

# Project: Quantum Machine Learning for Conspicuity Detection in Production

## TASK 1

Familiarize yourself with PennyLane. The tutorials in the PennyLane codebook are a good way to get started. We recommend the codebooks from the sections “Introduction to Quantum Computing”, “Single-Qubit Gates” and “Circuits with Many Qubits”, which can be found on the following page (registration is required): PennyLane Codebook. Document your progress and share your learnings as you follow these PennyLane tutorials.

- Codercise— Normalization of quantum states

Ques:

In this codercise, you are given an unnormalized vector

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \quad |\alpha|^2 + |\beta|^2 \neq 1.$$

We can turn this into an equivalent, valid quantum state  $|\psi'\rangle$  by *normalizing* it. Your task is to complete the function `normalize_state` so that, given  $\alpha$  and  $\beta$ , it normalizes this state to

$$|\psi'\rangle = \alpha'|0\rangle + \beta'|1\rangle, \quad |\alpha'|^2 + |\beta'|^2 = 1.$$

Theory:

Of course,  $|0\rangle$  and  $|1\rangle$  are not the only possible states for a qubit (otherwise, they wouldn't be any different than bits!). What makes qubits so special is that they can exist in a **superposition** state somewhere “between”  $|0\rangle$  and  $|1\rangle$ . Mathematically, the state of a qubit in superposition is a linear combination of the basis states,

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}, \quad (6)$$

where  $\alpha$  and  $\beta$  are complex numbers such that

$$\alpha\alpha^* + \beta\beta^* = 1, \quad (7)$$

and the  $*$  indicates the complex conjugate. These  $\alpha$  and  $\beta$  are called **amplitudes**, or **probability amplitudes**. The amplitudes carry information about the relative strength of  $|0\rangle$  and  $|1\rangle$  in the state.

*Tip.* A common misconception is that a qubit in a superposition of two states is *in* both states at the same time. This is false; the qubit is only ever *in* one state. It's just that sometimes, that state may be a linear combination of the basis states.

Now that we have complex numbers in the mix, we have to be a bit more careful. Let's suppose we have two states

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \quad \text{and} \quad |\phi\rangle = \gamma|0\rangle + \delta|1\rangle \quad (8)$$

and we would like to take the inner product between them,  $\langle\phi|\psi\rangle$ . First, we must compute the bra of  $|\phi\rangle$ . When a qubit is in a superposition of the basis states, we can compute the bra by taking the bra of each basis state one at a time, remembering to take the *conjugate* of the amplitudes:

$$\langle\phi| = \gamma^*\langle 0| + \delta^*\langle 1| = (\gamma^* \quad \delta^*). \quad (9)$$

With more interesting quantum states, the inner product tells us something about the overlap, or, in a sense, similarity between two states. The result of the inner product is a complex number. If it's 0, the two states are orthogonal, and if it is 1, they are the same and the state is normalized (as we observed when we computed the inner product of  $|1\rangle$  with itself). There are two ways to compute the inner product of two superposition states. You could write out the matrices:

$$\langle\phi|\psi\rangle = (\gamma^* \quad \delta^*) \begin{pmatrix} \alpha \\ \beta \end{pmatrix}. \quad (10)$$

Alternatively, you can work purely in bra-ket notation. The inner product is **linear**, and we can compute the inner product by expanding out the expression, much like we would a polynomial in algebra:

$$\begin{aligned} \langle\phi|\psi\rangle &= (\gamma^*\langle 0| + \delta^*\langle 1|) \cdot (\alpha|0\rangle + \beta|1\rangle) \\ &= \gamma^*\alpha\langle 0|0\rangle + \delta^*\alpha\langle 1|0\rangle + \gamma^*\beta\langle 0|1\rangle + \delta^*\beta\langle 1|1\rangle. \end{aligned} \quad (11)$$

### Code:

```

1  # Here are the vector representations of |0> and |1>, for convenience
2  ket_0 = np.array([1, 0])
3  ket_1 = np.array([0, 1])
4
5
6  def normalize_state(alpha, beta):
7      """Compute a normalized quantum state given arbitrary amplitudes.
8
9      Args:
10         alpha (complex): The amplitude associated with the |0> state.
11         beta (complex): The amplitude associated with the |1> state.
12
13     Returns:
14         np.array[complex]: A vector (numpy array) with 2 elements that represents
15         a normalized quantum state.
16     """
17
18     #####
19     # YOUR CODE HERE #
20
21     # Calculate the normalization factor
22     norm = np.sqrt(np.abs(alpha)**2 + np.abs(beta)**2)
23
24     # Normalize the amplitudes
25     alpha_normalized = alpha / norm
26     beta_normalized = beta / norm
27
28     # Return the normalized state as a numpy array
29     return np.array([alpha_normalized, beta_normalized])
30     #####
31
32     # CREATE A VECTOR [a', b'] BASED ON alpha AND beta SUCH THAT |a'|^2 + |b'|^2 = 1
33
34     # RETURN A VECTOR
35
36

```

[Reset Code](#)

**Submit**

Correct!

### Learning:

The function `normalize_state` normalizes a given quantum state vector represented by complex amplitudes `alpha` and `beta`. Here's a step-by-step breakdown:

#### 1. Normalization Factor Calculation:

- The normalization factor (`norm`) is calculated using the formula:

$$\text{norm} = \sqrt{|\alpha|^2 + |\beta|^2}$$

- Here,  $|\alpha|^2$  and  $|\beta|^2$  represent the squared magnitudes of the complex numbers  $\alpha$  and  $\beta$ , respectively.

- This ensures that the sum of the probabilities (squared magnitudes) of both states equals 1.

## 2. Amplitude Normalization:

- Each amplitude is divided by the normalization factor:

$$\alpha_{\text{normalized}} = \frac{\alpha}{\text{norm}}$$

$$\beta_{\text{normalized}} = \frac{\beta}{\text{norm}}$$

- This step ensures that the normalized amplitudes satisfy the condition  $|\alpha'|^2 + |\beta'|^2 = 1$ .

## 3. Returning the Normalized State:

- The function returns a numpy array containing the normalized amplitudes  $[\alpha_{\text{normalized}}, \beta_{\text{normalized}}]$ .

