QCOMPPW101: Quantum computing kickoff practical work

Lotus Noir Quantum Computing Research Group

Based on:
MITx: 8.370.1x Quantum Information Science I, Part I

Practical work Outline: During This practical work we will simulate a quantum computer.

Contents

1 Introduction

The goal of this practical work is to make you program a simulation of a quantum computer.You will have to program it in **Python3**. The only librairie you have the right to use is **NumPy**, **math** and **Enum**. You will have to program the function:

```
quantumComputer (nbQbits: int, quantumGates: list)
```

nbQbits is the number of qubits in the circuit of the circuit.
quantumGates is the list of quantum gates of circuit.

1.1 Forbiden function

You does not have the right to use the use the function numpy.kron()

1.2 Given file

You can download the given file here: LINK
The content of the only given file, **quantumComputer.py**:

```
import numpy as np
from enum import Enum


#Enumeration for each type of gate our quantum computer handle
class TypeOfQuantumGate(Enum):
    NOT = 1
    HADAMARD = 2
    CNOT = 3

#The class quantum gate
#TypeOfGate is the type of the gate (a value of TypeOfQuantumGate)
#fQbit is the position in the circuit of the first input Qbit of the gate
#sQbit is the position in the circuit of the second input Qbit of the gate ( if the
    gate has two input)
class QuantumGate:
    def __init__(self, typeOfGate: TypeOfQuantumGate, fQbit: int, sQbit: int =0 ):
        self.typeOfGate = typeOfGate
        self.fQbit = fQbit
        self.sQbit = sQbit


#The main function, you need to program it without using qiskit
#nbQbits is the number of Qbits of the circuit
#QuantumGates is the list of quantum gates of the circuits
#This function output the state vector of the circuit after executing all the gate
def quantumComputer(nbQbits: int, quantumGates: list):
    pass
```

You will have to program in this file.
TypeOfQuantumGate is an **enumeration** for each type of gate our quantum computer handle.
QuantumGate is the class of the quantum gate.
TypeOfGate is the type of the gate (a value of **TypeOfQuantumGate**).
fQbit is the position in the circuit of the first input qubit of the gate.
sQbit is the position in the circuit of the second input qubit of the gate (if the gate has two input). You can add functions to quantum gate.

1.3 Reference

You can find an implementation of this practical work at : LINK. This implementation uses Qiskit and you does not have the right to do so. Yet this can be useful to test your code and

disambiguate what you are ask to do. Your practical work has to behave the same way[1].
The reference raise exception for some type of input. Your quantum computer will never be tested for those kind of input.

1.4 Submission

For the submission you will have to download the repository: LINK. You will have to submit you work before TIME.

2 Generate state vector

For validating this section, your quantum computer will have to handle an **empty** list of quantum gate as input. This function output a vector of $2^{nb}Qbit$ values, with the first value being equal to one and the other values must be equal to zeroes. The function must pass this test (no assert must be thrown):

```
arr = np.array([1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j])
comparison = np.array_equal(arr, quantumComputer(3,[]))
assert(comparison)
```

Tips 1: This code does not throw any error assert

```
 assert(1. == 1. + 0.j)
```

Tips 2: checks the function numpy.zeros ;)

3 Kronecker product

To validate this section you will have to program the function **kroneckerProduct**. This function takes in input to matrix m1 and m2. This function ouput the Kronecker product of m1 and m2. If you do not remember who does work the Kronecker product you can check: LINK
The function must pass this test:

```
m1 = [1,2]
m2 = [[4,5],[6,7]]
assert(np.array_equal(kroneckerProduct(m1,m2) , [[ 4,  5,  8, 10],[ 6,  7, 12, 14]]))
```

4 Handle one gate having one input

The goal of this section is to make your quantum computer handle one gate with one input.

4.1 Iterative kronecker product

To do so, you will have to program the function computeMatrix. This function takes in input an integers : **nbQbit**, a 2x2 matrix, and an integer **fQbit**. This function output:

$$I_2^{\otimes(nbQbit-fQbit-1)} \otimes matrix \otimes I_2^{\otimes fQbit}$$

. The function must pass this test:

```
m = [[0,1],
     [1,0]]
arr = [[0., 1., 0., 0.],
 [1., 0., 0., 0.],
 [0., 0., 0., 1.],
 [0., 0., 1., 0.]]
assert(np.array_equal(createMatFromBaseMatrix(m,2,0), arr))
```

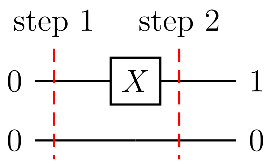
[1]When it comes to float value the values will be compared using the NumPy function isclose

