_state, shifted_parameters

# 例
raw_state, shifted_params_and_coefs = get_raw_param_state_and_shifted_par
    param_state, [0.2, 0.3]
)

for i, params_and_coefs in enumerate(shifted_params_and_coefs):
    print(f"Parameter shifts for circuit parameter {i}:")
    for p, c in params_and_coefs:
        print(f"  gate params: {p}, coefficient: {c}")
```

```
Parameter shifts for circuit parameter 0:
  gate params: (1.7707963267948965, -0.08333333333333326), coefficient:
0.16666666666666666
  gate params: (3.3415926535897933, -1.6541296601282298), coefficient: 0
.25
  gate params: (0.19999999999999996, -1.6541296601282298), coefficient:
-0.25
  gate params: (1.7707963267948965, -3.224925986923126), coefficient: -0
.16666666666666666
Parameter shifts for circuit parameter 1:
  gate params: (1.7707963267948965, -0.08333333333333326), coefficient:
-0.25
  gate params: (3.3415926535897933, -1.6541296601282298), coefficient: 0
.16666666666666666
  gate params: (0.19999999999999996, -1.6541296601282298), coefficient:
-0.16666666666666666
  gate params: (1.7707963267948965, -3.224925986923126), coefficient: 0.
25
```

続いて以下に示す通り、1) 各シフトパラメータに対して、演算子の期待値を推定する、2) それを対応する係数を掛けて足し合わせる、ことによって勾配が計算できます。

In [ ]:
```python
from quri_parts.qulacs.estimator import create_qulacs_vector_concurrent_p

def get_parameter_shift_gradient(op, raw_state, shifted_params_and_coefs)
    # 評価対象のゲートパラメータを集める
    gate_params = set()
    for params_and_coefs in shifted_params_and_coefs:
        for p, _ in params_and_coefs:
            gate_params.add(p)
    gate_params_list = list(gate_params)

    # Parametric estimatorを準備する
    estimator = create_qulacs_vector_concurrent_parametric_estimator()

    # 期待値を推定する
    estimates = estimator(op, raw_state, gate_params_list)
    estimates_dict = dict(zip(gate_params_list, estimates))

    # 係数を掛けながら期待値を足し上げる
    gradient = []
    for params_and_coefs in shifted_params_and_coefs:
        g = 0.0
        for p, c in params_and_coefs:
            g += estimates_dict[p].value * c
        gradient.append(g)

    return gradient

# 例
gradient = get_parameter_shift_gradient(op, raw_state, shifted_params_and
print("Estimated gradient:", gradient)
```

```
Estimated gradient: [(0.0004866565766964703 3-0.013872819366718303j), (0.
04216566140053679+0.020809229050077496j)]
```

これらの関数は以下のように、`GradientEstimator`としてまとめることができます。

In [ ]:
```python
from collections.abc import Sequence
from dataclasses import dataclass

# GradientEstimatorの戻り値
@dataclass
class _Estimates:
    values: Sequence[complex]
    error_matrix = None

def parameter_shift_gradient_estimator(op, state, params):
    raw_state, shifted_params_and_coefs = get_raw_param_state_and_shifted
    gradient = get_parameter_shift_gradient(op, raw_state, shifted_params
    return _Estimates(gradient)

# 例
gradient = parameter_shift_gradient_estimator(op, param_state, [0.2, 0.3]
print("Estimated gradient:", gradient.values)
```

```
Estimated gradient: [(0.0004866565766964703 3-0.013872819366718303j), (0.
04216566140053679+0.020809229050077496j)]
```

変分量子固有値ソルバ (VQE)

変分量子固有値ソルバ (VQE) はパラメータを持つ量子状態に対して、演算子の期待値 (例えば分子のエネルギーなど) を最適化する手法です。

分子や物質の性質の一部は、以下のようなシュレディンガー方程式を解くことで、明らかにできます。

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

ここで、Hはハミルトニアンと呼ばれる演算子 (行列) で、分子の形など、系によって決まっています。シュレディンガー方程式を解くことは、ハミルトニアンHの固有値問題を解き、固有値E_iと対応する固有ベクトル (固有状態) $|\phi_i\rangle$を求めることと同値です。この時固有値E_iは固有状態$|\phi_i\rangle$のエネルギーとなります。

特殊な状況を除いて、電子の状態は基底状態にあることがほとんどなので、固有状態の中でも特に基底状態に興味を持たれることが多いです。変分法では、任意の状態$|\psi\rangle$について、そのエネルギー期待値が基底エネルギーE_0以上となることを利用します。

$$\langle\psi|H|\psi\rangle \geq E_0$$

仮にランダムに状態$\{|\psi_i\rangle\}$を用意し、その中で一番エネルギーが低い状態を見つければ、それは$\{|\psi_i\rangle\}$の中では、最も基底状態に近い状態になるでしょう。実際には、ランダムに状態を取ってくるのは効率がわるいので、経験的にパラメータ付きの量子状態$|\psi(\theta)\rangle$を用意し、エネルギー期待値を最小化するようなθを見つけるという方法がとられています。

VQEは、変分法において、量子計算機で効率的に記述できる量子状態を用いて基底状態を探索するアルゴリズムです。VQEの実行手順は以下の通りです。

1. 量子計算機上で量子状態$|\psi(\theta)\rangle$を生成する
2. $\langle H(\theta)\rangle = \langle\phi(\theta)|H|\psi(\theta)\rangle$を測定する
3. 測定結果をもとに、古典計算機で$\langle\psi(\theta)|H|\psi(\theta)\rangle$が小さくなるような$\theta$を求める

この手順を$\langle\psi(\theta)|H|\psi(\theta)\rangle$が収束するまで繰り返すことで、近似的な基底状態を求めます。

VQEには大きく2つの部品があります。

- *Ansatz*: パラメータ化された量子状態を生成するための、パラメータ付きの量子回路
- *Optimizer*: 数値的に演算子の期待値を最適化する方法

Ansatz

VQEの文脈では、ansatzは対象の演算子の期待値を評価するための、パラメータ化された量子状態を生成するための、パラメータ化された量子回路を指します。`(LinearMapped)UnboundParametricQuantumCircuit`を直接定義することもできますし、`quri_parts.algo.ansatz`パッケージに含まれる、よく使われるansatzを使用することもできます。ここでは、例としてhardware-efficient ansatz [1] を使用しています。

[1]: Kandala, A., Mezzacapo, A., Temme, K. et al. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. Nature **549**, 242–246 (2017).

In [ ]:
```
from quri_parts.algo.ansatz import HardwareEfficient

hw_ansatz = HardwareEfficient(qubit_count=4, reps=3)
```

期待値を評価するためにはパラメータ化された量子状態が必要ですが、これは初期状態にansatzを適用することで得られます。ここでは計算基底状態 $|0011\rangle$ を使ってみましょう。

In [ ]:
```
from quri_parts.core.state import ComputationalBasisState, ParametricCirc

# 現状ではパラメータ化された量子状態の準備はすこし複雑になっています
def prepare_parametric_state(initial_state, ansatz):
    circuit = LinearMappedUnboundParametricQuantumCircuit(initial_state.q
    circuit += initial_state.circuit
    circuit += ansatz
    return ParametricCircuitQuantumState(initial_state.qubit_count, circu

cb_state = ComputationalBasisState(4, bits=0b0011)
parametric_state = prepare_parametric_state(cb_state, hw_ansatz)
```

Optimizer

オプティマイザはコスト関数を最小化するような最適なパラメータを探索します。VQEの文脈ではコスト関数は対象の演算子の期待値が使用されます。オプティマイザの種類によって、コスト関数それ自体のみが使用されたり、コスト関数の勾配が使用されたりします。`scipy.optimize`ライブラリや`quri_parts.algo.optimizer`パッケージのオプティマイザを自由に使用することができます。この例ではAdam [1] を使用しています。

[1]: Diederik P. Kingma, Jimmy Ba, Adam: A Method for Stochastic Optimization. arXiv:1412.6980 (2014)

In [ ]:
```
from quri_parts.algo.optimizer import Adam

# 引数で設定を調整することができます。詳細はリファレンスをご参照ください。
adam_optimizer = Adam()
```

VQEを実行する

まずは対象の演算子を定義します。この演算子の期待値が最適化の対象になります。

In [ ]:
```
from quri_parts.core.operator import Operator, pauli_label, PAULI_IDENTIT

# Jordan-Wigner変換した水素分子のハミルトニアンを例として使用します
hamiltonian = Operator({
    PAULI_IDENTITY: 0.03775110394645542,
    pauli_label("Z0"): 0.18601648886230593,
    pauli_label("Z1"): 0.18601648886230593,
    pauli_label("Z2"): -0.2694169314163197,
    pauli_label("Z3"): -0.2694169314163197,
    pauli_label("Z0 Z1"): 0.172976101307451,
    pauli_label("Z0 Z2"): 0.12584136558006326,
    pauli_label("Z0 Z3"): 0.16992097848261506,
    pauli_label("Z1 Z2"): 0.16992097848261506,
    pauli_label("Z1 Z3"): 0.12584136558006326,
    pauli_label("Z2 Z3"): 0.17866777775953396,
    pauli_label("X0 X1 Y2 Y3"): -0.044079612902551774,
    pauli_label("X0 Y1 Y2 X3"): 0.044079612902551774,
    pauli_label("Y0 X1 X2 Y3"): 0.044079612902551774,
    pauli_label("Y0 Y1 X2 X3"): -0.044079612902551774,
})
```

この演算子とパラメータ化された状態を使用して、回路パラメータの関数としてコスト関数を定義できます。

In [ ]:
```
from quri_parts.qulacs.estimator import create_qulacs_vector_parametric_e

estimator = create_qulacs_vector_parametric_estimator()

def cost_fn(param_values):
    estimate = estimator(hamiltonian, parametric_state, param_values)
    return estimate.value.real
```

また数値勾配を使用して、コスト関数の勾配を定義します。

In [ ]:
```
import numpy as np
from quri_parts.core.estimator.gradient import create_numerical_gradient_
from quri_parts.qulacs.estimator import create_qulacs_vector_concurrent_p

qulacs_concurrent_parametric_estimator = create_qulacs_vector_concurrent_
gradient_estimator = create_numerical_gradient_estimator(
    qulacs_concurrent_parametric_estimator,
    delta=1e-4,
)

def grad_fn(param_values):
    estimate = gradient_estimator(hamiltonian, parametric_state, param_va
    return np.asarray([g.real for g in estimate.values])
```

それでは、QURI Partsのオプティマイザを使用してVEQを実行してみましょう。

In [ ]:
```
from quri_parts.algo.optimizer import OptimizerStatus

def vqe(operator, init_params, cost_fn, grad_fn, optimizer):
    opt_state = optimizer.get_init_state(init_params)
    while True:
        opt_state = optimizer.step(opt_state, cost_fn, grad_fn)
        if opt_state.status == OptimizerStatus.FAILED:
            print("Optimizer failed")
            break
        if opt_state.status == OptimizerStatus.CONVERGED:
            print("Optimizer converged")
            break
    return opt_state

init_params = [0.1] * hw_ansatz.parameter_count
result = vqe(hamiltonian, init_params, cost_fn, grad_fn, adam_optimizer)
print("Optimized value:", result.cost)
print("Optimized parameter:", result.params)
print("Iterations:", result.niter)
print("Cost function calls:", result.funcalls)
print("Gradient function calls:", result.gradcalls)
```

```
Optimizer converged
Optimized value: -1.11198134059955
Optimized parameter: [ 5.47178291e-02  8.40762191e-02  5.12253346e-02  8
.19750455e-02
 -9.72099552e-03 -1.16141816e-01 -3.06727503e-03  9.66792839e-01
  1.27323903e-01  1.04790840e-01  1.27097745e-01  9.40512846e-02
 -1.60419273e-02  9.92326575e-01 -3.35897820e-02  9.91027219e-01
  6.44048147e-02  2.49935143e-04  6.43611653e-02 -5.72090665e-03
 -1.48640070e-02 -1.16555429e-01 -3.59503991e-02  9.79005523e-01
  1.67652638e-02 -2.35033577e-01  1.34115104e-02 -2.24492671e-01
 -2.91851967e-02  4.35033433e-01 -3.52284759e-03  4.24492881e-01]
Iterations: 24
Cost function calls: 25
Gradient function calls: 24
```

SciPyのオプティマイザを使用してVQEを実行することもできます。

In [ ]:
```
from scipy.optimize import minimize

def vqe_scipy(operator, init_params, cost_fn, grad_fn, method, options):
    return minimize(cost_fn, init_params, jac=grad_fn, method=method, opt

init_params = [0.1] * hw_ansatz.parameter_count
bfgs_options = {
    "gtol": 1e-6,
}
result = vqe_scipy(hamiltonian, init_params, cost_fn, grad_fn, "BFGS", bf
print(result.message)
print("Optimized value:", result.fun)
print("Optimized parameter:", result.x)
print("Iterations:", result.nit)
print("Cost function calls:", result.nfev)
print("Gradient function calls:", result.njev)
```

