Codercise I.1.1 — Normalization of quantum states

Open related theory

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 α and β, it normalizes this state to

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

Solution :

```
# Here are the vector representations of |0> and |1>, for convenience
ket_0 = np.array([1, 0])
ket_1 = np.array([0, 1])


def normalize_state(alpha, beta):
    """Compute a normalized quantum state given arbitrary amplitudes.

    Args:
        alpha (complex): The amplitude associated with the |0> state.
        beta (complex): The amplitude associated with the |1> state.

    Returns:
        np.array[complex]: A vector (numpy array) with 2 elements that represents
        a normalized quantum state.
    """

    ##################
    vectorstate = np.array([alpha,beta])
    norm = np.linalg.norm(vectorstate)
    if norm == 0:
      return v
    vector = vectorstate / norm
    ##################

    # CREATE A VECTOR [a', b'] BASED ON alpha AND beta SUCH THAT |a'|^2 + |b'|^2 = 1

    # RETURN A VECTOR
    return vector
```

Qiskit Program:

```
import numpy as np
import random
from qiskit.quantum_info import Statevector

ket_0 = np.array([1, 0])
ket_1 = np.array([0, 1])
```

```
def generaterandomcomplexnumber ():
    # Generate the real part
    real_part = random.uniform(1, -1)
    # Generate the imaginary part
    imag_part = random.uniform(1, -1)
    # Form the complex number
    complex_number = complex(real_part, imag_part)
    # Print the complex number
    print(complex_number)
    return complex_number

def normalize(v):
    norm = np.linalg.norm(v)
    if norm == 0:
       return v
    return v / norm

def return_normalized_vector(alpha , beta):
   vectorstate = np.array([alpha,beta])
   norm = np.linalg.norm(vectorstate)
   if norm == 0:
      return vectorstate
   print("vectorstate :",vectorstate)
   return ( vectorstate / norm )

alpha = generaterandomcomplexnumber()
beta = generaterandomcomplexnumber()
vector = return_normalized_vector(alpha,beta)
Statevector(vector).draw('latex')
```

O/P:

```
(-0.525059845510538+0.5196887910606383j)
(-0.0031775003541036906+0.11399858788785244j)
vectorstate : [-0.52505985+0.51968879j -0.0031775 +0.11399859j]
```

$$(-0.7024125206 + 0.6952272523i)|0\rangle + (-0.0042507841 + 0.1525045881i)|1\rangle$$

Codercise I.1.2 — Inner product and orthonormal bases

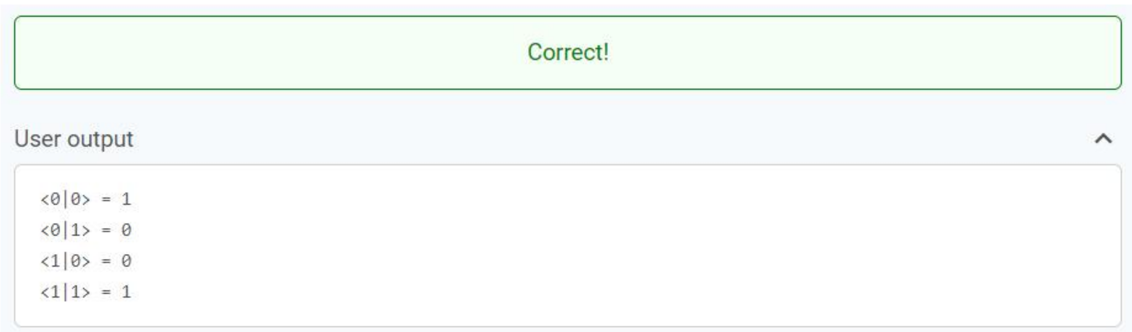
Open related theory

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.

