# **Question 1 – PennyLane Tutorials**

All exercises from PennyLane modules “Introduction to Quantum Computing”, “Single-Qubit Gates”, and “Circuits with Many Qubits” were done correctly. In this report, all the correct code for the exercises is presented.

# TABLE OF CONTENTS

# Introduction to Quantum Computing

### All About Qubits

### Quantum Circuits

### Unitary Matrices

# Single-Qubit Gates

### X and H

* It’s Just a Phase
* From a Different Angle
* Universal Gate Sets
* Prepare Yourself
* Measurements
* What Did You Expect?

1. Circuits with Many Qubits

* Multi-Qubit Systems
* All Tied Up
* We’ve Got It Under Control
* Multi-Qubit Gate Challenge

RESULTS

1. Introduction to Quantum Computing:

|  |
| --- |
| **All About Qubits** |
| Codercise I.1.1 – Normalization of quantum states |
| # 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.  """  denominator = np.abs(alpha)\*\*2 + np.abs(beta)\*\*2  k = np.sqrt(1/denominator)  a = k \* alpha  b = k \* beta  result = np.array([a, b])  return result |
| Codercise I.1.2 – Inner product and orthonormal bases |
| 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\_prod = np.vdot(state\_1, state\_2)  return inner\_prod  # 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)}") |
| Codercise I.1.3 – Sampling measurement outcomes |
| 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.  """  outcomes = [0, 1]  probabilities = [np.abs(state[0])\*\*2, np.abs(state[1])\*\*2]  measurements = np.random.choice(outcomes, size=num\_meas, p=probabilities)  return measurements |
| Codercise I.1.4 – Applying a quantum operation |
| 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.  """  result = np.dot(U, state)  return result |
| Codercise I.1.5 – A simple quantum algorithm |
| 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>.  """  return np.array([1, 0])  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 = initialize\_state()  apply\_u(state)  result = measure\_state(state, 100)  return result |

|  |
| --- |
| **Quantum Circuits** |
| Codercise I.2.1 – Order of operations |
| def my\_circuit(theta, phi):  qml.CNOT(wires=[0, 1])  qml.RX(theta, wires=2)  qml.Hadamard(wires=0)  qml.CNOT(wires=[2, 0])  qml.RY(phi, wires=1)  return qml.probs(wires=[0, 1, 2]) |
| Codercise I.2.2 – Building a QNode |
| # This creates a device with three wires on which PennyLane can run computations  dev = qml.device("default.qubit", wires=3)  def my\_circuit(theta, phi, omega):  qml.RX(theta, wires=0)  qml.RY(phi, wires=1)  qml.RZ(omega, wires=2)  qml.CNOT(wires=[0,1])  qml.CNOT(wires=[1,2])  qml.CNOT(wires=[2,0])  return qml.probs(wires=[0, 1, 2])  # This creates a QNode, binding the function and device  my\_qnode = qml.QNode(my\_circuit, dev)  # We set up some values for the input parameters  theta, phi, omega = 0.1, 0.2, 0.3  # Now we can execute the QNode by calling it like we would a regular function  my\_qnode(theta, phi, omega) |
| Codercise I.2.3 – The QNode decorator |
| dev = qml.device("default.qubit", wires=3)  @qml.qnode(dev)  def my\_circuit(theta, phi, omega):  qml.RX(theta, wires=0)  qml.RY(phi, wires=1)  qml.RZ(omega, wires=2)  qml.CNOT(wires=[0, 1])  qml.CNOT(wires=[1, 2])  qml.CNOT(wires=[2, 0])  return qml.probs(wires=[0, 1, 2])  theta, phi, omega = 0.1, 0.2, 0.3  my\_circuit(theta, phi, omega) |
| Codercise I.2.4 – Circuit depth |
| dev = qml.device("default.qubit", wires=3)  @qml.qnode(dev)  def my\_circuit(theta, phi, omega):  qml.RX(theta, wires=0)  qml.RY(phi, wires=1)  qml.RZ(omega, wires=2)  qml.CNOT(wires=[0, 1])  qml.CNOT(wires=[1, 2])  qml.CNOT(wires=[2, 0])  return qml.probs(wires=[0, 1, 2])  # Execute the circuit to generate the tape  theta, phi, omega = 0.1, 0.2, 0.3  my\_circuit(theta, phi, omega)  tape = my\_circuit.qtape  # Function to compute the depth of the circuit  def compute\_depth(tape):  depth = {wire: 0 for wire in tape.wires}  for op in tape.operations:  max\_depth = max(depth[wire] for wire in op.wires)  for wire in op.wires:  depth[wire] = max\_depth + 1  return max(depth.values())  # Calculate the depth of the circuit  depth = compute\_depth(tape) |

