Problem Set III - QPROG - HT2023

Mateus de Oliveira Oliveira
Department of Computer and System Sciences
Stockholm University

December 2023

1 Implementing the Quantum Fourier Transform in Qiskit

The Quantum Fourier Transform (QFT) is a cornerstone in the field of quantum computing. It forms the basis of several important quantum algorithms, such as algorithms for integer factorization, for phase estimation, for solving systems of linear equations, and for solving special cases of the hidden subgroup problem. In this assignment, you will be implementing the QFT in qiskit.

1.1 Preparation

You will need to import the `QuantumCircuit` class from the `qiskit` module. It is also a good idea to import the `numpy` module so that you are able to use standard mathematical constants and arithmetic functions.

```
import numpy as np
from qiskit import QuantumCircuit
```

To implement the QFT, you will need the following ingredients.

- Hadamard gates. To add a Hadamard gate on the i-th qubit of the circuit `circuit`, call the method `circuit.h(i)`.
- Controlled Phase Gates. To add a controlled phase gate with phase shift `theta`, control qubit `i` and target qubit `j`, call the method `circuit.cp(theta,i,j)`.
- Swap gates. This gates swaps two qubits. To swap qubits `i` and `j`, call the method `circuit.swap(i,j)`.
- Barriers, to help visualize subblocks in the circuit. To apply a barrier, call the method `circuit.barrier()`.

- A sufficiently good approximation of the number $\pi = 3.1415....$ For instance, you can call `#import numpy as np` and then use `np.pi`.

1.2 QFT acting on the first n qbits

Implement a function `apply_qft(n,circuit)` that takes as input a number `n` and a quantum circuit `circuit` and applies the Quantum Fourier Transform (QFT) on the first n qubits of `circuit`. Recall that the QFT acting on qubits $q_{n-1} \dots q_1 q_0$ qubits is defined as follows.

1. For each j ranging through the sequence $n-1, \dots, 1, 0$, apply a Hadamard gate on qubit j followed by a sequence of $j-1$ controlled phase gates. To apply this sequence, use a variable k to range through the sequence $j-1, \dots, 1, 0$, and then apply the controlled phase gate with phase shift $\pi/2^{j-k}$.
2. For each i in `range(n//2)`, swap qubit i with qubit $n-i-1$. Note that `n//2` is the integer part of `n` divided by `2`.

If everything went well with your implementation, Figure 1 should printed when executing the following code with $n = 3$, while Figure 2 should be printed when executing the code with $n = 4$.

```
n = 3  #n=4
circuit = QuantumCircuit(n,0)
apply_qft(n,circuit)
print(circuit)
print()
```

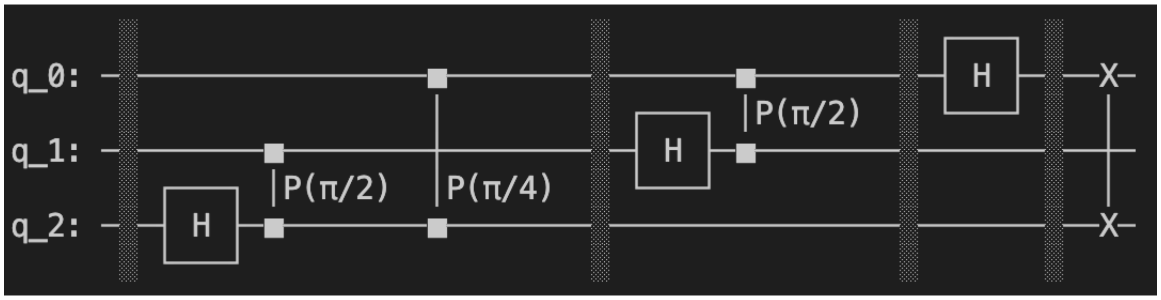

Figure 1: Quantum Fourier Transform on 3 Qubits.