- This represents the quantum state as a unit vector in the Hilbert space, where the total probability is 1.

## Summary

This code takes arbitrary amplitudes for the quantum states  $|0\rangle$  and  $|1\rangle$ , normalizes them, and returns the normalized state vector. Normalization is crucial in quantum mechanics to ensure that the total probability of all possible states is equal to 1.

- Codercise— Inner product and orthonormal bases

### Ques:

Complete the `inner_product` function below that computes the inner product between two arbitrary states. Then, use it to verify that  $|0\rangle$  and  $|1\rangle$  form an **orthonormal basis**, i.e., the states are normalized and orthogonal.

### Theory:

Of course,  $|0\rangle$  and  $|1\rangle$  are not the only possible states for a qubit (otherwise, they wouldn't be any different than bits!). What makes qubits so special is that they can exist in a **superposition** state somewhere "between"  $|0\rangle$  and  $|1\rangle$ . Mathematically, the state of a qubit in superposition is a linear combination of the basis states,

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}, \quad (6)$$

where  $\alpha$  and  $\beta$  are complex numbers such that

$$\alpha\alpha^* + \beta\beta^* = 1, \quad (7)$$

and the  $*$  indicates the complex conjugate. These  $\alpha$  and  $\beta$  are called **amplitudes**, or **probability amplitudes**. The amplitudes carry information about the relative strength of  $|0\rangle$  and  $|1\rangle$  in the state.

*Tip.* A common misconception is that a qubit in a superposition of two states is *in* both states at the same time. This is false; the qubit is only ever *in* one state. It's just that sometimes, that state may be a linear combination of the basis states.

Now that we have complex numbers in the mix, we have to be a bit more careful. Let's suppose we have two states

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \quad \text{and} \quad |\phi\rangle = \gamma|0\rangle + \delta|1\rangle \quad (8)$$

and we would like to take the inner product between them,  $\langle\phi|\psi\rangle$ . First, we must compute the bra of  $|\phi\rangle$ . When a qubit is in a superposition of the basis states, we can compute the bra by taking the bra of each basis state one at a time, remembering to take the *conjugate* of the amplitudes:

$$\langle\phi| = \gamma^*\langle 0| + \delta^*\langle 1| = (\gamma^* \quad \delta^*). \quad (9)$$

With more interesting quantum states, the inner product tells us something about the overlap, or, in a sense, similarity between two states. The result of the inner product is a complex number. If it's 0, the two states are orthogonal, and if it is 1, they are the same and the state is normalized (as we observed when we computed the inner product of  $|1\rangle$  with itself).

There are two ways to compute the inner product of two superposition states. You could write out the matrices:

$$\langle\phi|\psi\rangle = (\gamma^* \quad \delta^*) \begin{pmatrix} \alpha \\ \beta \end{pmatrix}. \quad (10)$$

Alternatively, you can work purely in bra-ket notation. The inner product is **linear**, and we can compute the inner product by expanding out the expression, much like we would a polynomial in algebra:

$$\begin{aligned} \langle\phi|\psi\rangle &= (\gamma^*\langle 0| + \delta^*\langle 1|) \cdot (\alpha|0\rangle + \beta|1\rangle) \\ &= \gamma^*\alpha\langle 0|0\rangle + \delta^*\alpha\langle 1|0\rangle + \gamma^*\beta\langle 0|1\rangle + \delta^*\beta\langle 1|1\rangle. \end{aligned} \quad (11)$$

## Code:

```
1 def inner_product(state_1, state_2):
2     """Compute the inner product between two states.
3
4     Args:
5         state_1 (np.array[complex]): A normalized quantum state vector
6         state_2 (np.array[complex]): A second normalized quantum state vector
7
8     Returns:
9         complex: The value of the inner product <state_1 | state_2>.
10    """
11
12    #####
13    # YOUR CODE HERE #
14    #####
15
16    # COMPUTE AND RETURN THE INNER PRODUCT
17
18    # Compute the inner product
19    return np.vdot(state_1, state_2)
20
21
22 # Test your results with this code
23 ket_0 = np.array([1, 0])
24 ket_1 = np.array([0, 1])
25
26 print(f"<0|0> = {inner_product(ket_0, ket_0)}")
27 print(f"<0|1> = {inner_product(ket_0, ket_1)}")
28 print(f"<1|0> = {inner_product(ket_1, ket_0)}")
29 print(f"<1|1> = {inner_product(ket_1, ket_1)}")
30
```

[Reset Code](#)[Submit](#)

Correct!

### User output

```
<0|0> = 1
<0|1> = 0
<1|0> = 0
<1|1> = 1
```

## Learning:

Brief explanation of the inner\_product function and the associated test code:

### 1. Inner Product Function:

- The function `inner_product(state_1, state_2)` computes the inner product (also known as the dot product in the complex space) between two normalized quantum states `state_1` and `state_2`.
- It takes two arguments:
  - `state_1`: A numpy array representing the first normalized quantum state.
  - `state_2`: A numpy array representing the second normalized quantum state.
- The function uses the `np.vdot` function from the NumPy library to compute the inner product between `state_1` and `state_2`. `np.vdot` handles complex conjugation of the first vector (as required in quantum mechanics) and then computes the dot product.
- The result is returned as a complex number representing the inner product  $\langle \text{state}_1 | \text{state}_2 \rangle$ .

## 2. Testing the Function:

- Four test cases are provided to check the inner product between different quantum states:
  - $\langle 0|0\rangle$ : The inner product between ket\_0 and ket\_0.
  - $\langle 0|1\rangle$ : The inner product between ket\_0 and ket\_1.
  - $\langle 1|0\rangle$ : The inner product between ket\_1 and ket\_0.
  - $\langle 1|1\rangle$ : The inner product between ket\_1 and ket\_1.
- Here, ket\_0 is defined as  $[1, 0]$  (representing the quantum state  $|0\rangle$ ), and ket\_1 is defined as  $[0, 1]$  (representing the quantum state  $|1\rangle$ ).