Solution:

```
def inner_product(state_1, state_2):
    """Compute the inner product between two states.
```

```
    Args:
      state_1 (np.array[complex]): A normalized quantum state vector
      state_2 (np.array[complex]): A second normalized quantum state vector

    Returns:
      complex: The value of the inner product <state_1 | state_2>.
    """

    ##################
    inner_product_value = np.dot(np.conj(state_1), state_2)
    ##################

    # COMPUTE AND RETURN THE INNER PRODUCT

    return inner_product_value


# Test your results with this code
ket_0 = np.array([1, 0])
ket_1 = np.array([0, 1])

print(f"<0|0> = {inner_product(ket_0, ket_0)}")
print(f"<0|1> = {inner_product(ket_0, ket_1)}")
print(f"<1|0> = {inner_product(ket_1, ket_0)}")
print(f"<1|1> = {inner_product(ket_1, ket_1)}")
```



Qiskit Program:

```
import numpy as np
import random
from qiskit.quantum_info import Statevector

def inner_product(state_1, state_2):
    """Compute the inner product between two states.

    Args:
      state_1 (np.array[complex]): A normalized quantum state vector
      state_2 (np.array[complex]): A second normalized quantum state vector

    Returns:
      complex: The value of the inner product <state_1 | state_2>.
    """

    ##################
    inner_product_value = np.dot(np.conj(state_1), state_2)
```

```
    ##################

    # COMPUTE AND RETURN THE INNER PRODUCT

    return inner_product_value


# Test your results with this code
ket_0 = np.array([1, 0])
ket_1 = np.array([0, 1])

print(f"<0|0> = {inner_product(ket_0, ket_0)}")
print(f"<0|1> = {inner_product(ket_0, ket_1)}")
print(f"<1|0> = {inner_product(ket_1, ket_0)}")
print(f"<1|1> = {inner_product(ket_1, ket_1)}")
```

O/P:

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

Codercise I.1.3 — Sampling measurement outcomes

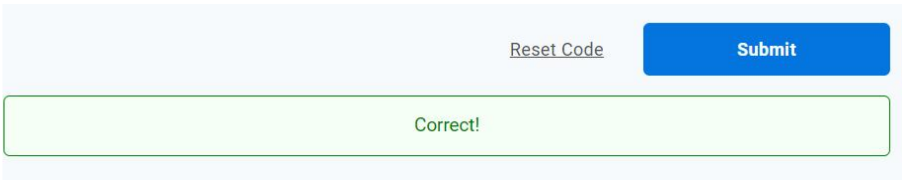
Open related theory

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.

Solution:

```
def measure_state(state, num_meas):
    """Simulate a quantum measurement process.

    Args:
        state (np.array[complex]): A normalized qubit state vector.
        num_meas (int): The number of measurements to take

    Returns:
        np.array[int]: A set of num_meas samples, 0 or 1, chosen according to the probability
        distribution defined by the input state.
    """
    ##################
    # Calculate the probability for each basis state
    prob_ket_0 = np.abs(state[0])**2
    prob_ket_1 = np.abs(state[1])**2

    # Ensure the probabilities are normalized
    total_probability = prob_ket_0 + prob_ket_1
    prob_ket_0 /= total_probability
    prob_ket_1 /= total_probability
```

```
    # Print the probabilities
    print("Probability of ket 0:", prob_ket_0)
    print("Probability of ket 1:", prob_ket_1)

    # Generate measurement outcomes based on the probabilities
    outcomes = np.random.choice([0, 1], size=num_meas, p=[prob_ket_0, prob_ket_1])
    ##################

    # COMPUTE THE MEASUREMENT OUTCOME PROBABILITIES

    return outcomes
```



