print (qml.draw (qn_sv) (np.pi/2))

from pennylane import numpy as pnp

Codercise

Introduction to Quantum Computing

1.1.1

norm = np.sqrt (np.abs(alpha)**2 + np.abs (beta)**2)

a = alpha / norm

Mistake:-
→ didn't put absolute value

1.1.2

$\langle state1 | state2 \rangle = state_1^{\dagger} . state2$

(complex conjugate → †)

∴ np.dot (np.conj (state1), state2)

→ remember np.conj

1.1.3

P0 = np.abs (state[0])**2

outcome = np.random.choice ([0,1], size=num_meas, p=[P0,P1])

1.1.4

$|\psi'\rangle = U |\psi\rangle$

ψ = np.dot (U, state)

1.1.5

Easy after solving previous exercises

1.2.1

Rearrange →

0
1
2

1.2.2

my_qnode = qml.QNode(my_circuit, dev)

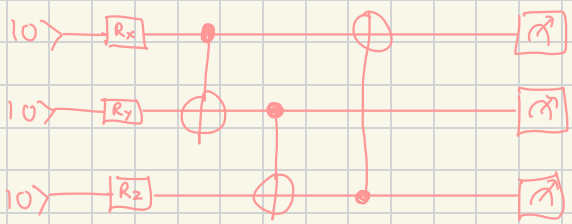
↓ can be called like a function

1.2.3

@qml.qnode(dev)

↓ or decorator can be used

1.2.4



depth=4

1.3.1

qml.QubitUnitary(U, wires=wire) → Alternative way to apply unitary operation

return qml.state()

1.3.2

Unitary matrices can be parameterized. A single qubit-unitary operation can be expressed in terms of just three real numbers:

$$U(\phi,\theta,\omega) = \begin{bmatrix} e^{-i(\phi+\omega)/2}\cos(\theta/2) & -e^{i(\phi-\omega)/2}\sin(\theta/2) \\ e^{-i(\phi-\omega)/2}\sin(\theta/2) & e^{i(\phi+\omega)/2}\cos(\theta/2) \end{bmatrix}$$

qml.Rot(phi, theta, omega, wires=wire)

Single-Qubit Gates

1.4.1

```
if (state == 1):
    qml.PauliX (wires = 0)
```

1.4.2

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

qml.Hadamard (wires = 0)

1.4.3

Simple combination of previous exercises

1.4.4

Coded from scratch,

- → device
- → decorator
- → input argument for state
- → X and H
- → returned state

1.5.1

PauliZ $|0\rangle \longrightarrow |0\rangle$, $|1\rangle \longrightarrow -|1\rangle$ qml.PauliZ (wires = wire)

$|0\rangle \rightarrow H \rightarrow Z \rightarrow |-\rangle$

1.5.2

$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ angle of rotatin (w) in radians

$\begin{pmatrix} e^{-i\omega/2} & 0 \\ 0 & e^{i\omega/2} \end{pmatrix}$ $RZ(\omega)|\psi\rangle = e^{-i\omega/2}\alpha|0\rangle + \beta e^{i\omega/2}|1\rangle$

$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\omega} \end{pmatrix}$ $e^{-i\omega/2}$ is a global phase so can be factored out

$\therefore RZ(\omega)|\psi\rangle \sim \alpha|0\rangle + \beta e^{i\omega}|1\rangle$

qml.RZ (angle, wires = wire) → PauliZ ≡ qml.RZ (np.pi, wires=0)

1.5.3

$RZ(\frac{\pi}{2})$ ⟶ phase gate / S gate qml.S(wires=wire)

$RZ(\pi/4)$ ⟶ T gate qml.T(wires=wire)

adjoints can be applied as,

qml.adjoint(qml.RZ)(omega, wires=0)

$\Downarrow$

qml.RZ(-omega, wires=0)

function whose adjoint is to be calculated

$$RZ^{\dagger}(\omega) = RZ(-\omega)$$

1.6.1

np.pi ⟹ ≈ Pauli X

qml.RX(angle, wires=wire)

qml.RY(angle, wires=wire)

1.6.2

$$RX(\theta) = \begin{bmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ -i\sin(\theta/2) & \cos(\theta/2) \end{bmatrix}$$

```
def apply_rx(theta)
qml.RX(theta, wires=0)
Code for plotting
angles = np.linspace(0, 4*np.pi, 200)
output_states = np.array([apply_rx(t,0) for t in angles])
plot = plotter(angles, output_states)
```