## 3. Expected Output:

- The output of the test cases:
  - $\langle 0|0\rangle = 1$ : Since ket\_0 is orthonormal with itself, the inner product equals 1.
  - $\langle 0|1\rangle = 0$ : Since ket\_0 and ket\_1 are orthonormal, the inner product equals 0.
  - $\langle 1|0\rangle = 0$ : The inner product between ket\_1 and ket\_0 is also 0 because they are orthonormal.
  - $\langle 1|1\rangle = 1$ : Since ket\_1 is orthonormal with itself, the inner product equals 1.

## Summary:

This code defines a function to compute the inner product between two quantum states. It then tests this function with basic quantum states  $|0\rangle$  and  $|1\rangle$  to confirm that the function works as expected. The inner product is a fundamental operation in quantum mechanics, used to determine the overlap or similarity between two quantum states.

- Codercise— Sampling measurement outcomes

### Ques:

Write the function `measure_state` that takes a quantum state vector as input and simulates the outcomes of an arbitrary number of quantum measurements, i.e., return a list of samples 0 or 1 based on the probabilities given by the input state.

### Theory:

Creating qubit states and putting them in superposition happens at the beginning of an algorithm. In order to perform a meaningful quantum computation, we'll need a way to get information *from* the qubits at the end of an algorithm. That is, we need a way to measure qubits. Measurement in quantum computing is probabilistic. When we measure, we can't see whether a qubit is in a superposition, rather we observe the qubit either in state  $|0\rangle$  or state  $|1\rangle$ . The amplitudes  $\alpha$  and  $\beta$  contain the information about the probability of each of those outcomes:

$$\begin{aligned}\text{Prob}(\text{measure and observe } |0\rangle) &= |\alpha|^2, \\ \text{Prob}(\text{measure and observe } |1\rangle) &= |\beta|^2.\end{aligned}\tag{12}$$

This notation,  $|\cdot|^2$  is called the "mod squared", and is simply multiplication of a number by its complex conjugate, i.e.,  $|\alpha|^2 = \alpha\alpha^*$ . Since there are only two possible outcomes for the measurement, it must be that these probabilities sum to 1 for a valid qubit state; this is why quantum states must be *normalized* ( $|\alpha|^2 + |\beta|^2 = 1$ ).

Colloquially, we often refer to observing the qubit in state  $|0\rangle$  after measurement as a "measurement outcome of 0", and similarly for  $|1\rangle$ . In other words, if we measure a  $|0\rangle$  we can map that to a classical "0" and if we measure a  $|1\rangle$  we can map that to a classical "1". After measurement, the qubit itself remains in the observed state. This means we can't tell right away what some original state  $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$  might have been. Measurement has given us just a single bit of information, 0 or 1, that we associate with the corresponding outcome. In order to determine the full state, we must take many, many measurements in order to estimate the outcome probabilities, and thus  $\alpha$  and  $\beta$ .

### Code:

```
1 def measure_state(state, num_meas):
2     """Simulate a quantum measurement process.
3
4     Args:
5         state (np.array[complex]): A normalized qubit state vector.
6         num_meas (int): The number of measurements to take
7
8     Returns:
9         np.array[int]: A set of num_meas samples, 0 or 1, chosen according to the probability
10            distribution defined by the input state.
11     """
12
13     #####
14     # YOUR CODE HERE #
15     # Compute the probability of measuring 0 and 1
16     prob_0 = np.abs(state[0])**2
17     prob_1 = np.abs(state[1])**2
18
19     # Generate measurement outcomes based on these probabilities
20     measurements = np.random.choice([0, 1], size=num_meas, p=[prob_0, prob_1])
21
22     return measurements
23     #####
24
25     # COMPUTE THE MEASUREMENT OUTCOME PROBABILITIES
26
27     # RETURN A LIST OF SAMPLE MEASUREMENT OUTCOMES
28
29
30
```

[Reset Code](#)

Submit

Correct!

## Learning:

Brief explanation of the `measure_state` function:

### 1. Purpose:

- The `measure_state` function simulates the measurement process of a quantum qubit state multiple times. The result of each measurement will be either 0 or 1, determined by the probabilities associated with the state vector.

### 2. Function Arguments:

- `state`: A numpy array representing a normalized quantum state vector with complex amplitudes for the basis states  $|0\rangle$  and  $|1\rangle$ .
- `num_meas`: The number of measurements to be simulated.

### 3. Calculating Measurement Probabilities:

- The probability of measuring the state as  $|0\rangle$  (`prob_0`) is calculated by taking the squared magnitude of the first element in the state vector
- Similarly, the probability of measuring the state as  $|1\rangle$  (`prob_1`) is calculated by taking the squared magnitude of the second element in the state vector
- These probabilities are guaranteed to sum to 1, as the state vector is normalized.

### 4. Generating Measurement Outcomes:

- The function uses `np.random.choice` to simulate `num_meas` measurements. This function randomly selects either 0 or 1 for each measurement, with the probabilities `prob_0` and `prob_1`, respectively.
- The `size=num_meas` argument specifies the number of measurements to generate.
- The `p=[prob_0, prob_1]` argument provides the probability distribution used to select 0 or 1.

### 5. Returning the Results:

- The function returns a numpy array `measurements` containing the results of the `num_meas` simulated measurements. Each entry in this array will be either 0 or 1.

## Summary:

This code simulates the process of measuring a quantum state multiple times. The results of these measurements are determined by the probability distribution of the quantum state, which is derived from the squared magnitudes of the state's complex amplitudes. This function is useful in quantum computing for understanding the behavior of qubits under repeated measurements.



- Codercise— Applying a quantum operation

### Ques:

Recall that quantum operations are represented as matrices. To preserve normalization, they must be a special type of matrix called a **unitary** matrix. For some  $2 \times 2$  complex-valued unitary matrix  $U$ , the state of the qubit after an operation is

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

Let's simulate the process by completing the function `apply_u` below to apply the provided quantum operation `U` to an input `state`.

