QOSF - Simulator Task - pdfVersion

January 30, 2021

[1]:
```python
import numpy as np
import pandas as pd
import random
from collections import Counter
import math
from operator import itemgetter
import sympy as sp
```

[14]:
```python
## Function to get the ground state in a multi-qubit system. ##

def get_ground_state(num_qubits):

    ## qubit numbering starts from 0,
    ## i.e., in a 3 qubit system, 1st qubit is q0, 2nd is q1, etc.

    q0 = np.array([[1], [0]])                               ## |0> state
    q1 = np.array([[0], [1]])                               ## |1> state
    gs = 1                                          ## gs = ground state variable

    ## tensor product of |0>, |0>, ... num_qubits times
    for i in range(num_qubits):
        gs = np.kron(gs, q0)

    print("Info: The ground state is initialised with each qubit at |0>.\n")
    print("Total no. of qubits in this system: ", num_qubits)
    print("\n Therefore, the ground state of the system is: \n", gs)
    return gs



## Function to get the matrix representation of gates in a multi-qubit system. ##
## The gates included are one-qubit and CNOT (CX) gates.   ##

def get_operator(total_qubits, gate_unitary, target_qubits):
    q0 = np.array([[1], [0]])
    q1 = np.array([[0], [1]])
    qplus = (q0 + q1)/np.sqrt(2)                ## (|0> + |1>)/sqrt(2) state
```

```
    qminus = (q0 - q1)/np.sqrt(2)                ## (|0> - |1>)/sqrt(2) state
    phase_T = np.exp((1j)*math.pi/4)
    P0x0 = q0 @ q0.T                                 ## Projection op. |0><0|
    P1x1 = q1 @ q1.T                                 ## Projection op. |1><1|
    I = np.identity(2)                               ## Identity Gate
    HadamardGate = qplus @ q0.T + qminus @ q1.T      ## HGate, H = |+><0| + |-><1|
    XGate = q0 @ q1.T + q1 @ q0.T                ## XGate, X = |0><1| + |1><0|
    ZGate = q0 @ q0.T - q1 @ q1.T                ## ZGate, Z = |0><0| - |1><1|
    YGate = (-1j)*(q0 @ q1.T - q1 @ q0.T)        ## YGate, X = -j|0><1| + j|1><0|
    TGate = q0 @ q0.T + phase_T*q1 @ q1.T    ## TGate, T = |0><0| + exp(i*pi/
↪4)|1><1|



    SingleQubit_Gates = {'h' : HadamardGate,
                         'x' : XGate,
                         'z' : ZGate,
                         'y' : YGate,
                         't' : TGate}

    GateKeys = list(SingleQubit_Gates.keys())

    ## To generate operators for single-qubit gates
    ## from the SingleQubit_Gates dictionary
    ## with various target qubit positions in a multi-qubit system.
    ## Info: Acts on only one-qubit so
    ## [target_qubits] should be one-item list s.t. [target]
    ## for e.g. in a 3-qubit system,
    ## to make a 1-qubit gate act on qubit 2: target_qubits = [1]

    if gate_unitary in GateKeys:
        GateOp = 1
        for m in range(total_qubits):
            if [m] == target_qubits:
                U = SingleQubit_Gates[gate_unitary]
            else:
                U = I
            GateOp = np.kron(GateOp, U)
        return GateOp


    ## To generate an operator for CNOT (CX) gate with various
    ## control and target qubits in a multi-qubit system :
    ## Info: Acts on only one target-qubit with single control-qubit
    ## so [target_qubits] should be two-item list s.t. [control, target]
    ## for e.g. in 3-qubit system, to have control at qubit 0
    ## and target at qubit 1: target_qubits = [0, 1]
```

```python
    if gate_unitary == 'cx':
        X = SingleQubit_Gates['x']
        GateOp1 = 1
        GateOp2 = 1
        for m in range(total_qubits):
            if m == target_qubits[0]:
                U1 = P0x0
                U2 = P1x1
            elif m == target_qubits[1]:
                U1 = I
                U2 = X
            else:
                U1 = I
                U2 = I
            GateOp1 = np.kron(GateOp1, U1)  ## tensor prod. for |0><0| part
            GateOp2 = np.kron(GateOp2, U2)  ## tensor prod. for |1><1| part
        GateOp = GateOp1 + GateOp2
        return GateOp

    return print("Check parameters. \n")             ## Trouble-shooting.



## Function to get U3 (theta, phi, lambda) gate's operator - matrix.

def get_parametric_gateU3(total_qubits, gate_unitary, target_qubits, theta,
                          phi, lam):
    cos = round(np.cos(theta / 2), 1)
    sin = round(np.sin(theta / 2), 1)
    exp_lam = round(np.exp(1j * lam), 1)
    exp_phi = round(np.exp(1j * phi), 1)

    U3 = np.array([[cos, -exp_lam * sin],
                [exp_phi * sin, (exp_lam * exp_phi) * cos]
            ])
    I = np.identity(2)
    if gate_unitary == 'u3':
        GateOp = 1
        for m in range(total_qubits):
            if [m] == target_qubits:
                U = U3
            else:
                U = I
            GateOp = np.kron(GateOp, U)
    return GateOp
```

```python
## Function to run the circuit given as 'program'.

def run_program(initial_state, program):

    ## Calculates total # of qubits
    ## in the initial_state.
    num_qubits = int(math.log(len(initial_state), 2))

    ## # of "loops"(i.e., sub-circuits) in the circuit.
    NumOfLoops = len(program)

    gates = list(map(itemgetter('gate'), program))          ## Gates.
    targets = list(map(itemgetter('target'), program))      ## Target qubits.

    ## Initialising cumulative gate-matrix operator.

    gate_final = np.identity(2**num_qubits)
    print("\n ***Performing circuit operations*** \n")

    for i in range(NumOfLoops):

        ## To check for loops with parametric gate U3:
        if gates[i] == 'u3':
            temp = program[i]
            theta = temp['params']['theta']
            phi = temp['params']['phi']
            lam = temp['params']['lambda']

            gate_in = get_parametric_gateU3(num_qubits, gates[i], targets[i],
                                            theta, phi, lam)
        else:
            ## Getting operator for gates in each loop
            gate_in = get_operator(num_qubits, gates[i], targets[i])

        ## Final op. to act on initial state.
        gate_final = gate_in @ gate_final

        print("You have applied", gates[i],
              "gate to qubit(s)", targets[i], ".\n")

    ## Final state = fs (column vec.)
    fs = gate_final @ initial_state
    print("\n ***Retrieving the final state*** \n")

    ## Computational basis states
    CompBasis = []
```

```python
    for i in range(len(fs)):
        CompBasis.append(decbin(i, num_qubits))

    print("The final state of the system is: \n", fs)
    print("\n The computational basis states are: ", CompBasis, "\n")

    return fs



## Function to convert decimal to binary number.

def decbin(number, bits):
    a = bin(number)[2:]
    c = a.zfill(bits)
    return c



## Function to simulate 'measurement' of states.

def measure_all(state_vector):
    num_qubits = int(math.log(len(state_vector), 2))
    ListOfProbabilities = []                        ## List of probabilities

    ListOfIndices = []                              ## List of indices
                                                    ## (which actually represent
                                                    ## the basis states.)
    ## Probabilities = |<psi|psi>|^2
    ## if |psi> = some state vector.

    weights = np.multiply(state_vector, state_vector.conj())

    for i in range(len(state_vector)):
        ListOfIndices.append(decbin(i, num_qubits))
        ListOfProbabilities.append(weights[i, 0])

    ## Weighted-random choice
    ChosenIndex = random.choices(ListOfIndices, ListOfProbabilities)

    return ChosenIndex



## Function to give counts of the measured states.

def get_counts(state_vector, num_shots):
```

```
    counts = []
    print("\n ***Making measurements*** \n")
    for i in range(num_shots):
        m = measure_all(state_vector)
        counts.append(m[0])
    results = dict(Counter(counts))
    print("Counts of the final state on measurement are: \n", results)
    return results
```