1.3 Inverse QFT

In many applications, we actually need to use the Inverse Quantum Fourier Transform, instead of QFT. Implement a function `apply_iqft(n,circuit)` that takes as input a number `n` and a circuit `circuit` and applies the inverse QFT to the first n qubits of `circuit`.

This can be achieved by applying reversing the order of the gates used in the implementation of `apply_qft(n,circuit)`, and by applying the inverse of each gate, instead of the gate itself.

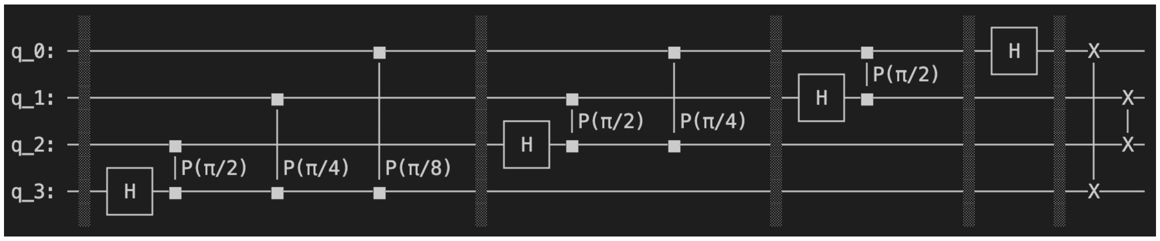

Figure 2: Quantum Fourier Transform on 4 Qubits.

- The inverse of the Hadamard gate is the hadamard gate itself.
- The inverse of a swap gate acting on qubits i and j is the swap gate itself.
- The inverse of a phase gate with shift phase θ i s the corresponding phase gate with shift phase $-\theta$.

If everything went well with your implementation, Figure 3 should printed when executing the following code with $n = 4$. Note the minus signs in the phase shifts.

```
n = 4
circuit = QuantumCircuit(n,0)
apply_iqft(n,circuit)
print(circuit)
print()
```

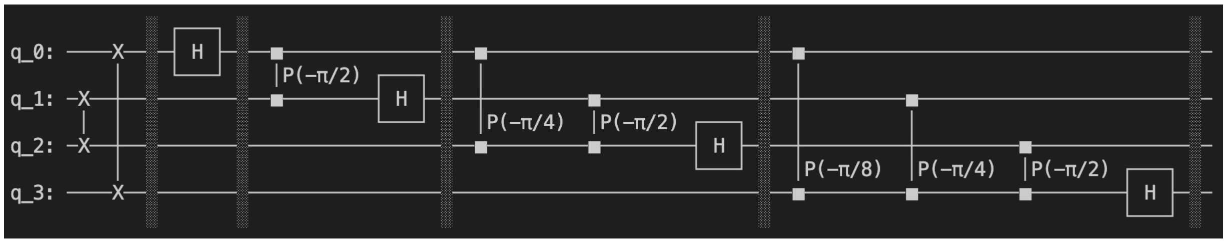

Figure 3: Inverse Quantum Fourier Transform on 4 Qubits.

1.4 Generalizing the functions

In the functions implemented in the previous subsections are applied to the first n qubits of the input circuit. Generalize each of these functions to act on a list of qubits. More specifically, instead of passing the number n as input to the function, pass a list `qubits` of qubit indices. Here, `qubits[0]` plays the role of q_0, `qubits[1]` plays the role of q_1, and so on.

- `apply_qft_gen(qubits,circuit)` takes as input a list `qubits` of qubits and a circuit `circuit`, and applies the QFT to the qubits indexed in the list.
- `apply_iqft_gen(qubits,circuit)` is defined similarly.

