YOOBEE

COLLEGE OF CREATIVE INNOVATION

Programme

Master of Software Engineering – 180 credits

Course

Quantum Computing

MSE802

(Level 8, 15 credits, Version 1)

Assessment 2

Quantum project

Submitted by

Prabesh Tandukar

Contents

Task 1: Entanglement Demonstrations (LO2)

- Create an entangled bell state using Cirq and run the system on the Quokka device. The Bell state is given as:

$$|\psi>=\frac{1}{\sqrt{2}}(|00> \; +|11>)$$

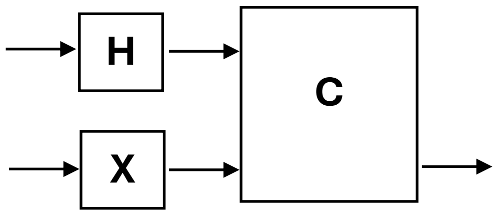
Generate qubits and apply necessary gates to generate the above state. Generate the necessary results accordingly.

```
!pip install cirq --quiet

import cirq
import requests

#Quokka API URL
quokka_url = "http://quokka1.quokkacomputing.com/qsim/qasm"


#Define the qubits
q0, q1 = cirq.LineQubit.range(2)

#Create a circuit
circuit = cirq.Circuit(
    cirq.H(q0), #Apply Hadamard gate to qubit 0
    cirq.CNOT(q0, q1), #Apply CNOT gate with qubit 0 as control and qubit 1 as target
    cirq.measure(q0, key='q0'), #Measure qubit 0 and store the result in the key 'q0'
    cirq.measure(q1, key='q1'), #Measure qubit 1 and store the result in the key 'q1'
)

#Display the circuit
print("Circuit:")
print(circuit)

#Simulate the circuit
simulator = cirq.Simulator()
result = simulator.simulate(circuit)

#Display the results
print("----------------------")
print("Results:")
print(result)
print("----------------------")
print("State vector:")
print(result.final_state_vector)
print("----------------------")

# Manually convert the circuit to QASM
qasm_code = """OPENQASM 2.0;
include "qelib1.inc";

qreg q[2];
creg c[2];

h q[0];
cx q[0], q[1];
measure q[0] -> c[0];
measure q[1] -> c[1];
"""
print(qasm_code)


import requests
import json
import numpy as np

```

```python
# The quokka device
req_str_qasm = 'http://quokka1.quokkacomputing.com/qsim/qasm'

# the Quokka accepts and returns a JSON object
data = {'script': qasm_code, 'count': 100000}

# Send the QASM code to the Quokka device
quokka_result = requests.post(req_str_qasm, json=data)

# Process the results
final_result = json.loads(quokka_result.content)
bits = list(np.concatenate(list(final_result['result'].values())).flat)
print(bits)


```

Task 2: Qiskit circuits (LO1,2)

- Using Qiskit, generate the following circuit:



  Measure the output by running it on a quokka device and print the output.

- Write a QASM script for a circuit of your own. Use different qubits and various gates to generate the final circuit. Measure the output by running it on a quokka device and print the output.

```
# !pip uninstall qiskit
!pip install --upgrade qiskit --quiet
!pip install qiskit-aer --quiet


import qiskit as qc
from qiskit_aer import Aer
from qiskit.primitives import BackendSampler
from qiskit import QuantumCircuit


#Create a Quantum circuit
qc = QuantumCircuit(2, 2)

#Apply a Hadamard gate to qubit 0
qc.h(0)

#Apply an X gate to qubit 1
qc.x(1)

#Apply a CNOT gate with qubit 0 as control and qubit 1 as target
qc.cx(0,1)

qc.measure([0, 1], [0, 1])

#Simulate the circuit
backend = Aer.get_backend('statevector_simulator')
sampler = BackendSampler(backend)
job = sampler.run(qc)

result = job.result()

print(result)


qc.draw()

#Manually generate qasm code
qasm_code = """OPENQASM 2.0;
include "qelib1.inc";

qreg q[2];
creg c[2];

h q[0];
```

