

Utility scale quantum computing
量子ユーティリティー授業

ユーティリティー実験 II

2025/10/9

Translated and modified by Toru Imai
Created by Yukio Kawashima

今井 亨 パートナシップマネージャー, IBM Quantum



Toru Imai

LinkedIn:
<https://www.linkedin.com/in/toru-imai-85678026a/>

- ビジネス経歴
 - キヤノン株式会社、シンクサイト株式会社（東大・阪大発スタートアップ）でのR&D、プロダクト開発を経て、IBM Quantumに2024年に参画
 - キヤノン株式会社
 - 熱・変形・光学シミュレーションによる次世代装置の研究開発
 - 光超音波イメージング技術を利用した顕微鏡の研究開発
 - 京都大学医学部、ワシントン大学、カリフォルニア工科大学での研究開発
 - シンクサイト株式会社
 - 光学部門研究開発グループ長として、機械学習と応用光学を利用した細胞分析装置の新規技術開発
 - 製品開発プロジェクトリードとして、初号機開発とハーバード大系研究機関に納品
 - IBM Quantum
 - IBMの量子コンピューティングシステムをご利用頂いているお客様・コミュニティとのパートナシップマネジメントを担当
- アカデミア経歴
 - 早稲田大学 物理学科（学士）（量子物性理論）
 - 早稲田大学大学院 物理学及び応用物理学専攻（修士）（量子物性理論）
 - Washington University in St. Louis, Biomedical Engineering, 博士課程単位取得中退
 - California Institute of Technology, Medical Engineering, 特別博士課程学生在籍

本日のアジェンダ

1. (第7回 8/22) 量子シミュレーションの振り返り (ハミルトニアンシミュレーション、量子ダイナミクス)
 1. ハミルトニアン (モデル)
 2. トロッタ分解
2. シミュレータ及び量子コンピュータによる実機計算の紹介
 1. 20-qubit 問題 (state-vector と matrix product state シミュレータによる計算)
 2. 70-qubit 問題 (matrix product state シミュレータと 量子コンピュータによる計算)

量子シミュレーション（ハミルトニアンシミュレーション）

時間依存シュレーディンガーエ方程式を解く

$$i\frac{d}{dt}|\Psi(t)\rangle = \hat{H}|\Psi(t)\rangle$$

↑
ハミルトニアン演算子

↑
波動関数

下記を解くのがゴール

$$|\Psi(t)\rangle = e^{-i\hat{H}t}|\Psi(0)\rangle$$

問題を可能な限り正確かつ効率的に数値的に解く

$$|\Psi(t + \Delta t)\rangle = e^{-i\hat{H}\Delta t}|\Psi(t)\rangle \approx \left(1 - iH\Delta t - \frac{\hat{H}^2\Delta t^2}{2} + \dots \right) |\Psi(t)\rangle$$

↑
非常に小さな時間スライス

↑
テイラー展開

一般的なハミルトニアン

- 量子系のハミルトニアンは、系の総エネルギーを表す演算子
 - 運動エネルギーとポテンシャルエネルギー $\hat{H} = \hat{T} + \hat{V}$
- 時間依存ハミルトニアンと時間非依存ハミルトニアン
 - 本講義では時間非依存ハミルトニアンのみを考える
- 多くの分野で重要
 - 量子化学(材料科学)
 - 凝縮系物質物理学
 - 高エネルギー物理学

ハミルトニアン(スピンハミルトニアン)

スピニシステムの格子モデルを用いて磁気システムを研究する

- n -ベクトルモデル

- イジングモデル ($n=1$)

- XYモデル ($n=2$)

- ハイゼンベルグモデル ($n=3$)

スピニ相互作用 外場

$$H = - \sum_{\langle i,j \rangle} J \sigma_{Z_i} \sigma_{Z_j} - \sum_i h_i \sigma_{X_i}$$



$$H = - \sum_{\langle i,j \rangle} J \left(\sigma_{X_i} \sigma_{X_j} + \sigma_{Y_i} \sigma_{Y_j} \right) - \sum_i h_i \sigma_{Z_i}$$

$$H = - \sum_{\langle i,j \rangle} \left(J_X \sigma_{X_i} \sigma_{X_j} + J_Y \sigma_{Y_i} \sigma_{Y_j} + J_Z \sigma_{Z_i} \sigma_{Z_j} \right) - \sum_i h_i \sigma_{Z_i}$$

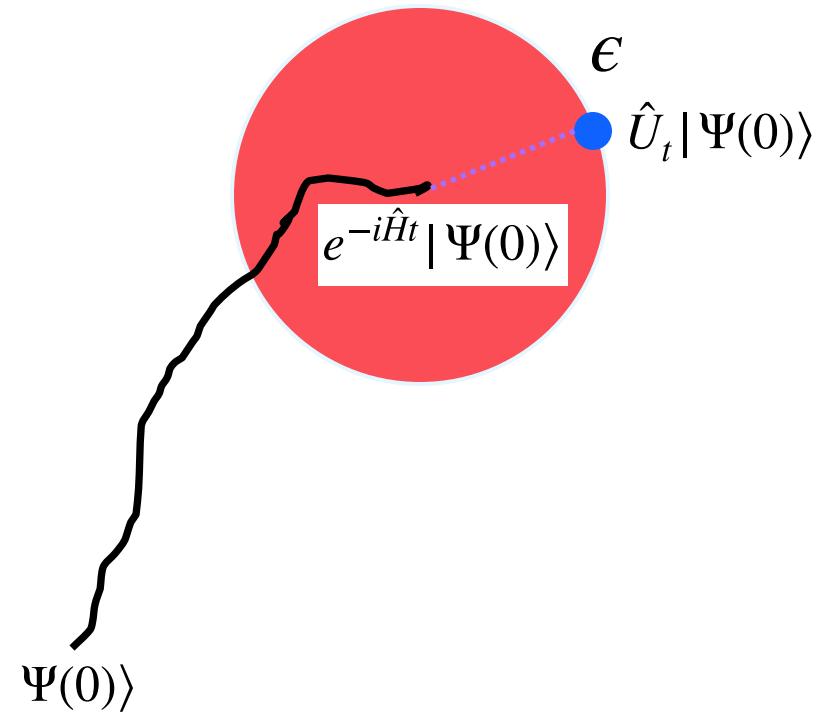
量子シミュレーションのアルゴリズム

ハミルトニアンが既知でも、 $e^{-i\hat{H}t}$ の計算方法は自明ではない
正確に計算することは難しい

次の式を満たすUを実装する $\|\hat{U}|\Psi\rangle - e^{-i\hat{H}t}|\Psi\rangle\| \leq \epsilon$

- 効率的に計算するのに幾つかの戦略がある
 - 誤差を小さく保つ
 - 回路深さを浅く保つ
- 戰略
 - Trotter 公式
 - Randomization (QDrift)
 - ”ポストTrotter”
 - ユニタリー演算子の線形組み合わせ
 - Qubitization (量子シグナルプロセッシング)

$$|\Psi(t)\rangle = e^{-i\hat{H}t}|\Psi(0)\rangle$$



Trotterization

仮定：ハミルトニアンが k -local (P は高々“ k ”個のPauliストリングに作用する)

$$\hat{H} = \sum_{i=1}^L a_i P_i$$

2項からなるハミルトニアンを考える

$$\hat{H} = \hat{H}_1 + \hat{H}_2$$

Lie Product公式：

$$e^{-it(H_1+H_2)} = \lim_{n \rightarrow \infty} \left(e^{-iH_1 \frac{t}{n}} e^{-iH_2 \frac{t}{n}} \right)^n$$