|  |
| --- |
| **Unitary Matrices** |
| Codercise I.3.1 – Unitaries in PennyLane |
| dev = qml.device("default.qubit", wires=1)  U = np.array([[1, 1], [1, -1]]) / np.sqrt(2)  @qml.qnode(dev)  def apply\_u():  qml.QubitUnitary(U, wires=0)    return qml.state() |
| Codercise I.3.2 – Parametrized unitaries |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def apply\_u\_as\_rot(phi, theta, omega):  qml.Rot(phi, theta, omega, wires=0)  return qml.state() |

1. Single-Qubit Gates

|  |
| --- |
| **X and H** |
| Codercise I.4.1 – Flipping bits |
| dev = qml.device("default.qubit", wires=1)  U = np.array([[1, 1], [1, -1]]) / np.sqrt(2)  @qml.qnode(dev)  def varied\_initial\_state(state):  """Complete the function such that we can apply the operation U to  either |0> or |1> depending on the input argument flag.  Args:  state (int): Either 0 or 1. If 1, prepare the qubit in state |1>,  otherwise, leave it in state 0.  Returns:  np.array[complex]: The state of the qubit after the operations.  """  if state == 1:  qml.PauliX(wires=0)  qml.QubitUnitary(U, wires=0)    return qml.state() |
| Codercise I.4.2 – Uniform superposition |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def apply\_hadamard():  qml.Hadamard(wires=0)    return qml.state() |
| Codercise I.4.3 – Combining X and H |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def apply\_hadamard\_to\_state(state):  """Complete the function such that we can apply the Hadamard to  either |0> or |1> depending on the input argument flag.  Args:  state (int): Either 0 or 1. If 1, prepare the qubit in state |1>,  otherwise, leave it in state 0.  Returns:  np.array[complex]: The state of the qubit after the operations.  """  if state ==1:  qml.PauliX(wires=0)  qml.Hadamard(wires=0)  return qml.state()  print(apply\_hadamard\_to\_state(0))  print(apply\_hadamard\_to\_state(1)) |
| Codercise I.4.4 – A QNode with X and H |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def apply\_hxh(state):  if state == 1:  qml.PauliX(wires=0)  qml.Hadamard(wires=0)  qml.PauliX(wires=0)  qml.Hadamard(wires=0)    return qml.state()    print(apply\_hxh(0))  print(apply\_hxh(1)) |

|  |
| --- |
| **It’s Just a Phase** |
| Codercise I.5.1 – The Pauli Z gate |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def apply\_z\_to\_plus():  """Write a circuit that applies PauliZ to the |+> state and returns  the state.  Returns:  np.array[complex]: The state of the qubit after the operations.  """  qml.Hadamard(wires=0)  qml.PauliZ(wires=0)    return qml.state()  print(apply\_z\_to\_plus()) |
| Codercise I.5.2 – The Z Rotation |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def fake\_z():  """Use RZ to produce the same action as Pauli Z on the |+> state.  Returns:  np.array[complex]: The state of the qubit after the operations.  """  qml.Hadamard(wires=0)  phi = np.pi  qml.RZ(phi, wires=0)    return qml.state() |
| Codercise I.5.3 – The S and T gates |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def many\_rotations():  """Implement the circuit depicted above and return the quantum state.  Returns:  np.array[complex]: The state of the qubit after the operations.  """  qml.Hadamard(wires=0)  qml.S(wires=0)  qml.adjoint(qml.T)(wires=0)  qml.RZ(0.3, wires=0)  qml.adjoint(qml.S)(wires=0)  return qml.state() |