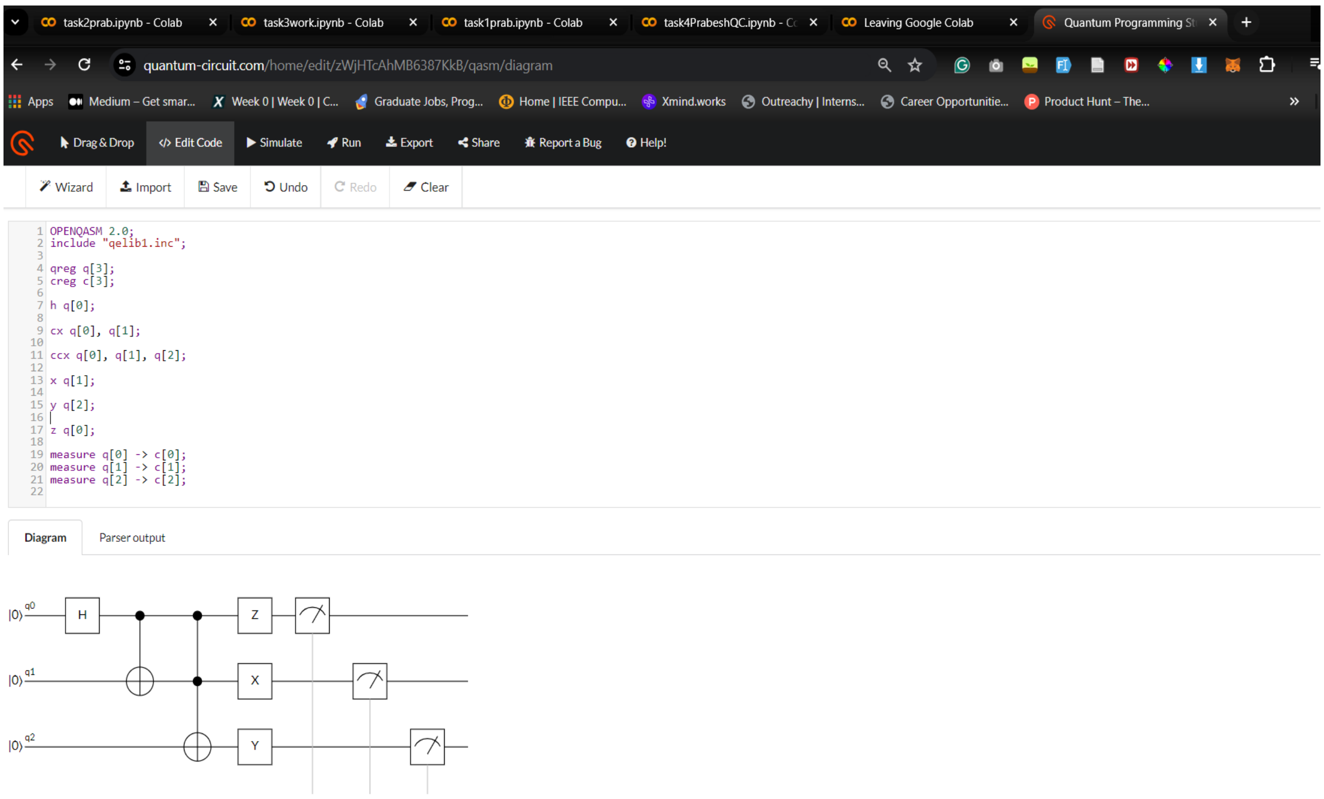
```
x q[1];
cx q[0], q[1];
measure q[0] -> c[0];
measure q[1] -> c[1];
"""

print(qasm_code)

import requests
import json
import numpy as np

# The quokka device
req_str_qasm = 'http://quokka1.quokkacomputing.com/qsim/qasm'

# the Quokka accepts and returns a JSON object
data = {'script': qasm_code, 'count': 100000}

# Send the QASM code to the Quokka device
quokka_result = requests.post(req_str_qasm, json=data)

# Process the results
final_result = json.loads(quokka_result.content)
bits = list(np.concatenate(list(final_result['result'].values())).flat)
print(bits)


#QASM script for my own circuit
qasm_code = """OPENQASM 2.0;
include "qelib1.inc";

qreg q[3];
creg c[3];

// Apply a Hadamard gate to qubit 0
h q[0];

// Apply a CX gate to qubits 0 and 1
cx q[0], q[1];

// Apply a Toffoli (CCX) gate to qubits 0, 1, and 2
ccx q[0], q[1], q[2];

// Apply a Pauli-X gate to qubit 1
x q[1];

// Apply a Pauli-Y gate to qubit 2
y q[2];

// Apply a Z gate to qubit 0
z q[0];

// Measure all qubits
measure q[0] -> c[0];
measure q[1] -> c[1];
measure q[2] -> c[2];
"""

#In this circuit, we have various single-qubit and multi-qubit gates (Hadamard, CNOT, Toffoli, Pauli-X, Pauli-Y and Z gates)

import requests
import json
import numpy as np

# The quokka device
req_str_qasm = 'http://quokka1.quokkacomputing.com/qsim/qasm'

# the Quokka accepts and returns a JSON object
data = {'script': qasm_code, 'count': 100000}
```

```python

# Send the QASM code to the Quokka device
quokka_result = requests.post(req_str_qasm, json=data)

# Process the results
final_result = json.loads(quokka_result.content)
bits = list(np.concatenate(list(final_result['result'].values())).flat)
print(bits)


#implementing the circuit on qiskit
import qiskit as qc
from qiskit_aer import Aer
from qiskit.primitives import BackendSampler
from qiskit import QuantumCircuit


#Create a Quantum circuit
qc = QuantumCircuit(3, 3)

#Apply a Hadamard gate to qubit 0
qc.h(0)

# Apply a CX gate to qubits 0 and 1
qc.cx(0, 1)

# Apply a Toffoli (CCX) gate to qubits 0, 1, and 2
qc.ccx(0, 1, 2)

# Apply a Pauli-X gate to qubit 1
qc.x(1)

# Apply a Pauli-Y gate to qubit 2
qc.y(2)

# Apply a Z gate to qubit 0
qc.z(0)

# Measure all qubits
qc.measure([0, 1, 2], [0, 1, 2])

#Simulate the circuit
backend = Aer.get_backend('statevector_simulator')
sampler = BackendSampler(backend)
job = sampler.run(qc)

result = job.result()

print(result)
qc.draw()