2 Converting Classical Circuits to Quantum Circuits

Any Boolean function $f : \{0,1\}^n \to \{0,1\}^m$ can be computed by a classical circuit with n inputs and m outputs. In this exercise, you will show that if such a function f can be computed by a classical circuit C with s (internal and output) gates, then it can also be computed by a quantum circuit Q with $O(s)$ gates. More specifically, this quantum circuit will specify a unitary transformation U_f that acts on three quantum registers: an n-qubit input register, an m-qubit output register, and a t-qubit auxiliary register where $t = s$. One crucial property of this unitary transformation is that for each $x \in \{0,1\}^n$,

$$U_f|x\rangle_n|0\rangle_m|0\rangle_t = |x\rangle_n|f(x)\rangle_m|0\rangle_t. \tag{1}$$

In other words, after applying U_f to the state $|x\rangle_n|0\rangle_m|0\rangle_t$, the first register is sent to $|x\rangle_n$, the second register is sent to $|f(x)\rangle_m$, and the third register is sent to $|0\rangle_t$. Note that, since the auxiliary register is equal to $|0\rangle_t$ both before and after the application of the unitary U_f, irrespectively of the state of the input register, it is often the case that we refer to U_f as an oracle that computes the transformation $|x\rangle_n|0\rangle_m \to |x\rangle_n|f(x)\rangle_m$. Nevertheless, when actually implementing such transformation, we need to take care of the auxiliary bits.

In what follows, whenever clear from the context we may omit the subscript indicating the number of qubits, and write simply $|x\rangle|y\rangle|z\rangle$ instead of $|x\rangle_n|y\rangle_m|z\rangle_t$.

2.1 Circuit Specification

We specify a circuit using a file with the following structure:

```
n_inputs
n_outputs
n_internal
x_1 x_2 ... x_{n_inputs}
y_1 y_2 ... y_{n_outputs}
i_1 i_2 ... i_{n_internal}
a_0 and b_0 c_0  |  a_0 not b_0
a_1 and b_1 c_1  |  a_1 not b_1
...
a_{r-1} and b_{r-1} c_{r-1}  |  a_{r-1} not b_{r-1}
```

More specifically, `n_inputs` is the number of inputs, `n_outputs` the number of output gates, and `n_internal` the number of internal gates. The next three lines are sequences of space separated numbers specifying which are the input, the output and the internal gates respectively. Finally, the remaining `r` lines (`r = n_outputs + n_internal`) specify the gates of the circuit. One gate per line. We assume that all gates in the circuit are either `and`-gates or `not`-gates, and that gates are listed in topological order. That is to say, for each two gates a and a' if a is an input of a' then a is specified before a'.

- `a and b c` specifies that gate number `a` is an `and`-gate with inputs `b` and `c`.
- `a not b` specifies that gate number `a` is a `not`-gate with input `b`.

We also assume that inputs and gates of the circuit are numbered from `0` to `n_inputs+n_outputs+n_internal`-1. For example, the following file specifies a circuit with 3 input gates, 2 output gates, and 2 internal gates. The input gates are $[0, 1, 2]$, the output gates are $[3, 4]$ and the internal gates are $[5, 6]$. Gate number 5 is an `and`-gate whose inputs are gates 0 and 1. Gate number 3 is a `not`-gate whose input is gate number 5. Gate number 6 is a `not`-gate whose input is gate number 2. Finally, gate number 4 is an `and`-gate whose inputs are gates 5 and 6 (See Figure 4).

```
3
2
2
0 1 2
3 4
5 6
5 and 0 1
3 not 5
6 not 2
4 and 5 6
```

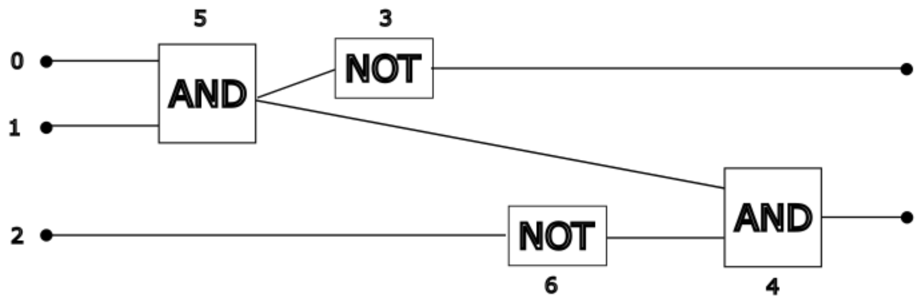

Figure 4: Example of a Classical Circuit.

