QCOMPPW102: Qiskit practical work

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

Practical work Outline: During This practical work we will learn how to program on a quantum computer by using **Qiskit**.

Contents

1 Introduction **2**
1.1 Documentation 2
1.2 Given file 2
1.3 Python requirements 2
1.4 Submission 2
1.5 Debugging 2

2 Example **3**

3 Bell state **4**

4 AND gate **4**

5 Quantum Fourier transformation for 3 qubits **4**

6 Deutsch problem **6**

7 General quantum Fourier transformation **6**

8 Deutsch-Joza algorithm **7**

9 Bonus: execute your code on a real quantum computer **8**

1 Introduction

For this practical work, you will have to program by using **Python3**. The only librairies you will need are **Qiskit** and **Numpy**.
If you can't download **Qiskit** with Pycharm, you can program online with the **IBM quantum lab**. Click on the link, create an account and then choose quantum lab.

1.1 Documentation

You will find here all the information on **Qiskit** you would need: `https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.html`

1.2 Given file

You can download the given file **here**.

1.3 Python requirements

The Python requirements are the same of the previous workshop and you can install them with the given **requirements.txt**.

1.4 Submission

Your code will need to be **submitted using github**, on this **assignment**. The only files that matter to us are the given Python files named exercise_*.py, and these are the only files that you should 'git add'.
Every practical has a deadline. Today's one is on **Sunday the 12*th* of december at 23:59**. The grade is determined by the script **run_test_suite.py**. To run it through Pycharm, you have to open it and click on the green arrow:

```
1  from big_file import print_the_key
2
3  if __name__ == '__main__':
4      print_the_key()
```

Please don't mind the red underlining of big_file. It will output something like this:

```
Your mark is:  0
Your proof of work is: 0,9967204840567662
```

You will have to put the proof of work in this **Excel file**. Your proof of work represent your mark, so update it every time you have a better mark. You also have to push your corresponding work on GitHub.

1.5 Debugging

Do not try to debug by looking at the output of the circuit, but debug by printing the circuit. You can print a **Qiskit** circuit named **circuit** this way:

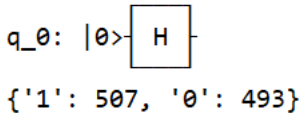
```
print(circuit.draw())
```

2 Example

This section is given as an example, you do not have to submit it. The solution to this exercise is given below.
For validating this section, you have to program the function **example**. This function takes no input. This function output a one qubit **qiskit** circuit having one Hadamard gate. You need to program it in the file: **example.py**.
Here is the content of **example.py**:

```
import math
import numpy as np
from qiskit import QuantumCircuit, execute, Aer


def example():
    pass

circuit = example()


print("the circuit:",circuit.draw())

sim = Aer.get_backend('statevector_simulator')
job = execute(circuit, sim)
arr = job.result().get_statevector(circuit)
assert(np.isclose(arr,[1/math.sqrt(2), 1/math.sqrt(2)]).all())


circuit.measure_all()
simulator = Aer.get_backend('qasm_simulator')
job = execute(circuit, simulator, shots=1000)
result = job.result()
counts = result.get_counts(circuit)
print("Execution of a quantum computer:\n",counts)
```

When you have solved the exercise, executing this file should print something close to this:



The circuit has to be exactly the same, yet it is not the case of the execution on simulation of quantum computer. Indeed the value only have to be close (ex: {'1': 485, '0': 515}, {'1': 492, '0': 508}, {'1': 503, '0': 497}).
Here is the function example when you have solved the question:

```
def example():
    circuit = QuantumCircuit(1)
    circuit.h(0)
    return circuit
```

3 Bell state

For validating this section you have to program the function **bellState** in the file **bellState.py**. This function takes no input. This function output a 2 qubits **Qiskit** circuit. The qubits **bellState** outputs must be in a bell state. A bell state is the simplest example of quantum entanglement. It means that by knowing the value of one qubit you can deduce the value of the other one.

After executing your script, it should output:

```
It is a bell state!
{'00': 498, '11': 502}
```

Or :

```
It is a bell state!
{'01': 519, '10': 481}
```

4 AND gate

For validating this section you have to program the function **AND** in the file **AND.py**. This function takes no input. This function output a 3 qubits **qiskit** circuit. This circuit must compute an AND between the first and second qubits. The value of the second and third qubits must be 0 or 1 (both are fine).
After executing **AND.py**, it should print:

```
Best 'AND' EU!
```

5 Quantum Fourier transformation for 3 qubits

For validating this section you have to program the function **qft** in the file **QFT3.py**. This function takes no input. This function output a 3 qubits **Qiskit** circuit. This circuit must compute a quantum Fourier transform. A Fourier transform performs a change of basis on numbers. This change of basis has many uses, for example, it is used in digital image processing. The complexity of this algorithm scales exponentially on a classical computer, yet fortunately, it scales in a quadratic way on a quantum computer.
For this exercise, you will have to use the gate: **CU1**. This gate does not change the state $|00\rangle$, $|01\rangle$ and $|10\rangle$. Yet it transforms the state $|11\rangle$ in $|11\rangle\, e^{i\theta}$. Where θ is an angle in radian.