```

QASM script for my own circuit:

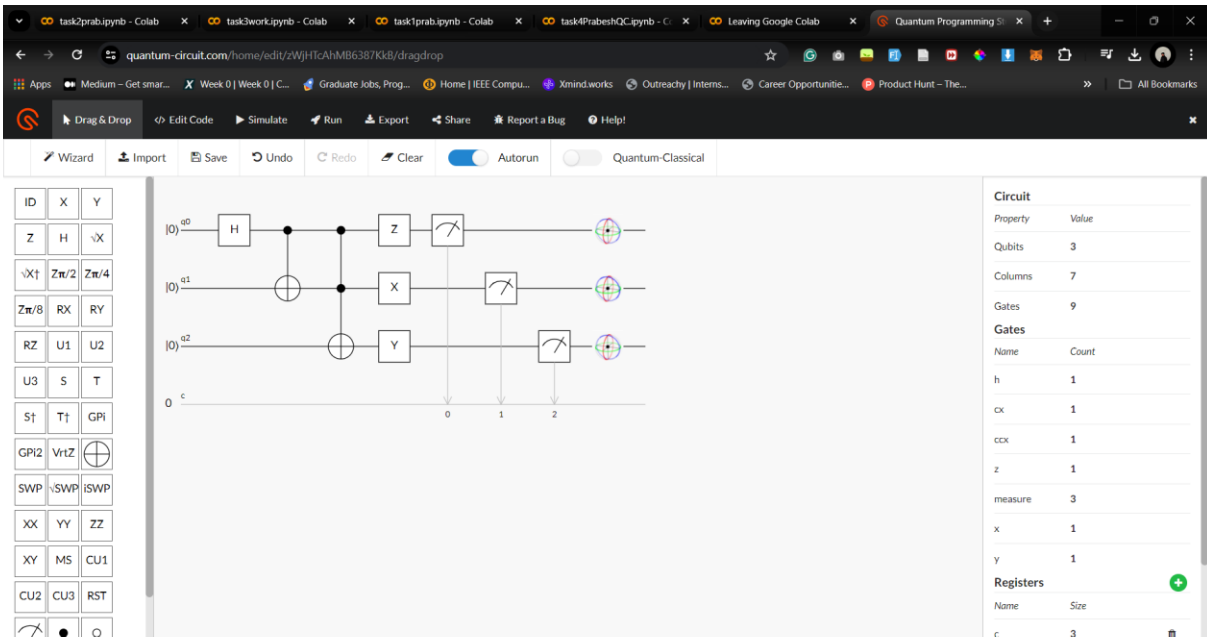
In this circuit, I have included various single-qubit and multi-qubit gates like Hadamard, CNOT, Toffoli, Pauli-X, Pauli-Y and Z gates.

```
OPENQASM 2.0;
include "qelib1.inc";

qreg q[3];
creg c[3];

h q[0];

cx q[0], q[1];

ccx q[0], q[1], q[2];

x q[1];

y q[2];

z q[0];

measure q[0] -> c[0];
measure q[1] -> c[1];
measure q[2] -> c[2];

```



```
OPENQASM 2.0;
include "qelib1.inc";

qreg q[3];
creg c[3];

h q[0];

cx q[0], q[1];

ccx q[0], q[1], q[2];

x q[1];

y q[2];

z q[0];

measure q[0] -> c[0];
measure q[1] -> c[1];
measure q[2] -> c[2];