|  |
| --- |
| **From a Different Angle** |
| Codercise I.6.1 – Applying RX |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def apply\_rx\_pi(state):  """Apply an RX gate with an angle of \pi to a particular basis state.  Args:  state (int): Either 0 or 1. If 1, initialize the qubit to state |1>  before applying other operations.  Returns:  np.array[complex]: The state of the qubit after the operations.  """  if state == 1:  qml.PauliX(wires=0)  qml.RX(np.pi, wires=0)  return qml.state()  print(apply\_rx\_pi(0))  print(apply\_rx\_pi(1)) |
| Codercise I.6.2 – Plotting RX |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def apply\_rx(theta, state):  """Apply an RX gate with an angle of theta to a particular basis state.  Args:  theta (float): A rotation angle.  state (int): Either 0 or 1. If 1, initialize the qubit to state |1>  before applying other operations.  Returns:  np.array[complex]: The state of the qubit after the operations.  """  if state == 1:  qml.PauliX(wires=0)  qml.RX(theta, wires=0)  return qml.state()  # 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) |
| Codercise I.6.3 – Plotting RY |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def apply\_ry(theta, state):  """Apply an RY gate with an angle of theta to a particular basis state.  Args:  theta (float): A rotation angle.  state (int): Either 0 or 1. If 1, initialize the qubit to state |1>  before applying other operations.  Returns:  np.array[complex]: The state of the qubit after the operations.  """  if state == 1:  qml.PauliX(wires=0)  qml.RY(theta, wires=0)  return qml.state()  # Code for plotting  angles = np.linspace(0, 4 \* np.pi, 200)  output\_states = np.array([apply\_ry(t, 0) for t in angles])  plot = plotter(angles, output\_states) |

|  |
| --- |
| **Universal Gate Sets** |
| Codercise I.7.1 – Universality of rotations |
| dev = qml.device("default.qubit", wires=1)  phi = np.pi/2  theta = np.pi/2  omega = np.pi/2  @qml.qnode(dev)  def hadamard\_with\_rz\_rx():  qml.RZ(phi, wires=0)  qml.RX(theta, wires=0)  qml.RZ(omega, wires=0)  return qml.state() |
| Codercise I.7.2 – Synthesizing a circuit |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def convert\_to\_rz\_rx():  qml.RZ(np.pi/2, wires=0)  qml.RX(np.pi/2, wires=0)  qml.RZ(np.pi\*3/4, wires=0)  qml.RX(np.pi, wires=0)  qml.RZ(np.pi, wires=0)    return qml.state() |
| Codercise I.7.3 – Universality of H and T |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def unitary\_with\_h\_and\_t():  # HTHTTH  qml.Hadamard(wires=0)  qml.T(wires=0)  qml.Hadamard(wires=0)  qml.T(wires=0)  qml.T(wires=0)  qml.Hadamard(wires=0)    return qml.state() |

|  |
| --- |
| **Prepare Yourself** |
| Codercise I.8.1 – State preparation |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def prepare\_state():  qml.Hadamard(wires=0)  qml.RZ(5/4\*np.pi, wires=0)  return qml.state() |
| Codercise I.8.2 – State preparation revisited |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def prepare\_state():  qml.RX(np.pi/3, wires=0)  return qml.state() |
| Codercise I.8.3 – State preparation with Mottonen’s method |
| v = np.array([0.52889389 - 0.14956775j, 0.67262317 + 0.49545818j])  dev = qml.device("default.qubit", wires=1)  def prepare\_state(state=v):  @qml.qnode(dev)  def circuit(state):  qml.MottonenStatePreparation(state, wires=0)  return qml.state()  return circuit(state)  # This will draw the quantum circuit and allow you to inspect the output gates  print(prepare\_state(v))  print()  print(qml.draw(prepare\_state, expansion\_strategy="device")(v)) |

|  |
| --- |
| **Measurements** |
| Codercise I.9.1 – Measuring a superposition |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def apply\_h\_and\_measure(state):  """Complete the function such that we apply the Hadamard gate  and measure in the computational basis.  Args:  state (int): Either 0 or 1. If 1, prepare the qubit in state |1>,  otherwise leave it in state 0.  Returns:  np.array[float]: The measurement outcome probabilities.  """  if state == 1:  qml.PauliX(wires=0)  qml.Hadamard(wires=0)  return qml.probs(wires=0)  print(apply\_h\_and\_measure(0))  print(apply\_h\_and\_measure(1)) |
| Codercise I.9.2 – Y basis rotation |
| dev = qml.device("default.qubit", wires=1)    def prepare\_psi():  state = np.array([1/2, 1j\*np.sqrt(3)/2])  qml.MottonenStatePreparation(state\_vector= state, wires=0)  def y\_basis\_rotation():  qml.Hadamard(wires=0)  qml.S(wires=0) |
| Codercise I.9.3 – Measurement in the Y basis |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def measure\_in\_y\_basis():  prepare\_psi()  qml.adjoint(qml.S)(wires=0)  qml.adjoint(qml.Hadamard)(wires=0)  return qml.probs(wires=0)  print(measure\_in\_y\_basis()) |