```
Optimization terminated successfully.
Optimized value: -1.1299047843085266
Optimized parameter: [ 4.87553205e-04  4.56074903e-02  6.61171239e-01  2
.02766648e-03
  3.12909972e-01 -4.22660254e-02 -1.39132536e+00 -6.93360425e-04
  3.36873792e-01  5.34139319e-05  6.57235219e-01 -2.91003674e-01
  6.78450243e-01  1.14192251e-01  2.19148540e+00 -1.42766239e-03
  1.57057211e+00 -6.99471236e-07 -2.75616085e-04  1.81141451e-01
  1.79719554e-01 -1.41103229e-01 -2.29963858e-01  1.33017500e-02
  1.23408687e+00  1.10949475e-01 -3.70379541e-04  1.10970471e-01
 -5.20762503e-01  8.90505617e-02 -6.27526106e-01  8.90295061e-02]
Iterations: 175
Cost function calls: 181
Gradient function calls: 181
```

量子回路学習 (QCL)

近年、機械学習の分野では、ディープラーニングが脚光を浴びています。ディープラーニングでは、生物の神経細胞のネットワークをモデル化したニューラルネットワークに対して、大規模な入出力データによって学習を行うことで、複雑な関数の近似を行い、新しいデータに対して予測ができるようになります。

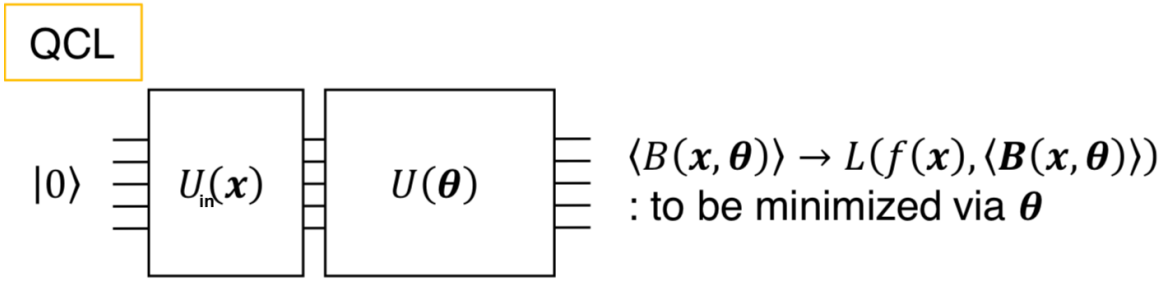
QCLはこのニューラルネットワークを量子回路に置き換えた機械学習手法です。量子回路を用いることで、指数関数的に多数の基底関数を用いて学習でき、モデルの表現力が向上し、また量子回路の性質により自動的に過学習を防げると期待されています。

[1]: K. Mitarai, M. Negoro, M. Kitagawa, and K. Fujii, “Quantum circuit learning”, Phys. Rev. A 98, 032309 (2018), https://arxiv.org/abs/1803.00745

ニューラルネットワークでは、各層の重みパラメータを調整することで関数の近似を行っていますが、QCLでも考え方は同様です。QCLで使用する量子回路には複数の回転ゲートが含まれていますが、この回転ゲートの回転角を調整することで関数の近似を行います。

学習の手順

1. 学習データ$\{(x_i, y_i)\}_i$を用意する (x_iは入力データ、y_iはx_iから予測したい出力データ)
2. $U_{in}(x)$という、入力xから何らかの規則で決まる回路を用意し、x_iの情報を埋め込んだ入力状態$\{|\psi_{in}(x_i)\rangle\} = \{U_{in}(x_i)|0\rangle\}_i$を作る
3. 入力状態に、パラメータΘに依存した回路$U(\theta)$を掛けたものを出力状態$\{|\psi_{out}(x_i, \theta)\rangle = U(\theta)|\psi_{in}(x_i)\rangle\}_i$とする
4. 出力状態のもとで何らかのオブザーバブルを測定し、期待値を得る (例: 1番目の量子ビットのZの期待値$\langle Z_1\rangle = \langle\phi_{out}|Z_1|\psi_{out}\rangle)$
5. Fを適当な関数 (sigmoidやsoftmax、定数倍など) として、$F(\text{測定値}_i)$をモデルの出力$y(x_i, \theta)$とする
6. 正解データ$\{y_i\}_i$とモデルの出力$\{y(x_i, \theta)\}_i$の間の乖離を表すコスト関数$L(\theta)$を計算する
7. コスト関数を最小化する$\theta = \theta*$を求める
8. $y(x, \theta*)$が所望の予測モデルとなる

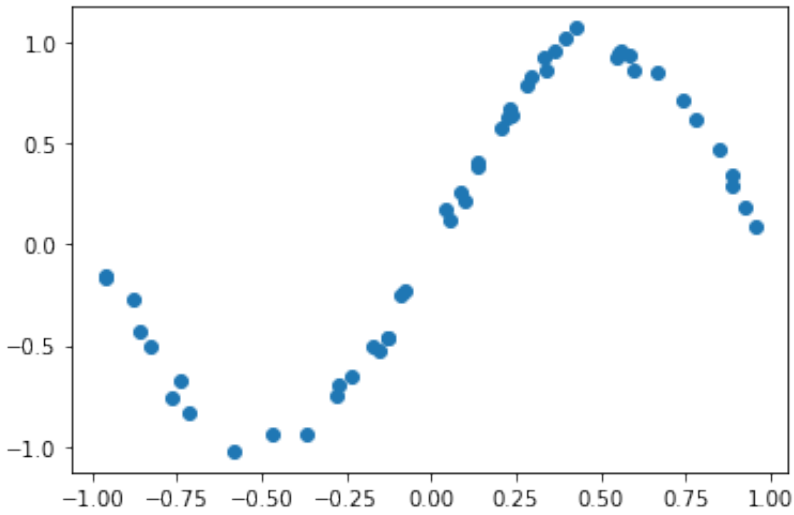


学習データの準備

ここでは、デモンストレーションとして$y = sin(\pi x)$のフィッティングを行ってみましょう。元の関数に乱数でノイズを加えたデータ点を用意し、これを学習データとして使用してみます。

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)  # 再現性のため乱数を固定

x_train = np.random.rand(50) * 2.0 - 1.0
y_train = np.sin(x_train * np.pi)
y_train += 0.05 * np.random.randn(y_train.size)

plt.plot(x_train, y_train, "o")
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7f2fe0033cd0>]
```



入力状態の構成

まずは、入力値x_iを初期状態$|00\ldots0\rangle$に埋め込むための回路$U_{in}(x_i)$を作成します。参考文献[1]に従って、$U_{in}(x) = \prod_j R_j^Z(\cos^{-1} x^2) R_j^Y(\sin^{-1} x)$と定義します。入力値$x_i$は$|\psi_{in}(x_i)\rangle = U_{in}(x_i)|00\ldots0\rangle$という量子状態に変換されます。

In [ ]:
```python
from quri_parts.circuit import LinearMappedUnboundParametricQuantumCircui

def gen_U_in(qubit_count: int) -> LinearMappedUnboundParametricQuantumCir
    circuit = LinearMappedUnboundParametricQuantumCircuit(qubit_count)
    angle_y, angle_z = circuit.add_parameters("angle_y", "angle_z")

    for i in range(qubit_count):
        circuit.add_ParametricRY_gate(i, angle_y)
        circuit.add_ParametricRZ_gate(i, angle_z)

    return circuit
```

変分量子回路の構成

次に、最適化される変分量子回路$U(\theta)$を作ります。パラメータθの最適な値が学習されることで、回路が目的の関数を再現できるようになります。ここでは$U(\theta)$は以下の手順で作成します。

1. 横磁場イジングハミルトニアンの作成
2. ゲートの作成と回路への追加

1. 横磁場イジングハミルトニアンの作成

量子回路の複雑性 (エンタングルメント) を増すことで、モデルの表現力を高めるため、横磁場イジングハミルトニアンによるユニタリ演算子を用意し、回路に追加します。

横磁場イジングモデルのハミルトニアンは以下の通りで、$U_{rand} = e^{-iHt}$という時間発展演算子を定義します。(詳細は量子ダイナミクスのパートで扱います。)

$$H = \sum_{j=1}^{N} a_j X_j + \sum_{j=1}^{N} \sum_{k=1}^{j-1} J_{jk} Z_j Z_k$$

ここで係数a, Jは$[-1, 1]$の一様分布乱数です。

In [ ]:
```
import functools

def make_fullgate(site_and_ops, qubit_count):
    i_mat = [[1, 0], [0, 1]]

    sites = [sop[0] for sop in site_and_ops]
    single_gates = []
    c = 0
    for i in range(qubit_count):
        if i in sites:
            single_gates.append(site_and_ops[c][1])
            c += 1
        else:
            single_gates.append(i_mat)
    return functools.reduce(np.kron, single_gates)


def gen_time_evol_unitary(qubit_count: int, time_step: float = 0.77):
    x_mat = [[0, 1], [1, 0]]
    z_mat = [[1, 0], [0, -1]]

    ham = np.zeros((2**qubit_count, 2**qubit_count), dtype=complex)
    for i in range(qubit_count):
        jx = -1 + 2 * np.random.rand()
        ham += jx * make_fullgate([[i, x_mat]], qubit_count)
        for j in range(i + 1, qubit_count):
            j_ij = -1 + 2 * np.random.rand()
            ham += j_ij * make_fullgate([[i, z_mat], [j, z_mat]], qubit_c

    diag, eigen_vecs = np.linalg.eigh(ham)
    return np.dot(
        np.dot(eigen_vecs, np.diag(np.exp(-1j * time_step * diag))), eige
    )
```

ゲートのユニタリ演算子を表現する行列が作成されます。

2. 回転ゲートの追加

上で作成したU_{rand}と、各量子ビットに回転ゲートをかけたU_{rot}を組み合わせて、変分量子回路$U(\theta)$を構成します。

$$U_{rot}(\theta_j^{(i)}) = R_j^X(\theta_{j1}^{(i)})R_j^Z(\theta_{j2}^{(i)})R_j^X(\theta_{j3}^{(i)})$$

ここではiは量子回路の層を表す添字で、jは量子ビットの番号です。U_{rand}とU_{rot}を一定の回数繰り返すので、全体では以下のような回路になります。

$$U\left(\{\theta_j^{(i)}\}_{i,j}\right) = \prod_{i=1}^{d}\left(\left(\prod_{j=1}^{n} U_{rot}(\theta_j^{(i)})\right)U_{rand}\right)$$

パラメータ数は3nd個になります。

In [ ]:
```
from quri_parts.circuit import UnboundParametricQuantumCircuit

def gen_U_out(
    qubit_count: int, reps: int
) -> UnboundParametricQuantumCircuit:
    unitary = gen_time_evol_unitary(qubit_count)

    circuit = UnboundParametricQuantumCircuit(qubit_count)
    for _ in range(reps):
        circuit.add_UnitaryMatrix_gate(tuple(range(qubit_count)), unitary
        for i in range(qubit_count):
            circuit.add_ParametricRX_gate(i)
            circuit.add_ParametricRZ_gate(i)
            circuit.add_ParametricRX_gate(i)

    return circuit
```

予測関数の作成

ここまでの部品を組み合わせて、あたえられたthetasと、データ点xs、U_in、U_outに対して、オペレータの期待値を返すような関数を作ってみましょう。

ここでは、ある特定のthetasに対して、各データ点の予測値をすべて計算して返すような構成にしてみます。1. まず、U_outに対してthetasをバインドし、さらにU_inと連結することで、1つの量子回路にします。2. 次に、ParametricQuantumEstimatorを使って、各データ点に対する期待値を計算していきます。3. 最後に、得られた期待値を取り出して返します。

オブザーバブルには2 * Zを設定しています。2をかけているのは$\langle Z \rangle$の値域を広げるためです。未知の関数に対応するためには、この定数もパラメータとして最適化する必要があります。

In [ ]:
```
from quri_parts.core.operator import pauli_label, Operator
from quri_parts.core.state import ParametricCircuitQuantumState
from quri_parts.qulacs.estimator import create_qulacs_vector_parametric_e

def qcl_preds(thetas, xs, U_in, U_out):
    op = Operator({pauli_label("Z0"): 2})

    circuit = U_in + U_out.bind_parameters(thetas)
    state = ParametricCircuitQuantumState(3, circuit)

    estimator = create_qulacs_vector_parametric_estimator()
    estimates = [estimator(op, state, (np.arcsin(x), np.arccos(x**2))) fo

    return [e.value.real for e in estimates]
```

コスト関数の作成、学習

コスト関数$L(\theta)$には、教師データと予測値の平均二乗誤差 (MSE) を使用してみます。

いよいよ学習を行います。ここでは簡単のため、勾配の計算式をあたえる必要のないNelder-Mead法を用いて最適化します。勾配を用いる最適化手法 (BFGS法など) を使う場合は、勾配の計算式が参考文献[1]に紹介されているので、参照してください。