```

Figure 1: QASM script of my own circuit



Circuit

Property	Value
Qubits	3
Columns	7
Gates	9

Gates

Name	Count
h	1
cx	1
ccx	1
z	1
measure	3
x	1
y	1

Registers

Name	Size
c	3

Figure 2: Circuit diagram in quantum cirucit.com

Task 3: Investigate a quantum code (LO1,2,4)

This is a partially developed interactive tic-tac-toe game designed to illustrate the generation of a quantum circuit through gameplay. To engage with this educational tool effectively, you are required to undertake the following tasks:

- Complete the provided code by filling in any missing instructions or comments.
- Once the code is finalized, participate in multiple instances of the game, and perform an in-depth analysis of the generated code.
- In your capacity as a quantum software engineer, elucidate the purpose, functionality of the code, the gameplay dynamics, and the process through which quantum circuits are generated. Compile your findings in a comprehensive Word document.
- Additionally, explore the various quantum gates employed in circuit generation and provide succinct descriptions of each gate's characteristics and functions in a separate section of your document.
- **Note: the code file is tic_tac_toe_quantum.ijpynb**

In-depth Analysis of the Code:

1. Game Setup ('Board' class):

The 'Board' class represents the game board and the corresponding quantum circuit. It initializes a quantum circuit with 9 qubits and 9 classical bits. Each cell in the game board corresponds to a qubit in the quantum circuit.

Initially the board is initialized with all the cells empty and all qubit in a superposition state which means all of them had the Hadamard gate applied for default.

The 'make_move' method adds a gate to the circuit based on the player's move. The player can choose from 4 gates which are Hadamard (H), Phase Gate (S), π/8 gate(T), Controlled-NOT gate (Cnot).

And the 'results' method displays the quantum circuit.

The game allows the player to measure the quantum circuit, collapsing the quantum state and determining the final game state. Measurement result s are used to update the game board toshow the final positions of X and O.

2. Game Logic ('Game' class):

The 'Game class manages the game logic and user interactions. It initializes a game board and sets up buttons for different moves (Hadamard, S, T, CNOT, Measure).

Players can make moves by clicking buttons, which are translated into quantum gates on the circuit. The 'handle_game' game method processes player clicks and updates the game state accordingly.

The 'scoreboard' method displays the current score meaning number of wins for X and O. And the 'replay' method allows the players to start a brand-new game.

The game follows the standard rules of tic-tac-toe, where players take turns making moves until one player wins or the game ends in a draw. Quantum gates are applied to the circuit based on the player's moves, with the goal of achieving a wining state.

3. Interactivity and Visualization:

   The game provides an interactive way to explore quantum circuits by linking games moves to quantum gates. Players can see the quantum circuit evolve with each move, providing a visual representation of quantum operations.

   Players can interact with the game through a graphical user interface (GUI) that allows them to make moves and view the quantum circuit. The GUI updates in real-time to reflect the current state of the game board and quantum circuit.

4. Education and Learning:

   The game is designed to be educational, helping players learn about the quantum gates and circuits through gameplay. It allows players to experiment with different gate combinations and observe their effects on the quantum state. We get to learn about quantum computing concepts, such as quantum gates, superposition and measurement.

5. Limitations and Issues:

   The code currently has an issue with the send_to_the_quokka function, which prevents the measurement results from being properly processed. This prevents the game from fully functioning as intended. The game could be further enhanced with better error handling, a more intuitive user interface, and additional features to make the gameplay more engaging.

My Finding from the interactive tic-tac-toe game.

Title: Quantum Tic-Tac-Toe: An Educational Quantum Computing Game

Introduction:

Quantum Tic-Tac-Toe is an interactive game designed to educate users about quantum computing concepts through gameplay. The game combines traditional tic-tac-toe rules with quantum mechanics principles. Players can experiment with quantum circuits and see how they impact the game in real-time.

Purpose:

The main purpose of Quantum Tic-Tac-Toe is to act as a teaching tool for explaining quantum computing concepts in a fun and interactive way. Through playing the game, individuals can develop a natural understanding of quantum gates, superposition, entanglement, and measurement - key principles in quantum computing.

Functionality:

1. Game Setup: The game initializes a quantum circuit with 9 qubits and 9 classical bits, representing the tic-tac-toe grid. Each cell in the grid corresponds to a qubit.
2. Gameplay: Players take turns making moves by applying quantum gates (Hadamard, S, T, CNOT) to the qubits. Each gate alters the quantum state of the circuit, potentially changing the outcome of the game.
3. Measurement: Players can measure the quantum circuit to collapse the quantum state and determine the final positions of X and O on the game board.
4. Visualization: The game provides a graphical user interface (GUI) that displays the current state of the game board and the quantum circuit, allowing players to visualize the effects of their moves.
5. Education: Quantum Tic-Tac-Toe is designed to be an educational tool, providing players with an interactive way to learn about quantum computing concepts through gameplay.