2.2 The Class ClassicalCircuit

In Code 1 below, you will find the specification of a class `ClassicalCircuit`, which is used to represent classical circuits. This class is also specified in the file `ClassicalCircuitExercise.py`. The first six attributes are self-explanatory. The attribute `self.gates` is a list of gates. Each gate is itself represented as a list. For instance, an `and`-gate is stored in python as a list of the form `[a,'and',b,c]` while a `not`-gate is stored as a list of the form `[a,'not',b]`. In general, a gate `g` with r inputs is represneted as a list of length $r+2$, where the first position is the number of the gate, the second position is a string specifying the type of the gate, and the subsequent positions are the numbers of the gates that are input to g. The method `read(self,filenate)` reads a classical circuit from a file, while the method `print(self)` prints the circuit at the standard output. These two methods are already implemented in the file `"ClassicalCircuitExercise.py"`.

```
class ClassicalCircuit:
    def __init__(self,filename):
        self.n_inputs = 0
        self.n_outputs = 0
        self.n_internal = 0
        self.input_gates = [] # list of input-gate numbers
        self.output_gates = [] # list of output-gate numbers
        self.internal_gates = [] # list of internal-gate numbers
        self.gates = [] # list of gates. Each gate is a list.
        # and-gate: [a,'and',b,c]. not-gate: [a,'not',b].
        self.read(filename)

    def read(self,filename):
        # Reads circuit from file. Already implemented.

    def print(self):
        # Prints the circuit in the terminal. Already implemented.

    def convert_step_1(self,quantumCircuit):
        # First step in the conversion. To implement.

    def convert_step_2(self,quantumCircuit):
        # Second step in the conversion. To implement.

    def convert(self,quantumCircuit):
        # Full conversion. To implement.
```
Code 1: Specification of the class Classical Circuit.

In this exercise, you will be implementing the method `convert(self,quantumCircuit)`, that takes a quantum circuit `quantumCircuit` as input and applies to it the gates corresponding to the unitary transformation U_f, where $f : \{0,1\}^n \to \{0,1\}^m$ is the function computed by the classical circuit represented by the object `self`. The implementation is split into three parts. In the first part, you will implement the auxiliary method `convert_step_1(self,quantumCircuit)`, in the second part you will implement auxiliary method `convert_step_2(self,quantumCircuit)`, and finally, in the third part you will implement `convert(self,quantumCircuit)` by combining the two previously defined auxiliary methods. The implementation of each part will be dealt with in a separate subsection.

2.3 Step 1

The first step of the conversion consists of applying the the gates specified in the classical circuit in the order they appear in the classical circuit. Note that the quantum circuit is assumed to have at least `n_inputs + n_outputs + n_internal` qubits. The number of each input specified in the classical circuit is the number of the corresponding input qubit of the classical circuit. The number of each gate in the quantum circuit is the number of the qubit that will store the output of that result. For instance `[a,'and',b,c]` specifies that the qubit number `a` will store the result of applygin the `and` operation to qubits numbers `b` and `c`.

For simplicity, we assume that the classical circuit is made only of `and` and `not` gates. The conversion from quantum gates to classical gates is done accoding to the following rules.

- For each gate of type `[a,'and',b,c]`, create a Tofolli (CCNOT) gate with control qubits b and c and target qubit a. In Qiskit, such gate is created by calling the method `ccx(b,c,a)` of the `QuantumCircuit` class. Note that qubit a is an auxiliary qubit initialized with $|0\rangle$. When the Tofolloi gate is applied to the state $|x\rangle|y\rangle|0\rangle$, the result is the state $|x\rangle|y\rangle|x \cdot y\rangle$.