1.6.3

$$RY(\theta) = \begin{bmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{bmatrix}$$

→ very similar to previous code

1.7.1

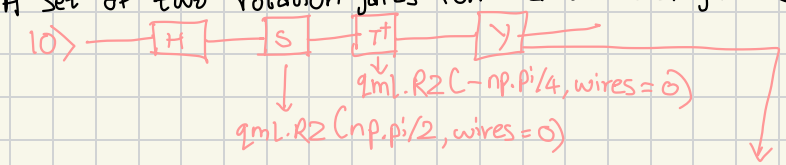
```
qml.Rot(phi, theta, omega, wires=wire)
```

≈

```
qml.RZ(phi, wires=wire)
qml.RY(theta, wires=wire)
qml.RX(omega, wires=wire)
```

```
qml.Hadamard(wires=0) ≡ qml.RZ(np.pi/2, wires=0)
                        qml.RX(np.pi/2, wires=0)
                        qml.RZ(np.pi/2, wires=0)
```

1.7.2 A set of two rotation gates form a universal gate set



Mistake
- Hadamard
- Y gate {visualize rotations}

```
qml.RZ(np.pi, wires=0)
qml.RX(np.pi, wires=0)
```

1.7.3

H and T are also a universal gate set

$$T=\begin{pmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{pmatrix} \qquad H=\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$$

This process is called quantum circuit synthesis, and is a part of quantum compilation.

$$U=\frac{1}{\sqrt{2}^3}\begin{pmatrix}1+e^{i\pi/4}+i(1-e^{i\pi/4}) & 1-e^{i\pi/4}+i(1+e^{i\pi/4})\\ 1+e^{i\pi/4}-i(1-e^{i\pi/4}) & 1-e^{i\pi/4}-i(1+e^{i\pi/4})\end{pmatrix}$$

$H \cdot H = I$ (1) → I solved with Permutation Combination

⇒ 3 H gates ⇒ 3 T gates (2)

Try permutations according to (1) and (2). ∴ HTHTTH

1.8.1 Quantum state preparation

$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}} e^{i5\pi/4}|1\rangle$

$H \longrightarrow RZ(5 * np.pi/4)$

1.8.2

$\frac{\sqrt{3}}{2}|0\rangle - \frac{i}{2}|1\rangle$

$R_X(\theta)|0\rangle = \begin{bmatrix} \cos\theta/2 \\ -i\sin\theta/2 \end{bmatrix}$ $\quad \frac{\theta}{2} = \frac{\pi}{6}$ $\quad \theta = \frac{\pi}{3}$

1.8.3

Template — sub-routine that can be used in the circuit

qml. MottonenStatePreparation ⟶ automatically prepares normalized qubit-state vector.

eg:
```
V = np.array([0.53 - 0.15j, 0.67 + 0.5j])
def func(state=V)
    qml.MottonenStatePreparation(V, wires=0)
    return qml.state()
print(func(V))
```

1.9.1

Outcome probabilities can be returned by

return qml.probs(wires = wire)

We must specify the wire labels of the qubits we would like to measure

For $n|0\rangle$ or $n|1\rangle$

returns [0.5 0.5]

1.9.2

$$|\psi\rangle = \frac{1}{2}|0\rangle + i\frac{\sqrt{3}}{2}|1\rangle$$

$$R_X|0\rangle = \begin{pmatrix} \cos\theta/2 \\ -i\sin\theta/2 \end{pmatrix}$$

$$\cos\theta/2 = \frac{1}{2}$$

$$\sin\theta/2 = -\frac{\sqrt{3}}{2}$$

$$\theta/2 = \frac{-\pi}{3} \qquad \theta = -\frac{2\pi}{3}$$

$$|y+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle)$$

$$|y-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle)$$

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$$

$|0\rangle \rightarrow H \rightarrow R_Z(\pi/2) \rightarrow |y+\rangle$

$|1\rangle \rightarrow H \rightarrow R_Z(\frac{\pi}{2}) \rightarrow |y-\rangle$

1.9.3

```
# Prepare the state
prepare_psi()
# Perform the rotation back to computational basis
qml.adjoint(y_basis_rotation)()
return qml.probs(wires=0)
```

1.10.1

To compute expectation value, qml.expval and specify the observable to be measured.