In [ ]:
```
import scipy

qubit_count = 3
reps = 3
U_in = gen_U_in(qubit_count)
U_out = gen_U_out(qubit_count, reps)

def cost_func(thetas):
    y_pred = qcl_preds(thetas, x_train, U_in, U_out)
    L = ((y_pred - y_train) ** 2).mean()
    return L

theta_init = 2 * np.pi * np.random.rand(qubit_count * reps * 3)
result = scipy.optimize.minimize(cost_func, theta_init, method="Nelder-Me
theta_opt = result.x
print(theta_opt)
```

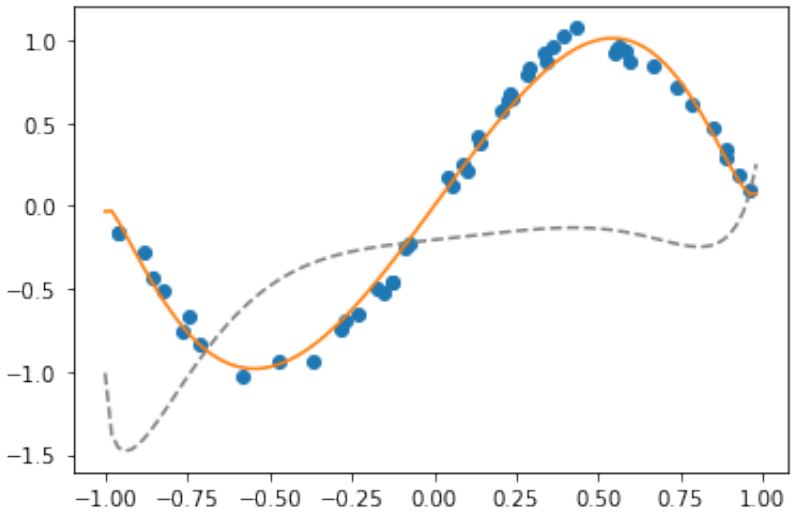
```
[4.85759345 3.61092621 1.69189655 3.44158295 0.05680401 1.98713228
 5.23384759 1.91278537 4.52216681 2.82392333 1.07333744 1.91661789
 3.75494355 2.86200065 3.48937179 3.95942674 3.72667005 3.27249143
 4.60579436 3.02948241 2.96250716 5.0571217  5.2012388  5.42962805
 0.993223   4.18945599 4.45484048]
```

結果のプロット

sin関数の近似に成功しました。ここでは入出力ともに1次元のとてもシンプルなタスクを扱いましたが、多次元の関数の近似や分類問題にも拡張が可能です。興味のある方は、Quantum Native Dojo 5.2c などもご参照ください。

In [ ]:
```
plt.figure()
plt.plot(x_train, y_train, "o", label="Train Data")

xlist = np.arange(-1.0, 1.0, 0.02)
y_init = qcl_preds(theta_init, xlist, U_in, U_out)
plt.plot(xlist, y_init, "--", label="Initial Model Prediction", c="gray")

y_pred = qcl_preds(theta_opt, xlist, U_in, U_out)
plt.plot(xlist, y_pred, label="Final Model Prediction")
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x7f2fd96c54f0>]
```



Exercise 4

以下のリストは2002年7月から2022年12月までのドル円為替レートです。このデータを使って、(前述のコードがそのまま使えるように) x_trainの範囲が [-1,1]、y_trainの範囲が [0,1] になるように正規化して、QCLで学習してみてください。

In [ ]:
```
data = np.array([119.82, 119.37, 118.63, 108.99, 103.95, 109.86, 103.58,
```

In [ ]:
```

```

第5章 ハミルトニアンダイナミクス

(共通講義資料では、(P69) 6-2.量子コンピュータを用いたハミルトニアンダイナミクス (量子系の実時間発展) をご参照ください。)

量子コンピュータの応用に「ハミルトニアンを用いた量子系のダイナミクスのシミュレーション」があります。量子系はシュレーディンガー方程式

$$i\frac{\partial|\psi(t)\rangle}{\partial t} = H|\psi(t)\rangle$$

に従い時間発展するので、量子系のダイナミクスをシミュレーションするには、シュレーディンガー方程式を数値的に解く必要があります。ここで H は考えている量子系のハミルトニアン、$|\psi(t)\rangle$ は量子状態を表します。

ハミルトニアンが時間に依存しない場合は、シュレーディンガー方程式は形式的に解くことができて

$$|\psi(t)\rangle = e^{-iHt}|\psi(0)\rangle$$

となります。よって、初期状態 $|\psi(0)\rangle$ に時間発展を表すユニタリー演算子 $U_H(t) = e^{-iHt}$ を作用させることで量子系のダイナミクスをシミュレーションできます。

ハミルトニアンが対角化できる場合

ハミルトニアンが対角化できる場合は、計算は簡単になります。ハミルトニアンの固有状態 $|\phi_i\rangle$ と固有値 E_i は次の式を満たします

$$H|\phi_i\rangle = E_i|\phi_i\rangle.$$

この固有状態に時間発展演算子を作用させると

$$e^{-iHt}|\phi_i\rangle = e^{-iE_it}|\phi_i\rangle.$$

となるので、初期状態を

$$|\psi(0)\rangle = \sum_{i=0}^{N} c_i|\phi_i\rangle$$

と展開することで、時刻 t での状態は

$$|\psi(t)\rangle = \sum_{i=0}^{N} c_i e^{-iHt}|\phi_i\rangle = \sum_{i=0}^{N} c_i e^{-iE_it}|\phi_i\rangle$$

つまり、各固有状態に対応する位相 e^{-iE_it} をかけて足し合わせることでシミュレーションできます。

しかし一般にハミルトニアン H の次元は考えている量子系に関係する自由度の数で指数関数的に大きくなります。例えば n qubit系を考えるとハミルトニアンは 2^n 次元となり、比較的小さな n でも古典コンピュータで対角化することは困難です。

トロッター分解を用いた量子系のシミュレーション

ハミルトニアンが特定の構造を持つ場合、量子コンピュータは $U_H(t)$ を効率よく計算できることが知られています。その場合に用いられるのがトロッター分解です。トロッター分解とは、正方行列 A, B の和の指数関数を、それぞれの指数関数の積に近似する公式です。

$$e^{\delta(A+B)} = e^{\delta A}e^{\delta B} + \mathcal{O}(\delta^2)$$

ハミルトニアンが $H = \sum_{k=1}^{L} H_k$ のように複数の部分ハミルトニアンの和に分解できるとき、この公式を使って時間発展演算子を次のように分解することができます

$$e^{-iHt} = \left(e^{-i\sum_k H_k \frac{t}{M}}\right)^M = \left(e^{-iH_1\frac{t}{M}} \cdot e^{-iH_2\frac{t}{M}} \dots\right)^M + \mathcal{O}\left(\left(\frac{t}{M}\right)^2\right).$$

ここで M は近似の精度 $\mathcal{O}((\frac{t}{M})^2)$ が十分小さくなるよう選ぶことができます。このようにトロッター分解を使うことで、時間発展演算子を部分ハミルトニアンの時間発展演算子 $U_{H_k}(t) = e^{-iH_kt}$ の積に書き直すことができます。量子回路で $U_{H_k}(t)$ を簡単に実装できる場合には、トロッター分解を用いて量子コンピュータで量子系のダイナミクスを効率よくシミュレーションすることができます。

まとめると、トロッター分解を用いて量子系のダイナミクスを効率よくシミュレーションするためにはハミルトニアンに対して次の条件が成り立つ必要があります。

- ハミルトニアンが $H = \sum_k H_k$ のように複数の部分ハミルトニアンに分解できる。
- 個々の H_k に対して、 $U_{H_k}(t) = e^{-iH_kt}$ が量子回路で簡単に実装できる。

幸いなことに、物理、量子化学の分野で興味のあるハミルトニアンは、大抵はこれらの条件を満たしていることが知られています。

ハイゼンベルグ模型のシミュレーション

それでは、具体的なハミルトニアンを使って量子コンピュータで量子系のダイナミクスをシミュレーションしてみましょう。今回は磁性体のモデルとしてよく使われる、1次元ハイゼンベルグ模型のハミルトニアンを使います。これは空間一列にスピンを持つ粒子が配置された系で、今回は簡単のため二つの粒子が存在する場合を考えます。具体的なハミルトニアンは

$$H = J(X_0X_1 + Y_0Y_1 + Z_0Z_1).$$

ここで $J < 0$ は結合定数、 X_i, Y_i, Z_i はそれぞれ i 番目のスピンに作用するパウリ演算子です。$J < 0$では強磁性と呼ばれる性質を持ち、基底状態は以下のスピン三重項です。

$$|\uparrow\uparrow\rangle,\ \frac{1}{\sqrt{2}}(|\uparrow\downarrow\rangle + |\downarrow\uparrow\rangle),\ |\downarrow\downarrow\rangle.$$

このハミルトニアンは上であげた条件を満たします。すなわち

- $H = \sum_{k=1}^{3} H_k, H_1 = X_0X_1, H_2 = Y_0Y_1, H_3 = Z_0Z_1$ と部分ハミルトニアンに分解できる。
- 後で詳しくみますが、$U_{H_k}(t) = e^{-iH_kt}$ は量子回路で簡単に実装できる形になっています。

今回は z 方向のスピン期待値の全系での平均値（z方向の全磁化）m_z という物理量の時間発展をシミュレーションしてみます。

$$m_z = \frac{1}{2}\sum_{j=0}^{1}\langle Z_j\rangle.$$

それぞれのスピンについて、$+z$ 方向に向いた状態を $|0\rangle$, $-z$ 方向に向いた状態を $|1\rangle$に対応づけ、初期状態は二つのスピンが$+z$ 方向に揃った状態 $|00\rangle$ とします。

In [ ]:
```
# 必要なパッケージをインポート
from quri_parts.circuit import PauliRotation, QuantumCircuit
from quri_parts.core.operator import Operator, pauli_label
from quri_parts.core.state import ComputationalBasisState, GeneralCircuit
from quri_parts.qulacs.estimator import create_qulacs_vector_estimator

import matplotlib.pyplot as plt
```

In [ ]:
```
# 今回は2スピン系なので2qubit系を考える。
nqubits = 2
# ハイゼンベルグ模型のパラメータ
J = -1
# ダイナミクスをシミュレーションする時間
t = 1.0
# トロッター分解の分割数
m = 100
# 時間の刻み幅
delta = t / m

# estimator
estimator = create_qulacs_vector_estimator()

# 全磁化に対応するオペレーターを用意する。
magnetization_op = Operator()
for i in range(nqubits):
    magnetization_op += Operator({pauli_label(f"Z{i}"): 1 / 2})

# 初期状態 |00> を準備する。
state = ComputationalBasisState(n_qubits=nqubits, bits=0)
print(state)
```

次にトロッター分解を考えます。時間発展演算子は

$$U_H(t) = \exp\left(-iJ(X_0X_1 + Y_0Y_1 + Z_0Z_1)t\right).$$

ハミルトニアンの各項は可換でないので、トロッター分解を用いると

$$U_H(t) \approx \left(\exp\left(-iJ\frac{t}{M}X_0X_1\right)\exp\left(-iJ\frac{t}{M}Y_0Y_1\right)\exp\left(-iJ\frac{t}{M}Z_0Z_1\right)\right)^M$$

と分解できるので、分割された各時間ステップ $\Delta t = \frac{t}{M}$ での近似的な時間発展は

$$U_{H,\Delta t} = \exp\left(-iJ\Delta t X_0X_1\right)\exp\left(-iJ\Delta t Y_0Y_1\right)\exp\left(-iJ\Delta t Z_0Z_1\right)$$

となります。

この演算子を量子回路で実装します。まず $\exp\left(-iJ\Delta t Z_0Z_1\right)$ を考えます。この演算子の2スピン系の４つの状態に対する作用は

$$\begin{aligned}
|00\rangle &\to e^{-iJ\Delta t}|00\rangle \\
|01\rangle &\to e^{iJ\Delta t}|01\rangle \\
|10\rangle &\to e^{iJ\Delta t}|10\rangle \\
|11\rangle &\to e^{-iJ\Delta t}|11\rangle
\end{aligned}$$

となっています。これは $[CNOT_{0,1}, RZ(2J\Delta t)_1, CNOT_{0,1}]$ という回路を考えると同じ作用をすることが確認できます。同様に、 $\exp\left(-iJ\Delta t X_0X_1\right)$ も $[H_0, H_1, CNOT_{0,1}, RZ(2J\Delta t)_1, CNOT_{0,1}, H_0, H_1]$ と書き直すことができます。（ $\exp\left(-iJ\Delta t Y_0Y_1\right)$ はどのようにすれば回路で実装できるか考えてみましょう。）

