

# 虎の巻：補足解説

出題範囲 (Section)	出題範囲 (Subsection)	解説テーマ	チェック
Section 1: Perform Operations on Quantum Circuits	a. Construct multi-qubit quantum registers	量子回路の作成	
	b. Measure quantum circuits in classical registers	measure( ) methodの使い方	
	d. Use multi-qubit gates	各種ゲートと等価な回路	
	e. Use barrier operations	Barrierのはたらき	
	f. Return the circuit depth	Circuit Depth	
	h. Return the OpenQASM string for a circuit	OpenQASMへの変換とファイル出力	
Section 3: Implement BasicAer: Python-based Simulators	a. Use the available simulators	BasicAerで利用できるシミュレータ	
Section 4: Implement Qasm	a. Read a QASM file and string	OpenQASMの読み込み	
Section 5: Compare and Contrast Quantum Information	c. Measure fidelity	Fidelityの計算	
Section 6: Return the Experiment Results	a. Return and understand the histogram data of an experiment	実行結果をヒストグラムで表示する方法	
	b. Return and understand the statevector of an experiment	実行結果をstatevectorで表示する方法	
	c. Return and understand the unitary of an experiment	実行結果をunitaryで表示する方法	

出題範囲 (Section)	出題範囲 (Subsection)	解説テーマ	チェック
Section 7: Use Qiskit Tools	a. Monitor the status of a job instance	ジョブ・インスタンスのステータス確認	
Section 8: Display and Use System Information	a. Perform operations around the Qiskit version	Qiskitのバージョンの確認	
	b. Use information gained from %qiskit_backend_overview	%qiskit_backend_overviewの表示内容	
Section 9: Construct Visualizations	b. Plot a histogram of data	ヒストグラムの表示内容	
	c. Plot a Bloch multivector	plot_bloch_multivector( )の表示内容	
	d. Plot a Bloch vector	plot_bloch_vector( )の表示内容	
	e. Plot a QSphere	plot_state_qsphere()の表示内容	
	g. Plot a gate map with error rates	plot_error_map( )の表示内容	
Section 10: Access Aer Provider	a. Access a statevector_simulator backend	statevector_simulatorの実行結果	
	b. Access a qasm_simulator backend	qasm_simulatorの実行結果	
	c. Access a unitary_simulator backend	unitary_simulatorの実行結果	

## 出題範囲

- Section 1: Perform Operations on Quantum Circuits
- a. Construct multi-qubit quantum registers

## 解説

- 量子回路の作成

```
qr = QuantumRegister(2, 'quantum')  
cr = ClassicalRegister(2, 'classical')
```

量子レジスタの定義

古典レジスタの定義

```
qc = QuantumCircuit(qr, cr)  
qc.draw('mpl')
```

量子回路の作成

- $quantum_0$  — ←量子レジスタ1
- $quantum_1$  — ←量子レジスタ2
- $classical$   $\frac{2}{=}$  ←古典レジスタ

## ポイント

- 別名を付けた場合にどのように表示されるか。  
左図の場合には、量子レジスタに quantum、古典レジスタに classical と別名を付けている。
- 簡易な作り方(例:  
`qc=QuantumCircuit(2,2)`)との違い。
- [参考]Qiskitドキュメント  
[QuantumCircuit — Qiskit 0.37.0 documentation](https://docs.quantum.ibm.com/api/qiskit/QuantumCircuit)

## 出題範囲

- Section 1: Perform Operations on Quantum Circuits
- b.Measure quantum circuits in classical registers

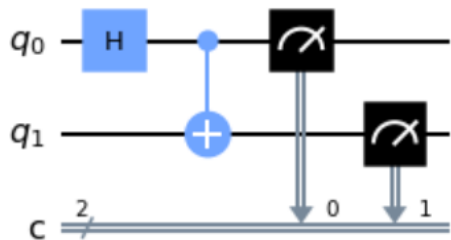
## 解 説

- measure( )

```
qc = QuantumCircuit(2, 2)
qc.h(0)
qc.cx(0, 1)
qc.measure((0, 1), (0, 1))

qc.draw('mpl')
```

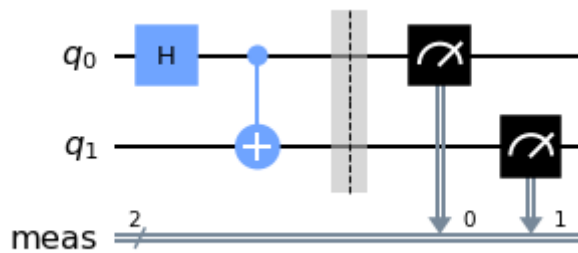
量子レジスタと古典レジスタをタプル形式、またはリスト形式で指定



- measure\_all( )

```
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
qc.measure_all()

qc.draw('mpl')
```



qiskitバージョン 0.34.2

## ポイント

- 古典レジスタがない場合、ビットを測定することはできないが、QuantumCircuit.measure\_all( )は使用できる。
- QuantumCircuit.measure( ) methodにて、測定する量子ビットと古典ビットの指定の方法。
- [参考]Qiskitドキュメント

[qiskit.circuit.QuantumCircuit.measure\\_all — Qiskit 0.37.0 documentation](https://docs.quantum.ibm.com/api/qiskit/circuit/QuantumCircuit.measure_all)

[qiskit.circuit.QuantumCircuit.measure — Qiskit 0.37.0 documentation](https://docs.quantum.ibm.com/api/qiskit/circuit/QuantumCircuit.measure)

## 出題範囲

- Section 1: Perform Operations on Quantum Circuits
- d.Use multi-qubit gates

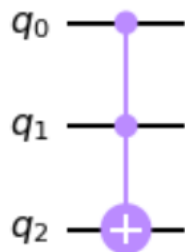
## 解 説

- ここでは各種のゲートと等価な回路を紹介する。
- Toffoliゲートは3量子ビットのゲートである。Qiskitではccxゲートで紹介されているが、mctゲートでも実装可能である：

```
In [88]:  from qiskit import QuantumCircuit
```

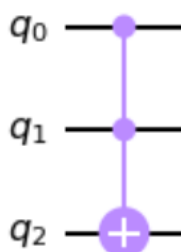
```
qc = QuantumCircuit(3)
qc.ccx(0,1,2)
qc.draw('mpl')
```

Out[88]:



```
In [2]:  qc = QuantumCircuit(3)
         qc.mct([0,1],[2])
         qc.draw('mpl')
```

Out[2]:



## ポイント

- 等価なゲートはさまざまなものがある。古典ゲートの論理的な動作と同様に、量子ゲートの論理的な動作を理解しておけば、等価な回路をすばやく選択できる。
- ccxゲートは、mctゲートでも実装ができる。

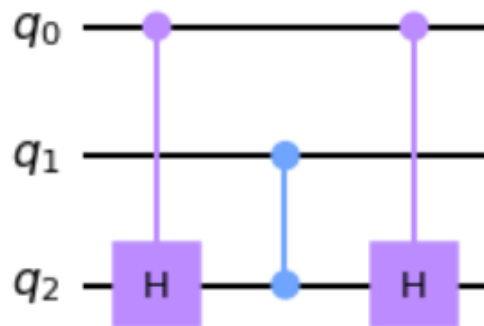
## 解 説

- ccxゲートを使ってToffoliゲートを実装したが、以下のような回路もToffoliゲートと等価である。(等価な回路はこのほかにもあり、一種類とは限らない):

```
In [89]: ► from qiskit import QuantumCircuit

qc = QuantumCircuit(3)
qc.ch(0,2)
qc.cz(1,2)
qc.ch(0,2)
qc.draw('mpl')
```