Gameplay Dynamics:

1. Move Selection: Players can select from five different moves: Hadamard, S, T, CNOT, and Measure.
2. Strategy: Players can develop strategies based on quantum principles, such as superposition and entanglement, to outplay their opponents and win the game.
3. Real-Time Updates: The GUI updates in real-time to reflect the current state of the game, allowing players to see the effects of their moves immediately.
4. Learning Curve: The game has a gentle learning curve, introducing players to quantum concepts gradually as they progress through the game.

Process of Quantum Circuit Generation:

1. Initialization: The game initializes the quantum circuit with all qubits in a superposition state (Hadamard gate applied).
2. Move Application: Players apply quantum gates to the qubits based on their moves, altering the quantum state of the circuit.

3. Measurement: Players can measure the quantum circuit to collapse the quantum state and determine the final game state.
4. Visualization: The GUI visualizes the quantum circuit, allowing players to see the effects of their moves on the circuit.

Conclusion:

Quantum Tic-Tac-Toe is an innovative educational game that combines the principles of quantum computing with the classic game of tic-tac-toe. By playing the game, users can gain a deeper understanding of quantum concepts while having fun. The game's interactive nature and real-time visualization make it an effective tool for learning about quantum computing in a hands-on way.

Quantum Gates in Circuit Generation:

1. Hadamard Gate (H):

- Characteristics: The Hadamard gate creates superposition by putting a qubit into a state where it has an equal probability of being measured as 0 or 1.
- Function: It is used to create superposition, which is a key principle in quantum computing for parallel computation and quantum algorithms like the Quantum Fourier Transform.

2. Phase Gate (S):

- Characteristics: The Phase gate introduces a phase of i (imaginary unit) to the state |1⟩, resulting in a rotation of 90 degrees around the Z-axis on the Bloch sphere.
- Function: It is used to manipulate the phase of a qubit, which is important for quantum algorithms like quantum phase estimation and quantum error correction.

3. π/8 gate (T):

- Characteristics: The T gate introduces a phase of e^(iπ/4) to the state |1⟩, resulting in a rotation of 45 degrees around the Z-axis on the Bloch sphere.
- Function: It is used for more precise phase manipulation, especially in algorithms where phase coherence is critical.

4. Controlled-NOT Gate (CNOT):

- Characteristics: The CNOT gate is a two-qubit gate that flips the second qubit (target) if the first qubit (control) is in state |1⟩.
- Function: It is a fundamental entangling gate used in many quantum algorithms, including quantum error correction and quantum teleportation.

Task 4: Machine learning Quantum Analysis (LO1,2,4)

You have been provided with a code designed to address a machine learning problem cantered on image classification. The code generates binary images and employs an optimization function to assess sample images, ultimately yielding a metric. This entire process is executed using a specifically tailored circuit.

Your tasks include:

- Analysing the code, with a particular focus on the circuit component and the problem's formulation. Determine the data input into the circuit and identify the portion that serves as the input. Additionally, elucidate the code's functionality, particularly its quantum aspects, excluding the optimization algorithm.
- Executing the code and monitoring the final output at each iteration. Create additional code to plot the iteration number against the output measure. Also, generate a graph depicting the iteration number against the time required for producing the output.
- Modifying the code to run on your local machine, eliminating the need for the quantum computer or circuit. Measure and document the disparities in performance, both in terms of efficiency and effectiveness.
- Compile a comprehensive report summarizing the tasks above.
- For the final task, you are required to submit a modified code (.ijpynb file) and a word document.

Task 4 a :

First, let's analyse the code focusing on the quantum circuit component and the problem's formulation.

Data Input into the circuit:

- The input data for the quantum circuit is a binary array representing either stripes or bars. Each element in the array corresponds to a qubit in the circuit.
- The input data is used to set the initial state of the qubits. If the value at an index is greater than 0.5, an X gate is applied to the corresponding qubit, effectively flipping its state.

Circuit Component:

- The quantum circuit consists of multiple blocks, each block handling a pair of qubits.
- Each block consists of two qubit rotations (ry gates) followed by a controlled-NOT (CX) gate between the two qubits.
- The ry gates are parameterized by angles, which are part of the optimization process to find the optimal angles that minimize the objective function.
- The circuit is constructed in a way that the final qubit is measured, and the measurement outcome is used to compute the objective function.

Problem Formulation:

- The problem is formulated as a binary image classification task, where the goal is to classify images as either stripes or bars.