QURI Partsでは`PauliRotation gate`を提供しています。このゲートを使えば毎回上のような変換を自分で考えずとも、直接 $\exp\left(-iJ\Delta t Z_0Z_1\right)$ の形の演算子を量子回路で実装できます。

In [ ]:
```
# 例として、\exp{\left(-iJ\Delta tZ_0Z_1\right)}を考える。
e_zz_circuit = PauliRotation(target_indices=[0, 1], pauli_ids=[3, 3], ang

# 他も同様にかける。\exp{\left(-iJ\Delta tX_0X_1\right)}
e_xx_circuit = PauliRotation(target_indices=[0, 1], pauli_ids=[1, 1], ang
# \exp{\left(-iJ\Delta tY_0Y_1\right)}
e_yy_circuit = PauliRotation(target_indices=[0, 1], pauli_ids=[2, 2], ang
```

後はこれらの回路を使って実際にシミュレーションしてみましょう。

In [ ]:
```
circuit_trotter_heisenberg = QuantumCircuit(qubit_count=nqubits)

# 時間と磁化を記録するリスト
x = [i * delta for i in range(m + 1)]
y = []

# t = 0 の時の全磁化のみ先に計算
y.append(estimator(magnetization_op, state).value.real)

#t=0以降の全磁化を計算
for i in range(m):
    # delta = t / M だけ時間発展
    circuit_trotter_heisenberg.add_gate(e_xx_circuit)
    circuit_trotter_heisenberg.add_gate(e_yy_circuit)
    circuit_trotter_heisenberg.add_gate(e_zz_circuit)
    circuit_state = GeneralCircuitQuantumState(nqubits, circuit_trotter_h
    # 磁化を計算して記録
    y.append(round(estimator(magnetization_op, circuit_state).value.real,

# グラフの描画
plt.xlabel("time")
plt.ylabel("Value of magnetization")
plt.title("Dynamics of Heisenberg model")
plt.plot(x, y, "-")
plt.show()
```

この結果から分かるように、 z 方向の全磁化 m_z は一定になっています。つまりこの模型では m_z が保存量になっていることがわかります。実際に確かめるためには、ハミルトニアン $H = J(X_0X_1 + Y_0Y_1 + Z_0Z_1)$ と 全磁化の演算子 $\frac{1}{2}(Z_0 + Z_1)$ の交換関係を求めればわかります。興味のある方は各自で調べてみてください。

QURI Partsを用いたトロッター・スズキ分解

ちなみに、今回はトロッター分解を直接行いましたが、QURI Partsではより一般化されたトロッター・スズキ分解が実装されており、使用することができます。一般のパウリ演算子の和 $P = \sum_i P_i$ に係数 x をかけた演算子の指数関数 e^{xP} に対するトロッター・スズキ分解は再帰的に次のように定義されます。

$$S_{2k}(x) = [S_{2k-2}(p_k x)]^2 S_{2k-2}((1-4p_k)x)[S_{2k-2}(p_k x)]^2.$$

ここで

$$S_2(x) = \prod_{j=1}^{m} e^{P_j x/2} \prod_{j'=m}^{1} e^{P_{j'} x/2},$$

であり、 $p_k = (4 - 4^{1/(2k-1)})^{-1}$ です。近似は k が大きくなるにつれ精度は良くなり、一般的に

$$e^{-iHt} = \left(e^{-i\sum_k H_k \frac{t}{M}}\right)^M = S_{2k}(-it/M)^M + \mathcal{O}\left(\left(\frac{t}{M}\right)^{2k+1}\right)$$

となることが知られています。以下のようにQURI Partsを使うことで簡単にトロッター・スズキ分解を行うことができます。

In [ ]:
```
from quri_parts.core.operator import trotter_suzuki_decomposition
from quri_parts.core.circuit.exp_single_pauli_gate import convert_exp_sin

# 今回は2スピン系なので2qubit系を考える。
nqubits = 2
# ハイゼンベルグ模型のパラメータ
J = -1
# ダイナミクスをシミュレーションする時間
t = 1.0
# トロッター・スズキ分解の分割数
m = 100
# 時間の刻み幅
delta = t / m

# トロッター・スズキ分解を適用するハミルトニアンを準備する。
hamiltonian = Operator({pauli_label(f"X0 X1"): J})
hamiltonian += Operator({pauli_label(f"Y0 Y1"): J})
hamiltonian += Operator({pauli_label(f"Z0 Z1"): J})

# トロッター・スズキ分解を行う。今回は k=2 とする。
decomposed_hamiltonian = trotter_suzuki_decomposition(hamiltonian, -1 * d

# 出力は分解された各項のリストになっている。各項はパウリ演算子積の指数関数で、`Exponen
# 各項は`convert_exp_single_pauli_gate()`を使うことで簡単にゲートに書き換えること
gates = [convert_exp_single_pauli_gate(i.pauli, i.coefficient) for i in d


circuit_trotter_suzuki_heisenberg = QuantumCircuit(qubit_count=nqubits)

# 時間と磁化を記録するリスト
x = [i * delta for i in range(m + 1)]
y = []

# t = 0 の時の全磁化のみ先に計算
y.append(estimator(magnetization_op, state).value.real)

#t=0以降の全磁化を計算
for i in range(m):
    # delta = t / M だけ時間発展
    circuit_trotter_suzuki_heisenberg.extend(gates)
    circuit_state = GeneralCircuitQuantumState(nqubits, circuit_trotter_s
    # 磁化を計算して記録
    y.append(round(estimator(magnetization_op, circuit_state).value.real,

# グラフの描画
plt.xlabel("time")
plt.ylabel("Value of magnetization")
plt.title("Dynamics of Heisenberg model")
plt.plot(x, y, "-")
plt.show()
```

横磁場を加えた場合のハイゼンベルグ模型

これだけだと面白くないので、 x 軸方向の一様な磁場をかけた横磁場ハイゼンベルグ模型を考えてみます。

$$H = J(X_0X_1 + Y_0Y_1 + Z_0Z_1) + h(X_0 + X_1).$$

ここで h は横磁場の強さを表す係数です。この場合にトロッター分解を行うと、先ほどの回路に $e^{-ih\frac{t}{M}X_0}e^{-ih\frac{t}{M}X_1}$ という項も加えて実装する必要があります。ただしこの場合は RX ゲートを使うだけで実装できるので難しくはありません。

In [ ]:
```
# 同様に2qubit系を考える。
nqubits = 2
# ハイゼンベルグ模型のパラメータ
J = -1
# ダイナミクスをシミュレーションする時間
t = 3.0
# トロッター分解の分割数
m = 100
# 時間の刻み幅
delta = t / m

# 横磁場の強さ
h = 3

# estimator
estimator = create_qulacs_vector_estimator()

# 全磁化に対応するオペレーターを用意する。
magnetization_op = Operator()
for i in range(nqubits):
    magnetization_op += Operator({pauli_label(f"Z{i}"): 1 / 2})

# 初期状態 |00> を準備する。
state = ComputationalBasisState(n_qubits=nqubits, bits=0)

circuit_trotter_heisenberg = QuantumCircuit(qubit_count=nqubits)

# 時間と磁化を記録するリスト
x = [i * delta for i in range(m + 1)]
y = []

# t = 0 の時の全磁化のみ先に計算
y.append(estimator(magnetization_op, state).value.real)

# t = 0 以降の全磁化を計算
for i in range(m):
    # delta = t / Mだけ時間発展
    circuit_trotter_heisenberg.add_gate(e_xx_circuit)
    circuit_trotter_heisenberg.add_gate(e_yy_circuit)
```

```
    circuit_trotter_heisenberg.add_gate(e_zz_circuit)
    # 横磁場の効果を RX ゲートで表す
    circuit_trotter_heisenberg.add_RX_gate(index=0, angle=2 * h * delta)
    circuit_trotter_heisenberg.add_RX_gate(index=1, angle=2 * h * delta)
    circuit_state = GeneralCircuitQuantumState(nqubits, circuit_trotter_h
    # 磁化を計算して記録
    y.append(estimator(magnetization_op, circuit_state).value.real)

# グラフの描画
plt.xlabel("time")
plt.ylabel("Value of magnetization")
plt.title("Dynamics of Heisenberg model")
plt.plot(x, y, "-")
plt.show()
```

$h = 0$ のハイゼンベルグ模型の場合は全磁化の値が一定だったのに対し、横磁場 $h = 3$ を入れた横磁場ハイゼンベルグ模型の場合は全磁化の値が 0 周りで振動しています。これは横磁場を加えたハミルトニアンと全磁化の演算子が可換でないことからもわかります。興味のある方は振動の物理的な描像を考えてみるのも、量子計算というより物理の問題ではありますが面白いと思います。

Exercise 5

横磁場をかけた場合も、QURI Partsの `trotter_suzuki_decomposition()` 関数使ってシミュレーションすることができます。シミュレーションを行って、トロッター分解と同じ結果が再現できていることを確認してみてください。

In [ ]:

第6章 位相推定

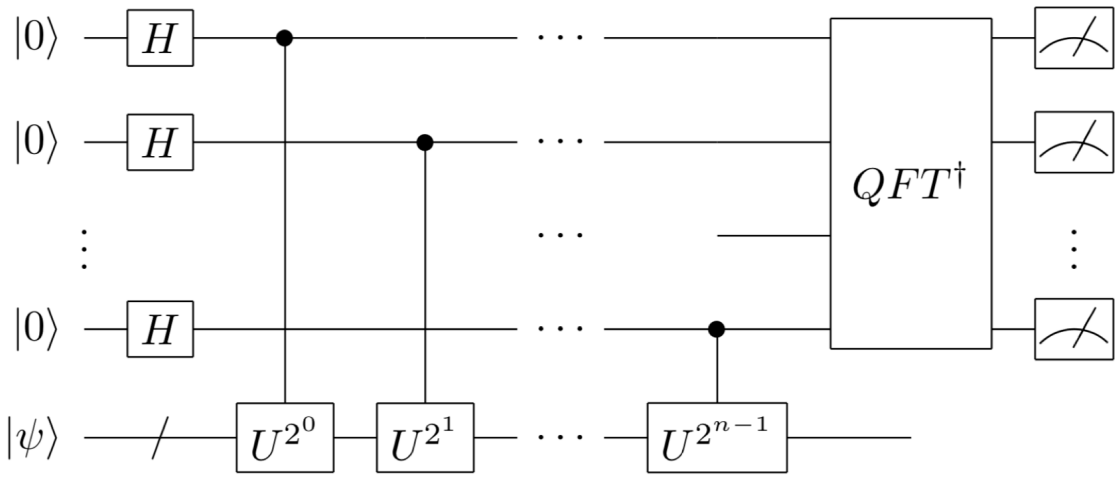
(共通講義資料では、(P71) 6-3.位相推定法をご参照ください。)

量子多体系のエネルギー計算や素因数分解、連立線型方程式といった様々な問題は、ユニタリ行列の固有値問題に帰着でき、量子位相推定アルゴリズム (QPE) によって、古典計算機と比較して非常に高速に解けるようになることが期待されています。

量子位相推定は、ユニタリ演算子Uがあたえられた時に、その固有値$e^{i\lambda}$の位相λを求めるアルゴリズムです。この説明では、入力状態$|\psi\rangle$を固有状態に限定して解説を行いますが、入力状態が固有状態の重ね合わせの場合でも同じ議論が行えるため、一般性は失われません。

$$U|\psi\rangle = e^{i\lambda}|\psi\rangle$$

量子位相推定を行う量子回路は以下のように表されます。



ユニタリ演算子Uの固有値$e^{i\lambda}$について、位相λの2進小数表現の各桁を$\lambda = (2\pi)0.j_1 j_2 \dots j_n$と表すと、位相推定回路によって 1 回の測定で$j_1, j_2, \dots, j_n$を求めることができます。

1. まず$|0\rangle$に初期化されたn個の量子ビットにアダマールゲートHを作用させ、次に制御ユニタリ演算$U^{2^k}(k = 0, \dots, n-1)$を作用させます。k番目の量子ビットには、位相キックバックによって$e^{i\lambda 2^k}$の位相が獲得され、以下のような状態が得られます。

$$\left(\frac{|0\rangle + e^{i(2\pi)0.j_1 \dots j_n}|1\rangle}{\sqrt{2}}\right) \otimes \left(\frac{|0\rangle + e^{i(2\pi)0.j_2 \dots j_n}|1\rangle}{\sqrt{2}}\right) \otimes \dots \otimes \left(\frac{|0\rangle + e^{i(2\pi)0.j_n}|1\rangle}{\sqrt{2}}\right)$$

2. 各補助量子ビットの状態は、量子フーリエ変換の結果と全く同じ形をしており、逆量子フーリエ変換をすることで、$|j_1 \dots j_n\rangle$が得られます。ノイズがない理想的な量子計算機では、この状態で1回の測定を行えば、$j_1, j_2, \dots, j_n$が求められます。