Commonly PauliX, PauliY and PauliZ, and all give expval as either 1 or -1, as these are their eigenvalues.

```
qml.expval(qml.PauliZ(0))
```

shorthand for (wires=0)

1.10.2

On hardware, we get a single data point for a particular run. So we have to perform the experiment many times.
Each time is called a shot or a sample.

```
dev = qml.device("default.qubit", wires=1, shots=1000)

shot_results = []
shot_values = [100, 1000, 10000, 100000, 1000000]
for shots in shot_values:
    dev = qml.device("default.qubit", wires=1, shots=shots)
    @qml.qnode(dev)
    def func():
        qml.RX(np.pi/4, wires=0)
        qml.Hadamard(0)
        qml.PauliZ(0)
        return qml.expval(qml.PauliY(0))
    shot_results.append(func())
pass
print(qml.math.unwrap(shot_results))
```

1.10.3

We can use samples to calculate expectation value, like calculating a weighted average.

$$\langle Y \rangle = \frac{1 \cdot (\text{number of } 1\text{'s}) + (-1) \cdot (\text{number of } -1\text{'s})}{\text{number of shots}}$$

```
return qml.sample(qml.PauliZ(wires=0))
x=0, y=0
for val in samples:        # array containing samples
    if (val==1):
        x=x+1
```

else :

y=y+1

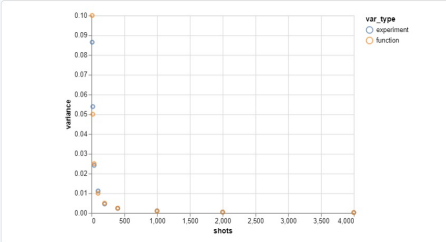
estimated_expval = (x-y)/100000 ← no. of shots

return estimated_expval

1.10.4

```python
def variance_experiment(n_shots):
    """Run an experiment to determine the variance in an expectation
    value computed with a given number of shots.

    Args:
        n_shots (int): The number of shots

    Returns:
        float: The variance in expectation value we obtain running the
        circuit 100 times with n_shots shots each.
    """

    # To obtain a variance, we run the circuit multiple times at each shot value.
    n_trials = 100

    ##################
    # YOUR CODE HERE #
    ##################

    # CREATE A DEVICE WITH GIVEN NUMBER OF SHOTS
    dev=qml.device("default.qubit", wires=0, shots=n_shots)

    # DECORATE THE CIRCUIT BELOW TO CREATE A QNODE
    @qml.qnode(dev)
    def circuit():
        qml.Hadamard(wires=0)
        return qml.expval(qml.PauliZ(wires=0))

    # RUN THE QNODE N_TRIALS TIMES AND RETURN THE VARIANCE OF THE RESULTS
    results=[]
    for i in range(n_trials):
        results.append(circuit())
    mean=np.mean(results)
    variance=np.mean((results-mean)**2)

    return variance


def variance_scaling(n_shots):
    """Once you have determined how the variance in expectation value scales
    with the number of shots, complete this function to programmatically
    represent the relationship.

    Args:
        n_shots (int): The number of shots

    Returns:
        float: The variance in expectation value we expect to see when we run
        an experiment with n_shots shots.
    """

    estimated_variance = 0

    ##################
    # YOUR CODE HERE #
    ##################

    # ESTIMATE THE VARIANCE BASED ON SHOT NUMBER
    estimated_variance=1/n_shots

    return estimated_variance


# Various numbers of shots; you can change this
shot_vals = [10, 20, 40, 100, 200, 400, 1000, 2000, 4000]

# Used to plot your results
results_experiment = [variance_experiment(shots) for shots in shot_vals]
results_scaling = [variance_scaling(shots) for shots in shot_vals]
plot = plotter(shot_vals, results_experiment, results_scaling)
```

User output



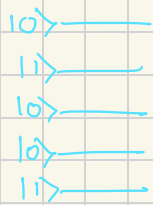
Circuits with Many Qubits

1.11.1

qubit order convention → left to right

$|01001\rangle$
0 1 2 3 4

for drawing, left most is on top



Accepts integer value and returns corresponding computational basis vector

```
bstr = np.binary_repr(basis_id, width=3)
for i, bit in enumerate(bstr):
    if (bit == '1'):
        qml.PauliX(i)
return qml.state()
```

1.11.2

$|+1\rangle = |+\rangle \otimes |1\rangle$

```
qml.Hadamard(0)
qml.PauliX(1)
return qml.expval(qml.PauliY(0)), qml.expval(qml.PauliZ(1))
```