### Theory:

Now that we have the starting and ending components of a quantum algorithm, we need the final ingredient that happens in between: the manipulation of qubit states. Qubit states are vectors, so we need a mathematical means of modifying a vector  $|\psi\rangle$  to produce another vector  $|\psi'\rangle$  :

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \rightarrow |\psi'\rangle = \alpha'|0\rangle + \beta'|1\rangle. \quad (13)$$

What sends a 2-dimensional vector to another 2-dimensional vector? Multiplication by a  $2 \times 2$  matrix,  $U$  :

$$|\psi'\rangle = U|\psi\rangle. \quad (14)$$

But not just any matrix will do. The matrix must preserve the normalization of the state. Even after an operation, the measurement outcome probabilities must sum to 1, i.e.,  $|\alpha'|^2 + |\beta'|^2 = 1$ . There is a special class of matrices that preserves the length of quantum states: **unitary matrices**. Their defining property is that  $UU^\dagger = I$ , where the  $\dagger$  indicates the taking complex conjugate of all elements in the transpose of  $U$ , and  $I$  is the  $2 \times 2$  identity matrix. As you progress through the rest of this module, you will become familiar with many common unitary operations.

### Code:

```

1  U = np.array([[1, 1], [1, -1]]) / np.sqrt(2)
2
3
4  def apply_u(state):
5      """Apply a quantum operation.
6
7      Args:
8          state (np.array[complex]): A normalized quantum state vector.
9
10     Returns:
11         np.array[complex]: The output state after applying U.
12     """
13
14     #####
15     # YOUR CODE HERE #
16     return np.dot(U, state)
17     #####
18
19     # APPLY U TO THE INPUT STATE AND RETURN THE NEW STATE
20
21

```

[Reset Code](#)

Submit

Correct!

## Learning:

brief explanation of the apply\_u function:

### 1. Purpose:

- The apply\_u function applies a specific quantum operation (represented by the unitary matrix  $U$ ) to a given quantum state vector.

### 2. Unitary Matrix U:

- The matrix  $U$  is defined as:  $U = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$
- This is the Hadamard matrix, a common quantum gate used to create superposition states. When applied to a qubit, it transforms the basis states  $|0\rangle$  and  $|1\rangle$  into equal superpositions of  $|0\rangle$  and  $|1\rangle$ .

### 3. Function Arguments:

- state: A numpy array representing a normalized quantum state vector, typically in the form  $[\alpha, \beta]$ , where  $\alpha$  and  $\beta$  are complex amplitudes associated with the basis states  $|0\rangle$  and  $|1\rangle$ .

### 4. Applying the Quantum Operation:

- The function uses the np.dot function to perform matrix multiplication between the matrix  $U$  and the state vector.
- This operation transforms the input state according to the rules defined by the matrix  $U$ .

### 5. Returning the New State:

- The function returns the resulting state vector after the application of  $U$ . This new state is the quantum state after the transformation.

## Summary:

This code applies the Hadamard gate (represented by matrix  $U$ ) to a quantum state vector. The Hadamard gate is a fundamental operation in quantum computing that creates superposition states by transforming the input state. The function multiplies the state vector by the Hadamard matrix and returns the resulting state, representing the qubit's new state after the operation.

- Codercise— Unitaries in PennyLane

### Ques:

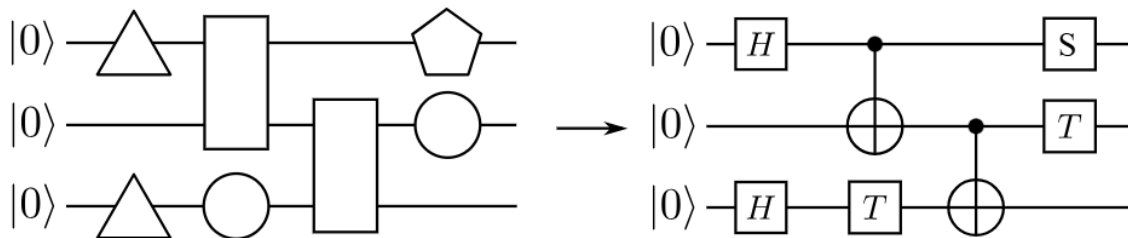
In PennyLane, unitary operations specified by a matrix can be implemented in a quantum circuit using the `QubitUnitary` operation. `QubitUnitary` is a parametrized gate, and can be called like so:

```
qml.QubitUnitary(U, wires=wire)
```

Complete the quantum function below to create a circuit that applies `U` to the qubit and returns its *state*. (Compare this to the earlier function `apply_u` that you wrote before - isn't it nice to not have to worry about the matrix arithmetic?)

### Theory:

Now that we know what qubits are, and how to express computations on them, it's time to make an important transition: what exactly are we *doing* to the qubits? What are the different possible gates, and how do they work?



We know already that single qubit operations must take valid qubit states to other valid qubit states, and this is done using matrix-vector multiplication by a  $2 \times 2$  matrix. Given an initial qubit state  $|\psi\rangle$ , a single-qubit operation  $U$  sends

$$|\psi\rangle \rightarrow |\psi'\rangle = U|\psi\rangle, \quad (1)$$

where  $|\psi'\rangle$  is the new state.

However, recall that qubit state vectors have some special properties - in particular, they are normalized, i.e., have length 1. Thus, any matrix that operates on qubits is going to require a structure that preserves this property. Matrices of this type are called **unitary matrices**. More formally, an  $n \times n$  complex-valued matrix  $U$  is unitary if

$$UU^\dagger = U^\dagger U = I_n, \quad (2)$$

where  $I_n$  is the  $n$ -dimensional identity matrix.  $U^\dagger$  is the notation for the conjugate transpose or adjoint of  $U$  (i.e., take the transpose of the matrix, and replace each entry with its complex conjugate).

## Code:

```
1 dev = qml.device("default.qubit", wires=1)
2
3 U = np.array([[1, 1], [1, -1]]) / np.sqrt(2)
4
5
6 @qml.qnode(dev)
7 def apply_u():
8
9     #####
10    # YOUR CODE HERE #
11    qml.QubitUnitary(U, wires=0)
12    #####
13
14    # USE QubitUnitary TO APPLY U TO THE QUBIT
15
16    # Return the state
17    return qml.state()
18
```

[Reset Code](#)[Submit](#)

Correct!