- Each image is represented as a binary array, with stripes represented by alternating 1s and 0s and bars represented by repeated 1s or 0s.
- The quantum circuit is used to process the input data and produce a measurement outcome, which is then used to compute the objective function.

Quantum Aspects:

- The quantum aspects of the code include the construction of the quantum circuit, which uses quantum gates (ry and CX gates) to manipulate qubits.
- The input data is encoded into the quantum state of the qubits, and the circuit operations are applied to this quantum state.
- The measurement outcome of the final qubit is used as a probabilistic indicator of the classification result, which is then used to compute the objective function.

Overall, the code implements a quantum circuit based approach for binary image classification, where the quantum circuit is used to process input data and produce a measurement outcome that is used in a classical optimization algorithm to minimize an objective function.

Task 4 b:

Additional code to plot the iteration number against the output measure. Also, generate a graph depicting the iteration number against the time required for producing the output.

# Binary Image Classification

### A) Generating dataset

```
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
import requests
import math
import json
import sympy
import time

# current Quokka address:
req_str_qasm = 'http://quokka1.quokkacomputing.com/qsim/qasm'
```

```
def generate_binary_code(bit_length):
    bit_combinations = np.zeros((int(2**bit_length), bit_length))
    for number in range(int(2**bit_length)):
        dividend = number
        bit_index = 0
        while dividend != 0:
            bit_combinations[number, bit_index] = np.remainder(dividend, 2)
            dividend = np.floor_divide(dividend, 2)
            bit_index += 1
    return bit_combinations

def generate_data(length):
    stripes = generate_binary_code(length)
    stripes = np.repeat(stripes, length, 0)
    stripes = stripes.reshape(2 ** length, length * length)

    bars = generate_binary_code(length)
    bars = bars.reshape(2 ** length * length, 1)
    bars = np.repeat(bars, length, 1)
    bars = bars.reshape(2 ** length, length * length)
    return np.vstack((stripes[1:stripes.shape[0]-1],bars[1:bars.shape[0]-1]))
```

```
length = 2
dataset = generate_data(length)
labels = np.concatenate((np.zeros(int(2**length-2)),np.ones(int(2**length-
2))))
n_parameters = int(2*(length**2-1))
```

```
#Plot sample data (all possiblities of straps and bars)
#Note the total number of samples fom each class is (2**length) - 2
print('THIS IS THE DATASET')
plt.figure(figsize=[10, 10])
j = 1
for i in dataset:
    print(i)
    print('-----------')
    plt.subplot(4, int(2**(length-1)-1), j)
    j += 1
    plt.imshow(np.reshape(i, [length,length]), cmap="gray")
    plt.xticks([])
    plt.yticks([])
print('THIS IS THE LABELS')
print(labels)
```

### B) Splitting dataset

```
# Select 75% of the entries in the array
selected_indices = np.random.choice(dataset.shape[0],
                                    int(dataset.shape[0] * 0.5),
                                    replace=False)

all_indices = np.arange(dataset.shape[0])
unselected_indices = np.setdiff1d(all_indices, selected_indices)

# Get the selected elements
training = dataset[selected_indices]
training_labels = labels[selected_indices]
test = dataset[unselected_indices]
test_labels = labels[unselected_indices]
```

```
# Training data visualization
print('THIS IS THE TRAINING DATASET')
plt.figure(figsize=[10, 10])
j = 1
for i in training:
    print(i)
    print('-----------')
    plt.subplot(1, training.shape[0], j)
    j += 1
    plt.imshow(np.reshape(i, [length,length]), cmap="gray")
    plt.xticks([])
    plt.yticks([])
print('THIS IS THE TRAINING LABELS')
print(training_labels)
```

```
# Testing data visualization
print('THIS IS THE TESTING DATASET')
plt.figure(figsize=[10, 10])
j = 1
for i in test:
    print(i)
    print('-----------')
    plt.subplot(1, test.shape[0], j)
    j += 1
    plt.imshow(np.reshape(i, [length,length]), cmap="gray")
    plt.xticks([])
    plt.yticks([])
print('THIS IS THE TESTING LABELS')
print(test_labels)
```