[10]:

```
## Basic Program: ##
## Big Endian Representation
## (left bit belongs to first qubit and right bit belongs to second qubit)

## Gate keys: H gate = 'h', X gate = 'x', Z gate = 'z', Y gate = 'y',
## T gate = 't', U3 gate = 'u3', CNOT (CX) gate = 'cx'

## For single qubit & U3 (single qubit parametric gate) gates,
## the target qubits are named as:
## [0], [1], ..., [n-1] for an n-qubit system.

## For CX gate gate:
## The control and target qubits are [control, target] == [n1, n2]
## where n1 != n2 and n1 & n2 <= n-1 in an n-qubit system.




# Define circuit:

my_circuit = [
{ "gate": "h", "target": [0] },
{ "gate": "cx", "target": [0, 1] }
]

# Fetches ground state all qubits initially in |0> state
# get_ground_state(num_qubits), where,
# num_qubits = # of qubits in the system
# in this e.g., num_qubits = 2

ground_state = get_ground_state(2)


final_state = run_program(ground_state, my_circuit)         # Final state

# Counts of states on measurement
counted_qubitstates = get_counts(final_state, 200)
```

```
Info: The ground state is initialised with each qubit equal to |0>.
```

```
Total no. of qubits in this system:  2

 Therefore, the ground state of the system is:
 [[1]
 [0]
 [0]
 [0]]

 ***Performing circuit operations***

You have applied h gate to qubit(s) [0] .

You have applied cx gate to qubit(s) [0, 1] .


 ***Retrieving the final state***

The final state of the system is:
 [[0.70710678]
 [0.        ]
 [0.        ]
 [0.70710678]]

 The computational basis states are:  ['00', '01', '10', '11']


 ***Making measurements***

Counts of the final state on measurement are:
 {'11': 105, '00': 95}
```