“ n ”を有限に取ると次式が成り立つ

$$e^{-i(\hat{H}_1+\hat{H}_2)\Delta t} = e^{-i\hat{H}_1\Delta t} e^{-i\hat{H}_2\Delta t}$$

ただし H_1 と H_2 が可換な場合に限る (大体非可換...)

memo Baker-Campbell-Hausdorff 公式

$e^X e^Y = e^Z$ を満たす Z は

$$Z = A + B + \frac{1}{2}[A, B] + \frac{1}{12}[A - B, [A, B]] + \dots$$

という形式で書ける。

$e^{X/n} e^{Y/n} = e^Z$ を満たす Z は

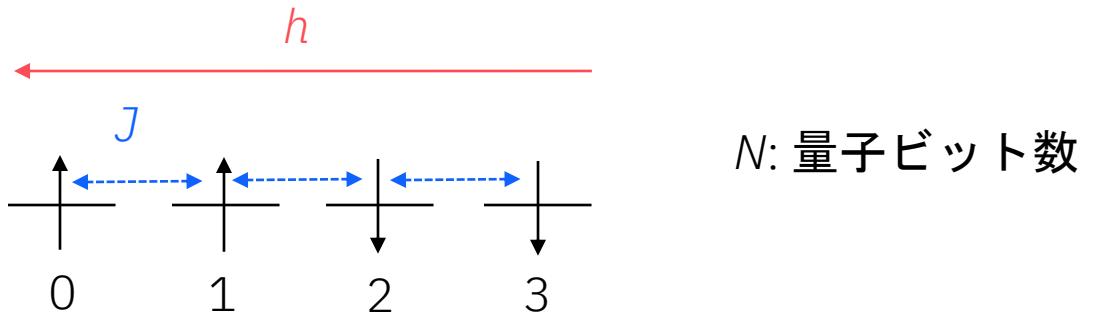
$$Z = A/n + B/n + \frac{1}{2n^2}[A, B] +$$

$$\frac{1}{12n^3}[A - B, [A, B]] + \dots$$

という形式で書ける。

例: Trotterization (一次の誤差まで) 横磁場イジングモデル

$$H = - \sum_{\langle i,j \rangle}^{N-1} J \sigma_{Z_i} \sigma_{Z_j} - \sum_i^N h_i \sigma_{X_i}$$



$$e^{-i\hat{H}\Delta t} = e^{-i\Delta t(-\sum_{i,j}^N J \sigma_{Z_i} \sigma_{Z_j} - \sum_i^N h_i \sigma_{X_i})} \approx e^{-i\Delta t(-\sum_{i,j}^N J \sigma_{Z_i} \sigma_{Z_j})} e^{-i\Delta t(-\sum_i^N h_i \sigma_{X_i})}$$

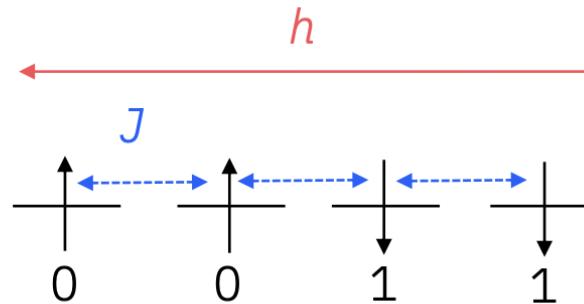
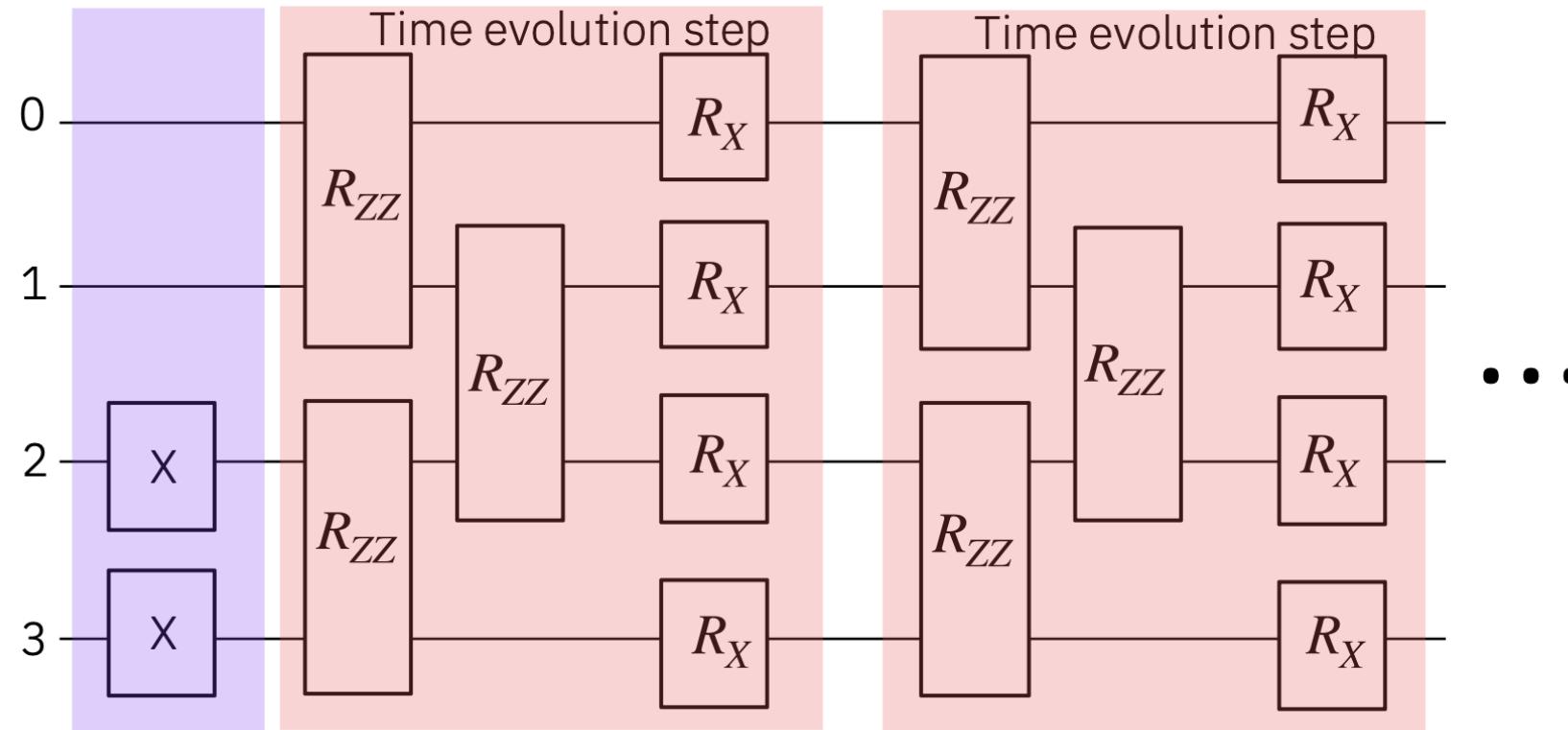
$R_{ZZ}(-2J\Delta t)$ $R_X(-2h\Delta t)$

$$R_{ZZ}(\theta) = e^{-i\frac{\theta}{2}\sigma_Z\sigma_Z} = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 & 0 & 0 \\ 0 & e^{i\frac{\theta}{2}} & 0 & 0 \\ 0 & 0 & e^{i\frac{\theta}{2}} & 0 \\ 0 & 0 & 0 & e^{-i\frac{\theta}{2}} \end{pmatrix}$$

$$R_X(\theta) = e^{-i\frac{\theta}{2}\sigma_X} = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}$$