1.11.3

Measure expval of two qubit observable. $(Z \otimes X)$

```
qml.expval(qml.PauliZ(0) @ qml.PauliX(1))
```

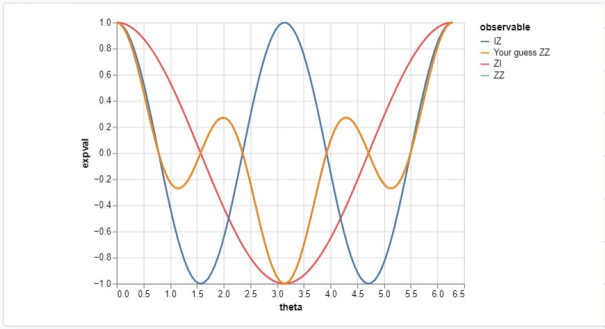
1.11.4

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


@qml.qnode(dev)
def circuit_1(theta):
    """Implement the circuit and measure Z I and I Z.

    Args:
        theta (float): a rotation angle.

    Returns:
        float, float: The expectation values of the observables Z I, and I Z
    """
    ##################
    # YOUR CODE HERE #
    ##################
    qml.RX(theta, wires=0)
    qml.RX(2*theta, wires=1)

    return qml.expval(qml.PauliZ(0)), qml.expval(qml.PauliZ(1))

```

```
@qml.qnode(dev)
def circuit_2(theta):
    """Implement the circuit and measure Z Z.

    Args:
        theta (float): a rotation angle.

    Returns:
        float: The expectation value of the observable Z Z
    """

    ##################
    # YOUR CODE HERE #
    ##################
    qml.RX(theta, wires=0)
    qml.RX(2*theta, wires=1)

    return qml.expval(qml.PauliZ(0)@qml.PauliZ(1))

```

```python
def zi_iz_combination(ZI_results, IZ_results):
    """Implement a function that acts on the ZI and IZ results to
    produce the ZZ results. How do you think they should combine?

    Args:
        ZI_results (np.array[float]): Results from the expectation value of
            ZI in circuit_1.
        IZ_results (np.array[float]): Results from the expectation value of
            IZ in circuit_2.

    Returns:
        np.array[float]: A combination of ZI_results and IZ_results that
        produces results equivalent to measuring ZZ.
    """

    combined_results = np.zeros(len(ZI_results))

    ##################
    # YOUR CODE HERE #
    ##################
    combined_results=ZI_results*IZ_results

    return combined_results


theta = np.linspace(0, 2 * np.pi, 100)

# Run circuit 1, and process the results
circuit_1_results = np.array([circuit_1(t) for t in theta])

ZI_results = circuit_1_results[:, 0]
IZ_results = circuit_1_results[:, 1]
combined_results = zi_iz_combination(ZI_results, IZ_results)