Qiskit Program:

```
import numpy as np
import random
from qiskit.quantum_info import Statevector

def measure_state(state, num_meas):
    """Simulate a quantum measurement process.

    Args:
        state (np.array[complex]): A normalized qubit state vector.
        num_meas (int): The number of measurements to take.

    Returns:
        np.array[int]: A set of num_meas samples, 0 or 1, chosen according to the probability
        distribution defined by the input state.
    """
    # Calculate the probability for each basis state
    prob_ket_0 = np.abs(state[0])**2
    prob_ket_1 = np.abs(state[1])**2

    # Ensure the probabilities are normalized
    total_probability = prob_ket_0 + prob_ket_1
    prob_ket_0 /= total_probability
    prob_ket_1 /= total_probability

    # Print the probabilities
    print("Probability of ket 0:", prob_ket_0)
    print("Probability of ket 1:", prob_ket_1)

    # Generate measurement outcomes based on the probabilities
    outcomes = np.random.choice([0, 1], size=num_meas, p=[prob_ket_0, prob_ket_1])

    return outcomes

def generaterandomcomplexnumber():
    real_part = random.uniform(-1, 1)
    imag_part = random.uniform(-1, 1)
```

```
    complex_number = complex(real_part, imag_part)
    print(complex_number)
    return complex_number

def return_normalized_vector(alpha, beta):
    vectorstate = np.array([alpha, beta])
    norm = np.linalg.norm(vectorstate)
    if norm == 0:
        return vectorstate
    print("vectorstate:", vectorstate)
    return vectorstate / norm

# Example usage
alpha = generaterandomcomplexnumber()
beta = generaterandomcomplexnumber()
vector = return_normalized_vector(alpha, beta)
Statevector(vector).draw('latex')

num_meas = 15  # Number of measurements
measurement_results = measure_state(vector, num_meas)
print("Measurement results:", measurement_results)
```

O/P:

```
(-0.26990776715789444-0.5630568559122677j)
(-0.9060243665925027+0.42884004558690636j)
vectorstate: [-0.26990777-0.56305686j -0.90602437+0.42884005j]
Probability of ket 0: 0.2795528825915502
Probability of ket 1: 0.7204471174084497
Measurement results: [0 1 1 0 1 1 1 1 1 1 1 1 0 0 1]
```

Codercise I.1.4 — Applying a quantum operation

Open related theory

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 2×2 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`.

Solution:

```
U = np.array([[1, 1], [1, -1]]) / np.sqrt(2)

def apply_u(state):
    """Apply a quantum operation.
```

```
    Args:
        state (np.array[complex]): A normalized quantum state vector.

    Returns:
        np.array[complex]: The output state after applying U.
    """

    ##################
    quantum_state = np.array(state)
    result = np.dot(U,quantum_state)
    ##################

    # APPLY U TO THE INPUT STATE AND RETURN THE NEW STATE
    return result
```

```
U = np.array([[1, 1], [1, -1]]) / np.sqrt(2)


def apply_u(state):
    """Apply a quantum operation.

    Args:
        state (np.array[complex]): A normalized quantum state vector.

    Returns:
        np.array[complex]: The output state after applying U.
    """

    ##################
    quantum_state = np.array(state)
    result = np.dot(U,quantum_state)
    ##################

    # APPLY U TO THE INPUT STATE AND RETURN THE NEW STATE
    return result

```

Reset Code   **Submit**

Correct!

Qiskit Program:

```
import numpy as np
import random
from qiskit.quantum_info import Statevector

U = np.array([[1, 1], [1, -1]]) / np.sqrt(2)

def apply_u(state):
    """Apply a quantum operation.

    Args:
        state (np.array[complex]): A normalized quantum state vector.

    Returns:
        np.array[complex]: The output state after applying U.
    """

    ##################
```

```
    quantum_state = np.array(state)
    result = np.dot(U,quantum_state)
    ##################

    # APPLY U TO THE INPUT STATE AND RETURN THE NEW STATE
    return result

# Example usage
state_as_input = np.array([0.8, 0.6])
Statevector(state_as_input).draw('latex')
```

$$\frac{4}{5}|0\rangle + \frac{3}{5}|1\rangle$$

```
output_state = apply_u(state_as_input)
Statevector(output_state).draw('latex')
```

$$0.9899494937|0\rangle + 0.1414213562|1\rangle$$

Codercise I.1.5 — A simple quantum algorithm

Open related theory

You may not have realized it, but you now have all the ingredients to write a very simple **quantum simulator** that can simulate the outcome of running quantum algorithms on a single qubit! Let's put everything together.

Use the functions below to simulate a quantum algorithm that does the following:

1. Initialize a qubit in state $|0\rangle$
2. Apply the provided operation U
3. Simulate measuring the output state 100 times

You'll have to complete a function for initialization, but we've provided functions for the other two.