Out[89]:



## ポイント

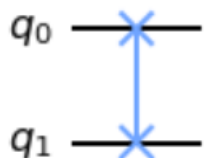
- Toffoliゲートの場合、入力→出力は
  - $|011\rangle \rightarrow |111\rangle$
  - $|111\rangle \rightarrow |011\rangle$の2つだけが入力≠出力となる。その他の場合は入力＝出力となる。

## 解 説

- Swapゲートと等価な回路のうち、3種類の等価な回路を示す:

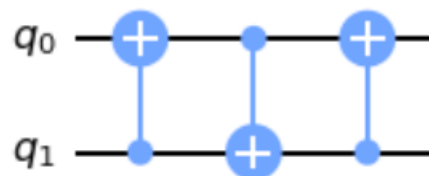
```
In [24]: ➤ qc_swap = QuantumCircuit(2)
qc_swap.swap(0,1)
qc_swap.draw('mpl')
```

Out[24]:



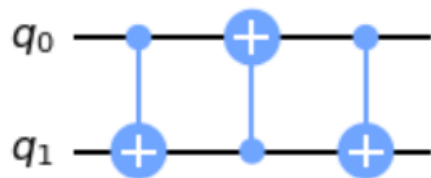
```
In [26]: ➤ qc_1 = QuantumCircuit(2)
qc_1.cnot(1,0)
qc_1.cnot(0,1)
qc_1.cnot(1,0)
qc_1.draw('mpl')
```

Out[26]:



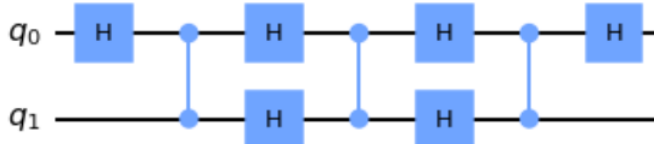
```
In [35]: ➤ qc_2 = QuantumCircuit(2)
qc_2.cnot(0,1)
qc_2.cnot(1,0)
qc_2.cnot(0,1)
qc_2.draw('mpl')
```

Out[35]:



```
In [38]: ➤ qc_3 = QuantumCircuit(2)
qc_3.h(0)
qc_3.cz(0,1)
qc_3.h(0)
qc_3.h(1)
qc_3.cz(0,1)
qc_3.h(0)
qc_3.h(1)
qc_3.cz(0,1)
qc_3.h(0)
qc_3.draw('mpl')
```

Out[38]:



## ポイント

- 等価な回路かどうかを確認するには、unitary\_simulatorを使って、回路を表わすユニタリ行列を作り、比較すればよい。

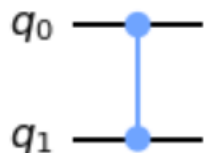


## 解 説

- 次にCZゲートと等価な回路の例を示す。
- CZゲートは制御ビットとターゲットビットを入れ替えても変わらない:

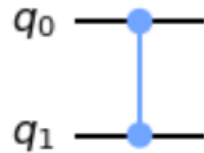
```
In [2]: ► qc_cz = QuantumCircuit(2)
        qc_cz.cz(0,1)
        qc_cz.draw('mpl')
```

Out[2]:



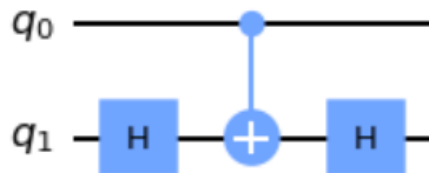
```
In [4]: ► qc_1 = QuantumCircuit(2)
        qc_1.cz(1,0)
        qc_1.draw('mpl')
```

Out[4]:



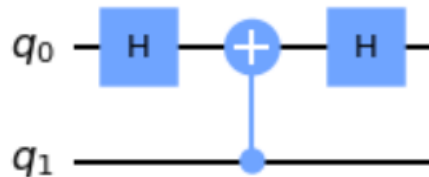
```
In [7]: ► qc_2 = QuantumCircuit(2)
        qc_2.h(1)
        qc_2.cx(0,1)
        qc_2.h(1)
        qc_2.draw('mpl')
```

Out[7]:



```
In [10]: ► qc_3 = QuantumCircuit(2)
        qc_3.h(0)
        qc_3.cx(1,0)
        qc_3.h(0)
        qc_3.draw('mpl')
```

Out[10]:



## ポイント

## 出題範囲

- Section 1: Perform Operations on Quantum Circuits
- e.Use barrier operations

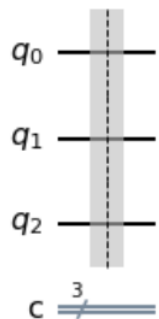
## 解 説

- Barrierはすべての量子ビットに設定することが多いと思われるが、一部の量子ビットだけに設定することも可能である:

```
In [4]: ► qc = QuantumCircuit(3,3)
        qc.barrier([0,1,2])

        qc.draw('mpl')
```

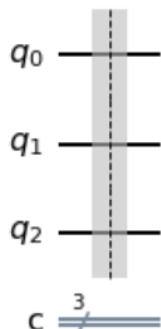
Out[4]:



```
In [5]: ► qc = QuantumCircuit(3,3)
        qc.barrier()

        qc.draw('mpl')
```

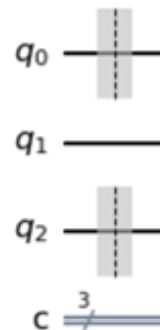
Out[5]:



```
In [6]: ► qc = QuantumCircuit(3,3)
        qc.barrier([0,2])

        qc.draw('mpl')
```

Out[6]:



## ポイント

- 量子回路は描画の際にゲートを左に詰めて表示するようになっているため、回路の構造を把握しづらい場合がある。
- ゲートの間にバリアを設定することで回路を見やすくすることができるが副作用があることを知ったうえで利用する(次頁参照)。

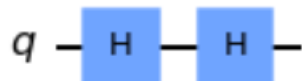
## 解 説

- Barrierを設定した場合には、両隣のゲートでキャンセルできる場合でも、キャンセルされずに量子回路が実行される。

## キャンセルされる

```
In [2]: qc = QuantumCircuit(1)
        qc.h(0)
        qc.h(0)
        qc.draw('mpl')
```

Out[2]:



```
In [3]: tqc = transpile(qc, lima)
        tqc.draw('mpl')
```

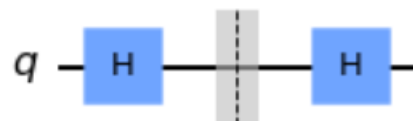
Out[3]:



## キャンセルされない

```
In [4]: qc = QuantumCircuit(1)
        qc.h(0)
        qc.barrier()
        qc.h(0)
        qc.draw('mpl')
```

Out[4]:



```
In [5]: tqc = transpile(qc, lima)
        tqc.draw('mpl')
```

Out[5]:



## ポイント

- 左記のように、barrierを設けたことによってゲートがキャンセルされない場合がある。

## 出題範囲

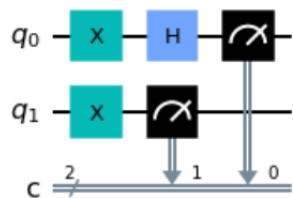
- Section 1: Perform Operations on Quantum Circuits
- f.Return the circuit depth

## 解 説

- Circuit Depth(量子回路の深さ)は、量子ゲートなどから構成される量子回路のクリティカルパスの長さと考えてよい。

```
In [18]: ► qc = QuantumCircuit(2,2)
          qc.x(0)
          qc.x(1)
          qc.h(0)
          qc.measure(0,0)
          qc.measure(1,1)
          qc.draw('mpl')
```

