2025 / Spring Semester

Topics on Quantum Computing

Lecture Note – Gate Operations & PENNYLANE SDK Basics

Junghee Ryu

KISTI

PENNYLANE

A quantum Software Development Kit (SDK)

- A python-based SDK with which quantum circuits can be programmed.
  - Released by XADADU Quantum Technology Inc. (https://pennylane.ai/)
  - A strong counterpart of the IBM qiskit (https://www.ibm.com/quantum/qiskit)
  - Open-source software; can be integrated with cuQuantum (NVIDIA)
- How to install (https://pennylane.ai/install)
  - Installation is super simple.
  - Python 3.9.6 or higher version is recommended.
  - Jupyter notebook is not mandatory, but would be fine if you have (encouraged)
  - Let’s install in your labtop and test the simple code in the next slide.

PENNYLANE

A hello-world code

```
import pennylane as qml
import numpy as np
import matplotlib.pyplot as plt

mydevice = qml.device("default.qubit", wires=1)          # device setup (plug-in)

@qml.qnode(mydevice)      # a quantum circuit that will be executed in the device
def quantum_circuit():
    qml.Hadamard(wires=0)
    return qml.state()

result = quantum_circuit() # run the circuit in the device
print(result)              # run classical command
print(result[0])
print(result[1])
```

PENNYLANE

Basic structure: Programming

```
import pennylane as qml
import numpy as np
import matplotlib.pyplot as plt
```

Designation of QPU environments

```
mydevice = qml.device("default.qubit", wires=1)          # device setup (plug-in)
```

```
@qml.qnode(mydevice)       # a quantum circuit that will be executed in the device
    def quantum_circuit():
    qml.Hadamard(wires=0)
    return qml.state()
```

QPU (or Emulator mimicking the QPU)

```
result = quantum_circuit() # run the circuit in the device
print(result)              # run classical command
print(result[0])
print(result[1])
```

My Computer

Output

```
[0.70710678+0.j 0.70710678+0.j]
(0.7071067811865475+0j)
(0.7071067811865475+0j)
```

What we do here …

With PENNYLANE SDK

- **Basic universal gate operations**
  - Single-qubit logics
  - Two-qubit logics
  - Multi-controlled sequence
  - `qml.ctrl` routine
- **Checking results**
  - `qml.state`
  - `qml.probs`
  - `qml.counts`
  - `qml.expval`
- **Some miscellaneous routines**
  - Circuit checking: `qml.draw_mpl`
  - Intermediate checking: `qml.Snapshot`
  - Multi-controlled sequence
  - `qml.ctrl` routine
- **Practice (can be your missions)**
  - 2-qubit Bell-state
  - 5-qubit Greeberg-Greenberger–Horne–Zeilinger (GHZ) state
  - Parameterized Quantum Circuit

Single-qubit gating

The most elementary level of programming

- ❑ RX, RY, RZ

$$R_x(\theta) = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix} \quad R_y(\theta) = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix} \quad R_z(\theta) = \begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$$

- ❑ Pauli-X (= RX(π)), Pauli-Y (= RY(π)), Pauli-Z (= RZ(π))

$$X = \sigma_x = \sigma_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad Y = \sigma_y = \sigma_2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad Z = \sigma_z = \sigma_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

- ❑ Hadamard $H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$

- ❑ PhaseShift $F_\theta = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}$