位相キックバックについてはQuantum Native Dojo 2-2 、量子フーリエ変換についてはQuantum Native Dojo 2-3をご参照ください。

Phaseゲートの位相推定

まずはシンプルな例を使って、位相推定回路が機能することを確認してみましょう。ここではターゲットのユニタリ演算子として、Phaseゲートを使用します。

$$Phase = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$$

Phaseゲートを制御Phaseゲートとして位相推定回路に埋め込むことで、$e^{i\theta} = e^{i2\pi x}$の$x$の値を求めてみましょう。

最初に必要なライブラリをまとめてimportしておきます。

In [ ]:
```
from quri_parts.circuit import PauliRotation, QuantumCircuit
from quri_parts.core.operator import Operator, pauli_label
from quri_parts.core.state import ComputationalBasisState, GeneralCircuit
from quri_parts.qulacs.estimator import create_qulacs_vector_estimator
from quri_parts.qulacs.sampler import create_qulacs_vector_sampler
from quri_parts.qiskit.circuit import convert_circuit

from math import pi
```

すべての補助量子ビットにHゲートを追加する関数を用意します。

In [ ]:
```
def add_h_gates(n: int, circuit: QuantumCircuit) -> None:
    for i in range(n):
        circuit.add_H_gate(i)
```

制御PhaseゲートをRZゲートとCNOTゲートの列に展開する関数を用意し、これを使って$U^{2^0}, U^{2^1}, \ldots, U^{2^n}$を回路に追加する関数を用意します。任意のユニタリゲートを制御ユニタリゲートとして回路に埋め込むのは少し面倒ですね。将来的には制御ユニタリゲートを直接扱える機能もQURI Partsに追加される予定です。

In [ ]:
```
def add_control_phase(circuit: QuantumCircuit, q0: int, q1: int, theta: f
    circuit.add_RZ_gate(q0, theta / 2)
    circuit.add_CNOT_gate(q0, q1)
    circuit.add_RZ_gate(q1, -theta / 2)
    circuit.add_CNOT_gate(q0, q1)
    circuit.add_RZ_gate(q1, theta / 2)


def add_cp_gates(n: int, circuit: QuantumCircuit, theta: float) -> None:

    for k in range(n):
        q0 = n
        for i in range(2 ** k):
            add_control_phase(circuit, k, q0, theta)
```

最後に、量子フーリエ変換の逆演算を行う回路を追加する関数を作成します。

In [ ]:
```
def add_qftdag_gates(n: int, circuit: QuantumCircuit) -> None:

    for k in range(n // 2):
        circuit.add_SWAP_gate(k, n - k - 1)

    for k in range(n):
        for i in range(k):
            add_control_phase(circuit, i, k, -pi / (2 ** (k - i)))
        circuit.add_H_gate(k)
```

これらの関数を使って、位相推定回路を作ってみましょう。ここでは、補助量子ビットの数nを4とし、求める位相xを1/4としてみます。回路はとても長くなっています。興味がある方はプリントしてみてください。

In [ ]:
```
n = 4
x = 1 / 4

circuit = QuantumCircuit(n + 1)

add_h_gates(n, circuit)
add_cp_gates(n, circuit, 2 * pi * x)
add_qftdag_gates(n, circuit)
# print(convert_circuit(circuit))
```

Phaseゲートの固有状態$|1\rangle$を入力状態として用意します。

In [ ]:
```
initial_state = ComputationalBasisState(n + 1, bits=1 << n)
circuit = initial_state.circuit + circuit
# print(convert_circuit(circuit))
```

準備ができたので、回路を実行してみましょう。固有値が1つに定まっている場合、ノイズがない理想的な環境では、1回実行すれば100%の確率で正しい結果が得られますが、ここでは念の為1000回ほどサンプリングしてみます。

In [ ]:
```
sampler = create_qulacs_vector_sampler()
result = sampler(circuit, shots=1000)
print({bin(k): v for k, v in result.items()})  # 下位nビットを取り出しビット列
mask = int("1" * n, 2)
print({(k & mask) / (2 ** n): v for k, v in result.items()})  # 結果を10進
```

```
{'0b10100': 1000}
{0.25: 1000}
```

ここでは下位4ビットが補助量子ビットに相当します。x = 1/4の2進数表現である0b0100が得られています。

もう少し精度を上げた計算もしてみましょう。ビット数n = 7、x = 0.421875として計算してみます。

In [ ]:
```
n = 7
x = 0.421875  # 1/4 + 1/8 + 1/32 + 1/64

circuit = QuantumCircuit(n + 1)

add_h_gates(n, circuit)
add_cp_gates(n, circuit, 2 * pi * x)
add_qftdag_gates(n, circuit)

initial_state = ComputationalBasisState(n + 1, bits=1 << n)
circuit = initial_state.circuit + circuit

sampler = create_qulacs_vector_sampler()
result = sampler(circuit, shots=1000)
print({bin(k): v for k, v in result.items()})  # 下位nビットを取り出しビット列
mask = int("1" * n, 2)
print({(k & mask) / (2 ** n): v for k, v in result.items()})  # 結果を10進
```

```
{'0b10110110': 1000}
{0.421875: 1000}
```

期待通りの結果が得られているようです。

ハイゼンベルグ模型での基底状態エネルギーの計算

モデル

それでは具体的な例として、ハミルトニアンダイナミクスの章でも扱った1次元ハイゼンベルグ模型を取り上げて、量子位相推定によって基底状態エネルギーを計算してみましょう。このモデルのハミルトニアンは以下のようなものでした。

$$H = J(X_0X_1 + Y_0Y_1 + Z_0Z_1)$$

基底エネルギーは厳密解が計算できます。デモンストレーション的ではありますが、量子位相推定によって、この厳密解が再現できることを確認してみましょう。

厳密対角化

今回考えるハミルトニアンは、 4×4 行列なので簡単に対角化することができます。以下では位相推定して得られた固有値が厳密解と比較しやすくなるよう、$J = -\frac{\pi}{2}$ とします。対角化すると

$$\begin{pmatrix} -\frac{\pi}{2} & 0 & 0 & 0 \\ 0 & -\frac{\pi}{2} & 0 & 0 \\ 0 & 0 & -\frac{\pi}{2} & 0 \\ 0 & 0 & 0 & \frac{3\pi}{2} \end{pmatrix}$$

となります。いま $J < 0$ なのでハミルトニアンダイナミクスのチュートリアルで見たように、基底状態は三重項を構成していて、基底エネルギーが J で三重に縮退していることがわかります。（励起状態は一重項でエネルギーは $-3J$ となっています。）三重項を構成している基底状態は以下の通りです。

$$|\uparrow\uparrow\rangle,\ \frac{1}{\sqrt{2}}(|\uparrow\downarrow\rangle + |\downarrow\uparrow\rangle),\ |\downarrow\downarrow\rangle.$$

位相推定でハミルトニアンの固有値問題を解く際は、時間発展演算子 $U = e^{-iH\tau}$ の固有値を求めることで、ハミルトニアンの固有値を求めます。Uの固有値の位相には2π の不定性があるので、 $[E_{\mathrm{min}}, E_{\mathrm{max}}]$ が例えば $[-\pi, \pi)$ に収まるように調整することで、固有値が一意に求められます。ここでは$\tau = 1/2$とすることで調整してみましょう。そうすると、位相推定では固有値を $\lambda = (2\pi)0.j_1j_2\ldots j_n$ と書いた際の $j_1, j_2, \ldots, j_n$が得られるので、例えば初期状態に $|\uparrow\uparrow\rangle$ を選ぶと、Uの固有値は$2\pi \times 1/8$ となり、$1/8$ の2進数表示 0.001 に対応する $j_1 = 0, j_2 = 0, j_3 = 1, \ldots$ が得られるはずです。

基底エネルギーの計算手順

量子位相推定を用いた基底エネルギーの計算手順は以下の通りです。

0. ハミルトニアンのサイズを対称性などを用いて削減する
1. ハミルトニアンの時間発展演算子$U = e^{-iH\tau}$を精度よく近似する
2. 制御時間発展演算子を量子計算機で実行しやすいゲートセットに分解し実装する
3. 基底状態と十分重なりのある初期状態を準備する
4. 量子回路を実行し、エネルギー固有値を求める

それでは実際にハイゼンベルグ模型の基底エネルギーを、位相推定を用いて計算してみましょう。ハミルトニアンのサイズは 4×4 と十分小さいのでサイズの削減は考えないことにします。次に時間発展演算子 $U = e^{-iH\tau}$ を精度よく近似する必要がありますが、ここではハミルトニアンダイナミクスのチュートリアルでも用いたトロッター分解を使います。 時間発展演算子にトロッター分解を用いると

$$U_H(\tau) \approx \left(\exp\left(-iJ\frac{t}{M}X_0X_1\right) \exp\left(-iJ\frac{t}{M}Y_0Y_1\right) \exp\left(-iJ\frac{t}{M}Z_0Z_1\right) \right)^M$$

と分解できます。今回は単純のため $M = 1$ とし、時間発展演算子を次のように分解します。

$$U_{H,\tau} = \exp\left(-iJ\tau X_0X_1\right) \exp\left(-iJ\tau Y_0Y_1\right) \exp\left(-iJ\tau Z_0Z_1\right)$$

（モデルが単純な場合はこれでも良い精度で固有値が得られます。）各項のユニタリ演算子はハミルトニアンダイナミクスのチュートリアルで説明したように量子コンピュータで実行できるゲートセットに分解できます。

あとは基底状態と十分重なりのある初期状態を準備する必要がありますが、ここでは少しズルをして初期状態として基底状態 $\left|\uparrow\uparrow\right\rangle$ を選びます。あとは位相推定を行い、固有値を求めるだけです。

Phaseゲートの位相推定の例でみたように、各ユニタリ演算子をコントロールゲートに変換する必要があります。

In [ ]:
```
def add_cu_gates(n: int, circuit: QuantumCircuit, phi: float, n_trotter_s
    q0, q1 = n, n + 1
    # n: system qubits
    for k in range(n):
        for i in range(2 ** k):
            for _ in range(n_trotter_step):

                # CU(X0 X1)
                circuit.add_H_gate(q0)
                circuit.add_H_gate(q1)
                circuit.add_CNOT_gate(q1, q0)
                circuit.add_RZ_gate(q0, phi)
                circuit.add_CNOT_gate(k, q0)
                circuit.add_RZ_gate(q0, -phi)
                circuit.add_CNOT_gate(k, q0)
                circuit.add_CNOT_gate(q1, q0)
                circuit.add_H_gate(q0)
                circuit.add_H_gate(q1)

                # CU(Y0 Y1)
                circuit.add_Sdag_gate(q0)
                circuit.add_Sdag_gate(q1)
                circuit.add_H_gate(q0)
                circuit.add_H_gate(q1)
                circuit.add_CNOT_gate(q1, q0)
                circuit.add_RZ_gate(q0, phi)
                circuit.add_CNOT_gate(k, q0)
                circuit.add_RZ_gate(q0, -phi)
                circuit.add_CNOT_gate(k, q0)
                circuit.add_CNOT_gate(q1, q0)
                circuit.add_H_gate(q0)
                circuit.add_H_gate(q1)
                circuit.add_S_gate(q0)
                circuit.add_S_gate(q1)

                # CU(Z0 Z1)
                circuit.add_CNOT_gate(q1, q0)
                circuit.add_RZ_gate(q0, phi)
                circuit.add_CNOT_gate(k, q0)
                circuit.add_RZ_gate(q0, -phi)
                circuit.add_CNOT_gate(k, q0)
                circuit.add_CNOT_gate(q1, q0)
    return circuit
```

あとはこれらを用いて位相推定を行います。ここでは補助量子ビットを3つ用意します。前述の通り、位相推定するユニタリ演算子 $U = e^{-iH\tau}$ は 2π の不定性があるので、ここでは $\tau = 1/2$とすることで、$[E_{\min}, E_{\max}]$ が $[-\pi, \pi)$ に収まるように調整します。

In [ ]:
```
# 補助量子ビット数
n = 3

# モデルパラメーター
J = -pi / 2
tau = 1 / 2
phi = tau * J

circuit = QuantumCircuit(n + 2)

# 初期状態を作る回路を準備。補助ビットは0, 入力状態は |11>
initial_state = ComputationalBasisState(n + 2, bits=0b11 << n)
initial_circuit = initial_state.circuit

# アダマールゲートを補助量子ビットに作用させる
add_h_gates(n, circuit)

# 時間発展演算子をコントロールゲートにして作用させる
add_cu_gates(n, circuit, phi)

# 逆フーリエ変換を行う
add_qftdag_gates(n, circuit)

# 初期状態回路と位相推定回路を組み合わせる。
circuit = initial_circuit + circuit
```

