### Introduzione

L’assignment consiste nell’implementare un codice di correzione degli errori.

Durante la comunicazione è possibile che si generino dei weak error, cioè che si trasformino alcuni qubit con .

Per ovviare a ciò, si immagazzina l’informazione logica in un numero dispari sempre maggiore di bit, a seconda dell’importanza del rumore: per esempio, il qubit logico può essere rappresentato come stato composto dai qubit .

Per riuscire a capire l’errore e correggerlo si applica quindi il protocollo di voto di maggioranza: si misurano i bit e, grazie alla ridondanza d’informazione, è possibile riconoscere l’errore e correggerlo.

### Inizializzazione

Per implementare il codice abbiamo utilizzato PennyLane e le librerie per generare numeri random, per le operazioni matematiche e per la creazione dei grafici:

import pennylane as qml

import numpy as np

import matplotlib.pyplot as plt

import random

Siamo partiti creando lo stato a singolo qubit , con e a piacimento.

Abbiamo definito il numero di qubit per immagazzinare l’informazione

n\_qubits = 3

Poi abbiamo inizializzato il device

dev = qml.device('default.qubit', wires=n\_qubits, shots=1000)

Abbiamo scelto i coefficienti, e , in quanto la probabilità deve essere uguale a

alpha = 0.4

beta = np.sqrt(1 - alpha \*\* 2)

Infine abbiamo creato la funzione per inizializzare lo stato

def initial\_state(alpha, beta):

qml.RY(2 \* np.arccos(alpha), wires=0)

qml.CNOT(wires=[0, 1])

qml.CNOT(wires=[0, 2])

Per creare lo stato abbiamo utilizzato una rotazione sull’asse Y sul qubit in posizione 0.

Dopodichè abbiamo applicato 2 porte CNOT per poter inizializzare i qubit ancilla: sono i qubit aggiunti alla rappresentazione del qubit logico, necessari per la ridondanza e quindi il riconoscimento dell’errore.

In questo momento quindi il nostro stato generico è .

### Implementazione dell’errore

Ora è necessario implementare una trasformazione unitaria che simuli l’errore su un qubit del tipo con .

def apply\_weak\_error():

qubit\_to\_flip = random.randint(0, 2)

epsilon = 0.1

qml.RX(epsilon, wires=qubit\_to\_flip)

Abbiamo implementato una funzione che si occupa di scegliere un qubit randomico da flippare per poi applicare una rotazione sull’asse X di angolo che andrà a sviluppare l’errore.

@qml.qnode(dev)

def circuit1(alpha, beta):

initial\_state(alpha, beta)

apply\_weak\_error()

return qml.probs(wires=[0, 1, 2])

Quindi l’abbiamo applicate al nostro stato e abbiamo stampato la probabilità di errore:

State |000⟩: 0.1540

State |001⟩: 0.0010

State |010⟩: 0.0000

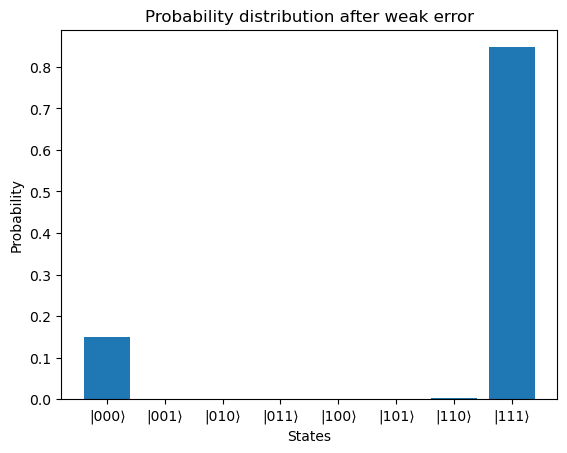
State |011⟩: 0.0000

State |100⟩: 0.0000

State |101⟩: 0.0000

State |110⟩: 0.0010

State |111⟩: 0.8440



L’output indica come si ha una grande probabilità di ricadere nello stato (come indicato da ), una minor probabilità di ricadere in (come indicato da ) e delle piccole probabilità di avere un bit flip randomico, che in questo caso provoca il flip del terzo qubit.

### Codice di correzione

Ora che è presente l’errore, bisogna implementare l’algoritmo che lo rilevi e lo risolva.

Una misura diretta dello stato distruggerebbe la sovrapposizione, per cui si devono trovare degli osservabili che permettono di estrarre l’informazione senza perturbare lo stato.

Si prendono quindi gli operatori:

* , che misura contemporaneamente il valore dell’operatore di Pauli del primo e del secondo qubit
* , che invece misura il valore dell’operatore di Pauli del primo e del terzo qubit

Essendo che il nostro stato è un autostato degli operatori e , avremo che lo stato collasserà o nello stato corretto autocorreggendo l’errore, oppure nello stato con un bit flippato: in quest’ultimo caso, uno dei due autovalori avrà valore -1

@qml.qnode(dev)

def circuit2(alpha, beta):

initial\_state(alpha, beta)

apply\_weak\_error()

Z1Z2 = qml.sample(qml.PauliZ(0) @ qml.PauliZ(1))

Z1Z3 = qml.sample(qml.PauliZ(0) @ qml.PauliZ(2))

if Z1Z2 == 1 and Z1Z3 == 1:

pass

elif Z1Z2 == -1 and Z1Z3 == -1:

qml.PauliX(wires=0)

elif Z1Z2 == -1 and Z1Z3 == 1:

qml.PauliX(wires=1)

elif Z1Z3 == 1 and Z1Z2 == -1:

qml.PauliX(wires=2)

corrected\_state = qml.probs(wires=[0, 1, 2])

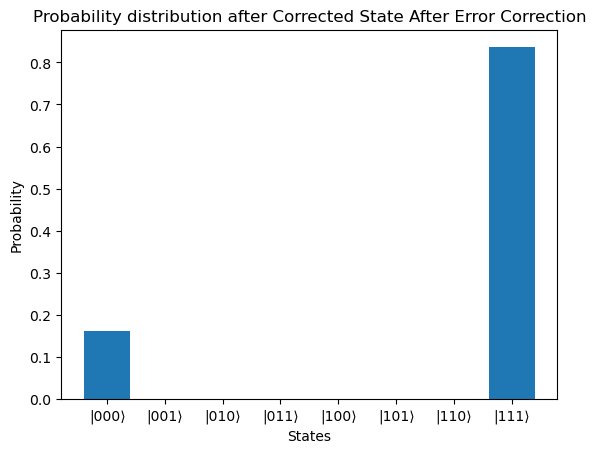
return Z1Z2, Z1Z3, corrected\_state

Abbiamo misurato gli operatori utilizzando delle porte di Pauli Z.

A seconda del risultato della misura applichiamo delle strategie di correzione diverse:

* Se entrambi gli operatori hanno riportato +1 vuol dire che non si è verificato errore
* Se entrambi hanno riportato -1, l’errore è nel primo qubit, quindi applichiamo una porta di Pauli in quella posizione
* Se e , l’errore è nel secondo qubit, quindi applichiamo la porta di conseguenza
* Se e , l’errore è nel terzo qubit

Dopodiché abbiamo misurato lo stato corretto e plottato il risultato.



### Conclusioni

Durante l’implementazione abbiamo avuto delle problematiche con l’utilizzo di PennyLane.

Infatti, non è possibile mantenere lo stato finale nella forma .

Si avrà invece uno stato con probabilità e con probabilità .

Avendo scelto , avremo e .