Single-qubit gating

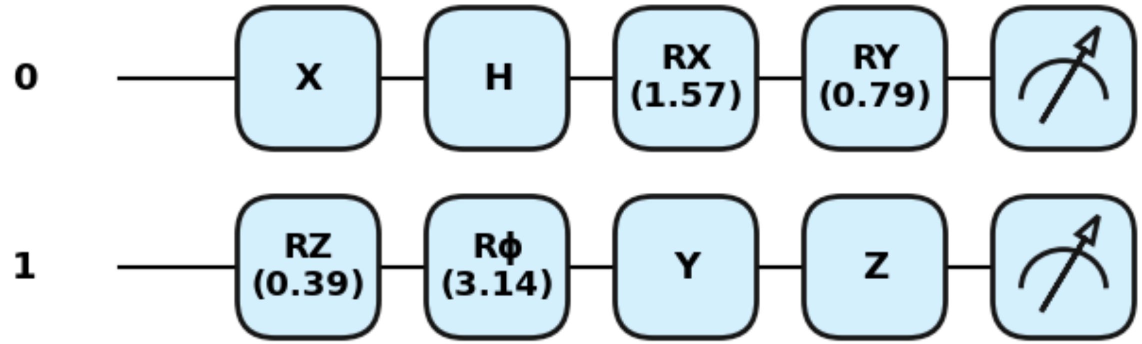
The most elementary level of programming

- RX, RY, RZ, Phase Shift gate
  - `qml.RX/RY/RZ(theta, wires)`, `qml.PhaseShift(theta, wires)`
  - `theta`: phase (in radian), `wires`: qubit index where the gate is applied
- Pauli-X (= RX(π)), Pauli-Y (= RY(π)), Pauli-Z (= RZ(π)), Hadamard gate
  - `qml.PauliX/PauliY/PauliZ/Hadamard(wires)`
  - `wires`: qubit index where the gate is applied
- There is no explicit variable of a state vector
  - The state vector can be accessed only with a function call
  - Initial state: $|0\rangle^{\otimes n} = |00\ldots0\rangle$
  - `qml.state()`: Will be discussed later in this lecture note

Single-qubit gating

The most elementary level of programming

```
dev = qml.device("default.qubit", wires=2)

@qml.qnode(dev)
def circuit1():
    qml.PauliX(wires=0)
    qml.Hadamard(wires=0)
    qml.RX(np.pi/2, wires=0)
    qml.RY(np.pi/4, wires=0)
    qml.RZ(np.pi/8, wires=1)
    qml.PhaseShift(np.pi, wires=1)
    qml.PauliY(wires=1)
    qml.PauliZ(wires=1)
    return qml.state()
```



OUTPUT

```
circuit1()
```

```
array([0. +0.j, 0.51327997-0.76817776j, 0. +0.j, -0.21260752+0.31818965j])
```

Two-qubit gating

The most elementary level of programming

- CNOT (Controlled-X), CZ (Controlled-Z), Controlled Phase Shift, SWAP gate
  - **CNOT:** Identity ($I_{2\text{-}by\text{-}2}$) if the control qubit = 0, X if the control qubit = 1
  - **CZ:** $I_{2\text{-}by\text{-}2}$ if the control qubit = 0, Z if the control qubit = 1
  - **Controlled Phase Shift:** $I_{2\text{-}by\text{-}2}$ if the control qubit = 0, F_θ if the control qubit = 1
  - **SWAP:** |ij⟩ = |i⟩⊗|j⟩ → |ji⟩ = |j⟩⊗|i⟩ (not an entangling operation)
  - `qml.CNOT/CZ/ControlledPhaseShift/SWAP(wires=[C,T])`

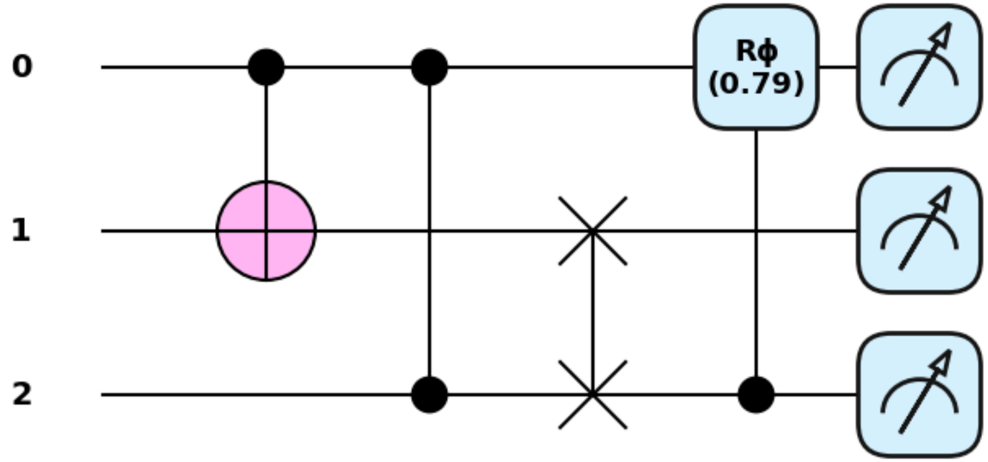
```
# C/T = control/target qubit, (Cth/Tth qubit for SWAP)
```