Out[18]:

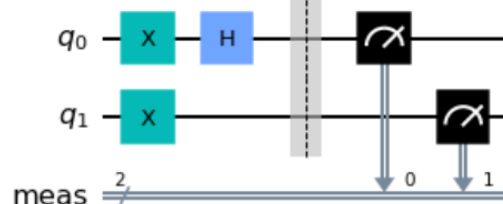


```
In [19]: ► qc.depth()
```

Out[19]: 3

```
In [14]: ► qc = QuantumCircuit(2)
          qc.x(0)
          qc.x(1)
          qc.h(0)
          qc.measure_all()
          qc.draw('mpl')
```

Out[14]:



```
In [15]: ► qc.depth()
```

Out[15]: 3

## ポイント

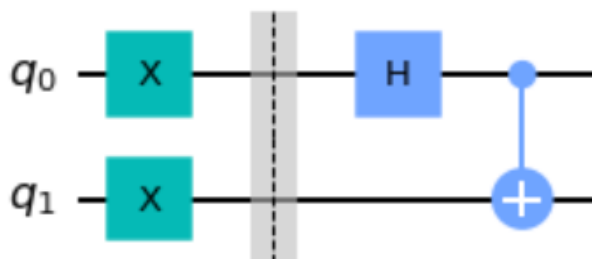
- Depthは量子回路で最も長いクリティカルパスを数えたもの。
- Measurementはdepthとしてカウントされる。measure、measure\_all()ともカウントされる。

## 解 説

Barrierはdepthとしてカウントされない。

```
In [49]: ► qc = QuantumCircuit(2)
          qc.x(0)
          qc.x(1)
          qc.barrier()
          qc.h(0)
          qc.cx(0,1)
          qc.draw('mpl')
```

Out[49]:



```
In [50]: ► qc.depth()
```

Out[50]: 3

## ポイント

- Barrierはdepthとしてカウントされない。

## 出題範囲

- Section 1: Perform Operations on Quantum Circuits
- h.Return the OpenQASM string for a circuit

## 解 説

- QuantumCircuit.qasmによるOpenQASMへの変換とファイル出力

```
qc = QuantumCircuit(1)
qc.x(0)
qc.qasm(formatted=True, filename='new_qc.qasm')
```

New\_qc.qasmという  
ファイル名で出力

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[1];
x q[0];
```

/new\_qc.qasm

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[1];
x q[0];
```

string(文字列)として作成される  
“OPENQASM 2.0;”で囲まれている

## ポイント

- 試験では、細かいオプションについては問われないが、OpenQASMのコードの変換とファイル出力
- OpenQASMのコマンドの違い、コマンド列の並び順の違いに注意 (Qiskitとの違い)
- [参考]Qiskitドキュメント  
[qiskit.circuit.QuantumCircuit.qasm — Qiskit 0.37.0 documentation](https://docs.quantum.ibm.com/api/qiskit/circuit/QuantumCircuit.qasm)

## 出題範囲

- Section 3: Implement BasicAer: Python-based Simulators
- a. Use the available simulators

## 解 説

- BasicAerで利用できるシミュレータは以下の3種類である:

```
In [1]:  from qiskit import BasicAer
```

```
In [2]:  BasicAer.backends()
```

```
Out[2]:  [<QasmSimulatorPy('qasm_simulator')>,  
          <StatevectorSimulatorPy('statevector_simulator')>,  
          <UnitarySimulatorPy('unitary_simulator')>]
```

## ポイント

- BasicAerはPythonベースのシミュレータであり、Qasm Simulator, Statevector Simulator, Unitary Simulator の3種類が利用できる。

## 解 説

- Aerで利用できるシミュレータは以下のとおりである:

```
In [1]: ▶ from qiskit import Aer
```

```
In [2]: ▶ Aer.backends()
```

```
Out[2]: [AerSimulator('aer_simulator'),  
AerSimulator('aer_simulator_statevector'),  
AerSimulator('aer_simulator_density_matrix'),  
AerSimulator('aer_simulator_stabilizer'),  
AerSimulator('aer_simulator_matrix_product_state'),  
AerSimulator('aer_simulator_extended_stabilizer'),  
AerSimulator('aer_simulator_unitary'),  
AerSimulator('aer_simulator_superop'),  
QasmSimulator('qasm_simulator'),  
StatevectorSimulator('statevector_simulator'),  
UnitarySimulator('unitary_simulator'),  
PulseSimulator('pulse_simulator')]
```

## ポイント

- AerはC++で書かれており、上記3種以外にAerSimulator, Pulse Simulatorが利用できる。



## 出題範囲

- Section 4: Implement Qasm
- a.Read a QASM file and string

## 解 説

- OpenQASMの読み込み


```
qasm = '''  
OPENQASM 2.0;  
include "qelib1.inc";  
qreg q[1];  
creg c[1];  
x q[0];  
'''
```

OpenQASM形式の  
回路を作成

Qiskit形式で  
読み込み

```
circuit = QuantumCircuit.from_qasm_str(qasm)  
circuit.draw()
```

<出力結果>

q:   
c: 1/═══════════

## ポイント

- `QuantumCircuit.from_qasm_str( )`で  
OpenQASMのコードを読み込むこと  
ができる
- ファイル読み込みは`from_qasm_file`  
のメソッド
- [参考]Qiskitドキュメント  
[qiskit.circuit.QuantumCircuit.from\\_qasm\\_str — Qiskit 0.37.0 documentation](https://docs.quantum.ibm.com/api/qiskit/circuit/QuantumCircuit.from_qasm_str)

## 出題範囲

- Section 5: Compare and Contrast Quantum Information
- c.Measure fidelity

## 解 説

- Qiskitではfidelityとして、
  - state\_fidelity
  - process\_fidelity
  - average\_gate\_fidelity

が提供されている。

▪ state\_fidelityは、2つの量子状態の近さ(=類似している)という忠実度を計算するもので、2つの状態ベクトルから計算する。

▪ process\_fidelityは、2つのオペレータ間の忠実度を計算するもので、2つのユニタリ行列の近さを表わす。

▪ average\_gate\_fidelityは量子チャネルとオペレータ、あるいは2つのオペレータ間の忠実度を計算するものだが、(両者のprocess\_fidelity × 次元数 + 1) / (次元数 + 1) に等しい。

## ポイント

- state\_fidelity、process\_fidelityは0以上1以下の値をとる。0の場合は比較している両者は直交しており、1の場合は両者は同一である。

## 解 説

- State\_fidelity, Process\_fidelity の計算例を示す:

```
In [19]: import numpy as np
from qiskit.quantum_info.operators import Operator
from qiskit.quantum_info import process_fidelity, state_fidelity
from qiskit.extensions import XGate
```

```
In [20]: psi_1 = np.array([ 0, 0, 0, 1 ])
psi_2 = np.array([ 0.5, 0.5, 0.5, 0.5 ])
print(state_fidelity(psi_1, psi_2))

0.25
```

2つの状態は直交していない  
(が平行でもない)ため、0より  
大きく1未満の値となる。

```
In [21]: psi_1 = np.array([ 0, 0, 0, 1 ])
psi_2 = np.array([ 1, 0, 0, 0 ])
print(state_fidelity(psi_1, psi_2))

0.0
```

2つの状態は直交しているた  
め、0となる

```
In [24]: Op_A = Operator(XGate())
Op_B = np.exp(1j*np.pi) * Operator(XGate())
F = process_fidelity(Op_A, Op_B)
print('Process fidelity =', F)

Process fidelity = 1.0
```