### D) The following codes will generate the required circut witht the any number of qbits as input

```
# One block circuit
def block(angles,qubits):
  qasm = '''
  ry({}) q[{}];
  ry({}) q[{}];
  cx q[{}], q[{}];
  '''.format(
      angles[0],qubits[0],
      angles[1],qubits[1],
      qubits[0],qubits[1]
  )
  return qasm
```

```
def generate_pairs(mylist, start, end):
  if start == end:
      pass
  else:
    mid = (start + end - 1) // 2
    first_half = generate_pairs(mylist, start, mid)
    second_half = generate_pairs(mylist, mid + 1, end)
    mylist.append((mid,end))
```

```
def create_circuit(data,angles):

  length = data.shape[0]

  qasm = '''
    OPENQASM 2.0;
    include "qelib1.inc";
    qreg q[{}];
    creg c[1];
    '''.format(length)

  for i in range(length):
    if data[i] > 0.5:
      qasm += '''
    x q[{}];
    '''.format(i)

  pairs = []
  generate_pairs(pairs,0,length-1)

  count = 0
  for (i,j) in pairs:
    qasm += block([angles[count],angles[count+1]],[i,j])
    count += 2

  qasm += '''
  measure q[{}] -> c[0];
  '''.format(length-1)

  return(qasm)
```

```
print(dataset[1])
print(np.zeros(n_parameters))
```

```
#Quickly testing the QASM
print(create_circuit(dataset[1],np.zeros(n_parameters)))
```

### E) Creating the objective function

```
def objective(x, dataset, labels, count):
    n_data = dataset.shape[0]
    to_return = 0

    for idx in range(n_data):
        qasm = create_circuit(dataset[idx],x)

        data = {'count': count,
                'script': qasm
                }

        result = requests.post(req_str_qasm, json=data)
        values = json.loads(result.content)['result']['c']

        #print(dataset[idx])
        #print(x)
        #print(values)

        to_return += np.abs(np.mean(values)-labels[idx])

    return to_return/n_data
```

### F) The actual training and ML/optimization algorithm

Method 1: Using Scipy built-in function

```
#Initialize list to store objective funciton value
obj_values = []

#To get the iteration infroamtion
def call_fn (xk):
    print("Iteration:", call_fn.iteration)
    p = objective(xk,training,training_labels,count=1)
    print("Objective Function Value:", sum(xk**2)**0.5)
    print("Parameter Values:", xk)
    print("")

    #Append objective function value to list for plotting
    obj_values.append(p)

    call_fn.iteration += 1
call_fn.iteration = 0

#The model and training process
sol = sp.optimize.minimize(lambda x:
objective(x,training,training_labels,count=10),
                           x0 = np.pi*np.random.randn(n_parameters),
                           method = 'Nelder-Mead', options
={'maxiter':5},callback=call_fn)
print(sol)
```

##Modify the training process

```
#The model and training process
start_time = time.time()
sol = sp.optimize.minimize(lambda x: objective (x, training,
training_labels,count=10),
                           x0 = np.pi*np.random.randn(n_parameters),
                           method = 'Nelder-Mead', options = {'maxiter':5},
callback=call_fn)
end_time = time.time()

print("Total Time Taken:", end_time - start_time)
```

## Plotting the Iteration Number against the Output Measure

```
plt.figure()
plt.plot(range(len(obj_values)), obj_values)
plt.xlabel('Iteration Number')
plt.ylabel('Objective Function Value')
plt.title('Iteration Number vs. Objective Function Value')
plt.show()
```

##Plotting the Iteration Number against the Time Required

```
# Initialize list to store time taken at each iteration
times = []

def call_fn(xk):
    start = time.time()
    # Your existing code here
    end = time.time()
    times.append(end - start)

# Modify the training process to use the new callback function

# The model and training process
start_time = time.time()
sol = sp.optimize.minimize(lambda x:
objective(x,training,training_labels,count=10),
                           x0 = np.pi*np.random.randn(n_parameters),
                           method = 'Nelder-Mead', options
={'maxiter':5},callback=call_fn)
end_time = time.time()

print("Total Time Taken:", end_time - start_time)

plt.figure()
plt.plot(range(1, len(times) + 1), times)
plt.xlabel('Iteration Number')
plt.ylabel('Time (seconds)')
plt.title('Iteration Number vs. Time Taken')
plt.show()
```

Method 2: Using spsa optimizer

```
#The actual code for the optimizer
def spsa(func, x0, a=0.1, c=0.01, alpha=0.602, gamma=0.101, maxiter=100,
verbose=False):

    k = 0
    x = x0

    while k < maxiter:
        ak = a / (k+1)**alpha
        ck = c / (k+1)**gamma
        delta = 2 * np.random.randint(0, 2, len(x0)) - 1
        xp = x + ck * delta
        xm = x - ck * delta
        grad = (func(xp) - func(xm)) / (2 * ck) * delta

        x = x - ak * grad

        # Print progress message
        if verbose and k % int(0.1*maxiter) == 0:
            fx = func(x)
            print(f"Iteration {k}: f = {fx}")

        k += 1

    return x

# Some parameters to set
a = 1
c = 0.5
maxiter= 10
alpha = 0.602
gamma = 0.101

f = lambda x: objective(x,training,training_labels,count=10)
x0 = np.pi*np.random.randn(n_parameters)

sol = spsa(f, x0, a=a, c=c,
            alpha=alpha, gamma=gamma,
            maxiter=maxiter, verbose=True)
```