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \quad CR = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{\theta i} \end{pmatrix} \quad SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Two-qubit gating

The most elementary level of programming

```
dev = qml.device("default.qubit", wires=3)

@qml.qnode(dev)
def circuit1():
    qml.CNOT(wires=[0,1])
    qml.CZ(wires=[0,2])
    qml.SWAP(wires=[1,2])
    qml.ControlledPhaseShift(np.pi/4,wires=[2,0])
    return qml.state()
```



Checking the output of your circuits

The most elementary level of programming

- ❑ Check elements of the state vector directly: `qml.state`
  - ▪ Only supported in emulator device
- ❑ Check the probability of each computational basis state: `qml.probs`
- ❑ Check the sample counts of each computational basis state: `qml.counts`
  - ▪ The device must be configured with a shot number
- ❑ Get the expectation value: `qml.expval`
- ❑ Get the sampling results: `qml.sample`
  - ▪ The device must be configured with a shot number
- ❑ Perform a mid-circuit measurement on the supplied qubit: `qml.measure`
  - ▪ Final results can be secured with `qml.probs`, `qml.counts`, `qml.expval`, etc.

Checking the output of your circuits

The most elementary level of programming

- Check elements of the state vector directly: `qml.state`
  - We already checked this utility in the hello-world example (slide #3)

```
...
mydevice = qml.device("default.qubit", wires=1) # device setup (plug-in)
# a quantum circuit that will be executed in the device
@qml.qnode(mydevice)
def quantum_circuit():
    qml.Hadamard(wires=0)
    return qml.state()

result = quantum_circuit() # run the circuit in the device
print(result)              # run classical command
print(result[0])
print(result[1])
```

Output

```
[0.70710678+0.j 0.70710678+0.j]
(0.7071067811865475+0j)
(0.7071067811865475+0j)
```

Checking the output of your circuits

The most elementary level of programming

❑ Check the probability of each computational basis state: `qml.prob`

```
...
mydevice2 = qml.device("default.qubit", wires=1) # device setup (plug-in)
# a quantum circuit that will be executed in the device
@qml.qnode(mydevice2)
def quantum_circuit2():
    qml.Hadamard(wires=0)
    return qml.probs()

result = quantum_circuit2() # run the circuit in the device
print(result)               # run classical command
print(result[0])
print(result[1])
```

Output

```
[0.5 0.5]
0.4999999999999999
0.4999999999999999
```

What happens with `qml.probs(op=qml.X(0))`?

Checking the output of your circuits

The most elementary level of programming

- ❑ Check the sample counts of each computational basis state: `qml.counts`
  - Device should be configured with a shot number

```
...
mydevice3 = qml.device("default.qubit", wires=1, shots=10) # device setup (plug-in)
# a quantum circuit that will be executed in the device
@qml.qnode(mydevice3)
def quantum_circuit3():
    qml.Hadamard(wires=0)
    return qml.counts()

result = quantum_circuit3() # run the circuit in the device
print(result)               # run classical command
```

Output

```
{'0': array(7), '1': array(3)}
```

Run multiple times. What do you observe?