2つの状態は同一の  
ため、1となる。

## ポイント

- state\_fidelity、average\_gate\_fidelity、process\_fidelityの計算ではグローバル位相の差は無視される。

## 出題範囲

- Section 6: Return the Experiment Results
- a.Return and understand the histogram data of an experiment

## 解 説

- Qasm\_simulatorの結果を確認

```
qc = QuantumCircuit(3)
qc.h([0, 1, 2])
qc.measure_all()
qasm_sim = Aer.get_backend('qasm_simulator')
job = execute(qc,backend=qasm_sim, shots=1000)
```

測定が必要

1000回シミュレーション  
(指定しないと、デフォルト1024回)

```
result = job.result()
counts = result.get_counts(qc)
print(counts)
```

集計結果を表示

出力結果がdict型

```
{'111': 117, '001': 127, '101': 119, '110': 129, '100': 128, '010': 115, '000': 123, '011': 142}
```

## ポイント

- qasm\_simulatorの実行方法を理解
- ヒストグラムで確認する方法を理解。

- [参考]Qiskitドキュメント

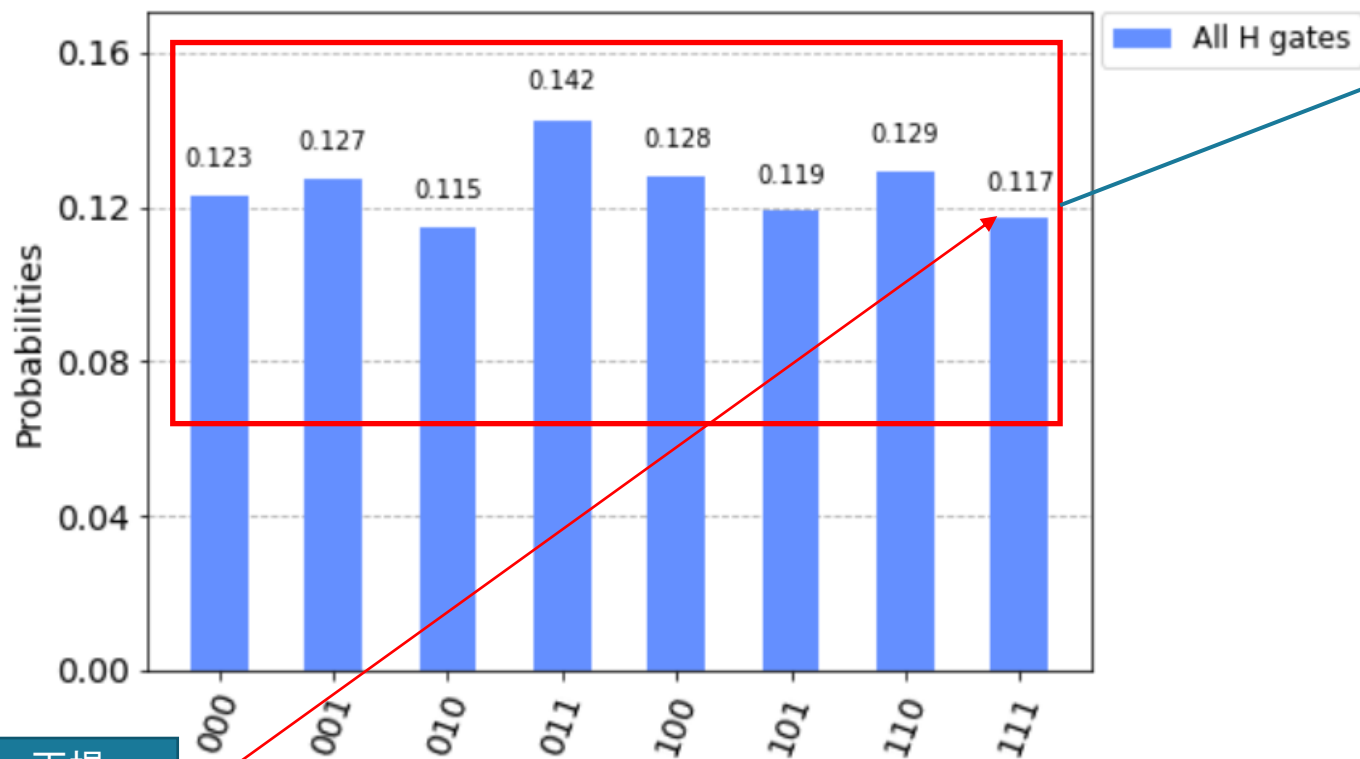
[Executing Experiments \(qiskit.execute\\_function\) — Qiskit 0.37.0 documentation](#)

[qiskit.visualization.plot\\_histogram — Qiskit 0.37.0 documentation](#)

## 解説

- ヒストグラムで実行結果の確認

```
plot_histogram(counts, legend=['All H gates'])
```



再掲

```
{'111': 117, '001': 127, '101': 119, '110': 129, '100': 128, '010': 115, '000': 123, '011': 142}
```

## ポイント

- ヒストグラムでは確率として表示される
- バーの色や、legend(凡例)の変更の仕方を理解しておく

## 出題範囲

- Section 6: Return the Experiment Results
- b.Return and understand the statevector of an experiment

## 解 説

- Statevectorの実行結果

```
qc = QuantumCircuit(1)
qc.x(0)
qc.h(0)

backend = Aer.get_backend('statevector_simulator')
final_state = execute(qc,backend).result().get_statevector()

from qiskit_textbook.tools import array_to_latex
array_to_latex(final_state, pretext="¥¥text{Statevector = }")
```

Statevectorシミュレーションを実行

Latex形式で結果を確認

$$\text{Statevector} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}$$

```
Statevector([ 0.70710678+0.00000000e+00j, -0.70710678-8.65956056e-17j],
            dims=(2,))
```

qiskitバージョン 0.34.2

## ポイント

- statevector\_simulatorの実行方法
- 実行結果の形式を理解する。
- [参考]Qiskitドキュメント  
[Qiskit Aer API Reference — Qiskit 0.37.0 documentation](https://qiskit.org/documentation/qiskit-aer/0.37.0/qiskit-aer.html)

(参考)単純に出力  
(print(final\_state))した結果

## 出題範囲

- Section 6: Return the Experiment Results
- c.Return and understand the unitary of an experiment

## 解 説

- unitaryの実行結果

```
qc = QuantumCircuit(1)
qc.x(0)
qc.h(0)
```

```
simulator = Aer.get_backend('unitary_simulator')
result = execute(qc,backend=simulator).result()
unitary = result.get_unitary(qc)
array_to_latex(unitary, pretext="¥¥text{Circuit = }¥n")
```

Unitaryシミュレーションを実行

Latex形式で結果を確認

(参考)単純に出力  
(print( unitary))した結果

$$\text{Circuit} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

```
Operator([[ 0.70710678+0.00000000e+00j,  0.70710678+8.65956056e-17j],
          [-0.70710678-8.65956056e-17j,  0.70710678+1.73191211e-16j]],
         input_dims=(2,), output_dims=(2,))
```

## ポイント

- unitary\_simulatorの実行方法
- 実行結果の形式を理解する。

- [参考]Qiskitドキュメント

[Qiskit Aer API Reference — Qiskit 0.37.0 documentation](#)