# Run circuit 2
ZZ_results = np.array([circuit_2(t) for t in theta])

# Plot your results
plot = plotter(theta, ZI_results, IZ_results, ZZ_results, combined_results)
```

User output

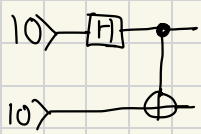


1.12.1

qml.CNOT(wires=[control, target])

bits = [int(x) for x in np.binary_repr(basis_id, width=num_wires)]
qml.BasisStatePreparation(bits, wires=[0,1])
qml.CNOT(wires=[0,1])

1.12.2



produces maximally entangled Bell state

$$\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

1.12.3

qml.CRX, qml.CRY, qml.CRZ
qml.CRX(theta, wires=[control, target])

return qml.probs(wires=range(3))

1.13.1

$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$ $Z = HXH$ $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

Similary,

Controlled Z can be constructed using Controlled X (C-NOT) and H.

or

qml.CZ(wires=[control, target])

$$CX = \left[\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{array}\right] \qquad CZ = \left[\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{array}\right]$$

→ Hadamard causes superposition of all computational basis states.

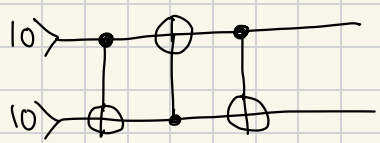
qml.Hadamard(0)
qml.CNOT(wires=[1,0])
qml.Hadamard(0)

1.13.2

SWAP → exchanges the state of two qubits

qml.SWAP(wires=[control, target])

$$\left[\begin{array}{cc|cc} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right]$$

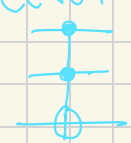


1.13.3

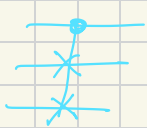
qml.Toffoli(wires=[control1, control2, target])

$$\begin{pmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0 \end{pmatrix}$$

CCNOT



controlled SWAP → Fredkin gate



qml.Toffoli(wires=[0,1,2])
qml.Toffoli(wires=[0,2,1])
qml.Toffoli(wires=[0,1,2])

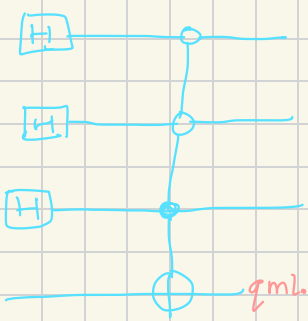
1. 13.4

Mixed polarity multi-controlled Toffoli gates

qml.MultiControlledX(control_wires=[0,1,2,3], wires=4, control_values="1011")

string of control bits

control on '0' is denoted by an open circle

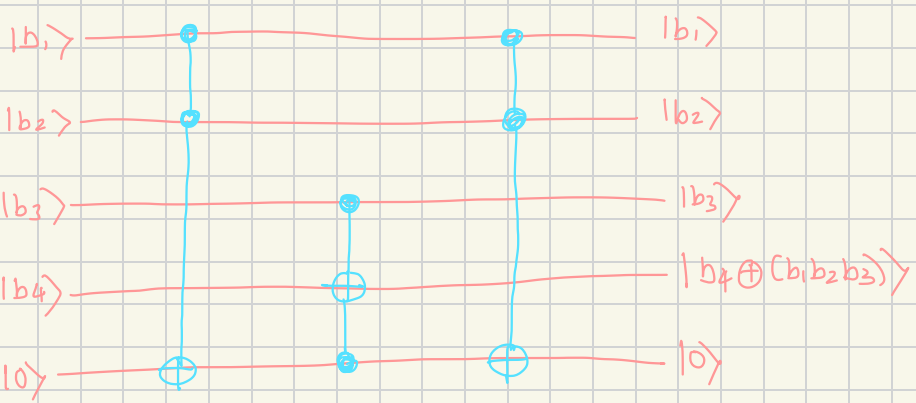


Control state $|001\rangle$

qml.Hadamard(0)
qml.Hadamard(1)
qml.Hadamard(2)
qml.MultiControlledX(control_wires=[0,1,2], wires=3, control_values="001")

1.13.5

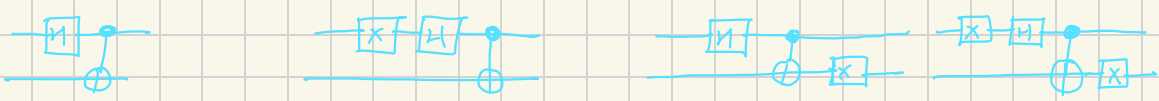
CCCNOT, we require an auxiliary qubit that starts and ends at $|0\rangle$



1.14.1

Bell states

$|\Psi+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ $|\Psi-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$ $|\Phi+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$ $|\Phi-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$



1.14.2

quantum multiplexer → When all 2^n possible cases of n control qubits are implemented, and the target operation is a single-qubit rotation, it is called a uniformly controlled rotation.

If first two qubits are both $|0\rangle$ do nothing
$|01\rangle \Rightarrow$ Pauli X on third qubit
$|10\rangle \Rightarrow$ Pauli Z on third qubit
$|11\rangle \Rightarrow$ Pauli Y on third qubit

```
qml.MultiControlledX(control_wires=[0,1], wires=2, control_values="01")

qml.Hadamard(2)
qml.MultiControlledX(control_wires=[0,1], wires=2, control_values="10")
qml.Hadamard(2)

qml.adjoint(qml.S(2))
qml.Toffoli([0,1,2])
qml.S(2)
```

Variational Classifier

quantum circuits that can be trained from labelled data to classify new data samples.

1) Fitting the Parity Function

$$f: x \in \{0,1\}^{\otimes n} \rightarrow y = \begin{cases} 1 \text{ if uneven number of 1's in } x \\ 0 \text{ else} \end{cases}$$

→ Variational classifiers usually define a "layer" or "block", which is an elementary circuit architecture that gets repeated to build the full variational circuit

→ 4 qubits, consisting of an arbitrary rotation on every qubit, as well a ring of CNOTs that entangles each qubit neighbour

→ Parameters of the layer → weights