|  |
| --- |
| **What Did You Expect?** |
| Codercise I.10.1 – Measurement of the PauliY observable |
| dev = qml.device("default.qubit", wires=1)  @qml.qnode(dev)  def circuit():  qml.RX(np.pi/4, wires=0)  qml.Hadamard(wires=0)  qml.Z(wires=0)  return qml.expval(qml.PauliY(wires=0))  print(circuit()) |
| Codercise I.10.2 – Setting up the number of experiment shots |
| # An array to store your results  shot\_results = []  # Different numbers of shots  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 circuit():  qml.RX(np.pi/4, wires=0)  qml.Hadamard(wires=0)  qml.Z(wires=0)  return qml.expval(qml.PauliY(wires=0))    shot\_results.append(circuit())  print(qml.math.unwrap(shot\_results)) |
| Codercise I.10.3 – Evaluating the samples |
| dev = qml.device("default.qubit", wires=1, shots=100000)  @qml.qnode(dev)  def circuit():  qml.RX(np.pi / 4, wires=0)  qml.Hadamard(wires=0)  qml.PauliZ(wires=0)  return qml.sample(qml.PauliY(wires=0))  def compute\_expval\_from\_samples(samples):  """Compute the expectation value of an observable given a set of  sample outputs. You can assume that there are two possible outcomes,  1 and -1.  Args:  samples (np.array[float]): 100000 samples representing the results of  running the above circuit.  Returns:  float: the expectation value computed based on samples.  """  estimated\_expval = 0  count\_ones = np.count\_nonzero(samples == 1)  count\_negatives = np.count\_nonzero(samples == -1)  total = count\_ones + count\_negatives  estimated\_expval = (1\*count\_ones - 1\*count\_negatives) / total  return estimated\_expval  samples = circuit()  print(compute\_expval\_from\_samples(samples)) |
| Codercise I.10.4 – The variance of sample measurements |
| 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.  """  n\_trials = 100    dev = qml.device("default.qubit", wires = 1, shots = n\_shots)    @qml.qnode(dev)  def circuit():  qml.Hadamard(wires=0)  return qml.expval(qml.PauliZ(wires=0))    variance = []  for \_ in range(n\_trials):  variance.append(circuit())    return np.var(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 = 1.0 / 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) |

1. Circuits with Many Qubits

|  |
| --- |
| **Multi-Qubit Systems** |
| Codercise I.11.1 – Preparing basis state |
| num\_wires = 3  dev = qml.device("default.qubit", wires=num\_wires)  @qml.qnode(dev)  def make\_basis\_state(basis\_id):  qml.BasisEmbedding(features=basis\_id, wires=range(3))  return qml.state()  basis\_id = 3  print(f"Output state = {make\_basis\_state(basis\_id)}") |
| Codercise I.11.2 – Separable Operations |
| # Creates a device with \*two\* qubits  dev = qml.device("default.qubit", wires=2)  @qml.qnode(dev)  def two\_qubit\_circuit():  qml.Hadamard(wires=0)  qml.PauliX(wires=1)  return qml.expval(qml.PauliY(wires=0)), qml.expval(qml.PauliZ(wires=1))  print(two\_qubit\_circuit()) |
| Codercise I.11.3 – Expectation value of two-qubit observable |
| dev = qml.device("default.qubit", wires=2)  @qml.qnode(dev)  def create\_one\_minus():  qml.PauliX(0)  qml.PauliX(1)  qml.Hadamard(1)    return qml.expval(qml.PauliZ(0) @ qml.PauliX(1))  print(create\_one\_minus()) |
| Codercise I.11.4 – Double Trouble |
| 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  """  qml.RX(theta, wires=0)  qml.RY(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  """  qml.RX(theta, wires=0)  qml.RY(2\*theta, wires=1)  return qml.expval(qml.PauliZ(0) @ qml.PauliZ(1))  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 = 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) |