これで回路が準備できたので、あとは実際にサンプリングしてみましょう。

In [ ]:
```
sampler = create_qulacs_vector_sampler()
result = sampler(circuit, shots=1000)
print(result)
print({bin(k): v for k, v in result.items()})
```

```
Counter({25: 1000})
{'0b11001': 1000}
```

結果を見ると、ビット列 11001 が得られています。最初の 11 は入力状態で、後ろの3つが得られた固有値を表しています。2進数表現で001、10進数表現では$1/8$なので、ハミルトニアンの固有値に直すと、$U = e^{-iH\tau}$より$\frac{2\pi \times 1/8}{-\tau} = -\frac{\pi}{2}$ となります。これは今考えているハミルトニアンの基底エネルギー$J = -\frac{\pi}{2}$と一致していることがわかります。

Exercise 6

Phaseゲートの位相推定で、$x = \pi/10$とした場合どうなるか試してみてください。また、10進数で3桁の精度を得るにはnを幾つにするべきか求めてみてください。

In [ ]:
```
```

第7章 ブロックエンコーディング

(共通講義資料では、(P93) 6-6.ブロックエンコーディングをご参照ください。)

量子回路を構成する量子ゲートはユニタリー演算なので、量子回路全体の量子状態に対する作用もユニタリーになります。一方で、多くの応用において、ユニタリーとは限らない演算子を量子状態に作用させたい場合があります。たとえば、物理系のハミルトニアン H を作用させた状態 $(H|0\rangle, H^2|0\rangle, \cdots)$は、多体計算のための基底として適していることが知られています。こうした状態を使った計算法は、クリロフ部分空間法と呼ばれ大規模固有値問題の計算で広く使われています。

ユニタリーとは限らない一般の演算子を量子状態に作用させるための方法として**ブロックエンコーディング**があります。ブロックエンコーディングは、一般の演算子の作用を、拡大した状態空間上のユニタリー演算子として実現する方法です。

なおブロックエンコーディングは、さまざまな量子アルゴリズムを統一的に理解するための基本的な要素にもなっています。日本語の参考資料として、大阪大学の藤井先生の量子と古典の物理と幾何@オンラインでの講演をおすすめします。

- 講演資料: https://www.dropbox.com/s/k2vzhwy33ohhz05/QSVT_fujii_20220428.pdf?dl=0
- 講演動画: https://youtube.com/live/bdKV3voK-QA?si=EnSIkaIECMiOmarE

n量子ビット系上の演算子Aのブロックエンコーディングとは、$n+m$量子ビット系上のユニタリー演算子Uとm量子ビット状態$|\psi_0\rangle_m$の組で、以下を満たすものです。

$$ {}_m\langle\psi_0|U|\psi_0\rangle_m = A $$

この資料では、n量子ビット系を「システム」、m量子ビット系を「補助レジスター」と呼ぶことにします。

言い換えると、Uのブロック要素のうち、システムに作用するブロック要素がAになっているということです。

$$ U = \begin{pmatrix} A & \cdot \\ \cdot & \cdot \end{pmatrix} $$

このようなUがユニタリーであるためには$\|A\| \leq 1$である必要があります。

補助レジスターの状態が$|\psi_0\rangle_m$の時、システム＋補助レジスターに対するUの作用は以下のようになります。

$$ U|\psi\rangle_n \otimes |\psi_0\rangle_m = (A|\psi\rangle_n) \otimes |\psi_0\rangle_m + |g\rangle, \quad {}_m\langle\psi_0|g\rangle = 0. $$

ブロックエンコーディングを実際に構成する方法は複数ありますが、以下ではLCUと呼ばれる手法を取り上げます。

ブロックエンコーディングの構成: LCU

演算子Hがユニタリー演算子の線形結合 (Linear combination of unitaries: LCU) になっている場合を考えます。各ユニタリーをH_l ($l = 0, \ldots, L-1$) と書くと

$$H = \sum_{l=0}^{L-1} w_l H_l$$

ただし、各H_lに対して適切に位相因子をかけておき、w_lはすべて正の実数になっているとします。 特にHがエルミートの場合、各H_lは（符号付き）パウリ演算子にとることができます。

補助レジスターを、LCUの添え字 $l = 0, \ldots, L-1$ をデータとして表現できる大きさに取ります。すなわち、lの2進表示に必要なビット数を考えると $m = \lceil \log L \rceil$ となります。

この時、以下のようにブロックエンコーディングが構成できます。

$$U = \mathrm{SELECT} \equiv \sum_{l=0}^{L-1} |l\rangle_m\langle l| \otimes H_l,$$

$$|\psi_0\rangle_m = \mathrm{PREPARE}|0\rangle_m \equiv \sum_{l=0}^{L-1} \sqrt{\frac{w_l}{\lambda}}|l\rangle_m, \quad \lambda \equiv \sum_{l=0}^{L-1} w_l.$$

SELECTはシステム＋補助レジスター上のユニタリー演算子で、補助レジスターの値lに応じたユニタリーH_lをシステムに作用させます。 PREPAREは補助レジスター上のユニタリー演算子で、初期状態$|0\rangle_m$から、lの取り得る値すべての重ね合わせ状態$|\psi_0\rangle_m$を生成します。 これらがブロックエンコーディングの定義 ${}_m\langle\psi_0|U|\psi_0\rangle_m = H$を満たすことは簡単に示すことができます。

ハイゼンベルグ模型の場合

前章でも扱った1次元ハイゼンベルグ模型のハミルトニアンは

$$H = J(X_0X_1 + Y_0Y_1 + Z_0Z_1)$$

で、固有値はJおよび$-3J$でした。上述の条件$\|H\| \leq 1$（$\|\cdot\|$はHの固有値の絶対値の最大値）を満たすため、以下では$J = 1/3$とします。LCUの定義に当てはめると

$$H_0 = X_0X_1, H_1 = Y_0Y_1, H_2 = Z_0Z_1$$

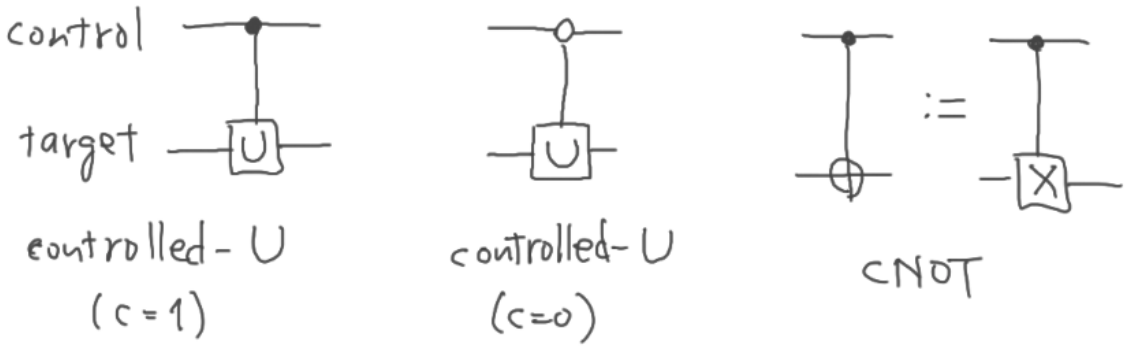
$$w_l = J = \frac{1}{3} \quad (l = 0, 1, 2)$$

システムは$n = 2$量子ビット、$l = 0, 1, 2$ なので補助レジスターは$m = \lceil \log 3 \rceil = 2$量子ビット、合計で4量子ビットを扱うことになります。

LCUブロックエンコーディングの回路構成

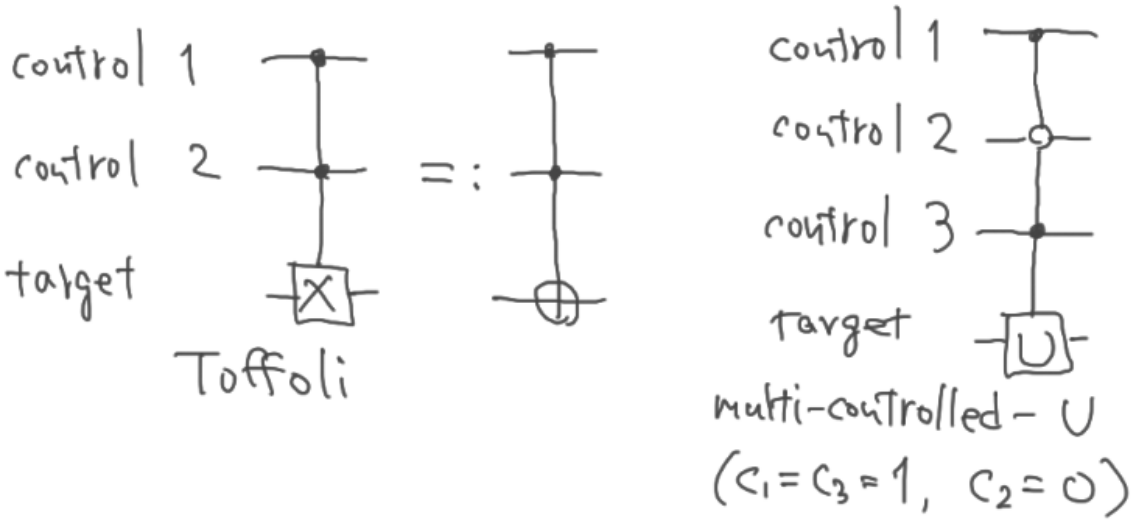
Multi-controlledゲート

SELECT演算子の定義には、 $|l\rangle_m\langle l| \otimes H_l$ という形の演算が出てきます。この演算は、補助レジスターの値が l の時に限りシステムに H_l を作用させるというものです。このように、補助レジスターの値を条件として行う演算を制御演算 (controlled operation) と呼びます。たとえばCNOTゲートは、1ビット目（制御ビット）が1の場合のみ、2ビット目にXゲートをかけるというものでした。より一般に、制御ビットが1（または0）の場合に2ビット目にユニタリー U を作用させるゲートを制御Uゲート (controlled-U) と呼びます。



図の小さい黒丸はそのビットが1の場合、白丸はそのビットが0の場合に作用することを意味します。

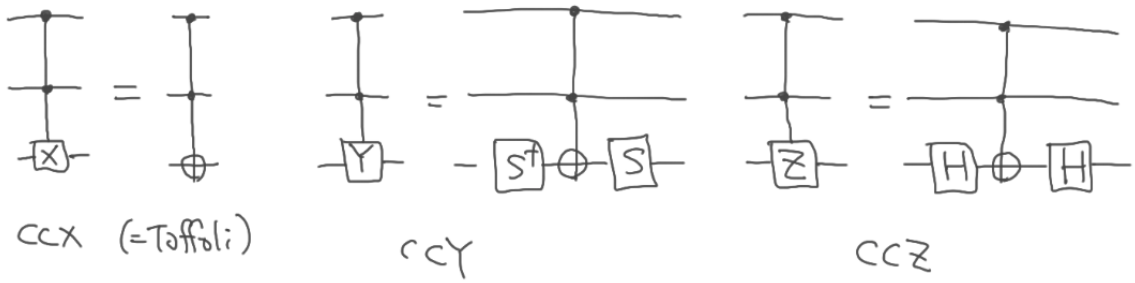
補助レジスター $|l\rangle_m$ は、 $l = 0, \ldots, L-1$ の値をm量子ビットで表すので、制御ビットが複数ある制御演算 (multi-controlled operation) が必要になります。特に、2つの制御ビットがともに1の場合に3ビット目にXゲートをかけるゲートはToffoliゲートと呼ばれ、よく使われます。