## Learning:

Brief explanation of the `apply_u` function using PennyLane:

### 1. Device Setup:

- The device `dev` is created with `qml.device("default.qubit", wires=1)`, which sets up a quantum simulation environment with a single qubit.

### 2. Matrix U:

- The matrix `U` is defined as:

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

- This is the Hadamard matrix, commonly used in quantum computing to create superposition states from standard basis states.

### 3. QNode Function:

- The `apply_u` function is defined as a quantum node (QNode) using the decorator `@qml.qnode(dev)`. QNodes are the core of PennyLane, allowing quantum functions to be executed on quantum devices.

### 4. Applying the Unitary Matrix:

- The `qml.QubitUnitary(U, wires=0)` function applies the unitary matrix `U` to the qubit at wire 0.
- `QubitUnitary` is used to apply any arbitrary unitary operation (in this case, the Hadamard operation) to a qubit.

## 5. Returning the State:

- The function returns the quantum state of the qubit after the unitary operation is applied using `qml.state()`.
- `qml.state()` retrieves the state vector of the qubit, which represents the superposition of the qubit after applying the Hadamard gate.

### Summary:

This function applies the Hadamard gate (represented by matrix  $U$ ) to a single qubit using PennyLane's `QubitUnitary` operation. The resulting state of the qubit is returned, which should be a superposition state if the qubit was initially in the  $|0\rangle$  or  $|1\rangle$  state. This process is fundamental in quantum computing for creating and manipulating superposition states.

- Codercise— Flipping bits

### Ques:

A common use of the  $X$  gate is in initializing the state of a qubit at the beginning of an algorithm. Quite often, we would like our qubits to start in state  $|0\rangle$  (which is the default in PennyLane), however there are many cases where we instead would like to start from  $|1\rangle$ . Complete the function below by using `qml.PauliX` to initialize the qubit's state to  $|0\rangle$  or  $|1\rangle$  based on an input flag. Then, use `qml.QubitUnitary` to apply the provided unitary `U`.

### Theory:

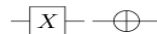
The first gate we will explore is the Pauli  $X$  gate. This operation is represented by the following unitary matrix  $X$  :

$$\begin{aligned} X|0\rangle &= |1\rangle, \\ X|1\rangle &= |0\rangle. \end{aligned} \quad (3)$$

This is also known as the **bit flip** operation, or **NOT gate**, due to its similarity to the Boolean NOT operation. From its action on the basis states, it is straightforward to deduce that  $X$  should have the following form (you can check this by hand, too):

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}. \quad (4)$$

There are two representations of  $X$  used in quantum circuits:



The first, an  $X$  in a box, is far more common. The second, a circle with a vertical line, in some special cases where  $X$  is involved in a 2-qubit operation (you might remember these from the graphics in I.2). While this notation can be still be used for the single-qubit gate, it is rarely encountered in the wild.

### Code:

```

1 dev = qml.device("default.qubit", wires=1)
2
3 U = np.array([[1, 1], [1, -1]]) / np.sqrt(2)
4
5
6 @qml.qnode(dev)
7 def varied_initial_state(state):
8     """Complete the function such that we can apply the operation U to
9     either |0> or |1> depending on the input argument flag.
10
11     Args:
12         state (int): Either 0 or 1. If 1, prepare the qubit in state |1>,
13         otherwise, leave it in state 0.
14
15     Returns:
16         np.array[complex]: The state of the qubit after the operations.
17     """
18     #####
19     # YOUR CODE HERE #
20     #####
21
22     # KEEP THE QUBIT IN |0> OR CHANGE IT TO |1> DEPENDING ON THE state PARAMETER
23
24     # APPLY U TO THE STATE
25     # Initialize the qubit to |1> if state == 1
26     if state == 1:
27         qml.PauliX(wires=0) # Apply X gate to flip the state from |0> to |1>
28
29     # Apply the unitary matrix U to the qubit
30     qml.QubitUnitary(U, wires=0)
31
32     # Return the state of the qubit
33     return qml.state()
34
35

```

[Reset Code](#)

Submit

Correct!

## Learning:

Key concepts and functionalities demonstrated in the varied\_initial\_state function:

### 1. Quantum Device Initialization:

- **PennyLane Device:** `qml.device("default.qubit", wires=1)` sets up a quantum device with a single qubit (wires=1) using the default simulator provided by PennyLane. This device is where all quantum operations will be performed.

### 2. Unitary Matrix U:

- The matrix U is defined as:

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

- This matrix represents the Hadamard gate, a fundamental quantum gate used to create superposition states. When applied to the  $|0\rangle$  state, it creates an equal superposition of  $|0\rangle$  and  $|1\rangle$ .

### 3. Function Purpose:

- **varied\_initial\_state(state):** The function prepares the qubit in either the  $|0\rangle$  or  $|1\rangle$  state based on the input parameter state. After preparing the initial state, it applies the unitary matrix U to the qubit.

### 4. State Preparation:

- The input state can be either 0 or 1:
  - If state is 0, the qubit remains in the default  $|0\rangle$  state.
  - If state is 1, the qubit is flipped to the  $|1\rangle$  state using the PauliX gate: `qml.PauliX(wires=0)`. The PauliX gate acts like a classical NOT gate, flipping  $|0\rangle$  to  $|1\rangle$  and vice versa.

### 5. Applying the Unitary Matrix U:

- The `qml.QubitUnitary(U, wires=0)` function is used to apply the unitary matrix U to the qubit.
- This step applies the Hadamard operation to the qubit, which will transform the qubit's state into a superposition if it was initially in  $|0\rangle$  or  $|1\rangle$ .

### 6. Returning the State:

- The function ends by returning the state of the qubit after all operations using `qml.state()`.
- `qml.state()` retrieves the current state vector of the qubit, which can be used to analyze or visualize the resulting quantum state.

## Learnings and Takeaways:

- **Control Flow with Quantum Operations:** The function demonstrates how classical control flow (using an if statement) can be combined with quantum operations to prepare a specific quantum state before applying further quantum gates.
- **Pauli Gates:** The PauliX gate is an essential operation for flipping the state of a qubit, showing how you can initialize a qubit in a desired basis state.
- **Unitary Operations:** Applying a unitary matrix U to a qubit transforms its state, which is fundamental to performing quantum computations. The Hadamard gate U is particularly important for creating superpositions, which are key to quantum parallelism.