4.2 Handle one not gate

Now you have all the key in hand to handle one not gate. To do so, if there is a NOT gate the circuit you will have to compute its corresponding matrix according to the number of qubits in the circuit and its position in the circuit. Then you will just have to multiply the transposition of your state vector with this matrix:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Here the evolution of the state vector during the computation:



At step 1 the state vector is equal to:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

At step 2 the state vector is equal to:

$$(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}) \times \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^t = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^t = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix}$$

Your quantum computer must pass this test:

```
arr = quantumComputer(2,[QuantumGate(TypeOfQuantumGate.NOT,0)])
assert(np.array_equal(arr,[0., 1., 0., 0.]))
```

4.3 Handle one Hadamard gate

Now you will have to handle one Hadamard gate. To compute the corresponding matrix of an Hadamard gate you just have to act like for the NOT gate, except that you use the Hadamard matrix as a 2x2 matrix:

$$\frac{1}{\sqrt{2}} \times \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Your quantum computer must pass this test:

```
arr = quantumComputer(1,[QuantumGate(TypeOfQuantumGate.HADAMARD,0)])
assert(np.isclose(arr,[1/math.sqrt(2), 1/math.sqrt(2) ]).all())
```

5 Handle many gates

For validating this section, your quantum computer must handle many gate. To do so first you have to compute the corresponding matrix of every gate and then multiply them in the reverse order[2] to get the matrix corresponding to you circuit.
Your quantum computer must pass this test:

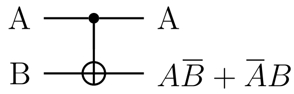
```
arr = quantumComputer(1,[QuantumGate(TypeOfQuantumGate.NOT,0),
QuantumGate(TypeOfQuantumGate.HADAMARD,0)])
assert(np.isclose(arr,[1/math.sqrt(2), -1/math.sqrt(2) ]).all())
```

[2] If as input you have such list of quantum gate : [gate A, gate B ,gate c], you have to compute: mat C X mat B X mat A.

6 Handle CNOT gate

6.1 Compute CNOT matrix

Program the function **computeCNOT**. This function takes in input an integers: **nbQbit**, an integers: **fQbit** and an integer: **sQbit**. Consedering **fQubit** as the control qubit[3]. This function ouputs the corresponding CNOT matrix. To compute this matrix there is an elegant way you can find here: **click here**. Yet this way is complicated, so we will explain a simpler, less elegant way.
For two bit in the classical computing world the CNOT gate act this way:



And has this truth table:

Input	Output
00	00
01	01
10	11
11	10

So in our quantum computing world, for 2 qubits, we have a such "truth table":

Input	Output
$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$	$\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$
$\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$	$\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$
$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$	$\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$
$\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$	$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$

So we have a such corresponding matrix:

[3] fQbit and SQbit must be different. Yet your work will never be tested with fQbit = sQbit

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Here a pseudo-code algorithm that computes the corresponding matrix:

```
    computeCNOT(nQubits, fQubit,sQubit):
    create  a (2**nbQubits) * (2**nbQubits) matrix m, filled with 0.
    for i in 0, 2**nbQubits :
        s = tobinarystring(i)
        if(i[fQubit] == 1):
            s[sQubit] = !s[sQubit]
            m[i,s] =1
        else:
        m[i,i] =1
    return m
```

Your function computeCNOT must pass this test:

```
    arr = [[1., 0., 0., 0.],
 [0., 0., 0., 1.],
 [0., 0., 1., 0.],
 [0., 1., 0., 0.]]
assert(np.array_equal(computeCNOT(2,0,1) , arr))
```

6.2 Back to our quantum computer

Now you have all you need to handle CNOT gate in our quantum computer. Just to do it! Your quantum computer must pass this test now:

```
arr =quantumComputer(2,[QuantumGate(TypeOfQuantumGate.NOT,1),
    QuantumGate(TypeOfQuantumGate.CNOT,1,0)])
assert(np.array_equal(arr , [0,0,0,1]))
```

7 Compute Probability

Program the function **computeProbability**. This function takes in inputs an array of complex numbers: **tab**. This function output an array of the same dimension. Every coefficient of the output is computed this way:

$\forall\, i \in [0,\ length(tab)[$

$$output[i] = \frac{|tab[i]|^2}{\Sigma_{k=0}^{tab.length()-1}|tab[k]|^2}$$

This function must pass this test:

```
arr =quantumComputer(1,[QuantumGate(TypeOfQuantumGate.NOT,0),
    QuantumGate(TypeOfQuantumGate.HADAMARD,0)])
assert(np.isclose(computeProbability(arr),[0.5,0.5]).all())
```

This function computes the probability of each combination to comes out when you mesure every qubit.