## 出題範囲

- Section 7: Use Qiskit Tools
- a. Monitor the status of a job instance

## 解 説

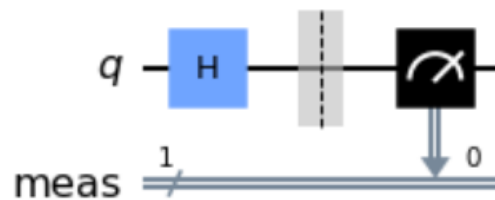
- IBMQに投入したジョブのステータスの確認の仕方について、例示する。
- まず、必要なライブラリ等をインポートし、簡単な回路を構成する

```
In [4]: ▶ from qiskit import QuantumCircuit, IBMQ, execute
        from qiskit.visualization import plot_histogram
        provider = IBMQ.load_account()
        machine = provider.get_backend('ibmq_armonk')
```

```
In [6]: ▶ qc = QuantumCircuit(1)
        qc.h(0)
        qc.measure_all()
        qc.draw('mpl')
```

## ポイント

Out[6]:





## 解 説

- ジョブを実行するときに、以下のようにジョブの実行状況を確認することができる。

```
In [7]: ▶ job = execute(qc,machine)
        job.status()
```

```
Out[7]: <JobStatus.QUEUED: 'job is queued'>
```

```
In [8]: ▶ job.status()
```

```
Out[8]: <JobStatus.RUNNING: 'job is actively running'>
```

```
In [11]: ▶ job.status()
```

```
Out[11]: <JobStatus.DONE: 'job has successfully run'>
```

```
In [12]: ▶ job.status()
```

```
Out[12]: <JobStatus.DONE: 'job has successfully run'>
```

## ポイント

- Statusコマンドを繰り返し実行
- ジョブが待ち行列に入っている
- ジョブは実行中
- ジョブは正常に実行された

## 解 説

- statusコマンドの場合、ステータスの変化を知るには、繰り返してコマンドを実行しなければならない。
- job\_monitorコマンドであれば、statusコマンドを繰り返し実行しているのと同様に、ステータスの変化を表示させることができる。(以下はジョブ実行完了のもの)

```
In [13]: ▶ from qiskit.tools import job_monitor  
         job = execute(qc,machine)  
         job_monitor(job)
```

Job Status: job has successfully run

## ポイント

- job\_monitorコマンドは1回実行すればよい。ステータスが変わるたびに、Job Statusの表示も変化する。

## 出題範囲

- Section 8: Display and Use System Information
- a.Perform operations around the Qiskit version

## 解説

- Qiskitのバージョンの確認方法は2種類。

```
import qiskit
qiskit.__qiskit_version__
```

方法①

```
{'qiskit-terra': '0.19.2', 'qiskit-aer': '0.10.3', 'qiskit-ignis': '0.7.0', 'qiskit-sket': '0.34.2', 'qiskit-nature': None, 'qiskit-finance': None, 'qiskit-op
```

```
import qiskit.tools.jupyter
%qiskit_version_table
```

方法②

## Version Information

Qiskit Software	Version
qiskit-terra	0.19.2
qiskit-aer	0.10.3
qiskit-ignis	0.7.0

qiskitバージョン 0.34.2

## ポイント

- バージョンの確認方法は2つ。
- 表示の違い。

- [参考]Qiskitテキストブック  
[Qiskitテキストブックで作業するための環境設定ガイド](#)

## 出題範囲

- Section 8: Display and Use System Information
- b.Use information gained from %qiskit\_backend\_overview

## 解 説

- backend\_overviewで表示される情報は以下のとおり

```
from qiskit.tools.monitor import backend_overview
backend_overview()
```

ibm\_nairobi

-----

Num. Qubits: 7	← 量子ビット数
Pending Jobs: 89	← 実行待ちのジョブ数
Least busy: False	
Operational: True	
Avg. T1: 124.9	← $ 1\rangle$ の励起状態を保持できる平均コヒーレンス時間
Avg. T2: 79.7	← $ 0\rangle+ 1\rangle$ の重ね合わせを保持できる平均コヒーレンス時間

## ポイント

- 自分の計算したいジョブを早く実行するためには、一覧を見て、実行待ちのジョブ数の少ないマシンを選択する。
- T1,T2の意味は左記のとおり。単位は $\mu s$ 。一般的にコヒーレンス時間が長いマシンのほうが、ゲート操作などでの誤りが少なくなる可能性が高い。

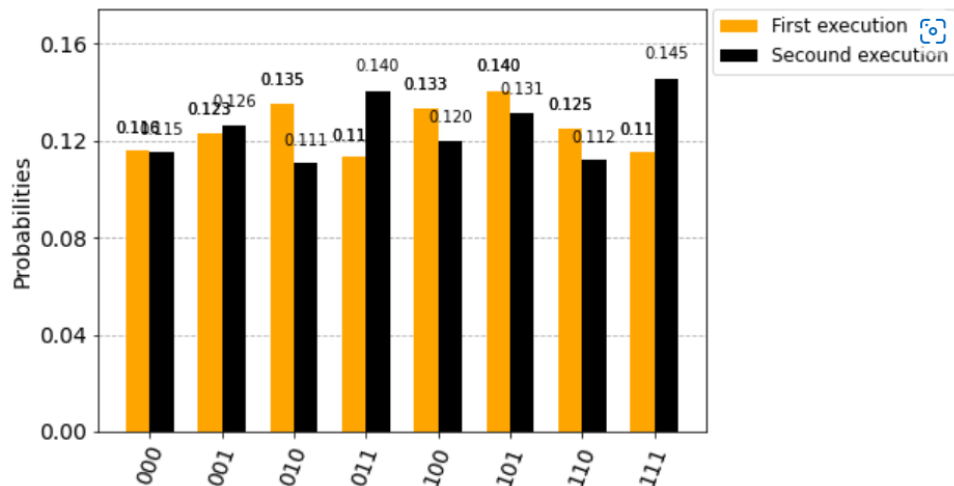
## 出題範囲

- Section 9: Construct Visualizations
- b.Plot a histogram of data

## 解説

- plot\_histogramで確認する方法

```
legend = ['First execution','Secound execution']  
plot_histogram([counts1,counts2],legend=legend,color=['orange','black'],bar_labels=True)
```



qiskitバージョン 0.34.2

## ポイント

- Countsをヒストグラムで表示する方法とオプションを知っている。
- 凡例、色の指定やバーのラベルやの付け方など
- [参考]Qiskitドキュメント  
[qiskit.visualization.plot\\_histogram — Qiskit 0.37.0 documentation](https://qiskit.org/documentation/visualization/plot_histogram.html)

## 出題範囲

- Section 9: Construct Visualizations
- c.Plot a Bloch multivector

## 解 説

- plot\_bloch\_multivector( ) の表示

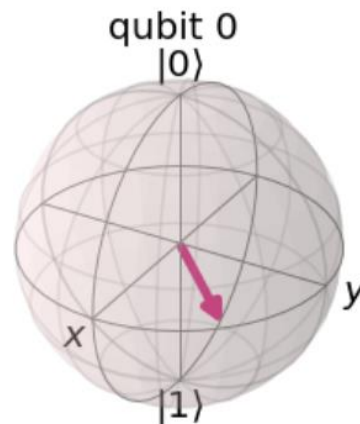
```
qc = QuantumCircuit(1)
qc.h(0)
qc.t(0)
```