|  |
| --- |
| **All Tied Up** |
| Codercise I.12.1 – Entangling Operations |
| num\_wires = 2  dev = qml.device("default.qubit", wires=num\_wires)  @qml.qnode(dev)  def apply\_cnot(basis\_id):  """Apply a CNOT to |basis\_id>.  Args:  basis\_id (int): An integer value identifying the basis state to construct.  Returns:  np.array[complex]: The resulting state after applying CNOT|basis\_id>.  """  # Prepare the basis state |basis\_id>  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])  return qml.state()  # REPLACE THE BIT STRINGS VALUES BELOW WITH THE CORRECT ONES  cnot\_truth\_table = {  "00" : "00",  "01" : "01",  "10" : "11",  "11" : "10"  }  # Run your QNode with various inputs to help fill in your truth table  print(apply\_cnot(0)) |
| Codercise I.12.2 – Separable or entangled? |
| dev = qml.device("default.qubit", wires=2)  @qml.qnode(dev)  def apply\_h\_cnot():  qml.Hadamard(wires=0)  qml.CNOT(wires=[0, 1])  return qml.state()  print(apply\_h\_cnot())  # SET THIS AS 'separable' OR 'entangled' BASED ON YOUR OUTCOME  state\_status = "entangled" |
| Codercise I.12.3 – Controlled rotations |
| dev = qml.device("default.qubit", wires=3)  @qml.qnode(dev)  def controlled\_rotations(theta, phi, omega):  """Implement the circuit above and return measurement outcome probabilities.  Args:  theta (float): A rotation angle  phi (float): A rotation angle  omega (float): A rotation angle  Returns:  np.array[float]: Measurement outcome probabilities of the 3-qubit  computational basis states.  """  qml.Hadamard(wires=0)  qml.CRX(theta, wires=[0,1])  qml.CRY(phi, wires=[1,2])  qml.CRZ(omega, wires=[2,0])  return qml.probs(wires=[0,1,2])  theta, phi, omega = 0.1, 0.2, 0.3  print(controlled\_rotations(theta, phi, omega)) |

|  |
| --- |
| **We’ve Got It Under Control** |
| Codercise I.13.1 – The imposter CS |
| dev = qml.device("default.qubit", wires=2)  # Prepare a two-qubit state; change up the angles if you like  phi, theta, omega = 1.2, 2.3, 3.4  @qml.qnode(device=dev)  def true\_cz(phi, theta, omega):  prepare\_states(phi, theta, omega)  qml.CZ(wires=[0,1])  return qml.state()  @qml.qnode(dev)  def imposter\_cz(phi, theta, omega):  prepare\_states(phi, theta, omega)  qml.Hadamard(wires=1)  qml.CNOT(wires=[0,1])  qml.Hadamard(wires=1)  return qml.state()  print(f"True CZ output state {true\_cz(phi, theta, omega)}")  print(f"Imposter CZ output state {imposter\_cz(phi, theta, omega)}") |
| Codercise I.13.2 – The SWAP gate |
| dev = qml.device("default.qubit", wires=2)  # Prepare a two-qubit state; change up the angles if you like  phi, theta, omega = 1.2, 2.3, 3.4  @qml.qnode(dev)  def apply\_swap(phi, theta, omega):  prepare\_states(phi, theta, omega)  qml.SWAP(wires=[0,1])  return qml.state()  @qml.qnode(dev)  def apply\_swap\_with\_cnots(phi, theta, omega):  prepare\_states(phi, theta, omega)  qml.CNOT(wires=[0, 1])  qml.CNOT(wires=[1, 0])  qml.CNOT(wires=[0, 1])  return qml.state()  print(f"Regular SWAP state = {apply\_swap(phi, theta, omega)}")  print(f"CNOT SWAP state = {apply\_swap\_with\_cnots(phi, theta, omega)}") |
| Codercise I.13.3 – The Toffoli gate |
| dev = qml.device("default.qubit", wires=3)  # Prepare first qubit in |1>, and arbitrary states on the second two qubits  phi, theta, omega = 1.2, 2.3, 3.4  # A helper function just so you can visualize the initial state  # before the controlled SWAP occurs.  @qml.qnode(dev)  def no\_swap(phi, theta, omega):  prepare\_states(phi, theta, omega)  return qml.state()  @qml.qnode(dev)  def controlled\_swap(phi, theta, omega):  prepare\_states(phi, theta, omega)  qml.Toffoli(wires=[0, 1, 2])  qml.Toffoli(wires=[0, 2, 1])  qml.Toffoli(wires=[0, 1, 2])  return qml.state()  print(no\_swap(phi, theta, omega))  print(controlled\_swap(phi, theta, omega)) |
| Codercise I.13.4 – Mixed Controlled Gates |
| dev = qml.device("default.qubit", wires=4)  @qml.qnode(dev)  def four\_qubit\_mcx():  qml.Hadamard(wires=0)  qml.Hadamard(wires=1)  qml.Hadamard(wires=2)  qml.MultiControlledX(control\_wires=[0, 1, 2], wires=3, control\_values="001")  return qml.state()  print(four\_qubit\_mcx()) |
| Codercise I.13.5 – The 3-controlled-NOT |
| # Wires 0, 1, 2 are the control qubits  # Wire 3 is the auxiliary qubit  # Wire 4 is the target  dev = qml.device("default.qubit", wires=5)  @qml.qnode(dev)  def four\_qubit\_mcx\_only\_tofs():  # We will initialize the control qubits in state |1> so you can see  # how the output state gets changed.  qml.PauliX(wires=0)  qml.PauliX(wires=1)  qml.PauliX(wires=2)  qml.Toffoli(wires=[0, 1, 3])  qml.Toffoli(wires=[2, 3, 4])  qml.Toffoli(wires=[0, 1, 3])  return qml.state()  # print(four\_qubit\_mcx\_only\_tofs()) |