例: 横磁場Ising model

$$H = - \sum_{\langle i,j \rangle}^{N-1} J \sigma_{Z_i} \sigma_{Z_j} - \sum_i^N h_i \sigma_{X_i}$$



0: up spin

Bit strings

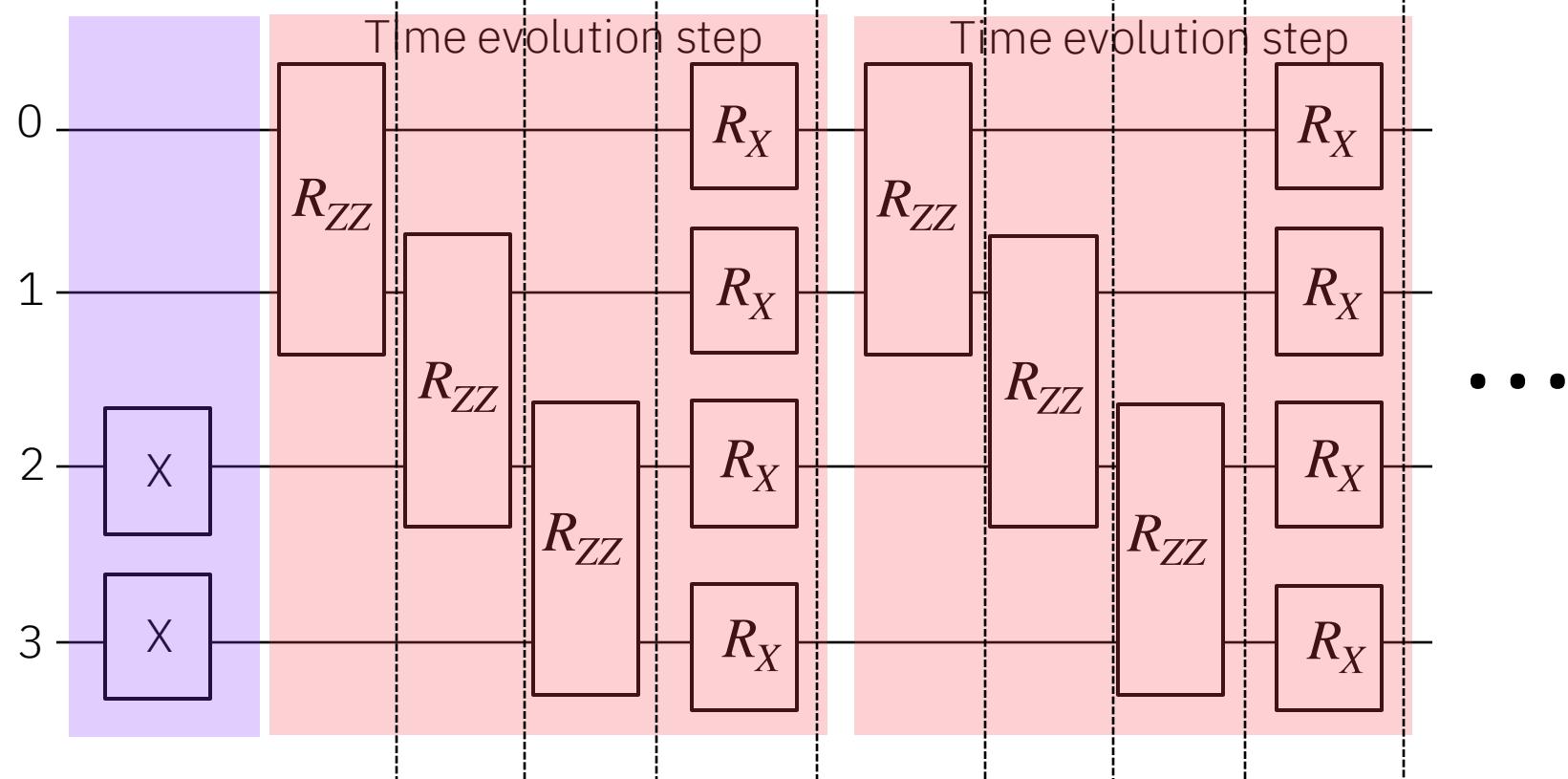
$|0011\rangle$

$|1100\rangle$

$$|\Psi(t)\rangle = e^{-i\hat{H}t} |\Psi(0)\rangle$$

回路の複雑さ

$$H = - \sum_{\langle i,j \rangle}^{N-1} J \sigma_{Z_i} \sigma_{Z_j} - \sum_i^N h_i \sigma_{X_i}$$



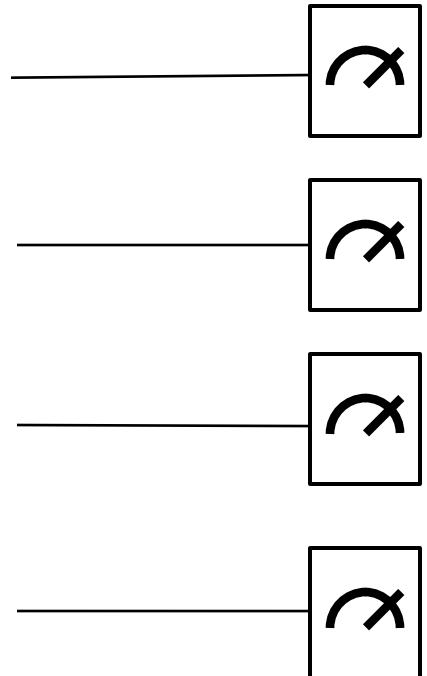
2量子ビットゲートの数や回路の深さを少なくすることが、計算精度の改善のために重要

0: up spin

1: down spin

Bit strings

$$\begin{array}{c} |0011\rangle \\ |1100\rangle \end{array}$$



磁化

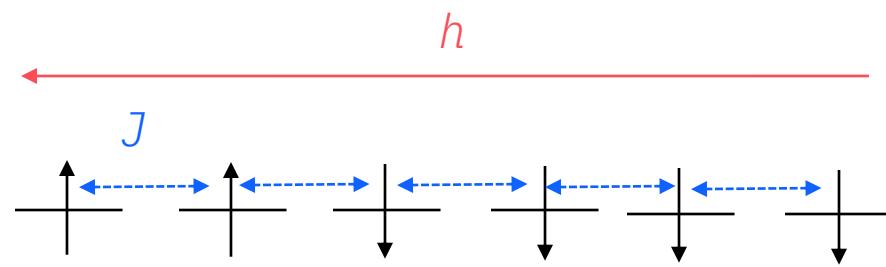
$$H = - \sum_{\langle i,j \rangle}^{N-1} J \sigma_{Z_i} \sigma_{Z_j} - \sum_i^N h_i \sigma_{X_i}$$