今回のハイゼンベルグ模型に対する回路構成では、制御ビットが2ビットの制御パウリ演算を扱います。

$$SXS^\dagger = Y, SS^\dagger = I, HXH = Z, HH = I$$

より、この制御パウリ演算はToffoliゲートを使って以下のように作れます。

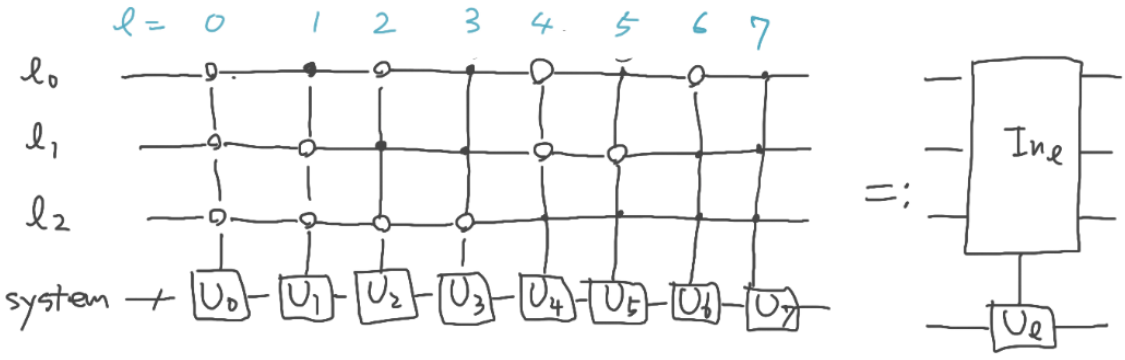


これらの性質を利用して、2ビット制御パウリ演算を以下のように作ることができます。

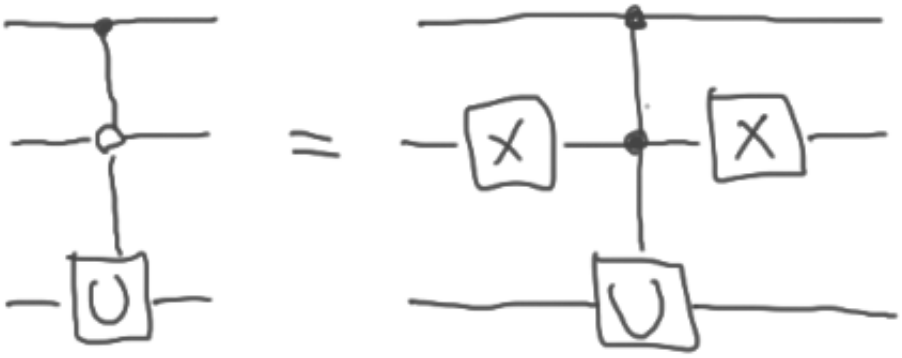
In [ ]:
```python
from quri_parts.circuit import H, QuantumGate, S, Sdag, TOFFOLI, X

def cc_pauli(control_bits: tuple[int, int], target_bit: int, pauli: str)
    """Two-qubit controlled Pauli operation.

    Args:
        control_bits: Indices of the control qubits.
        target_bit: Index of the target qubit.
        pauli: Specifies which Pauli operator to apply. "X" or "Y" or "Z"
    """
    gates = []
    if pauli == "Y":
        gates.append(Sdag(target_bit))
    if pauli == "Z":
        gates.append(H(target_bit))
    gates.append(TOFFOLI(control_bits[0], control_bits[1], target_bit))
    if pauli == "Y":
        gates.append(S(target_bit))
    if pauli == "Z":
        gates.append(H(target_bit))
    return gates
```

Indexed演算

SELECT演算子は、「補助レジスターの値がlの場合にシステムにユニタリー演算U_lをかける」という操作を、lが取り得るすべての値に対して行う、という形になっています。このような操作をindexed operationと呼びます。lは補助レジスターのビット上で2進表示で表されているので、以下のような回路で実現することができます。



制御ビットが0の場合の制御演算は、1に対する制御演算の前後にXゲートを入れてビットを反転させることで作れるので



2ビットの補助レジスターに対するindexed operationは、Xゲートと2ビット制御パウリ演算を組み合わせて作ることができます。

In [ ]:
```python
# 注: 通常indexed operationは全てのlに対する一連の操作を指すが、この関数では特定のl
def two_qubit_indexed_pauli(
    index_bits: tuple[int, int],
    index_value: int,
    target_bit: int,
    pauli: str
) -> list[QuantumGate]:
    """Two-qubit indexed Pauli operation for a specific index value.

    Args:
        index_bits: Indices of the qubits storing the index.
        index_value: The index value for the operation.
        target_bit: Index of the target qubit.
        pauli: Specifies which Pauli operator to apply. "X" or "Y" or "Z"
    """
    x_bits = []
    if (index_value & 0b01) == 0:
        x_bits.append(index_bits[0])
    if (index_value & 0b10) == 0:
        x_bits.append(index_bits[1])

    gates = []
    for i in x_bits:
        gates.append(X(i))
    gates += cc_pauli(index_bits, target_bit, pauli)
    for i in x_bits:
        gates.append(X(i))
    return gates
```

SELECT回路

前述のとおり、システムは$n = 2$量子ビット、補助レジスターは$m = 2$量子ビットですが、後述のPREPARE回路の構成で補助量子ビットを1個使うので、5量子ビットの回路を作ることにします。システム・補助レジスター・補助量子ビットに使う量子ビットのインデックスをあらかじめ変数に割り当てておきます。

In [ ]:
```
n_qubits = 5
# (s0, s1): システム、(r0, r1): 補助レジスター、a: 補助量子ビット
s0, s1, r0, r1, a = 0, 1, 2, 3, 4
# l = 0, 1, 2
L = 3
m = 2
```

SELECTの定義

$$\mathrm{SELECT} = |0\rangle_m\langle 0| \otimes X_0X_1 + |1\rangle_m\langle 1| \otimes Y_0Y_1 + |2\rangle_m\langle 2| \otimes Z_0Z_1$$

より、上述の `two_qubit_indexed_pauli()` を使って構成することができます。

In [ ]:
```
from quri_parts.circuit import QuantumCircuit

select = QuantumCircuit(n_qubits)

# X0X1
select += two_qubit_indexed_pauli((r0, r1), 0, s0, "X")
select += two_qubit_indexed_pauli((r0, r1), 0, s1, "X")

# Y0Y1
select += two_qubit_indexed_pauli((r0, r1), 1, s0, "Y")
select += two_qubit_indexed_pauli((r0, r1), 1, s1, "Y")

# Z0Z1
select += two_qubit_indexed_pauli((r0, r1), 2, s0, "Z")
select += two_qubit_indexed_pauli((r0, r1), 2, s1, "Z")
```

PREPARE回路

今扱っているハイゼンベルグ模型では、PREPAREは以下のように$l=0,\ldots,L-1$の等しい重みの重ね合わせを作る操作（$\equiv \mathrm{UNIFORM}_L$と書きます）です。

$$\mathrm{PREPARE}|0\rangle_m = \mathrm{UNIFORM}_L|0\rangle_m = \frac{1}{\sqrt{L}}\sum_{l=0}^{L-1}|l\rangle_m$$

この操作の作り方を考えるために、一般の状態 $|\alpha\rangle$ を $|\beta\rangle$ に移す方法を考えます。
$s\equiv\langle\beta|\alpha\rangle$ が $1/4<|s|^2<1$ を満たすとき、以下が成り立つことが示せます。

$$e^{i\phi|\alpha\rangle\langle\alpha|}e^{i\phi|\beta\rangle\langle\beta|}|\alpha\rangle=|\beta\rangle,\quad \phi=\arccos\left(1-\frac{1}{2|s|^2}\right).$$

（ヒント: $|\alpha\rangle$ を $|\beta\rangle$ とそれに直交する $|\beta^\perp\rangle$ の線形結合で書き、 $|\beta\rangle$ を $|\alpha\rangle$ とそれに直交する $|\alpha^\perp\rangle$ の線形結合で書く。 $e^{i\phi|\alpha\rangle\langle\alpha|}$ ($e^{i\phi|\beta\rangle\langle\beta|}$) は、 $|\alpha\rangle$ ($|\beta\rangle$) に $e^{i\phi}$ の位相を付け、 $|\alpha^\perp\rangle$ ($|\beta^\perp\rangle$) は不変に保つ。状態ベクトルを $\left(|\alpha\rangle,|\alpha^\perp\rangle\right)$ および $\left(|\beta\rangle,|\beta^\perp\rangle\right)$ 基底に変換しながら作用を計算していけば示せる）

ここでは、「始状態」$|\alpha\rangle_m$ を、 $|0\rangle_m$ の各ビットにHゲートをかけた状態とし、「終状態」$|\beta\rangle_m$ を目的の状態とします:

$$|\alpha\rangle_m = H^{\otimes m}|0\rangle_m = \frac{1}{\sqrt{2^m}}\sum_{l=0}^{2^m-1}|l\rangle_m,\quad |\beta\rangle_m = \frac{1}{\sqrt{L}}\sum_{l=0}^{L-1}|l\rangle_m.$$

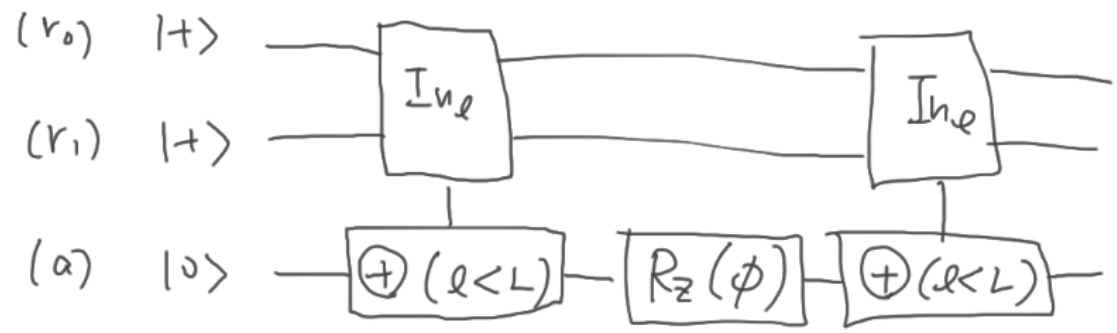
まず $e^{i\phi|\beta\rangle_m\langle\beta|}$ の作用について考えます。入力状態 $|\alpha\rangle_m$ は以下のように書けるので

$$|\alpha\rangle_m = \frac{1}{\sqrt{2^m}}\sum_{l=0}^{L-1}|l\rangle_m + \frac{1}{\sqrt{2^m}}\sum_{l=L}^{2^m-1}|l\rangle_m = \sqrt{\frac{L}{2^m}}|\beta\rangle_m + \frac{1}{\sqrt{2^m}}\sum_{l=L}^{2^m-1}|l\rangle_m$$

$e^{i\phi|\beta\rangle_m\langle\beta|}$ の作用は

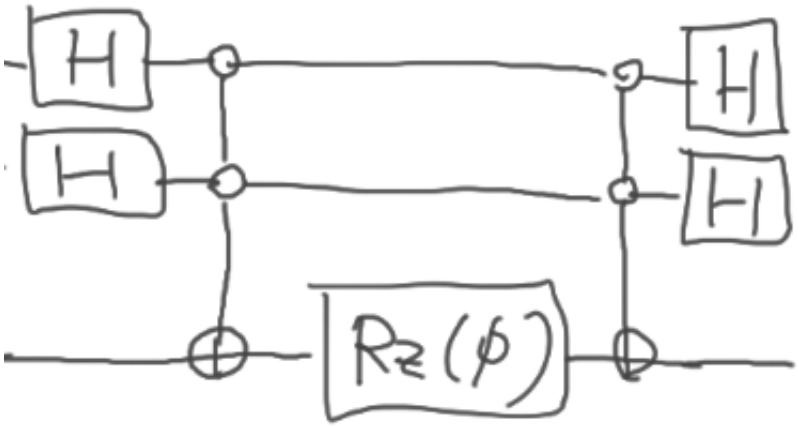
$$e^{i\phi|\beta\rangle_m\langle\beta|}|\alpha\rangle_m = \sqrt{\frac{L}{2^m}}e^{i\phi}|\beta\rangle_m + \frac{1}{\sqrt{2^m}}\sum_{l=L}^{2^m-1}|l\rangle_m$$

となりますが、これは「$|l\rangle_m(l<L)$ に対して $e^{i\phi}$ の位相を付ける」ことと等価なので、補助量子ビットを使って以下の回路で実現できます。



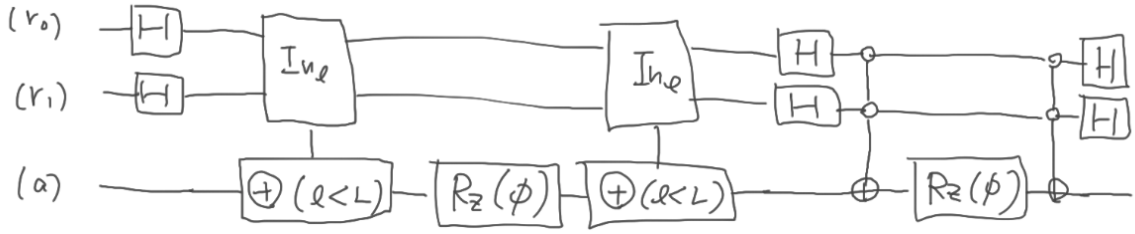
ここで、⊕は古典ビットの排他的論理和 (XOR) を表します。すなわちこのindexed XORは、補助量子ビット (a) に対して、$l < L$ の場合は1、$l \geq L$ の場合は0をXORすることになります。$b \oplus 0 = b, b \oplus 1 = \neg b$ なので、結局 $l < L$ の場合のみビット反転＝Xゲートをかけることになります。反転させた補助量子ビットに対してZ回転によって位相 $e^{i\phi}$ を付け、その後再度indexed XORをかけて補助量子ビットを $|0\rangle$ に戻します。