|  |
| --- |
| **Multi-Qubit Gate Challenge** |
| Codercise I.14.1 – The Bell states |
| dev = qml.device("default.qubit", wires=2)  # Starting from the state |00>, implement a PennyLane circuit  # to construct each of the Bell basis states.  @qml.qnode(dev)  def prepare\_psi\_plus():  # PREPARE (1/sqrt(2)) (|00> + |11>)  qml.Hadamard(wires=0)  qml.CNOT(wires=[0, 1])  return qml.state()  @qml.qnode(dev)  def prepare\_psi\_minus():  # PREPARE (1/sqrt(2)) (|00> - |11>)  qml.Hadamard(wires=0)  qml.CNOT(wires=[0, 1])  qml.PauliZ(wires=1)  return qml.state()  @qml.qnode(dev)  def prepare\_phi\_plus():  # PREPARE (1/sqrt(2)) (|01> + |10>)  qml.Hadamard(wires=0)  qml.PauliX(wires=1)  qml.CNOT(wires=[0, 1])    return qml.state()  @qml.qnode(dev)  def prepare\_phi\_minus():  # PREPARE (1/sqrt(2)) (|01> - |10>)  qml.Hadamard(wires=0)  qml.PauliX(wires=1)  qml.CNOT(wires=[0, 1])  qml.PauliZ(wires=0)  return qml.state()  psi\_plus = prepare\_psi\_plus()  psi\_minus = prepare\_psi\_minus()  phi\_plus = prepare\_phi\_plus()  phi\_minus = prepare\_phi\_minus()  # Uncomment to print results  # print(f"|ψ\_+> = {psi\_plus}")  # print(f"|ψ\_-> = {psi\_minus}")  # print(f"|ϕ\_+> = {phi\_plus}")  # print(f"|ϕ\_-> = {phi\_minus}") |
| Codercise I.14.2 – Quantum Multiplexer |
| dev = qml.device("default.qubit", wires=3)  # State of first 2 qubits  state = [0, 1]  @qml.qnode(device=dev)  def apply\_control\_sequence(state):  # Set up initial state of the first two qubits  if state[0] == 1:  qml.PauliX(wires=0)  if state[1] == 1:  qml.PauliX(wires=1)  # Set up initial state of the third qubit - use |->  # so we can see the effect on the output  qml.PauliX(wires=2)  qml.Hadamard(wires=2)    # IMPLEMENT THE MULTIPLEXER  # IF STATE OF FIRST TWO QUBITS IS 01, APPLY X TO THIRD QUBIT  qml.PauliX(wires=0)  qml.Toffoli(wires=[0,1,2])  qml.PauliX(wires=0)  # IF STATE OF FIRST TWO QUBITS IS 10, APPLY Z TO THIRD QUBIT  qml.PauliX(wires=1)  qml.Hadamard(wires=2)  qml.Toffoli(wires=[0,1,2])  qml.Hadamard(wires=2)  qml.PauliX(wires=1)  # IF STATE OF FIRST TWO QUBITS IS 11, APPLY Y TO THIRD QUBIT  qml.adjoint(qml.S(wires=2))  qml.Toffoli(wires=[0,1,2])  qml.S(wires=2)    return qml.state()  print(apply\_control\_sequence(state)) |