期待値

$$Z|0\rangle = |0\rangle \quad +1 \quad |0\rangle = |\uparrow\rangle$$

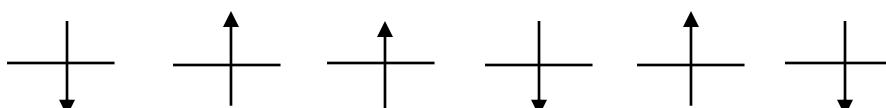
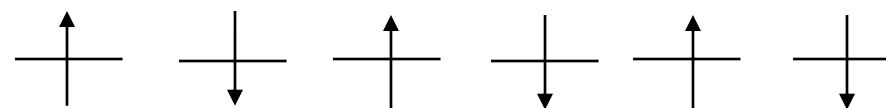
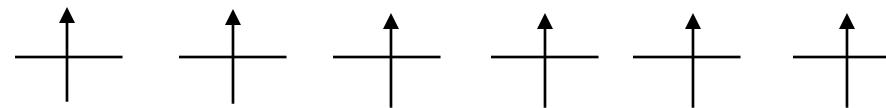
$$Z|1\rangle = -|1\rangle \quad -1 \quad |1\rangle = |\downarrow\rangle$$

- 強磁性
 - 隣接サイトとスピンが揃う
 - 常温で磁石となる（鉄）
- 反強磁性
 - 隣接サイトとスピンの向きが逆になる
 - Insulator (MnO)
- パラマグネティック (Paramagnetic)
 - 外場が存在しないと磁化しない



磁化

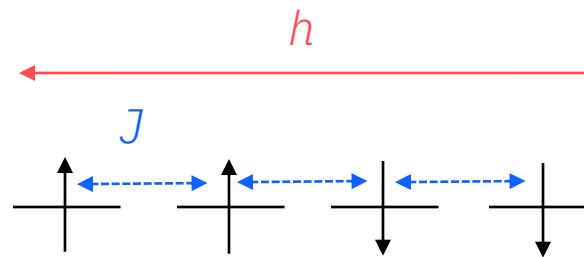
$$\sum_i^N Z_i / N$$



Metric for describing the magnetic state

1次元横磁場イジングモデルの基底エネルギー状態

$$H = - \sum_{\langle i,j \rangle}^{N-1} J \sigma_{Z_i} \sigma_{Z_j} - \sum_i^N h_i \sigma_{X_i}$$

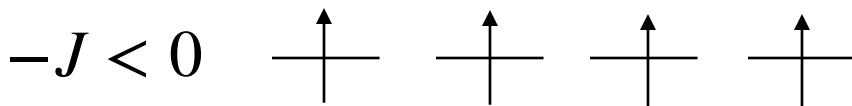


磁化
$$\sum_i^N Z_i / N$$

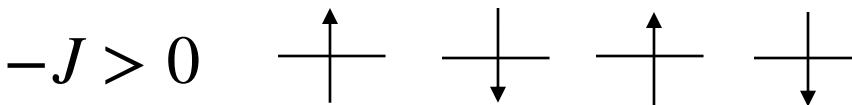
- 相互作用エネルギー

基底エネルギー状態 (h の項を無視した場合) $\sigma_{X_i}, \sigma_{Y_i}, \sigma_{Z_i} = X_i, Y_i, Z_i$

$$Z_k Z_{k+1} = 1 \quad (1,1), (-1, -1)$$



$$Z_k Z_{k+1} = -1 \quad (1, -1), (-1, 1)$$

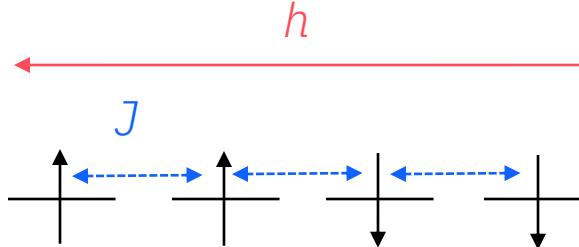


係数 h が大きい場合: スピンの向きの配置の秩序が失われる

基底エネルギー状態は係数に符号によって異なる

ダイナミカルな量子相転移

$$H = - \sum_{\langle i,j \rangle}^{N-1} J \sigma_{Z_i} \sigma_{Z_j} - \sum_i^N h_i \sigma_{X_i}$$



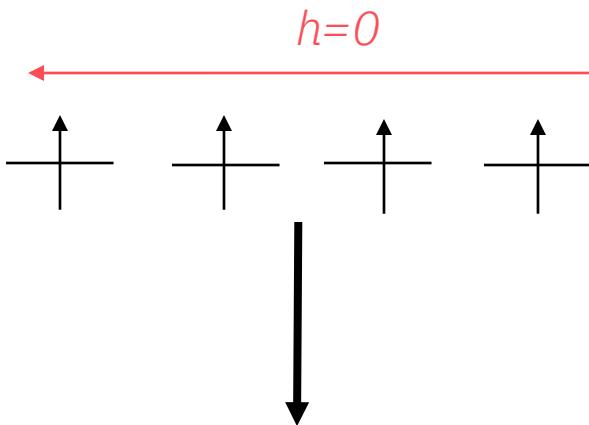
磁化

$$\sum_i^N Z_i / N$$

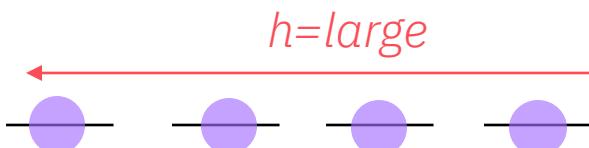
- 非平衡プロセスによる相転移

- 外部磁場の急な印加による磁気の相はどのように変化するのか？

Ferromagnetic ($-J < 0$)



?



磁化の時間発展を観察する

実際の計算例の紹介

- 磁化の時間発展を観察する
 - 理想的なシミュレーターによる量子シミュレーション
 - 20量子ビット問題
 - state-vector と matrix product state シミュレーターによる計算
 - 量子コンピュータ実機を用いた量子シミュレーション
 - 70量子ビット問題
 - matrix product state シミュレーターによる計算
 - 実機を用いた計算

計算モデル

一次元横磁場イジングモデル

$$H = -J \sum_{i=0}^{N-2} Z_i Z_{i+1} - h \sum_{i=0}^{N-1} (\sin \alpha Z_i + \cos \alpha X_i).$$

```
1 # Check the version of Qiskit
2 import qiskit
3
4 qiskit.__version__
```



Qiskit library のインポート

```
1 # Import the qiskit library
2
3 import numpy as np
4 import warnings
5
6 from qiskit import QuantumCircuit, QuantumRegister
7 from qiskit.circuit.library import PauliEvolutionGate
8 from qiskit.quantum_info import SparsePauliOp
9 from qiskit.synthesis import LieTrotter
10 from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager
11
12 from qiskit_aer import AerSimulator
13 from qiskit_ibm_runtime import QiskitRuntimeService, Estimator
14
15 warnings.filterwarnings("ignore")
```