Statevector\_simulator  
を使用

```
simulator = Aer.get_backend('statevector_simulator')
job = execute(qc,simulator)
result = job.result()
statevector = result.get_statevector(qc)
plot_bloch_multivector(statevector)
```

plot\_bloch\_multivector( )  
を使って出力

## 出力結果



qiskitバージョン 0.34.2

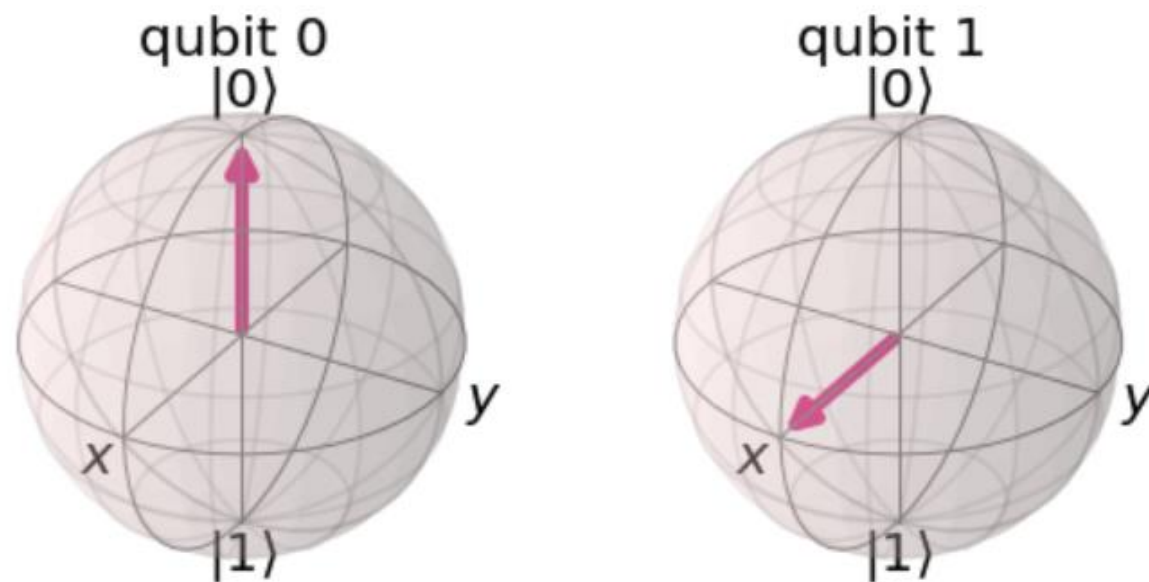
## ポイント

- plot\_bloch\_multivector( ) のコマンド自体を覚えておくこと
- ゲート操作の結果とmultivectorの状態を理解し、一致させることができること

## 解 説

- 数値で座標の位置を指定する方法

```
import numpy as np
vector = [1/np.sqrt(2), 0, 1/np.sqrt(2), 0]
plot_bloch_multivector(vector)
```



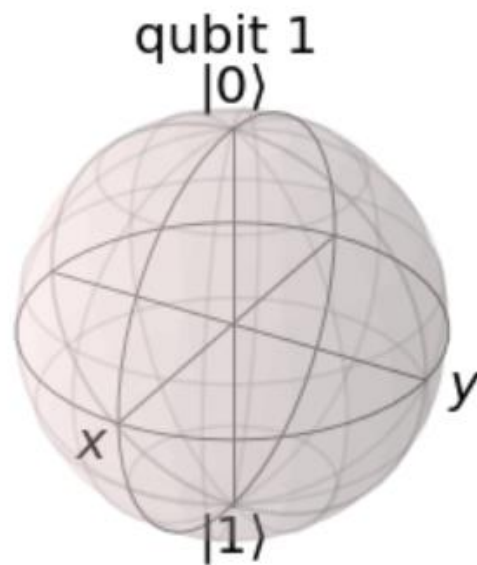
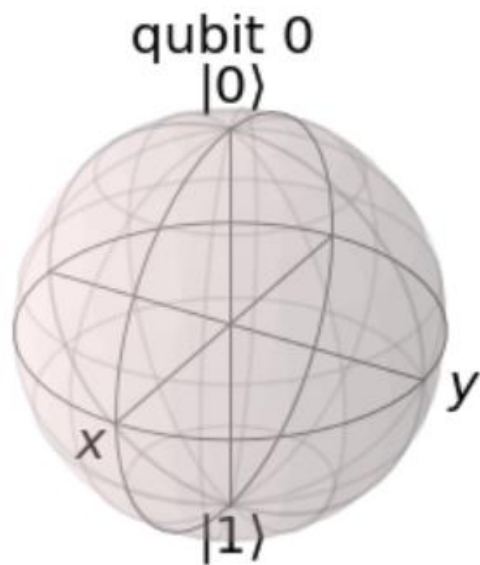
## ポイント

- 左記の形式で、問題が出されても、出力結果がわかるようにしておくこと

## 解 説

- Bell状態

```
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
plot_bloch_multivector(qc)
```



## ポイント

- エンタングル状態になり、矢印が表示されない



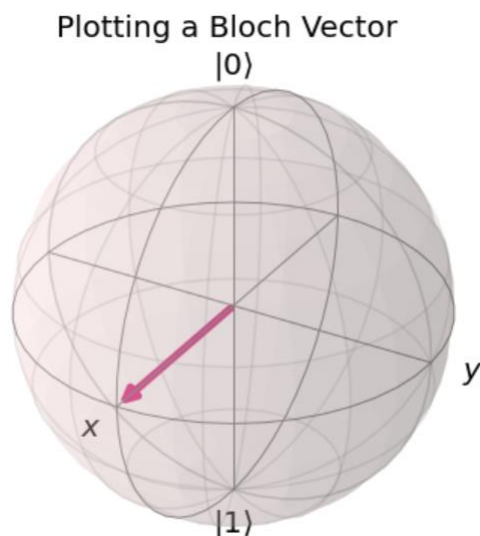
## 出題範囲

- Section 9: Construct Visualizations
- d.Plot a Bloch vector

## 解 説

- ブロッホ球のプロット

```
plot_bloch_vector([1, 0, 0],title='Plotting a Bloch Vector')
```



## ポイント

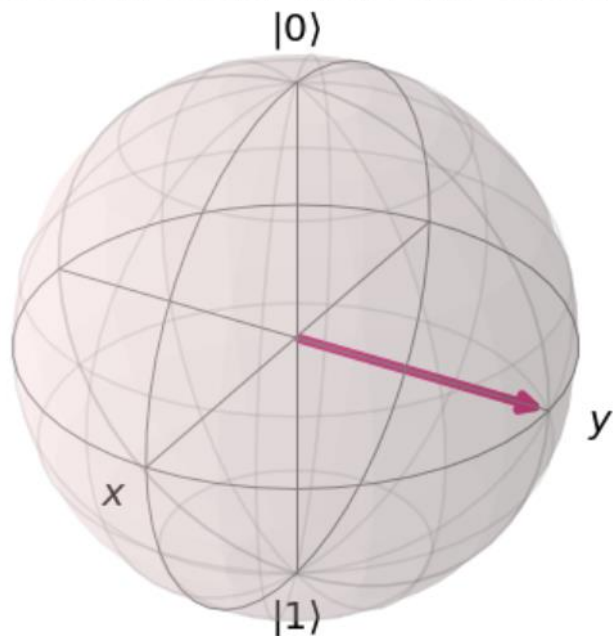
- 次の順序で関数にリストとして渡す  
`Plot_bloch_vector([x , y, z])`
  - x- x 座標
  - y- y 座標
  - z- z 座標
- [参考] Qiskit: Bloch Vector  
[Qiskit: Bloch Vector Tutorial – Deep Learning University](#)

## 解説

- 球座標によるブロッホ球のプロット

```
plot_bloch_vector([1, np.pi/2, np.pi/2],  
coord_type='spherical',  
title='Plotting a Bloch Vector with Polar Coordinates')
```

Plotting a Bloch Vector with Polar Coordinates



## ポイント

- 次の順序で関数にリストとして渡す。  
`plot_bloch_vector(r,theta,phi)`  
r- ブロッホ球の中心からの距離。  
theta- 正の Z 軸を持つブロッホベクトルによって作られた角度 (ラジアン単位)。  
phi- 正の X 軸 (ラジアン) を持つブロッホベクトルによって作られました。
- 2種類の方法があるので、違いを理解しておく。