Checking the output of your circuits

The most elementary level of programming

- ❑ Get the sampling results: `qml.sample`
  - ▪ Device should be configured with a shot number

```
...
mydevice3 = qml.device("default.qubit", wires=1, shots=10) # device setup (plug-in)
# a quantum circuit that will be executed in the device
@qml.qnode(mydevice3)
def quantum_circuit3():
    qml.Hadamard(wires=0)
    return qml.samples()

result = quantum_circuit3() # run the circuit in the device
print(result)               # run classical command
```

Output

```
[1 1 0 1 0 1 0 0 1 0]
```

Run multiple times. What do you observe?

Checking the output of your circuits

The most elementary level of programming

- Get the expectation value: `qml.expval`

```
...
mydevice3 = qml.device("default.qubit", wires=1, shots=10) # device setup (plug-in)
# a quantum circuit that will be executed in the device
@qml.qnode(mydevice3)
def quantum_circuit3():
    qml.Hadamard(wires=0)
    return qml.expval()

result = quantum_circuit3() # run the circuit in the device
print(result)               # run classical command
```

How is the result (1) when you specified a shot number and (2) you didn't?
Why do the results become different in these two cases?

Checking the output of your circuits

The most elementary level of programming

- ❑ Perform a mid-circuit measurement on the supplied qubit : `qml.measure`
  - Sometimes results of mid-circuit measurements are used for determination of next logic operations

```
...
mydevice3 = qml.device("default.qubit", wires=3)
@qml.qnode(mydevice3)
def circuit3(x, y):
    qml.RX(x, wires=0)
    qml.RY(y, wires=1)
    m0 = qml.measure(0) # measure qubit 0 a with computational basis
    m1 = qml.measure(1) # measure qubit 1 a with computational basis
    qml.cond(~m0 & m1 == 0, qml.X)(wires=2) # conduct Pauli-X to qubit 2 conditionally
    return qml.expval(qml.Z(2)) # expectation value of qubit 2
                                # (measured with a computational basis)
```

```
result = circuit3(np.pi/2, np.pi/2)
print(result)
```

Output `-0.5`

General N-qubit gating

Some miscellaneous routines

function ctrl(op, control, control_values=None, work_wires=None)

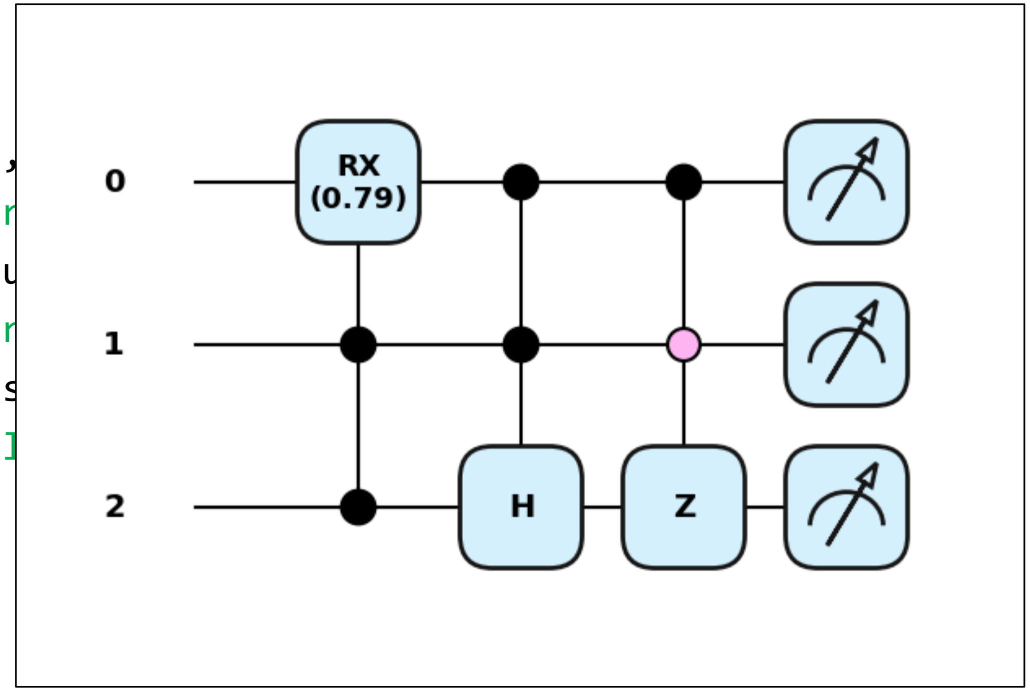
```
dev = qml.device("default.qubit", wires=3)

@qml.qnode(dev)
def circuit2():
    qml.ctrl(qml.RX, (1,2), control_values=(1,1))(np.pi/4, wires=0)
    # RX(pi/4) operation to qubit 0 when control qubits are (1,1)
    qml.ctrl(qml.Hadamard, (0,1), control_values=(1,1))(wires=2)
    # Hadamard operation to qubit 2 when control qubits are (1,1)
    qml.ctrl(qml.PauliZ, (0,1), control_values=(1,0))(wires=2)
    # PauliZ operation to qubit 2 when control qubits are (0,1)
    return qml.state()
```