20量子ビット問題 ハミルトニアン構築

ハミルトニアンの定義

```
1 def get_hamiltonian(nqubits, J, h):
2     # List of Hamiltonian terms as 3-tuples containing
3     # (1) the Pauli string,
4     # (2) the qubit indices corresponding to the Pauli string,
5     # (3) the coefficient.
6     ZZ_tuples = [("ZZ", [i, i + 1], -J) for i in range(0, nqubits - 1)]
7     X_tuples = [("X", [i], -h) for i in range(0, nqubits)]
8
9     # We create the Hamiltonian as a SparsePauliOp, via the method
10    # `from_sparse_list`, and multiply by the interaction term.
11    hamiltonian = SparsePauliOp.from_sparse_list(
12        [*ZZ_tuples, *X_tuples], num_qubits=nqubits
13    )
14    return hamiltonian.simplify()
```

パラメータ定義：量子ビット数（20）、係数（J, h）

```
1 n_qubits = 20
2 hamiltonian = get_hamiltonian(nqubits=n_qubits, J=1.0, h=-5.0)
3 hamiltonian
```

20量子ビット問題 時間発展パラメータ/初期状態定義

時間発展パラメータの定義 : Lie-Product公式の一次まで

```
1 num_timesteps = 20
2 evolution_time = 2.0
3 dt = evolution_time / num_timesteps
4 product_formula_lt = LieTrotter()
```



初期状態定義 : 強磁性体の基底状態 (ここでは全てスピン上向き状態を採用)

```
1 initial_circuit = QuantumCircuit(n_qubits)
2 initial_circuit.prepare_state("000000000000000000000000")
3 # Change reps and see the difference when you decompose the circuit
4 initial_circuit.decompose(reps=1).draw("mpl")
```



20量子ビット問題 時間発展量子回路生成

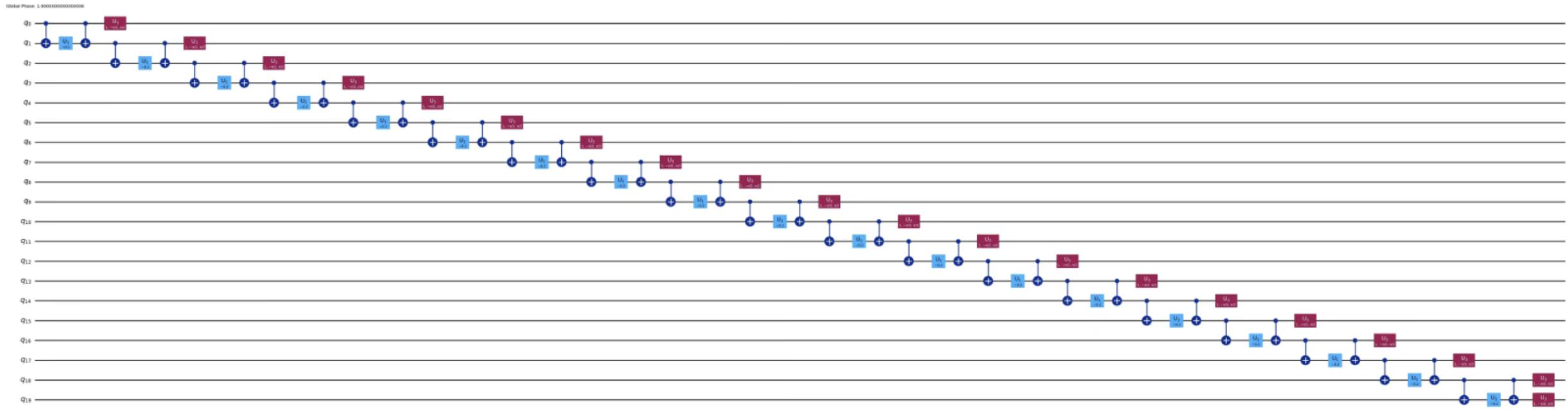
時間発展量子回路生成 : LieTrotterクラス（一次）を利用

$$e^{H_1+H_2} \approx e^{H_1}e^{H_2}$$

```
1 single_step_evolution_gates_lt = PauliEvolutionGate(□
2     hamiltonian, dt, synthesis=product_formula_lt
3 )
4 single_step_evolution_lt = QuantumCircuit(n_qubits)
5 single_step_evolution_lt.append(
6     single_step_evolution_gates_lt, single_step_evolution_lt.qubits
7 )
8
9 print(
10    f"""
11 Trotter step with Lie-Trotter
12 -----
13 Depth: {single_step_evolution_lt.decompose(reps=3).depth()}
14 Gate count: {len(single_step_evolution_lt.decompose(reps=3))}
15 Nonlocal gate count: {single_step_evolution_lt.decompose(reps=3).num_nonlocal_gates()}
16 Gate breakdown: {", ".join([f"{k.upper()}: {v}" for k, v in single_step_evolution_lt.
17   """])
18 }
19 single_step_evolution_lt.decompose(reps=3).draw("mpl", fold=-1)
```

20量子ビット問題 時間発展量子回路生成

時間発展量子回路生成：出力



```
Trotter step with Lie-Trotter
-----
Depth: 58
Gate count: 77
Nonlocal gate count: 38
Gate breakdown: CX: 38, U3: 20, U1: 19
```

20量子ビット問題 観測オペレータ（オブザーバブル）の定義

観測オペレータの定義：磁化 $\sum_i Z_i/N$

```
1 magnetization = (
2     SparsePauliOp.from_sparse_list(
3         [("Z", [i], 1.0) for i in range(0, n_qubits)], num_qubits=n_qubits
4     )
5     / n_qubits
6 )
7 print("magnetization : ", magnetization)
```



出力

```
magnetization : SparsePauliOp(['IIIIIIIIIIIIIIIZ', 'IIIIIIIIIIIIIZI', 'IIIII
coefs=[0.05+0.j, 0.05+0.j, 0.05+0.j, 0.05+0.j, 0.05+0.j, 0.05+0.j, 0.05+0.j,
0.05+0.j, 0.05+0.j, 0.05+0.j, 0.05+0.j, 0.05+0.j, 0.05+0.j, 0.05+0.j,
0.05+0.j, 0.05+0.j, 0.05+0.j, 0.05+0.j])
```



20量子ビット問題 時間発展計算(二種のシミュレータ)の実行

バックエンド指定 (MPS, Statevector) 、Estimatorの設定等

```
1 # Step 1. Map the problem
2 # Initiate the circuit
3 evolved_state = QuantumCircuit(initial_circuit.num_qubits)
4 # Start from the initial spin configuration
5 evolved_state.append(initial_circuit, evolved_state.qubits)
6
7 # Define backend (simulator)
8 # MPS
9 backend_mps = AerSimulator(method="matrix_product_state")
10 # Statevector
11 backend_sv = AerSimulator(method="statevector")
12
13 # Set Runtime Estimator
14 # MPS
15 estimator_mps = Estimator(mode=backend_mps)
16 # Statevector
17 estimator_sv = Estimator(mode=backend_sv)
```