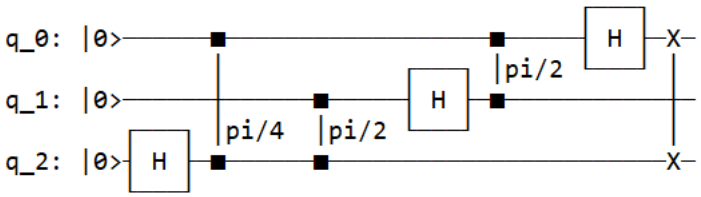
The gate does not change the probability to measure $|00\rangle$, $|01\rangle$, $|10\rangle$, or $|11\rangle$. Nevertheless it changes the quantum phase. Certain types of gate will have a different output according to the quantum phase.
Here is the matrix representation of this gate:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{bmatrix}$$

Here is how to use it on **qiskit** with $\theta = \frac{\pi}{4}$[1]:

```
circuit.cp(pi/4,0,2)
```

The main goal of the exercise is to make you understand how to draw a circuit using **qiskit**. Just focus on the visual representation of your circuit.
Here is the content of QFT3.py:
After this file is executed it should output:



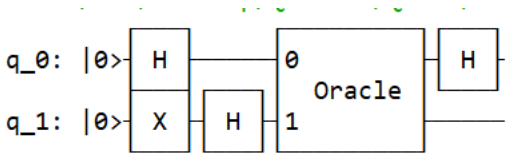

```
You are good at computing quantum Fourier transformation!

{'001': 133, '010': 133, '111': 115, '100': 114, '101': 117, '110': 126,
'000': 118, '011': 144}
```

6 Deutsch problem

For validating this section you have to program the function **deutsch** in the file **deutsch.py**. This function takes as input a **Qiskit** quantum gate: **oracle**. This oracle is supposed to be the Deutsch oracle. This function must output a 2 qubits **Qiskit** circuit, solving the Deutsch problem.
You are not used to Deutsch problem yet, so you should just focus on its visual representation. We provide you the function **dj-oracle**, this is IBM's implementation of the Deutsch oracle. This function takes into input the string "balanced" or the string "constant". This function outputs an oracle which is either constant or balanced according to the string.
To add the oracle to your **Qiskit** circuit use you can act this way:

```
circuit.append(oracle,[0,1])
```

Here is the content of the file deutsch.py:
After executing this file it should print this:



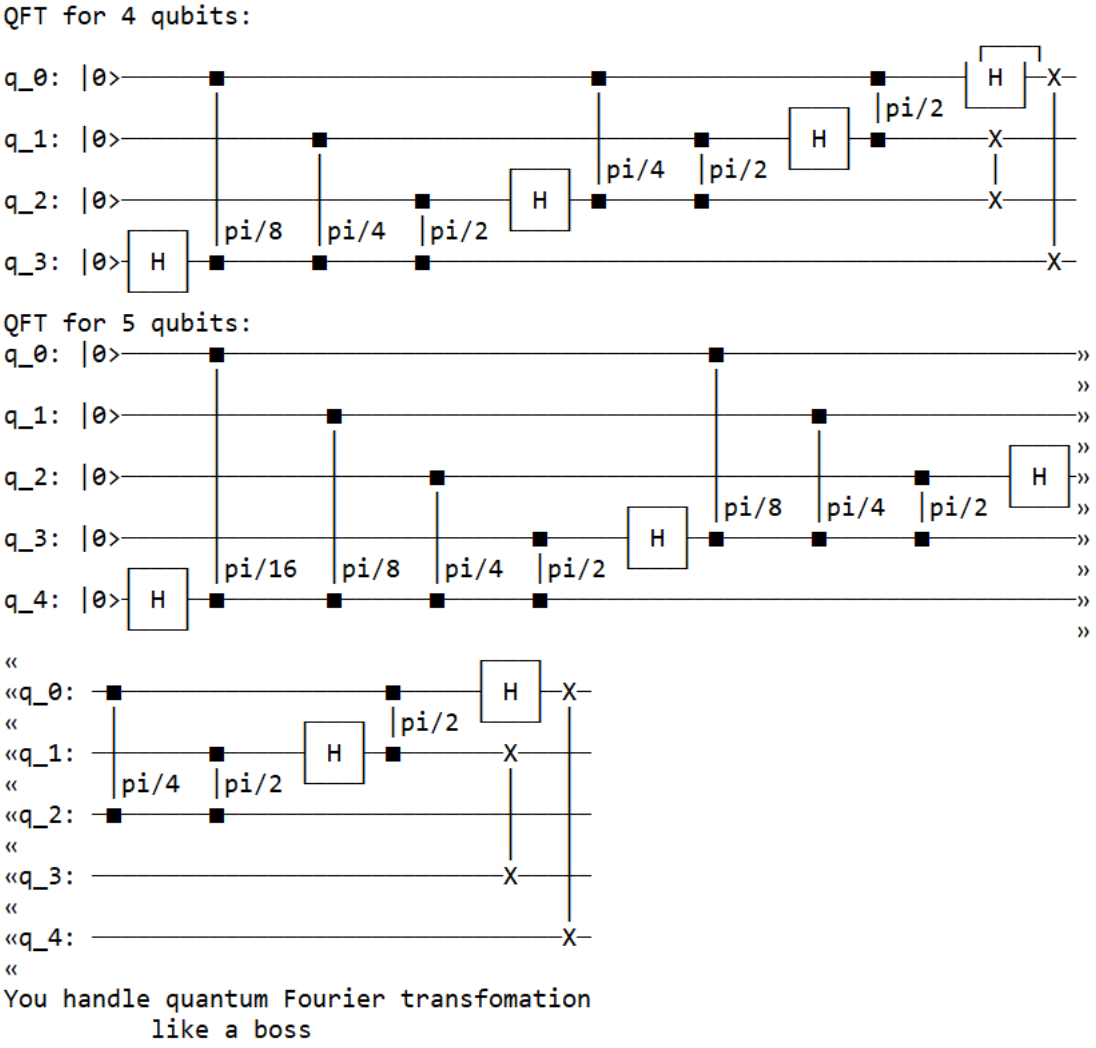

```
Your Deutsch algorithm handle well constant oracle.
Your Deutsch algorithm handle well balanced oracle.
```

7 General quantum Fourier transformation

For validating this section you have to program the function **qft** in the **QFTN.py** file. This function takes as input an integer: **n**. This function outputs an **n** qubits circuit computing the quantum Fourier transformation for **n** qubits. Once again the goal of this section is to make you draw the circuit, so you should rather focus on the emerging visual pattern.

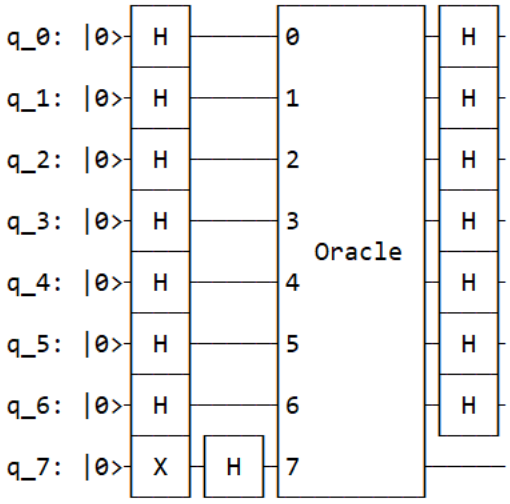
[1] You can use numpy to define the constant pi

After executing the file it should print this:



8 Deutsch-Joza algorithm

For validating this section you have to program the function **dj** in the file **deutsch_joza.py**. This function takes as input an integer: **n** and a **qiskit** quantum gate: **oracle** on **n +1** qubits. This oracle is supposed to be the Deustch-Joza oracle. The function **dj** must output a **n+1** qubits **qiskit** circuit, solving the deutsch problem.
You are not used to the Deutsch-Joza problem, so you just focus on its visual representation. We provide you the function **dj-oracle**, this is an IBM implementation of the Deutsch-Joza oracle. This function takes in input the string "balanced" or the string "constant" and an integer: **n**. This function outputs an oracle which is either constant or balanced according to the string. This oracle is a quantum gate on **n+1** qubits. Here is the content of **deutsch-joza.py**:
After executing the file it should output this:




```
Your Deutsch-Joza algorythm handle well constant oracle
Your Deutsch-Joza algorythm handle well balanced oracle
```

9 Bonus: execute your code on a real quantum computer

In the file `executing_on_real_quantum_computer.py`, you will find what you need to execute **Qiskit** code on a real quantum computer in Santiago. To use it:

- First you have to create an API key at: `https://quantum-computing.ibm.com/`.
- Then put your API key in file `executing_on_real_quantum_computer.py`.
- Execute the file.
- And wait for hours or even more...

I advise you to only execute algorithm needing five or less qubits. This is not evaluated.