General N-qubit gating

Some miscellaneous routines

function ctrl(op, control, control_values=None, work_wires=None)

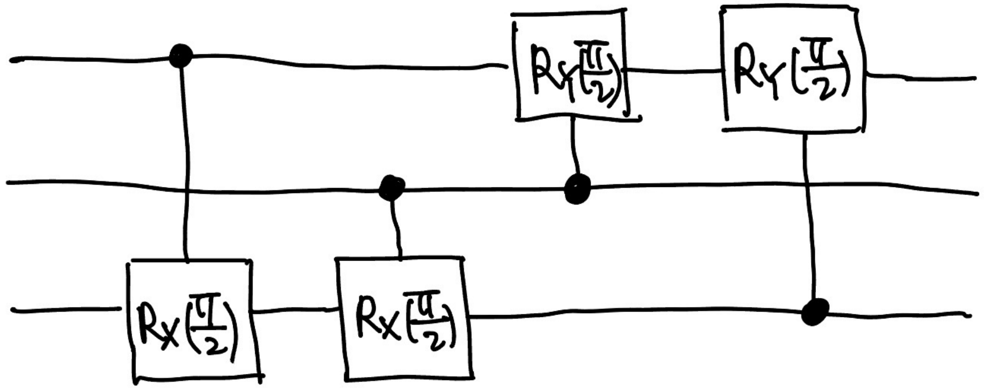
```
@qml.qnode(dev)
def circuit2():
    qml.ctrl(qml.RX, (1,2), control_values=(1,
    # RX(pi/4) operation to qubit 0 when contr
    qml.ctrl(qml.Hadamard, (0,1), control_valu
    # Hadamard operation to qubit 2 when contr
    qml.ctrl(qml.PauliZ, (0,1), control_values
    # PauliZ operation to qubit 2 when control
    return qml.state()
```



Sequence of entangling gating

Some miscellaneous routines

class ControlledSequence(base, ..., control, id=None)



