QCOMPPW Intro: Programming Warm-up

Lotus Noir Quantum Computing Research Group

Based on:
Qiskit textbook & numpy documentation

Practical Outline: During this practical we will quickly get familiar with Python and Qiskit

Contents

1 Introduction

To properly simulate quantum behaviours using a classical computer and Python 3, we are going to need two libraries. The first one is **numpy**, providing numerical computing tools like linear algebra. The second one, **qiskit** [kiss-kit], is an open source SDK for working with quantum computers at the level of pulses, circuits and algorithms.

Please mind 1.1. For Windows users, we recommend using Anaconda, which already provides an IDE and several packages like numpy. You will, though, need to install Qiskit by hand (*click this link*). For Unix user: 'pip3 install qiskit numpy' does the trick, as long as you have pip installed on your device. We still provide you a 'requirements.txt' file, if you want to work with 'pipenv' and virtual environments (which is a good practice).

Do not worry, this practical aims to be short and efficient. The main goal is to code simple functions, which are a pretext to packages installation. Please be thorough, and do not hesitate to ask for help.
Each practical is graded, and this one is a good opportunity to start properly while quantum computing might seem very straightforward. Of course, cheating and code sharing are not permitted. This rule applies to the upcoming sessions as well.

1.1 Submission

Your code will need to be **submitted using git**, on the repository given to you. The only files that matter to us are the given Python files to complete, and these are the only files that you should 'git add'.
Every practical has a deadline that is approximately 48 hours after the class. Today's one is on **Thursday, the 22nd of October, at 23h59**. At this exact time, your repositories are automatically cloned by our almighty scripts. Only the very last commit is taken into account. The grade is 70% about code results (compared to the reference), and 30% about code quality (a normalized opinion about your code from the teaching assistants).

1.2 Given files

You can download the given files to complete all along the next exercises here: LINK. If you are using Unix, you can use **pipenv** to install the needed libraries:

```
pipenv shell
```

At the root of your work directory. It performs a 'pip3 install -r requirements.txt' for you and create a virtualenv to work with. For Windows users, the similar command can be performed using a terminal, but we understand that 'pipenv' or 'pip3' installation and terminal use can be a little less natural than on Unix. For this purpose, we encourage you to use **Anaconda**, that you should have installed after checking on the early Slack channel's *#atelier-quantique* instructions.

1.3 Imports

Through these exercises, you will be asked to use **numpy** and, at the end, **qiskit**. They are the only allowed imports.

```
1 import numpy as np
2 from qiskit import *
```

1.4 Documentation

numpy: https://numpy.org/doc/1.19/
qiskit: https://qiskit.org/documentation/

2 Numpy

Please mind 2.1. Numpy is one of the top-used Python library, especially among data scientists and computer engineers. These following exercises aim to show you the power of this tool, and the wide range of functions it gives you. Even if some useful tips are given to you each time, do not hesitate to take a glimpse at the documentation.

All along these exercises, tricky cases (empty list, None, equal number of values...) won't be tested. The bonus exercise is worth extra 2 points.

2.1 Exercise 1: Nearest Value

This function should return the nearest value in a numpy array to x. **Using loops is forbidden**, you should use numpy functions. This exercise *can* be done by filling-up a single line. Take a look at the 'argmin' function, and remember you should return X, not its index.

```
arr = np.random.uniform(0,1,10) // arr == [0.73592 0.81708 0.22248 0.40447 0.22499]
x = 0.5
m = nearestValue(arr, x) // m == 0.40447
```

2.2 Exercise 2: Most frequent

This function should return the most frequent value in an n-dimensional numpy array (can be an array, a matrix...). **Using loops is forbidden**, you should use numpy functions. This exercise *can* be done in one line. Take a look at 'flat' and 'bincount'. Maybe that, if the function 'arg**min**' exists, you can get the intuition of another existing function...

```
arr = np.random.randint(0,3,(3,4)) // arr == [[0 1 0 1]
                                              [1 1 1 0]
                                              [2 1 0 1]]
m = mostFrequent(arr) // m = 1
```

2.3 Exercise 3: Kronecker Product

2.3.1 Without loops (using numpy)

This function should return the Kronecker product between two matrices. **Using loops is... forbidden**, you guessed it. As always, think about numpy functions, that is what this practical is about. If you don't remember what the Kronecker product is, below is a quick recap. Further details are given is the textbook.
Given A a matrix $m * n$ and B a matrix $p * q$. The Kronecker product symbolized $A \otimes B$, of size $mp * nq$, is defined as such:

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}$$

```
m1 = [[1,2]]
m2 = [[4,5],[6,7]]
res = kroneckerProduct(m1,m2) // res == [[4,  5,  8, 10],[ 6, 7, 12, 14]]
```

2.3.2 [BONUS] without numpy

From now, you should all be starving for loops / list comprehension. Go ahead, and recode the Kronecker product without the help of numpy. You only realize how much you love someone when you lose him, and this way you may truly understand the power of numpy The Great.

3 Qiskit

Please mind 3.1. Qiskit accelerates the development of quantum applications by providing the complete set of tools needed for interacting with quantum systems and simulators. It is one of the preferred choices for quantum computing in Python.

3.1 Introduction to Qiskit

This exercice will be pretty straightforward. We do not want you to get scared by quantum computing, and we will pretty much give you a hint for every single line of your code. So do not worry, and let's dive into it.

Please mind 3.2. Here are shortcuts link for this exercise:

1. *Single qubits gates*
2. And if you get the curiosity: *here is how to create more complex circuits*

To get more familiar about the Python library, here is a code sample for the **NOT** function. Try to understand it and, why not, run it. Keep in mind that the input is a string, not an integer.

```
from qiskit import *

def NOT(input):

    q = QuantumRegister(1) # a qubit in which to encode and manipulate the input
    c = ClassicalRegister(1) # a bit to store the output
    qc = QuantumCircuit(q, c) # this is where the quantum program goes

    # We encode '0' as the qubit state |0>, and '1' as |1>
    # Since the qubit is initially |0>, we don't need to do anything for an input of
    '0'
    # For an input of '1', we do an x to rotate the |0> to>|1>
    if input=='1':
        qc.x( q[0] )

    # Now we've encoded the input, we can do a NOT on it using x
    qc.x( q[0] )

    # We extract the |0>/|1> output of the qubit and encode it in the bit c[0]
    qc.measure( q[0], c[0] )

    # We'll run the program on a simulator
    backend = Aer.get_backend('qasm_simulator')
    # Since the output will be deterministic, we can use just a single shot to get it
    job = execute(qc,backend,shots=1)
    output = next(iter(job.result().get_counts()))

    return output
```

3.2 Exercice 4: XOR

Your goal is to fill out the **XOR** function code sample. Remember that, if $\{|0\rangle, |1\rangle\}$ are the only allowed input values for both qubits, then the TARGET output of the CNOT gate corresponds to the result of a classical XOR gate.

Before		After	
Control	**Target**	**Control**	**Target**
$\|0\rangle$	$\|0\rangle$	$\|0\rangle$	$\|0\rangle$
$\|0\rangle$	$\|1\rangle$	$\|0\rangle$	$\|1\rangle$
$\|1\rangle$	$\|0\rangle$	$\|1\rangle$	$\|1\rangle$
$\|1\rangle$	$\|1\rangle$	$\|1\rangle$	$\|0\rangle$

Figure 1: CNOT truth table