- For each gate of type `[a,'not',b]`, first create an X-gate on qubit a to flipt $|0\rangle$ to $|1\rangle$ and then apply, a `CNOT` gate with control bit b and target bit a. Note that qubit a is an auxiliary qubit that is initialized to $|0\rangle$. After applying the X gate to it, we obtain $|1\rangle$. After the application of the CNOT gate, on state $|x\rangle|1\rangle$, the result is $|x\rangle|1 \oplus x\rangle$.

If everything went well with your implementation, the the quantum circuit in Figure 5 should be printed by the following code.

```
cc = ClassicalCircuit("circuit.txt")
n_wires = cc.n_inputs + cc.n_outputs + cc.n_internal
qc = QuantumCircuit(n_wires,0)
cc.convert_step_1(qc)
print(qc)
```

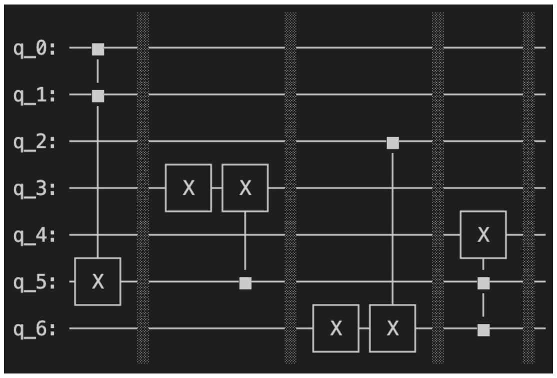

Figure 5: Conversion: Step 1.

2.4 Step 2

The next step consists of implementing a method `conversion_step_2(self,quantumCircuit)` that takes as input a quantum circuit `quantumCircuit` and applies the gates of the the circuit obtained in Step 1 in reverse order.

If everything went well with your implementation, the the quantum circuit in Figure 6 should be printed by the following code.

```
cc = ClassicalCircuit("circuit.txt")
n_wires = cc.n_inputs + cc.n_outputs + cc.n_internal
qc = QuantumCircuit(n_wires,0)
cc.convert_step_2(qc)
print(qc)
```

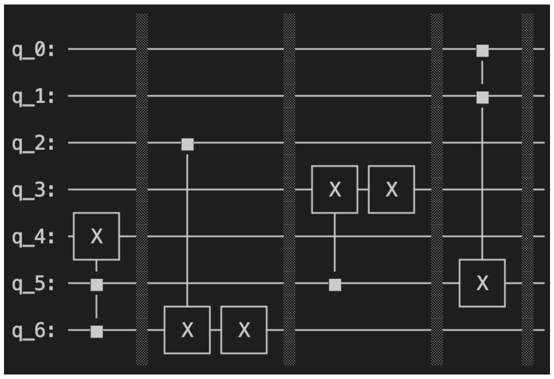

Figure 6: Conversion: Step 2.

2.5 Step 3

Finally, in step number 3, we combine the methods implemented in the last two steps to obtain a circuit implementing the unitary transformation U_f. This is achieved by applying the following steps.

- Apply the gates of the circuit constructed by the method `conversion_step_1`.
- For each output gate a, copy the bit in a to a fresh qubit `a_prime`. This is done by applying a CNOT gate with control a and target `a_prime` (initialized to $|0\rangle$). We assume that the i-th new qubit stores the bit in the i-th output bit of the circuit (where the order is specified by the list `output_qubits`).
- Finally, apply the gates of the circuit constructed by the method `conversion_step_2`.

Intuitively, the initial state of the overall circuit is $|x\rangle_n|0\rangle_m|0\rangle_t$ for some $x \in \{0,1\}^n$, where `t = m + r` for `m=n_outputs` and `r=n_internal`. Here, the second register is the output register of the quantum circuit, while the third register is an auxiliary register. This auxiliary register is used to store the outputs of the gates of the classical circuit. For this reason, there are `n_outputs+n_internal` qubits in this register. After the application of the circuit specified in Step 1, we obtain the state $|x\rangle_n|0\rangle_m(|y\rangle_r|f(x)\rangle_m)$. Note that the qubits corresponding to the m output qubits of the classical circuit are now at state $|f(x)\rangle$, while the qubits corresponding to the internal gates are in state $|y\rangle$ for some y that depends on x. After the application of the intermediate step, the output qubits of the classical circuit are copied to the output qubits of the quantum circuit giving rise to the setate $|x\rangle_n|f(x)\rangle_m(|y\rangle_r|f(x)\rangle_m)$.