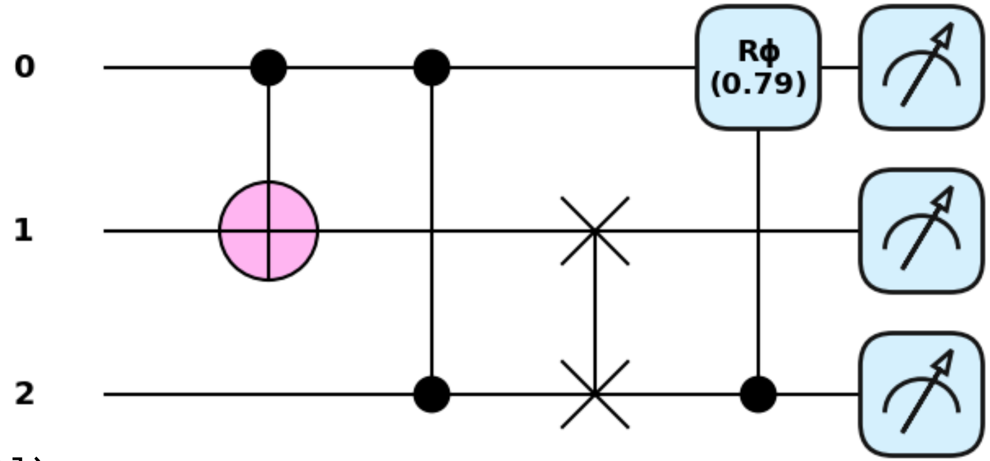
```
dev = qml.device("default.qubit", wires=3)

@qml.qnode(dev)
def circuit3():
    qml.ControlledSequence(qml.RX(np.pi/2, wires=2), control = [0,1])
    # Sequential conduction of two controlled RX(pi/2) gates to qubit 2 whose control bits
    # are qubit 0 & qubit 1
    qml.ControlledSequence(qml.RY(np.pi/2, wires=0), control = [1,2])
    # Sequential conduction of two controlled RY(pi/2) gates to qubit 0 whose control bits
    # are qubit 1 & qubit 2
    return qml.state()
```

Plot your circuits before their executions

Some miscellaneous routines

```
...
import matplotlib.pyplot as plt

dev = qml.device("default.qubit", wires=3)

@qml.qnode(dev)
def circuit1():
    qml.CNOT(wires=[0,1])
    qml.CZ(wires=[0,2])
    qml.SWAP(wires=[1,2])
    qml.ControlledPhaseShift(np.pi/4,wires=[2,0])
    return qml.state()

qml.draw_mpl(circuit1, decimals = 2, style = "pennylane")()
plt.show()
```

function draw_mpl(...)



[Only works in Jupyter notebook]

Snapshots

Some miscellaneous routines

❑ Check state @ intermediate steps in circuits: **`qml.Snapshot` & `qml.snapshots`**

1. Designate points of your interest

```
dev = qml.device("default.qubit", wires=2)

@qml.qnode(dev)
def circuit2():
    qml.Snapshot("before gate operations")
    qml.PauliX(wires=0)
    qml.Snapshot("after PauliX(0)")
    qml.Hadamard(wires=0)
    qml.Snapshot("after Hadamard(0)")
    qml.RX(np.pi/2, wires=0)
    qml.Snapshot("after RX(0)")
    qml.RY(np.pi/4, wires=0)
    qml.Snapshot("after RY(0)")
    qml.RZ(np.pi/8, wires=1)
    qml.Snapshot("after RZ(1)")
    qml.PhaseShift(np.pi, wires=1)
    qml.Snapshot("after PhaseShift(1)")
    qml.PauliY(wires=1)
    qml.Snapshot("after PauliY(1)")
    qml.PauliZ(wires=1)
    qml.Snapshot("after PauliZ(1)")
    return qml.state()
```