20量子ビット問題 時間発展計算(二種のシミュレータ)の実行

トランスパイル（量子回路を実行可能な形に最適化する作業）

```
18  
19 # Step 2. Optimize  
20 # Set pass manager  
21 # MPS  
22 pm_mps = generate_preset_pass_manager(optimization_level=3, backend=backend_mps)  
23 # Statevector  
24 pm_sv = generate_preset_pass_manager(optimization_level=3, backend=backend_sv)  
25  
26 # Transpile initial circuit  
27 # MPS  
28 evolved_state_mps = pm_mps.run(evolved_state)  
29 # Statevector  
30 evolved_state_sv = pm_sv.run(evolved_state)  
31  
32 # Apply layout to the operator  
33 # MPS  
34 magnetization_mps = magnetization.apply_layout(evolved_state_mps.layout)  
35 # Statevector  
36 magnetization_sv = magnetization.apply_layout(evolved_state_sv.layout)  
37  
38 mag_mps_list = []  
39 mag_sv_list = []  
40
```

20量子ビット問題 時間発展計算(二種のシミュレータ)の実行

時間発展量子回路の実行：MPS, Statevectorシミュレータの時刻0での期待値を計算

```
41 # Step 3. Run the circuit
42 # Estimate expectation values for t=0.0: MPS
43 job = estimator_mps.run([(evolved_state_mps, [magnetization_mps])])
44 # Get estimated expectation values: MPS
45 evs = job.result()[0].data.evs
46 # Collect data: MPS
47 mag_mps_list.append(evs[0])
48
49 # Estimate expectation values for t=0.0: Statevector
50 job = estimator_sv.run([(evolved_state_sv, [magnetization_sv])])
51 # Get estimated expectation values: Statevector
52 evs = job.result()[0].data.evs
53 # Collect data: Statevector
54 mag_sv_list.append(evs[0])
55
```

20量子ビット問題 時間発展計算(二種のシミュレータ)の実行

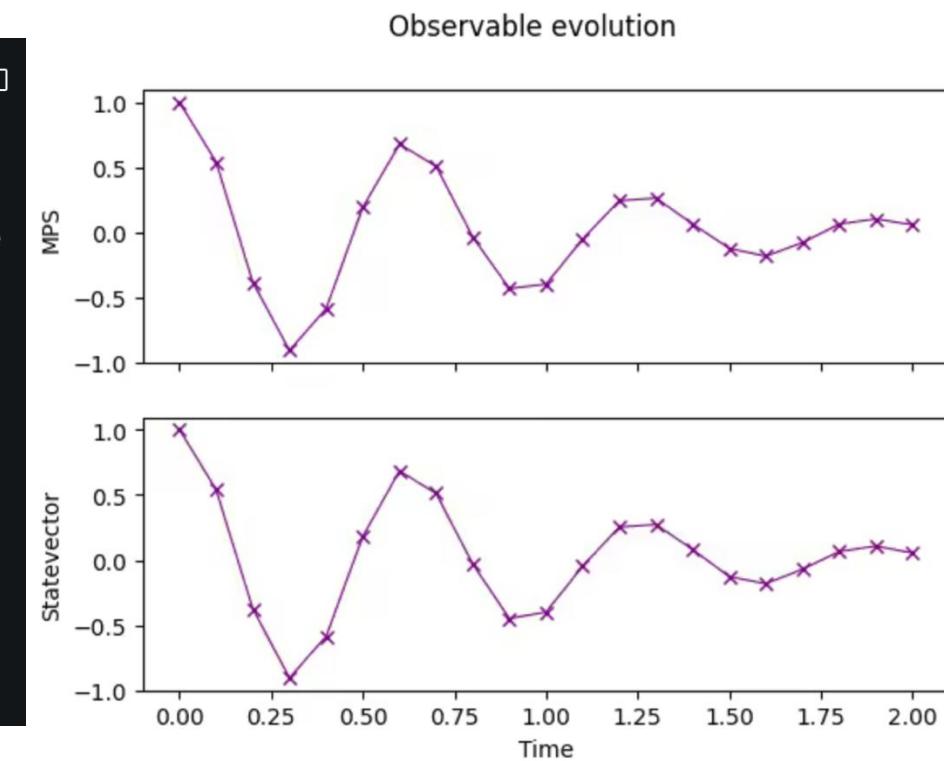
時間発展量子回路の実行：一連のプロセスをタイムステップごとに繰り返し計算

```
56 # Start time evolution
57 for n in range(num_timesteps):
58     # Step 1. Map the problem
59     # Expand the circuit to describe delta-t
60     evolved_state.append(single_step_evolution_lt, evolved_state.qubits)
61     # Step 2. Optimize
62     # Transpile the circuit: MPS
63     evolved_state_mps = pm_mps.run(evolved_state)
64     # Apply the physical layout of the qubits to the operator: MPS
65     magnetization_mps = magnetization.apply_layout(evolved_state_mps.layout)
66     # Step 3. Run the circuit
67     # Estimate expectation values at delta-t: MPS
68     job = estimator_mps.run([(evolved_state_mps, [magnetization_mps])])
69     # Get estimated expectation values: MPS
70     evs = job.result()[0].data.evs
71     # Collect data: MPS
72     mag_mps_list.append(evs[0])
73
74     # Step 2. Optimize
75     # Transpile the circuit: Statevector
76     evolved_state_sv = pm_sv.run(evolved_state)
77     # Apply the physical layout of the qubits to the operator: Statevector
78     magnetization_sv = magnetization.apply_layout(evolved_state_sv.layout)
79     # Step 3. Run the circuit
80     # Estimate expectation values at delta-t: Statevector
81     job = estimator_sv.run([(evolved_state_sv, [magnetization_sv])])
82     # Get estimated expectation values: Statevector
83     evs = job.result()[0].data.evs
84     # Collect data: Statevector
85     mag_sv_list.append(evs[0])
86
87     # Transform the list of expectation values (at each time step) to arrays
88     mag_mps_array = np.array(mag_mps_list)
89     mag_sv_array = np.array(mag_sv_list)
```

20量子ビット問題 時間発展計算(二種のシミュレータ)の実行

オブザーバブル（磁化）の期待値の時間発展をプロットする

```
1 import matplotlib.pyplot as plt
2
3 # Step 4. Post-processing
4 fig, axes = plt.subplots(2, sharex=True)
5 times = np.linspace(0, evolution_time, num_timesteps + 1) # includes initial state
6 axes[0].plot(
7     times, mag_mps_array, label="MPS", marker="x", c="darkmagenta", ls="-", lw=0.8
8 )
9 axes[1].plot(
10    times, mag_sv_array, label="SV", marker="x", c="darkmagenta", ls="-", lw=0.8
11 )
12
13 axes[0].set_ylabel("MPS")
14 axes[1].set_ylabel("Statevector")
15 axes[1].set_xlabel("Time")
16 fig.suptitle("Observable evolution")
```



70量子ビット問題

量子ビット数を70に設定する

```
1 # Set the number of qubits  
2 n_qubits2 = 70  
3 # Construct the Hamiltonian by calling the function you made in Act  
4 hamiltonian2 = get_hamiltonian(nqubits=n_qubits2, J=1.0, h=-5.0)  
5 hamiltonian2
```

その後、20量子ビット問題と同様に、初期状態を定義し、LieTrotterクラスを用いて量子回路を生成

70量子ビット問題

生成された量子回路（70量子ビット）

```
Trotter step with Lie-Trotter
-----
Depth: 208
Gate count: 277
Nonlocal gate count: 138
Gate breakdown: CX: 138, U3: 70, U1: 69
```