Solution:

```
U = np.array([[1, 1], [1, -1]]) / np.sqrt(2)


def initialize_state():
    """Prepare a qubit in state |0>.

    Returns:
        np.array[float]: the vector representation of state |0>.
    """

    ##################
```

```
    state_as_input = np.array([1, 0])
    ##################

    return state_as_input

def apply_u(state):
    """Apply a quantum operation."""
    return np.dot(U, state)


def measure_state(state, num_meas):
    """Measure a quantum state num_meas times."""
    p_alpha = np.abs(state[0]) ** 2
    p_beta = np.abs(state[1]) ** 2
    meas_outcome = np.random.choice([0, 1], p=[p_alpha, p_beta], size=num_meas)
    return meas_outcome


def quantum_algorithm():
    """Use the functions above to implement the quantum algorithm described above.

    Try and do so using three lines of code or less!

    Returns:
        np.array[int]: the measurement results after running the algorithm 100 times
    """

    ##################
    state_init_u = apply_u(initialize_state())
    outcomes = measure_state(state_init_u,100)
    ##################

    # PREPARE THE STATE, APPLY U, THEN TAKE 100 MEASUREMENT SAMPLES
    return outcomes
```

```python


def measure_state(state, num_meas):
    """Measure a quantum state num_meas times."""
    p_alpha = np.abs(state[0]) ** 2
    p_beta = np.abs(state[1]) ** 2
    meas_outcome = np.random.choice([0, 1], p=[p_alpha, p_beta], size=num_meas)
    return meas_outcome


def quantum_algorithm():
    """Use the functions above to implement the quantum algorithm described above.

    Try and do so using three lines of code or less!

    Returns:
        np.array[int]: the measurement results after running the algorithm 100 times
    """

    ##################
    state_init_u = apply_u(initialize_state())
    outcomes = measure_state(state_init_u,100)
    ##################

    # PREPARE THE STATE, APPLY U, THEN TAKE 100 MEASUREMENT SAMPLES
    return outcomes

```

Reset Code

Submit

Correct!

Qiskit Program:

```python
import numpy as np
import random
from qiskit.quantum_info import Statevector

U = np.array([[1, 1], [1, -1]]) / np.sqrt(2)

def initialize_state():
    """Prepare a qubit in state |0>.

    Returns:
        np.array[float]: the vector representation of state |0>.
    """

    ##################
    state_as_input = np.array([1, 0])
    ##################

    # PREPARE THE STATE |0>
    Statevector(state_as_input).draw('latex')
    return state_as_input

def apply_u(state):
    """Apply a quantum operation.

    Args:
        state (np.array[complex]): A normalized quantum state vector.
```

```
    Returns:
      np.array[complex]: The output state after applying U.
    """

    ##################
    quantum_state = np.array(state)
    result = np.dot(U,quantum_state)
    ##################

    # APPLY U TO THE INPUT STATE AND RETURN THE NEW STATE
    return result

def measure_state(state, num_meas):
    """Measure a quantum state num_meas times."""
    p_alpha = np.abs(state[0]) ** 2
    p_beta = np.abs(state[1]) ** 2
    meas_outcome = np.random.choice([0, 1], p=[p_alpha, p_beta], size=num_meas)
    return meas_outcome

def quantum_algorithm():
    """Use the functions above to implement the quantum algorithm described above.

    Try and do so using three lines of code or less!

    Returns:
      np.array[int]: the measurement results after running the algorithm 100 times
    """

    ##################
    state_init_u = apply_u(initialize_state())
    outcomes = measure_state(state_init_u,100)
    ##################

    # PREPARE THE STATE, APPLY U, THEN TAKE 100 MEASUREMENT SAMPLES
    return outcomes
```

```
stateinit = initialize_state()
print("stateinit :",stateinit)
```

```
stateinit : [1 0]
```

```
measurement_results = quantum_algorithm()
print("Measurement results:", measurement_results)
```

```
Measurement results: [0 0 1 1 0 0 0 1 0 0 1 0 1 1 0 0 1 0 1 0 1 0 0 1 0 1 1 0 1 1 0 0 1 1 1 1 1
 1 0 1 0 0 0 0 0 1 1 0 1 1 1 0 1 1 1 1 1 0 0 1 1 1 1 0 1 0 1 0 1 0 0 1 1 1
 0 0 0 0 0 1 1 0 0 0 0 1 0 1 1 1 0 1 0 0 0 0 0 1 0 1]
```