## 出題範囲

- Section 9: Construct Visualizations
- e.Plot a QSphere

## 解 説

- plot\_state\_qsphere()の表示

```
qc = QuantumCircuit(3)
qc.x([1, 2])
```

```
from qiskit.visualization import plot_state_qsphere
simulator = Aer.get_backend('statevector_simulator')
job = execute(qc, simulator)
result = job.result()
statevector = result.get_statevector(qc)
plot_state_qsphere(statevector)
```

Statevector\_simulator  
を使用

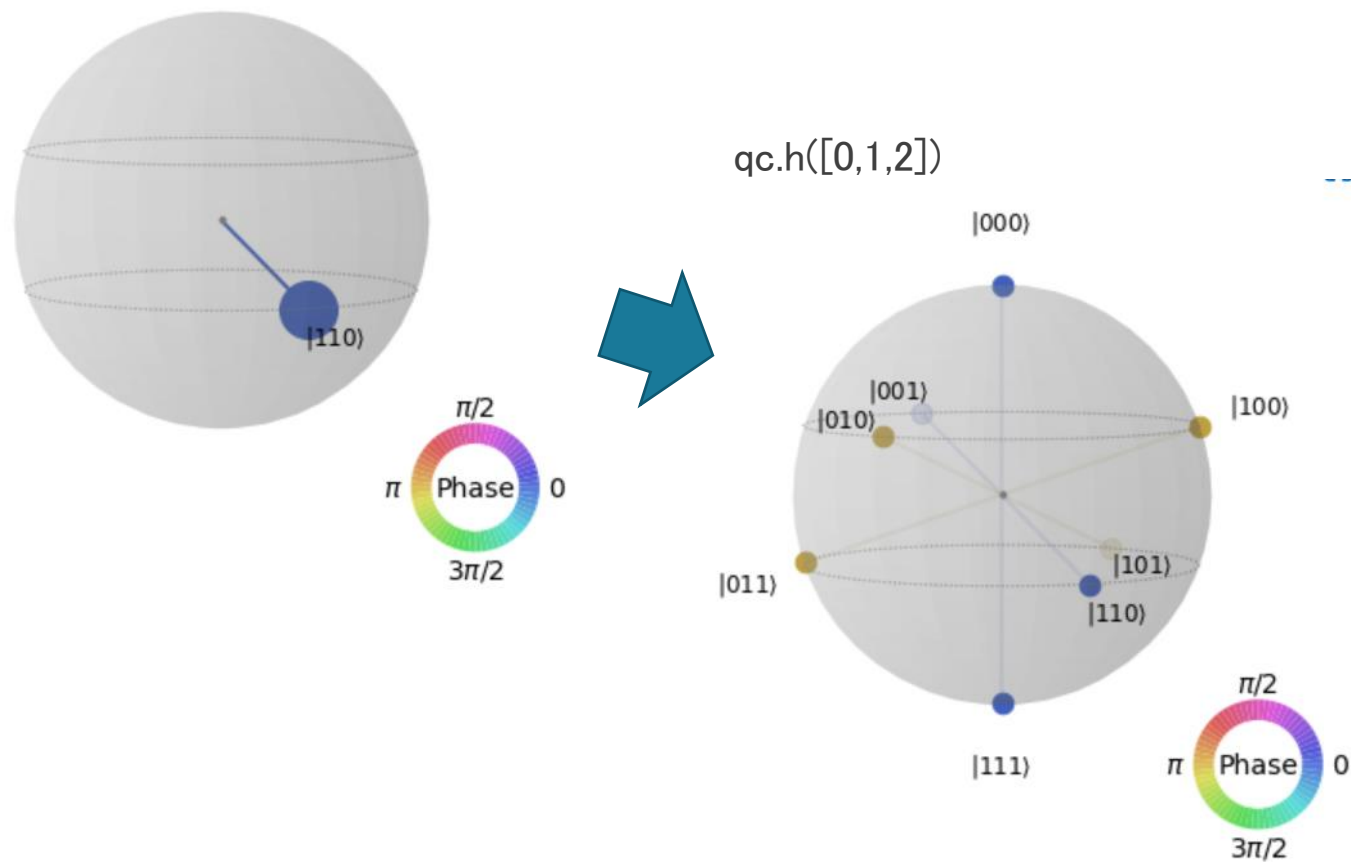
plot\_state\_qsphere( )を  
使って出力

## ポイント

- plot\_state\_qsphere( )のコマンド自体を覚えておくこと

## 解説

- 出力結果



qiskitバージョン 0.34.2

## ポイント

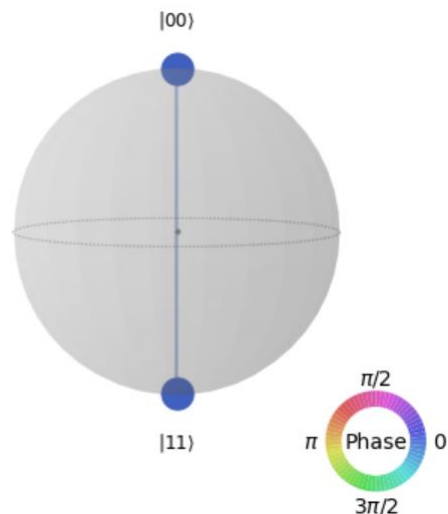
- ビット数が異なる場合、どのように表示されるかを理解
- ゲート操作の結果とQsphereの状態を理解し、一致させることができること

## 解 説

## • Bell状態

```
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0,1)

from qiskit.visualization import plot_state_qsphere
simulator = Aer.get_backend('statevector_simulator')
job = execute(qc,simulator)
result = job.result()
statevector = result.get_statevector(qc)
plot_state_qsphere(statevector)
```



## ポイント

- $|00\rangle$ と $|11\rangle$ の重ね合わせ状態となっている

## 出題範囲

- Section 9: Construct Visualizations
- g.Plot a gate map with error rates

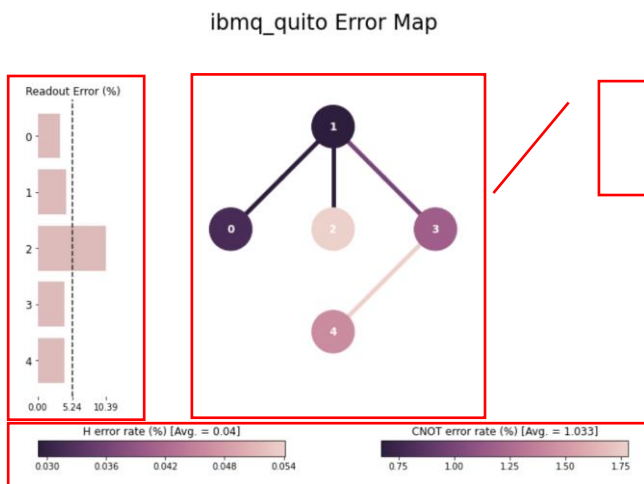
## 解 説

- plot\_error\_map( ) で表示される情報は以下のとおり

```
backend = provider.get_backend('ibmq_quito')  
job = execute(qc, backend)  
plot_error_map(backend)
```