回路の空白が多く、回路が深くなる。誤差が大きくなり、計算精度が悪化する

70量子ビット問題 実機計算：回路作成、トランスパイル

実機計算のための準備：バックエンド指定

```
1 service = QiskitRuntimeService()  
2 backend = service.least_busy(operational=True, simulator=False)  
3 backend.name
```

実機計算のための準備：量子回路作成、トランスパイル

```
1 pm_hw = generate_preset_pass_manager(optimization_level=3, backend=backend)  
2 circuit_isa = []  
3 # Step 1. Map the problem  
4 evolved_state_hw = QuantumCircuit(initial_circuit2.num_qubits)  
5 evolved_state_hw.append(initial_circuit2, evolved_state_hw.qubits)  
6 # Step 2. Optimize  
7 circuit_isa.append(pm_hw.run(evolved_state_hw))  
8  
9 for n in range(num_timesteps):  
10     # Step 1. Map the problem  
11     evolved_state_hw.append(single_step_evolution_lt2, evolved_state_hw.qubits)  
12     # Step 2. Optimize  
13     circuit_isa.append(pm_hw.run(evolved_state_hw))
```

70量子ビット問題 実機計算：ジョブ実行

Estimatorの定義

```
1 # Step 2. Optimize  
2 estimator_hw = Estimator(mode=backend)  
3 pub_list = []  
4 for circuit in circuit_isas:  
5     temp = (circuit, magnetization2.apply_layout(circuit.layout))  
6     pub_list.append(temp)
```

ジョブ実行

```
1 job = estimator_hw.run(pub_list)  
2 job_id = job.job_id()  
3 print(job_id)
```

ジョブ実行ステータス確認

```
1 # check job status  
2 job.status()
```

70量子ビット問題 実機計算：ポストプロセス

結果取得

```
1 | job = service.job(job_id)
2 | pub_result = job.result()
```



期待値抽出

```
1 | mag_hw_list = []
2 | for res in pub_result:
3 |     evs = res.data.evs
4 |     mag_hw_list.append(evs)
```



70量子ビット問題 量子回路の改善

量子回路の改善：より浅い回路の作成

```
1 # Define J
2 J = 1.0
3 # Define h
4 h = -5.0
5 # Create instruction for rotation around ZZ:
6 # Initiate the circuit (use 2 qubits)
7 Rzz_circ = QuantumCircuit(2)
8 # Add Rzz gate (do not forget to multiply the angle by 2.0)
9 Rzz_circ.rzz(-J * dt * 2.0, 0, 1)
10 # Transform the QuantumCircuit to instruction (QuantumCircuit.to_instruction())
11 Rzz_instr = Rzz_circ.to_instruction(label="RZZ")
12
13 # Create instruction for rotation around X:
14 # Initiate the circuit (use 1 qubit)
15 Rx_circ = QuantumCircuit(1)
16 # Add Rx gate (do not forget to multiply the angle by 2.0)
17 Rx_circ.rx(-h * dt * 2.0, 0)
18 # Transform the QuantumCircuit to instruction (QuantumCircuit.to_instruction())
19 Rx_instr = Rx_circ.to_instruction(label="RX")
20
21 # Define the interaction list
22 interaction_list =
23     [[i, i + 1] for i in range(0, n_qubits2 - 1, 2)],
24     [[i, i + 1] for i in range(1, n_qubits2 - 1, 2)],
25 ] # linear chain
```



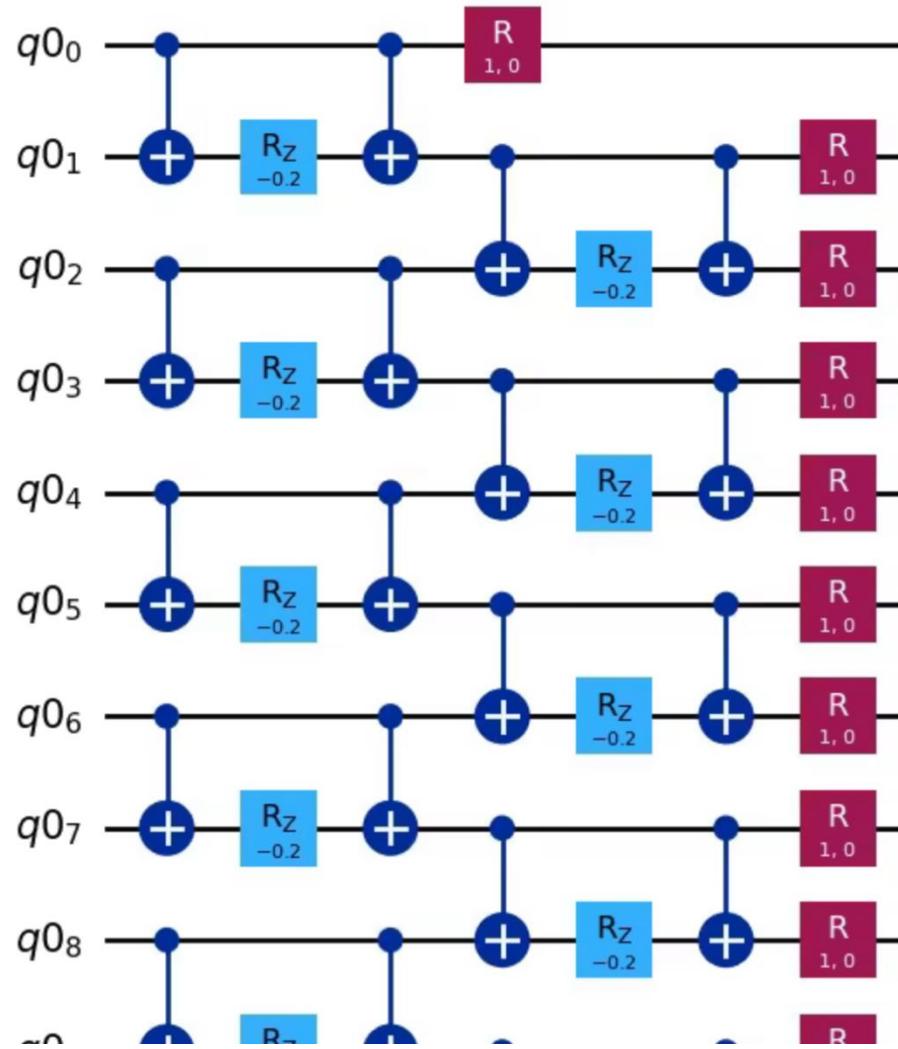
70量子ビット問題 量子回路の改善

量子回路の改善：より浅い回路の作成

```
26
27     # Define the registers
28     qr = QuantumRegister(n_qubits2)
29     # Initiate the circuit
30     single_step_evolution_sh = QuantumCircuit(qr)
31     # Construct the Rzz gates
32     for i, color in enumerate(interaction_list):
33         for interaction in color:
34             single_step_evolution_sh.append(Rzz_instr, interaction)
35
36     # Construct the Rx gates
37     for i in range(0, n_qubits2):
38         single_step_evolution_sh.append(Rx_instr, [i])
39
40     print(
41         f"""
42         Trotter step with Lie-Trotter
43         -----
44         Depth: {single_step_evolution_sh.decompose(reps=3).depth()}
45         Gate count: {len(single_step_evolution_sh.decompose(reps=3))} 
46         Nonlocal gate count: {single_step_evolution_sh.decompose(reps=3).num_nonlocal_gates()}
47         Gate breakdown: {", ".join([f"{k.upper()}: {v}" for k, v in single_step_evolution_sh.
48             """])
49     )
50
51     single_step_evolution_sh.decompose(reps=2).draw("mpl")
```