### G) Testing of data

```
# Actual testing function
def testing_circuit(x, dataset):
    count = 1
    n_data = dataset.shape[0]
    output_labels = []
    for idx in range(n_data):
        qasm = create_circuit(dataset[idx],x)
        data = {'count': count, 'script': qasm}
        result = requests.post(req_str_qasm, json=data)
        values = json.loads(result.content)['result']['c'][0]
        output_labels.append(values)
    return output_labels
```

```
# Calling the testing fuction to get the predicted classes
output_labels = testing_circuit(np.array(sol), test)
output_labels = np.array(output_labels).flatten()
```

```
output_labels
```

```
test_labels
```

Task 4 c:

Modifying the code to run on your local machine, eliminating the need for the quantum computer or circuit. Measure and document the disparities in performance, both in terms of efficiency and effectiveness.

#Modified code for simulating the quantum circuit on a classical machine

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy as sp
import requests
import json
import time

# Define functions for generating data and simulating quantum circuit
def generate_binary_code(bit_length):
    bit_combinations = np.zeros((int(2**bit_length), bit_length))
    for number in range(int(2**bit_length)):
        dividend = number
        bit_index = 0
        while dividend != 0:
            bit_combinations[number, bit_index] = np.remainder(dividend, 2)
            dividend = np.floor_divide(dividend, 2)
            bit_index += 1
    return bit_combinations

def generate_data(length):
    stripes = generate_binary_code(length)
    stripes = np.repeat(stripes, length, 0)
    stripes = stripes.reshape(2 ** length, length * length)

    bars = generate_binary_code(length)
    bars = bars.reshape(2 ** length * length, 1)
    bars = np.repeat(bars, length, 1)
    bars = bars.reshape(2 ** length, length * length)
    return np.vstack((stripes[1:stripes.shape[0]-1], bars[1:bars.shape[0]-1]))

def simulate_circuit(data, angles):
    length = len(data)
    qubits = np.zeros(length)

    for i in range(length):
        if data[i] > 0.5:
            qubits[i] = 1

    pairs = []
    generate_pairs(pairs, 0, length - 1)

    for (i, j) in pairs:
        qubits[j] = np.cos(angles[0]) * qubits[j] + np.sin(angles[0]) *
qubits[i]
        qubits[i] = np.cos(angles[1]) * qubits[i] + np.sin(angles[1]) *
qubits[j]

    return np.mean(qubits)

# Define objective function and training process
def objective(x, dataset, labels, count):
    n_data = dataset.shape[0]
    to_return = 0

    for idx in range(n_data):
        simulated_value = simulate_circuit(dataset[idx], x)
        to_return += np.abs(simulated_value - labels[idx])

    return to_return / n_data

def call_fn(xk):
    global obj_values
    print("Iteration:", call_fn.iteration)
    p = objective(xk, training, training_labels, count=1)
    print("Objective Function Value:", sum(xk**2)**0.5)
    print("Parameter Values:", xk)
    print("")

    obj_values.append(p)
    call_fn.iteration += 1
call_fn.iteration = 0

# Generate dataset
length = 2
dataset = generate_data(length)
labels = np.concatenate((np.zeros(int(2**length-2)),np.ones(int(2**length-
2))))
n_parameters = int(2*(length**2-1))

# Split dataset
selected_indices = np.random.choice(dataset.shape[0],
                                    int(dataset.shape[0] * 0.5),
                                    replace=False)
all_indices = np.arange(dataset.shape[0])
unselected_indices = np.setdiff1d(all_indices, selected_indices)
training = dataset[selected_indices]
training_labels = labels[selected_indices]
test = dataset[unselected_indices]
test_labels = labels[unselected_indices]
```

```python
# Training process
obj_values = []
start_time = time.time()
sol = sp.optimize.minimize(lambda x: objective(x, training, training_labels, 
count=10),
                           x0=np.pi*np.random.randn(n_parameters),
                           method='Nelder-Mead', options={'maxiter': 5}, 
callback=call_fn)
end_time = time.time()

print("Total Time Taken:", end_time - start_time)

# Plotting
plt.figure()
plt.plot(range(len(obj_values)), obj_values)
plt.xlabel('Iteration Number')
plt.ylabel('Objective Function Value')
plt.title('Iteration Number vs. Objective Function Value')
plt.show()
```