- **State Measurement and Retrieval:** The use of `qml.state()` shows how to retrieve the final quantum state of a system, allowing you to understand the outcome of your quantum circuit.

This code encapsulates basic yet critical concepts in quantum computing, making it a good practice example for understanding quantum gates, state preparation, and quantum operations in PennyLane.

- Codercise I.5.1 — The Pauli Z gate

**Ques:**

The Pauli  $Z$  gate is defined by its action on the computational basis states

$$\begin{aligned} |0\rangle &\rightarrow |0\rangle, \\ |1\rangle &\rightarrow -|1\rangle. \end{aligned} \tag{1}$$

In PennyLane, you can implement it by calling `qml.PauliZ`. It has the following circuit element:



Write a QNode that applies `qml.PauliZ` to the  $|+\rangle$  state and returns the state. What state is this? How do the measurement probabilities differ from those of the state  $|+\rangle$ ?

**Theory:**

Both the  $X$  and the  $H$  gate affect which basis states appear in the superposition. We know how to flip the states using  $X$ , and how to create uniform superpositions. But, how can we change the amplitudes more generally? For example, how do we change relative phases?

Consider the case of the **Pauli Z** gate (denoted  $Z$ ), which has the following effect:

$$\begin{aligned} |0\rangle &\rightarrow |0\rangle, \\ |1\rangle &\rightarrow -|1\rangle. \end{aligned}$$

Applying  $Z$  "flips the phase" (i.e., multiplies the phase by  $-1$ ) of the  $|1\rangle$  state, while leaving the  $|0\rangle$  state intact. However

$$\begin{aligned} Z|+\rangle &= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) &= |-\rangle, \\ Z|-\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) &= |+\rangle. \end{aligned}$$

That is, the Pauli  $Z$  gate introduces a relative phase when applied to the  $|+\rangle$  and  $|-\rangle$ , resulting on it "flipping" these states into one another. This statement can be expressed mathematically through the equality

$$Z = HXH,$$



## Code:

```
1 dev = qml.device("default.qubit", wires=1)
2
3
4 @qml.qnode(dev)
5 def apply_z_to_plus():
6     """Write a circuit that applies PauliZ to the |+> state and returns
7     the state.
8
9     Returns:
10         np.array[complex]: The state of the qubit after the operations.
11     """
12
13     #####
14     # YOUR CODE HERE #
15     #####
16
17     # CREATE THE |+> STATE
18
19     # APPLY PAULI Z
20
21     # RETURN THE STATE
22     # CREATE THE |+> STATE
23     qml.Hadamard(wires=0)
24
25     # APPLY PAULI Z
26     qml.PauliZ(wires=0)
27
28     # RETURN THE STATE
29     return qml.state()
30
31
32 print(apply_z_to_plus())
33
```

[Reset Code](#)[Submit](#)

Correct!

## Learning:

Brief explanation of the `apply_z_to_plus` function:

### 1. Quantum Device Initialization:

- The quantum device `dev` is initialized using `qml.device("default.qubit", wires=1)`, which sets up a simulation environment with one qubit.

### 2. QNode Function:

- The function `apply_z_to_plus` is decorated with `@qml.qnode(dev)`, making it a QNode. QNodes in PennyLane are used to define quantum circuits that can be executed on quantum devices.

### 3. Creating the $|+\rangle$ State:

- The Hadamard gate (`qml.Hadamard(wires=0)`) is applied to the qubit. The Hadamard gate transforms the standard basis state  $|0\rangle$  into the  $|+\rangle$  state.
- The  $|+\rangle$  state is defined as:  $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$
- This state is a superposition of  $|0\rangle$  and  $|1\rangle$ .

#### 4. Applying the Pauli-Z Gate:

- The Pauli-Z gate (`qml.PauliZ(wires=0)`) is applied to the qubit. The Pauli-Z gate flips the phase of the  $|1\rangle$  component of the qubit state, leaving  $|0\rangle$  component unchanged.
- The Pauli-Z gate is represented by the matrix:  $Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
- When applied to the  $|+\rangle$  state, the Pauli-Z gate transforms it into the  $|-\rangle$  state:  $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$

#### 5. Returning the Final State:

- The function returns the state of the qubit after applying the Pauli-Z gate using `return qml.state()`.
- `qml.state()` retrieves the current state vector of the qubit, which reflects the result of the quantum operations performed.

#### Summary:

- **Hadamard Gate:** The Hadamard gate creates the  $|+\rangle$  superposition state from the initial  $|0\rangle$  state.
- **Pauli-Z Gate:** The Pauli-Z gate changes the phase of the  $|1\rangle$  component in the superposition, transforming  $|+\rangle$  into  $|-\rangle$ .
- **Final State:** After the operations, the qubit is in the  $|-\rangle$  state, which is returned by the function.

This function demonstrates how to create a specific quantum state ( $|+\rangle$ ) and apply a phase-flipping operation (Pauli-Z), which is a fundamental concept in quantum computing.

- Codercise— Separable Operations

### Ques:

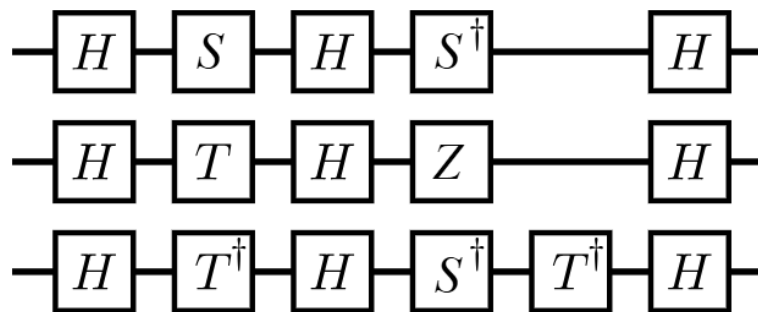
Use PennyLane to create the state  $|+1\rangle = |+\rangle \otimes |1\rangle$ . Then, return two measurements:

- the expectation value of  $Y$  on the first qubit
- the expectation value of  $Z$  on the second qubit

In PennyLane, you can return measurements of multiple observables as a tuple, as long as they don't share wires.

