|  |
| --- |
|  |
| **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:** |

|  |
| --- |
|  |
| **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:** |

|  |
| --- |
|  |
| **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**: |

|  |
| --- |
|  |
| **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 |
| **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')    output\_state = apply\_u(state\_as\_input)  Statevector(output\_state).draw('latex') |

|  |
| --- |
|  |
| **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 |
| **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 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 |