70量子ビット問題 量子回路の改善

量子回路の改善：より浅い回路の作成



Trotter step with Lie-Trotter

Depth: 7

Gate count: 277

Nonlocal gate count: 138

Gate breakdown: CX: 138, U3: 70, U1: 69

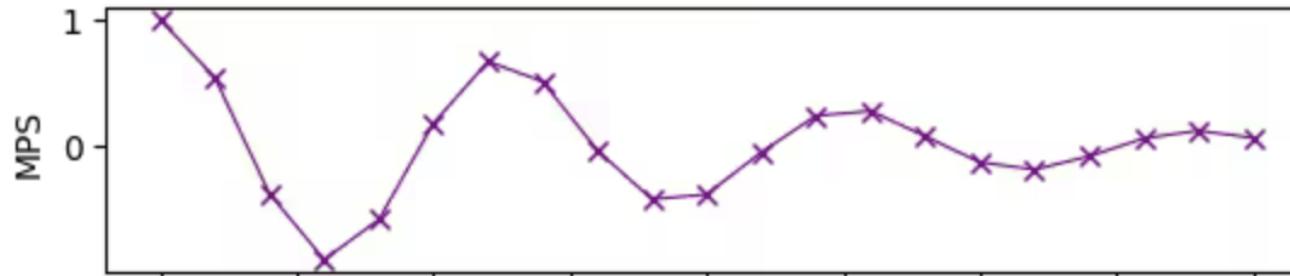
ゲート配置を工夫することにより、深い量子回路の作成が可能

70量子ビット問題 量子回路の改善

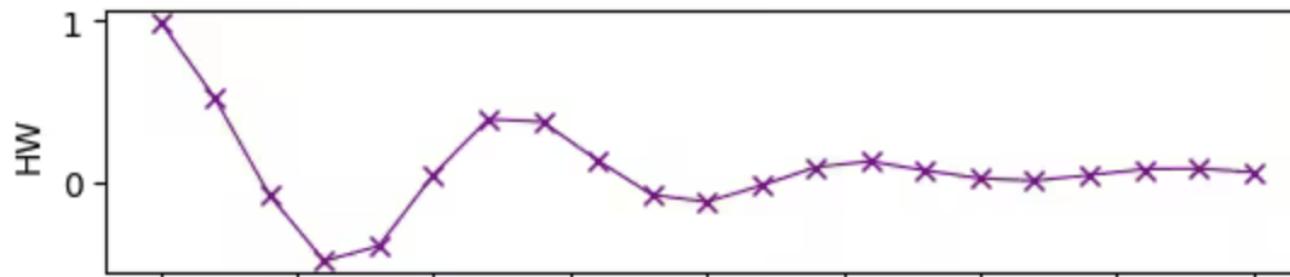
ノイズレスシミュレーション (MPS)

計算結果

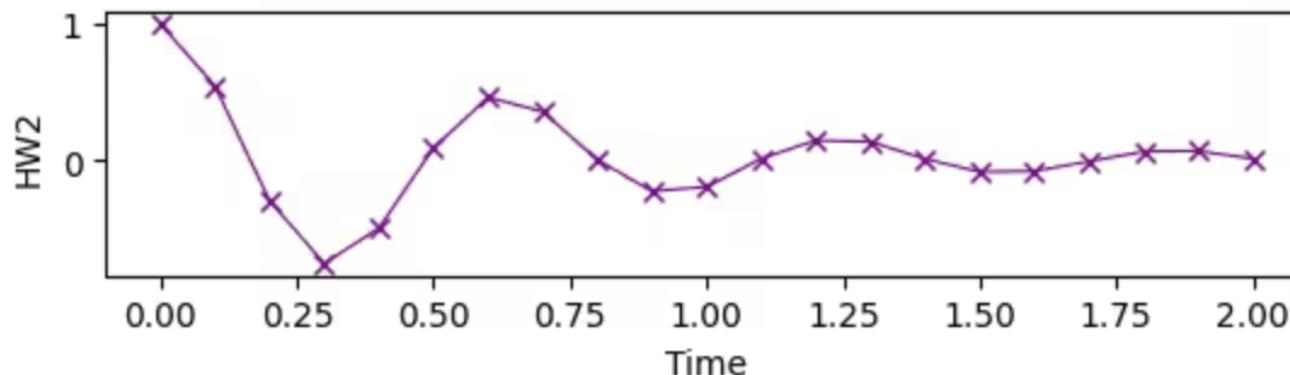
Observable evolution



実機（深い回路）



実機（改善した浅い回路）



まとめ

1. 量子シミュレーションの振り返り（ハミルトニアンシミュレーション、量子ダイナミクス）
 1. ハミルトニアン（モデル）
 2. トロッタ分解
2. シミュレータ及び量子コンピュータによる実機計算の紹介
 1. 20-qubit 問題 (state-vector と matrix product state シミュレータによる計算)
 2. 70-qubit 問題 (matrix product state シミュレータと 量子コンピュータによる計算)

最後に：もっと勉強したい方へ

Jupyter notebookの和訳版



三



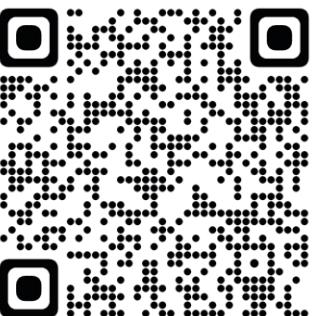
ユーティリティー・スケール量子コンピューティング

概要

このイベント・リプレイ・コースは、IBM Quantum®が東京大学と共同で開発し実施した14のLessonとLabで構成されています。このコースでは、量子コンピューティングにおける幅広い重要なトピックを網羅しつつ、実用規模（ユーティリティー・スケール）の量子計算を構築することに重点を置いています。最終的な結果として、2023年6月にNature誌の表紙を飾った論文と非常によく似た課題を扱います。

翻訳元はこれらです：[IBM Quantum Learning の Utility-scale quantum computing](#)

1. はじめに
 2. 量子ビット・量子ゲート・量子回路
 3. 量子テレポーテーション
 4. グローバーのアルゴリズム
 5. 量子位相推定
 6. 量子変分アルゴリズム
 7. 量子系のシミュレーション
 8. 古典計算によるシミュレーション
 9. 量子ハードウェア
 10. 量子回路の最適化
 11. 量子エラー緩和
 12. 量子ユーティリティーの実験 I
 13. 量子ユーティリティーの実験 II
 14. 量子ユーティリティーの実験 III



<https://quantum-tokyo.github.io/introduction/courses/utility-scale-quantum-computing/overview-ja.html>

これまでのセッションの録画



<https://www.youtube.com/playlist?list=PLA-UlpvIBvpuzFXRPNTqiK94kfRgYCBMs>