### Theory:

Operations on multiple qubits can take one of two forms. The first type looks essentially like performing single-qubit operations on individual qubits, but in a multi-qubit system, multiple single-qubit operations can be applied in parallel. We'll call these *separable* operations because they can be expressed simply as tensor products of individual qubit operations. For example,



You can imagine each "layer" of this circuit as a single multi-qubit operation. For example, the first step in the circuit is to apply a Hadamard to each qubit. This corresponds to applying  $H \otimes H \otimes H$ . The second layer applies  $S \otimes T \otimes T^\dagger$ , and so on. While circuits are read from left to right, the matrices are applied to the states from right to left, i.e., the result of applying the first two layers of gates to an input state  $|\psi\rangle$  would be  $(S \otimes T \otimes T^\dagger)(H \otimes H \otimes H)|\psi\rangle$ .

### Code:

```

1 # Creates a device with *two* qubits
2 dev = qml.device("default.qubit", wires=2)
3
4
5 @qml.qnode(dev)
6 def two_qubit_circuit():
7     #####
8     # YOUR CODE HERE #
9     #####
10
11     # PREPARE |+>|1>
12
13     # RETURN TWO EXPECTATION VALUES, Y ON FIRST QUBIT, Z ON SECOND QUBIT
14
15     # Prepare |+> on the first qubit (superposition state)
16     qml.Hadamard(wires=0)
17
18     # Prepare |1> on the second qubit
19     qml.PauliX(wires=1)
20
21     # Return two expectation values: Y on the first qubit, Z on the second qubit
22     return qml.expval(qml.PauliY(0)), qml.expval(qml.PauliZ(1))
23
24
25 print(two_qubit_circuit())
26

```

[Reset Code](#)

Submit

Correct!

## Learning:

This code defines a quantum circuit that operates on two qubits. It aims to prepare specific quantum states and then measure the expectation values of Pauli Y operator on the first qubit and Pauli Z operator on the second qubit.

Here's a breakdown of the code:

### 1. Initialization:

- **dev = qml.device("default.qubit", wires=2):** This line creates a quantum device with two qubits, using the 'default.qubit' simulator.

### 2. Circuit Definition:

- **@qml.qnode(dev):** This decorator indicates that the following function **two\_qubit\_circuit()** defines a quantum circuit that will be executed on the previously defined device.

### 3. Preparation:

- **qml.Hadamard(wires=0):** This applies a Hadamard gate to the first qubit (wire 0), creating a superposition state ( $|+\rangle$ ).
- **qml.PauliX(wires=1):** This applies a Pauli X gate to the second qubit (wire 1), flipping it to the  $|1\rangle$  state.

### 4. Measurement:

- **return qml.expval(qml.PauliY()), qml.expval(qml.PauliZ(1)):** This line returns the expectation values of Pauli Y on the first qubit and Pauli Z on the second qubit.

### 5. Execution:

- **print(two\_qubit\_circuit()):** This line executes the defined quantum circuit and prints the output, which will be the calculated expectation values.

This code exemplifies a basic example of preparing quantum states and measuring specific observables in a quantum circuit. It provides a foundation for understanding how to implement more complex quantum algorithms and experiments

- Codercise— The Bell states

## Ques:

Consider again the entangled state that we saw earlier,

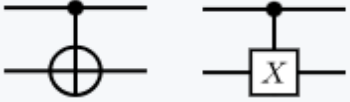
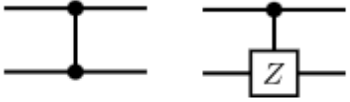
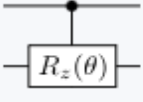
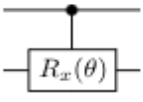
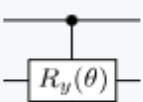
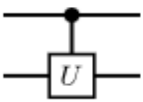
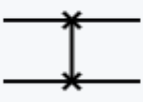
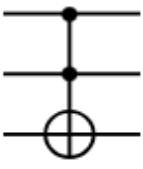
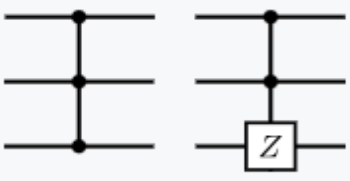
$$|\psi_+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).$$

This state is called a **Bell state**, and it has 3 siblings:

$$|\psi_-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle), \quad |\phi_+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle), \quad |\phi_-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle).$$

Together, these states form the **Bell basis**. Write a set of 4 circuits that prepare and return each of the four Bell states.