次に、$e^{i\phi|\alpha\rangle_m\langle\alpha|}$ は以下の回路で実行できます。



補助レジスター (r0, r1) に対してHゲートをかけた後に $|0\rangle_m$ の条件で補助量子ビット (a) を反転させていますが、各ビットにHゲートをかけた後に $|0\rangle_m$ となる状態は $|\alpha\rangle_m$ なので、結局入力状態が $|\alpha\rangle_m$ の場合のみ補助ビットが反転されることになります。反転させた補助量子ビットに対して上と同様Z回転した後、再度制御ゲート・Hゲートをかけて、各ビットの値を元に戻しています。

したがって、全体の回路は以下のようになります。



また、 $s = \langle\beta|\alpha\rangle = \sqrt{L/2^m}$ なので、 $\phi = \arccos(1 - 2^{m-1}/L)$ となります。 以上の回路を以下のように構成することができます。

In [ ]:
```
from math import acos

phi = acos(1-2**(m-1)/L)

prepare = QuantumCircuit(n_qubits)

# Hadamard
for q in r0, r1:
    prepare.add_H_gate(q)

indexed_a_l = []
for l in range(L):
    indexed_a_l += two_qubit_indexed_pauli((r0, r1), l, a, "X")

# Indexed XOR
prepare += indexed_a_l
# Rotation
prepare.add_RZ_gate(a, phi)
# Indexed XOR
prepare += indexed_a_l

gates = two_qubit_indexed_pauli((r0, r1), 0, a, "X")

# Hadamard
for q in r0, r1:
    prepare.add_H_gate(q)
# Multi-controlled NOT
prepare += gates
# Rotation
prepare.add_RZ_gate(a, phi)
# Multi-controlled Not
prepare += gates
# Hadamard
for q in r0, r1:
    prepare.add_H_gate(q)
```

作成したPREPARE回路で、実際に $|0\rangle_m$ が $|\psi_0\rangle_m$ に変換されることを確かめてみましょう。そのためにまず、QURI Partsの回路をQulacsで実行し、結果の状態を表示する関数を作成します。

In [ ]:
```
from math import isclose
from pprint import pprint
from quri_parts.qulacs.circuit import convert_circuit
import qulacs

def run_circuit(circuit, initial_state = None):
    """Run a QURI Parts circuit with Qulacs.

    Args:
        circuit: A QURI Parts circuit
        initial_state: A numpy vector specifying the initial state. If om
    """
    qs_circuit = convert_circuit(circuit)
    n = circuit.qubit_count
    state = qulacs.QuantumState(n)
    if initial_state is not None:
        state.load(initial_state)
    qs_circuit.update_quantum_state(state)
    return state

def print_state(state):
    """Print amplitudes in a qulacs.QuantumState."""
    n = state.get_qubit_count()
    amplitudes = {
        format(i, f"0{n}b"): a
        for i in range(2**n)
        # 数値誤差で残ったごく小さな項を落とす
        if not isclose(abs(a := state.get_amplitude(i)), 0, abs_tol=1e-9)
    }
    pprint(amplitudes)
```

作成した関数でPREPARE回路を実行してみます。補助レジスターの量子ビットを `(r0, r1) = (2, 3)` としているので、出力ビット列の（右から）3, 4桁目が `00`, `01`, `10` の3つの状態が均等に重ねあわされていることが分かります。（係数が複素数になっていますが、グローバルな位相因子のため問題ありません）

In [ ]:
```
print_state(run_circuit(prepare))
```

```
{'00000': (0.3333333333333331+0.47140452079103146j),
 '00100': (0.3333333333333315+0.4714045207910314j),
 '01000': (0.3333333333333331+0.47140452079103146j)}
```

PREPARE-SELECTによるブロックエンコーディング

ここまでで作ったSELECT, PREPARE回路を使って、実際にHのブロックエンコーディングが構成できていることを確認してみましょう。まず、回路実行後の状態を $|\psi_0\rangle_m$ に射影するために、対応するベクトルを準備します。

In [ ]:
```python
from math import sqrt
import numpy as np

psi0 = np.zeros(2**3, dtype=complex)
for l in (0b00, 0b01, 0b10):
    # 上記のPREPARE回路で生成される状態ベクトルにはグローバルな位相因子がついているの
    # 共役な係数を設定する
    psi0[l] = 0.3333333333333331-0.47140452079103146j

def project_on_psi0(state_vec):
    v = state_vec.reshape(2**3, 2**2)
    return np.einsum("i,ij->j", psi0, v)
```

PREPARE, SELECTを連結した回路を作り、 H の固有状態に対応する入力状態に対して実行した後 $|\psi_0\rangle_m$ に射影し、得られる（システムの）状態ベクトルを調べます。

In [ ]:
```
prepare_select = prepare + select

print("input: |00>")
state = run_circuit(prepare_select)
print("Full state")
print_state(state)
print("Projected state")
proj = project_on_psi0(state.get_vector())
print(proj)
print("\n")

print("input: (|01>+|10>)/sqrt(2)")
init_state = np.zeros(2**5)
init_state[0b01] = 1/sqrt(2)
init_state[0b10] = 1/sqrt(2)
state = run_circuit(prepare_select, init_state)
print("Full state")
print_state(state)
print("Projected state")
proj = project_on_psi0(state.get_vector())
print(proj)
print("\n")

print("input: (|01>-|10>)/sqrt(2)")
init_state = np.zeros(2**5)
init_state[0b01] = 1/sqrt(2)
init_state[0b10] = -1/sqrt(2)
state = run_circuit(prepare_select, init_state)
print("Full state")
print_state(state)
print("Projected state")
proj = project_on_psi0(state.get_vector())
print(proj)
print("\n")

print("input: |11>")
init_state = np.zeros(2**5)
init_state[0b11] = 1
state = run_circuit(prepare_select, init_state)
print("Full state")
print_state(state)
print("Projected state")
proj = project_on_psi0(state.get_vector())
print(proj)
print("\n")
```

```
input: |00>
Full state
{'00011': (0.333333333333333+0.4714045207910313j),
 '00111': (-0.33333333333333304-0.47140452079103123j),
 '01000': (0.333333333333333+0.4714045207910313j)}
Projected state
[0.33333333+0.00000000e+00j 0.        +0.00000000e+00j
 0.        +0.00000000e+00j 0.        +5.55111512e-17j]


input: (|01>+|10>)/sqrt(2)
Full state
{'00001': (0.23570226039551562+0.33333333333333304j),
 '00010': (0.23570226039551562+0.33333333333333304j),
 '00101': (0.23570226039551562+0.33333333333333304j),
 '00110': (0.23570226039551562+0.33333333333333304j),
 '01001': (-0.23570226039551562-0.33333333333333304j),
 '01010': (-0.23570226039551562-0.33333333333333304j)}
Projected state
[0.        +0.00000000e+00j 0.23570226-1.38777878e-17j
 0.23570226-1.38777878e-17j 0.        +0.00000000e+00j]


input: (|01>-|10>)/sqrt(2)
Full state
{'00001': (-0.23570226039551562-0.33333333333333304j),
 '00010': (0.23570226039551562+0.33333333333333304j),
 '00101': (-0.23570226039551562-0.33333333333333304j),
 '00110': (0.23570226039551562+0.33333333333333304j),
 '01001': (-0.23570226039551562-0.33333333333333304j),
 '01010': (0.23570226039551562+0.33333333333333304j)}
Projected state
[ 0.        +0.00000000e+00j -0.70710678+4.16333634e-17j
  0.70710678-4.16333634e-17j  0.        +0.00000000e+00j]


input: |11>
Full state
{'00000': (0.333333333333333+0.4714045207910313j),
 '00100': (-0.33333333333333304-0.47140452079103123j),
 '01011': (0.333333333333333+0.4714045207910313j)}
Projected state
[0.        +5.55111512e-17j 0.        +0.00000000e+00j
 0.        +0.00000000e+00j 0.33333333+0.00000000e+00j]
```

それぞれの固有状態についての H の作用は

$$H|00\rangle = J|00\rangle = \frac{1}{3}|00\rangle$$

$$H\frac{|01\rangle + |10\rangle}{\sqrt{2}} = J\frac{|01\rangle + |10\rangle}{\sqrt{2}} = \frac{1}{3\sqrt{2}}|01\rangle + \frac{1}{3\sqrt{2}}|10\rangle$$

$$H\frac{|01\rangle - |10\rangle}{\sqrt{2}} = -3J\frac{|01\rangle - |10\rangle}{\sqrt{2}} = -\frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|10\rangle$$

$$H|11\rangle = J|00\rangle = \frac{1}{3}|11\rangle$$

となり、 $1/3\sqrt{2} \simeq 0.2357, 1/\sqrt{2} \simeq 0.7071$ より出力が正しいことが分かります。

Qubitization

H のブロックエンコーディング $U, |\psi_0\rangle_m$ が $U^2 = I$ を満たすとき、以下のように H の新しいブロックエンコーディング $\mathcal{W}, |\psi_0\rangle_m$ を定義できます。

$$\mathcal{W} \equiv \left[(2|\psi_0\rangle_m\langle\psi_0| - I_m) \otimes I\right] U$$

$\mathcal{W}$ は量子ウォークの文脈でSzegedyによって導入された演算子と同じ形のため、Szegedy walk operatorと呼ばれます。

Exercise 上で構成したハイゼンベルグ模型のハミルトニアン H の（LCU）ブロックエンコーディングについて、 $U^2 = I$ であることを示してください。

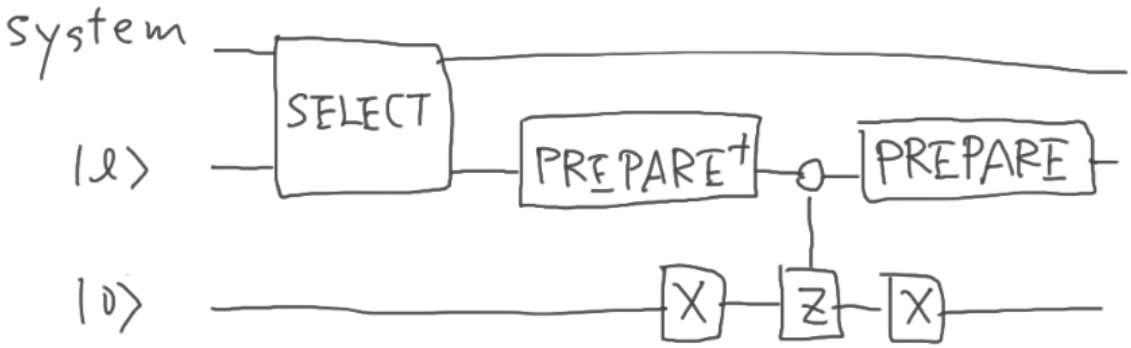
$\mathcal{W}$ の定義から、 H のブロックエンコーディングである（＝補助レジスターが $|\psi_0\rangle_m$ であるときのシステム上の作用が H である）ことは簡単に分かります。一方、 $\mathcal{W}$ の全体（システム＋補助レジスター）系上の作用は以下のようになっています。

$$\mathcal{W} = \bigoplus_{E_k} \begin{pmatrix} E_k/\lambda & \sqrt{1-(E_k/\lambda)^2} \\ -\sqrt{1-(E_k/\lambda)^2} & E_k/\lambda \end{pmatrix} = \bigoplus_{E_k} e^{i\arccos(E_k/\lambda)Y}$$

ここで、 E_k は H の固有値であり、直和の各項は $|E_k\rangle \otimes |\psi_0\rangle_m, \mathcal{W}|E_k\rangle \otimes |\psi_0\rangle_m$ が張る2次元空間に作用しています。すなわち、システムと補助レジスターを合わせた系全体の状態空間が、 H の各固有値 E_k に対応する2次元空間に直和分解され、 $\mathcal{W}$ の作用は各2次元空間内の Y 軸回転になっています。各2次元固有空間を量子ビットとみなして、この操作のことを qubitization と呼びます。各固有空間内での回転角が $\arccos(E_k/\lambda)$ となっているため、 $\mathcal{W}$ を位相推定することで H の固有値 E_k を復元することができます。

LCUの場合の回路構成

LCUブロックエンコーディング (PREPARE-SELECT) に対するqubitizationの回路構成を考えましょう。U は元のブロックエンコーディングの U なので、SELECTに対応します。残りの $2|\psi_0\rangle_m\langle\psi_0| - I_m$ は、補助レジスターが $|\psi_0\rangle_m\langle\psi_0|$ の時はそのままで、それ以外（直交する場合）は符号を反転するので、初期状態が $|0\rangle$ の補助量子ビットを使って、以下のように構成できます。



Exercise

- 上記の $\mathcal{W}$ に対応する回路をQURI Partsで作成してみましょう。ヒント: $\mathrm{PREPARE}^\dagger$ は、QURI Partsの機能で以下のコードで作れます
- 作成した回路を実際に実行し、システム上でどのような作用になっているか調べてみましょう

In [ ]:
```
from quri_parts.circuit import inverse_gate

prepare_dag = [inverse_gate(g) for g in reversed(prepare.gates)]
```

In [ ]:
```
```