読み取り時の  
エラー率

論理ゲート  
のエラー率



物理的な  
ビットの結びつき

qiskitバージョン 0.34.2

## ポイント

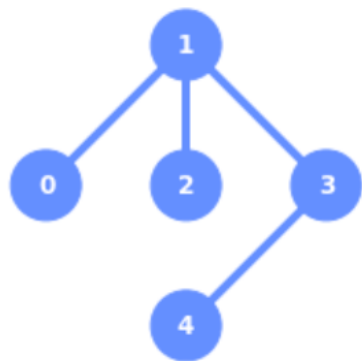
- plot\_error\_map( ) で表示できる内容を理解する。
- [参考]Qiskitドキュメント  
[qiskit.visualization.plot\\_error\\_map — Qiskit 0.37.0 documentation](https://qiskit.org/documentation/visualization/plot_error_map.html)

## 解説

- 結合マップ(coupling\_map)

```
from qiskit.visualization import plot_coupling_map
%matplotlib inline

num_qubits = 5
coupling_map = [[0, 1],[1, 2],[1, 3],[3, 4]]
qubit_coordinates = [[1, 0], [0, 1], [1, 1],[1, 2],[2, 1]]
plot_coupling_map(num_qubits, qubit_coordinates, coupling_map)
```



この場合の量子ビットの接続は、  
[0,1],[1,2],[1,3],[3,4]と表示される。  
[0,1] は、0と1の接続していることを  
表している

## ポイント

- 結合マップを見て、量子ビットごとの接続関係を理解する

- [参考]Qiskitドキュメント、関連サイト  
[qiskit.visualization.plot\\_gate\\_map — Qiskit 0.37.1 docu](https://qiskit.org/documentation/visualization/plot_gate_map.html)

[qiskit.visualization.plot\\_coupling\\_map — Qiskit 0.37.1 documentation](https://qiskit.org/documentation/visualization/plot_coupling_map.html)

## 出題範囲

- Section 10: Access Aer Provider
- a.Access a statevector\_simulator backend

## 解 説

- Statevector\_Simulatorの実行結果は以下のとおり

```
from qiskit import Aer
qc = QuantumCircuit(1)
qc.x(0)
qc.h(0)
```

Aer.get\_backend  
で呼び出し

```
backend = Aer.get_backend('statevector_simulator')
final_state = execute(qc,backend).result().get_statevector()
print(final_state)
```

状態ベクトルが  
表示される

```
Statevector([ 0.70710678+0.00000000e+00j, -0.70710678-8.65956056e-17j ],
            dims=(2,))
```

qiskitバージョン 0.34.2

## ポイント

- Aer.get\_backendの文法、使えるシミュレーターを理解しておく
- 出力結果が何を意味するか理解する
- [参考]Qiskitドキュメント  
[Qiskit Aer API Reference — Qiskit 0.37.0 documentation](https://docs.quantum.ibm.com/api/qiskit-aer/)



## 出題範囲

- Section 10: Access Aer Provider
- b.Access a qasm\_simulator backend

## 解説

- qasm\_simulatorの実行結果は以下のとおり

```
qc = QuantumCircuit(2)
qc.h([0, 1])
qc.measure_all()
qasm_sim = Aer.get_backend('qasm_simulator')
job = execute(qc, backend=qasm_sim, shots=1024)
```

Aer.get\_backend  
で呼び出し

1024回測定  
を実行

```
result = job.result()
counts = result.get_counts(qc)
```

```
print(counts)
```

Hゲートをかけた結果  
であり、等しく観測さ  
れたことがわかる

```
{'00': 271, '01': 261, '11': 232, '10': 260}
```

qiskitバージョン 0.34.2

## ポイント

- Aer.get\_backendの文法、使えるシミュレーターを理解しておく
- 出力結果が何を意味するか理解する
- [参考]Qiskitドキュメント  
[Qiskit Aer API Reference — Qiskit 0.37.0 documentation](https://docs.quantum.ibm.com/api/qiskit-aer/)
- execute機能のオプションにどのようなものがあるか知っておく(shots等)
- [参考]Qiskitドキュメント  
[Executing Experiments \(qiskit.execute function\) — Qiskit 0.37.1 documentation](https://docs.quantum.ibm.com/tutorials/running-experiments/)

## 出題範囲

- Section 10: Access Aer Provider
- c.Access a unitary\_simulator backend

## 解 説

- unitary\_simulatorの実行結果は以下のとおり

```
qc = QuantumCircuit(1)
qc.x(0)
qc.h(0)

simulator = Aer.get_backend('unitary_simulator')
result = execute(qc,backend=simulator).result().get_unitary()
print(unitary)
```

Aer.get\_backend  
で呼び出し

Operatorの形式  
で出力される

```
Operator([[ 0.70710678+0.00000000e+00j,  0.70710678+8.65956056e-17j],
          [-0.70710678-8.65956056e-17j,  0.70710678+1.73191211e-16j]],
         input_dims=(2,), output_dims=(2,))
```

qiskitバージョン 0.34.2

## ポイント

- Aer.get\_backendの文法、使えるシミュレーターを理解しておく
- 出力結果が何を意味するか理解する
- [参考]Qiskitドキュメント  
[Qiskit Aer API Reference — Qiskit 0.37.0 documentation](https://docs.quantum.ibm.com/api/qiskit-aer/)

本資料の著作権は、日本アイ・ビー・エム株式会社（IBM Corporationを含み、以下、IBMといいます。）に帰属します。

ワークショップ、セッション、および資料は、IBMまたはセッション発表者によって準備され、それぞれ独自の見解を反映したものです。それらは情報提供の目的のみで提供されており、いかなる参加者に対しても法律的またはその他の指導や助言を意図したものではなく、またそのような結果を生むものでもありません。本資料に含まれている情報については、完全性と正確性を期するよう努力しましたが、「現状のまま」提供され、明示または暗示にかかわらずいかなる保証も伴わないものとします。本資料またはその他の資料の使用によって、あるいはその他の関連によって、いかなる損害が生じた場合も、IBMまたはセッション発表者は責任を負わないものとします。本資料に含まれている内容は、IBMまたはそのサプライヤーやライセンス交付者からいかなる保証または表明を引きだすことを意図したものでも、IBMソフトウェアの使用を規定する適用ライセンス契約の条項を変更することを意図したものでもなく、またそのような結果を生むものでもありません。

本資料でIBM製品、プログラム、またはサービスに言及していても、IBMが営業活動を行っているすべての国でそれらが使用可能であることを暗示するものではありません。本資料で言及している製品リリース日付や製品機能は、市場機会またはその他の要因に基づいてIBM独自の決定権をもっていつでも変更できるものとし、いかなる方法においても将来の製品または機能が使用可能になると確約することを意図したものではありません。本資料に含まれている内容は、参加者が開始する活動によって特定の販売、売上高の向上、またはその他の結果が生じると述べる、または暗示することを意図したものでも、またそのような結果を生むものでもありません。パフォーマンスは、管理された環境において標準的なIBMベンチマークを使用した測定と予測に基づいています。ユーザーが経験する実際のスループットやパフォーマンスは、ユーザーのジョブ・ストリームにおけるマルチプログラミングの量、入出力構成、ストレージ構成、および処理されるワークロードなどの考慮事項を含む、数多くの要因に応じて変化します。したがって、個々のユーザーがここで述べられているものと同様の結果を得られると確約するものではありません。

記述されているすべてのお客様事例は、それらのお客様がどのようにIBM製品を使用したか、またそれらのお客様が達成した結果の実例として示されたものです。実際の環境コストおよびパフォーマンス特性は、お客様ごとに異なる場合があります。

IBM、IBM ロゴは、米国やその他の国におけるInternational Business Machines Corporationの商標または登録商標です。他の製品名およびサービス名等は、それぞれIBMまたは各社の商標である場合があります。現時点でのIBMの商標リストについては、[ibm.com/trademark](http://ibm.com/trademark)をご覧ください。