Finally the computation in the auxiliary register is undone by applying the circuit speficied by `conversion_step_2`. This yields the state $|x\rangle_n|f(x)\rangle_m(|0\rangle_r|0\rangle_m)$.

If everything went well with your implementation, the the quantum circuit in Figure 7 should be printed by the following code.

```
cc = ClassicalCircuit("circuit.txt")
n_wires = cc.n_inputs + 2*cc.n_outputs + cc.n_internal
qc = QuantumCircuit(n_wires,0)
cc.convert(qc)
print(qc)
```

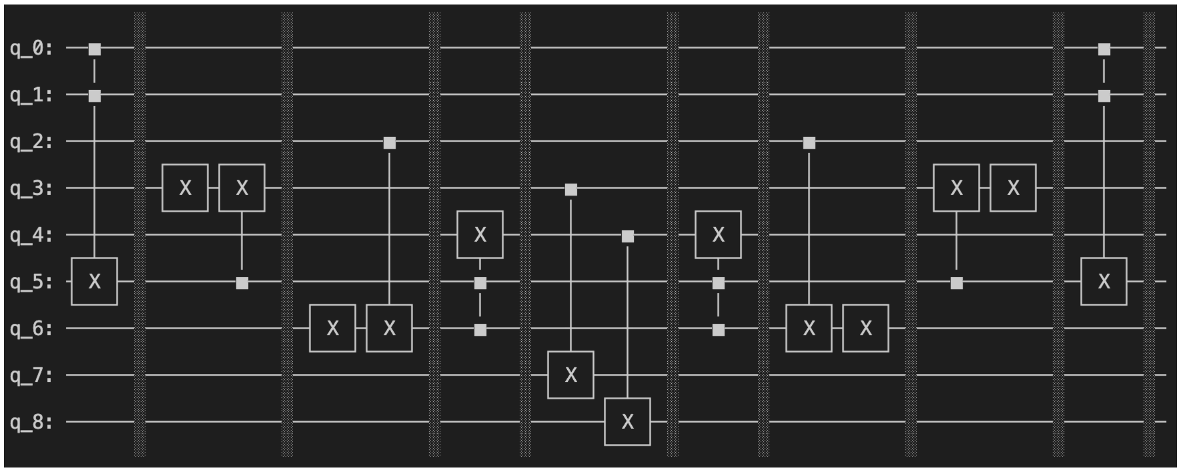

Figure 7: Full Conversion.

2.6 Controlled U_f

In many applications, we need to have a way of controlling when the unitary U_f will be applied. In this exercise, you will adapt the methods implemented in the previous sections to achieve this goal. The adaptation is in principle straightforward: for each gate `g` used in the definition of the circuit computing U_f, apply the gate `controlled-g`, instead of g. All having the same control bit.

More specifically define the following methods which apply the controlled versions of the unitaries defined in by the methods `convert_step_1`, `convert_step_2` and `convert` respectively. In each of these methods, qubit i is assumed to be the control bit (i is assumed to be distinct from all qubits used by the `convert` method.).

- `convert_contr_step_1(self,quantumCircuit,i)`
- `convert_contr_step_2(self,quantumCircuit,i)`,
- `convert_contr(self,quantumCircuit,i)`.

2.7 Bonus - Extend the Set of Basis Gates

In the previous exercises, we have assumed that the classical circuits are defined over the basis `{and,not}`. Remove this assumption by dealing with the gates of type `{or,xor,nand}`. This can be done by the application of a suitable sequence of `X` and `CNOT` gates.