2. Run the circuit with snapshots

```
results = qml.snapshots(circuit2)()
```

3. Print states

```
for k, result in results.items():
    print(f"{k}: {result}")
```

Snapshots

Some miscellaneous routines

❑ Check state @ intermediate steps in circuits: **`qml.Snapshot` & `qml.snapshots`**

```
dev = qml.device("default.qubit", wires=2)
```

```
before gate operations: [1.+0.j 0.+0.j 0.+0.j 0.+0.j]
after PauliX(0): [0.+0.j 0.+0.j 1.+0.j 0.+0.j]
after Hadamard(0): [0.70710678+0.j 0. +0.j -0.70710678+0.j 0. +0.j]
after RX(0): [0.5+0.5j 0. +0.j -0.5-0.5j 0. +0.j]
after RY(0): [0.65328148+0.65328148j 0. +0.j -0.27059805-0.27059805j 0. +0.j]
after RZ(1): [0.76817776+0.51327997j 0. +0.j -0.31818965-0.21260752j 0. +0.j]
after PhaseShift(1): [0.76817776+0.51327997j -0. +0.j -0.31818965-0.21260752j -0. +0.j]
after PauliY(1): [0. +0.j -0.51327997+0.76817776j 0. +0.j 0.21260752-0.31818965j]
after PauliZ(1): [0. +0.j 0.51327997-0.76817776j 0. +0.j -0.21260752+0.31818965j]
execution_results: [0. +0.j 0.51327997-0.76817776j 0. +0.j -0.21260752+0.31818965j]
```

```
    qml.RY(np.pi/4, wires=0)
    qml.Snapshot("after RY(0)")
```

```
    qml.RZ(np.pi/8, wires=1)
    qml.Snapshot("after RZ(1)")
```

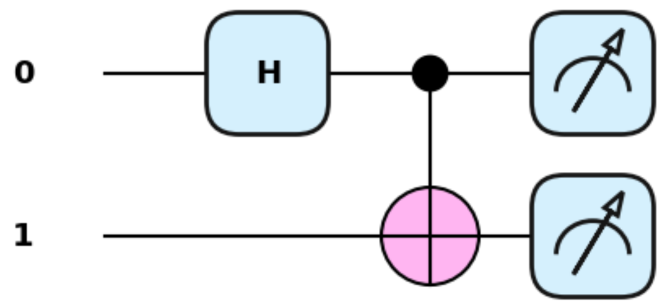
```
for k, result in results.items():
    print(f"{k}: {result}")
```

3. Print states

Missions

The most elementary level of programming

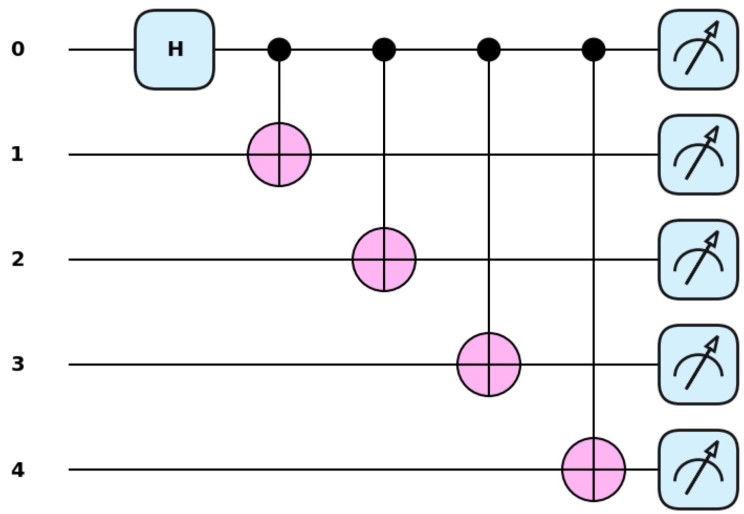
- ❑ Program a circuit that generates 2-qubit Bell-state
  - The circuit diagram is shown below
  - Configure your device with `default.qubit` (emulator) and 2048 shots
  - Execute the circuit 10 times and calculate the average sampling counts



Missions

The most elementary level of programming

- ❑ Program a circuit that generates a 5-qubit Bell-state
  - The circuit diagram is shown below
  - Configure your device with `default.qubit` (emulator)
  - Calculate the probability of computational basis states



Missions

The most elementary level of programming

- Program a 5-qubit parameterized circuit that takes a total of 4 input parameters
  - The circuit diagram is shown below ($R\phi(\theta)$ = a phase shift gate of angle θ)
  - The circuit returns an expectation value on the value 1st qubit obtained by measurement with a computational basis: What do you get with ($\theta 1$, $\theta 2$, $\theta 3$, $\theta 4$) = ($\pi/4$, $\pi/8$, $\pi/16$, $\pi/32$)?