# Theory:

Gate	Matrix	Circuit element(s)	Basis state action
$CNOT$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$		$CNOT 00\rangle =  00\rangle$ $CNOT 01\rangle =  01\rangle$ $CNOT 10\rangle =  11\rangle$ $CNOT 11\rangle =  10\rangle$
$CZ$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$		$CZ 00\rangle =  00\rangle$ $CZ 01\rangle =  01\rangle$ $CZ 10\rangle =  10\rangle$ $CZ 11\rangle = - 11\rangle$
$CRZ$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & e^{-i\frac{\theta}{2}} & 0 \\ 0 & 0 & 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$		$CRZ 00\rangle =  00\rangle$ $CRZ 01\rangle =  01\rangle$ $CRZ 10\rangle = e^{-i\frac{\theta}{2}} 10\rangle$ $CRZ 11\rangle = e^{i\frac{\theta}{2}} 11\rangle$
$CRX$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos(\frac{\theta}{2}) & -i\sin(\frac{\theta}{2}) \\ 0 & 0 & -i\sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{pmatrix}$		$CRX(\theta) 00\rangle =  00\rangle$ $CRX(\theta) 01\rangle =  01\rangle$ $CRX(\theta) 10\rangle = \cos\frac{\theta}{2} 10\rangle - i\sin\frac{\theta}{2} 11\rangle$ $CRX(\theta) 11\rangle = -i\sin\frac{\theta}{2} 10\rangle + \cos\frac{\theta}{2} 11\rangle$
$CRY$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos(\frac{\theta}{2}) & -\sin(\frac{\theta}{2}) \\ 0 & 0 & \sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{pmatrix}$		$CRY(\theta) 00\rangle =  00\rangle$ $CRY(\theta) 01\rangle =  01\rangle$ $CRY(\theta) 10\rangle = \cos\frac{\theta}{2} 10\rangle + \sin\frac{\theta}{2} 11\rangle$ $CRY(\theta) 11\rangle = -\sin\frac{\theta}{2} 10\rangle + \cos\frac{\theta}{2} 11\rangle$
$CU$	$\begin{pmatrix} I_2 & 0 \\ 0 & U \end{pmatrix}$		$CU 00\rangle =  00\rangle$ $CU 01\rangle =  01\rangle$ $CU 10\rangle =  1\rangle \otimes U 0\rangle$ $CU 11\rangle =  1\rangle \otimes U 1\rangle$
$SWAP$	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$		$SWAP 00\rangle =  00\rangle$ $SWAP 01\rangle =  10\rangle$ $SWAP 10\rangle =  01\rangle$ $SWAP 11\rangle =  11\rangle$
$TOF$	$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$		$TOF 000\rangle =  000\rangle$ $TOF 001\rangle =  001\rangle$ $\vdots = \vdots$ $TOF 101\rangle =  101\rangle$ $TOF 110\rangle =  111\rangle$ $TOF 111\rangle =  110\rangle$
$CCZ$	$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}$		$CCZ 000\rangle =  000\rangle$ $\vdots = \vdots$ $CCZ 110\rangle =  110\rangle$ $CCZ 111\rangle = - 111\rangle$

## Code:

```
1 dev = qml.device("default.qubit", wires=2)
2
3 # Starting from the state  $|00\rangle$ , implement a PennyLane circuit
4 # to construct each of the Bell basis states.
5
6
7 @qml.qnode(dev)
8 def prepare_psi_plus():
9     #####
10    # YOUR CODE HERE #
11    #####
12
13    # PREPARE  $(1/\sqrt{2})(|00\rangle + |11\rangle)$ 
14
15    # Apply a Hadamard gate on the first qubit
16    qml.Hadamard(wires=0)
17    # Apply a CNOT gate (controlled X gate)
18    qml.CNOT(wires=[0, 1])
19
20    return qml.state()
21
22
23
24 @qml.qnode(dev)
25 def prepare_psi_minus():
26     #####
27    # YOUR CODE HERE #
28    #####
29
30    # PREPARE  $(1/\sqrt{2})(|00\rangle - |11\rangle)$ 
31
32    # Apply a Hadamard gate on the first qubit
33    qml.Hadamard(wires=0)
34    # Apply a CNOT gate (controlled X gate)
35    qml.CNOT(wires=[0, 1])
36    # Apply a Pauli-Z gate on the second qubit
37    qml.PauliZ(wires=1)
38
39    return qml.state()
40
```

```
41 @qml.qnode(dev)
42 def prepare_phi_plus():
43     #####
44    # YOUR CODE HERE #
45    #####
46
47    # PREPARE  $(1/\sqrt{2})(|01\rangle + |10\rangle)$ 
48
49    # Apply a Hadamard gate on the first qubit
50    qml.Hadamard(wires=0)
51    # Apply a CNOT gate (controlled X gate)
52    qml.CNOT(wires=[0, 1])
53    # Apply a Pauli-X gate on the second qubit
54    qml.PauliX(wires=1)
55
56    return qml.state()
57
58
59 @qml.qnode(dev)
60 def prepare_phi_minus():
61     #####
62    # YOUR CODE HERE #
63    #####
64
65    # PREPARE  $(1/\sqrt{2})(|01\rangle - |10\rangle)$ 
66
67    # Apply a Hadamard gate on the first qubit
68    qml.Hadamard(wires=0)
69    # Apply a CNOT gate (controlled X gate)
70    qml.CNOT(wires=[0, 1])
71    # Apply a Pauli-Z gate on the first qubit
72    qml.PauliZ(wires=0)
73    # Apply a Pauli-X gate on the second qubit
74    qml.PauliX(wires=1)
75
76    return qml.state()
77
78 psi_plus = prepare_psi_plus()
79 psi_minus = prepare_psi_minus()
80 phi_plus = prepare_phi_plus()
81 phi_minus = prepare_phi_minus()
82
83 # Uncomment to print results
84 # print(f" $|\psi_+\rangle = \{psi\_plus\}$ ")
85 # print(f" $|\psi_-\rangle = \{psi\_minus\}$ ")
86 # print(f" $|\phi_+\rangle = \{phi\_plus\}$ ")
87 # print(f" $|\phi_-\rangle = \{phi\_minus\}$ ")
88
```

[Reset Code](#)

Submit

Correct!

## **Learning:**

This code is a quantum computing exercise designed to teach how to prepare the four Bell states using the PennyLane quantum machine learning framework. The Bell states are an important concept in quantum mechanics, representing the simplest examples of entanglement, which is a phenomenon where qubits become correlated in such a way that the state of one qubit is directly related to the state of another, regardless of the distance between them.

## **Learning Objectives**

### **1. Understanding Quantum Gates:**

- The code involves using basic quantum gates like the Hadamard gate (H), Controlled NOT gate (CNOT), Pauli-X gate (X), and Pauli-Z gate (Z).
- The Hadamard gate is used to create superposition, and the CNOT gate is used to entangle the qubits. The Pauli-X and Pauli-Z gates are used to modify the phase or flip the qubits.

### **2. Entanglement and Bell States:**

- The code demonstrates how to create each of the four Bell states, which are maximally entangled states of two qubits.
- Understanding how to prepare these states is crucial for learning about entanglement, quantum communication, and quantum cryptography.

### **3. Quantum Circuits and QNodes:**

- The code shows how to define quantum circuits using PennyLane's `@qml.qnode` decorator. Each function represents a quantum circuit that prepares a specific Bell state.
- Students learn how to construct quantum circuits and retrieve the resulting quantum states.