Implementation of Quantum Fourier Transform

Seong Axel Cho

Computer Science Department, Vanderbilt University
Nashville, TN USA

seong.g.cho@vanderbilt.edu

***Abstract*— Quantum Fourier Transform is implemented at IBM Quantum Lab using Qiskit library. The implementation is done with a manual step by step method, then with a build function, then with a Qiskit's QFT operator. This Fourier Transformation circuit represents an invertible operation, which would be presented with a simulator. Finally, this circuit will be showcased inside a working Shor Algorithm machine.**

***Keywords*— Quantum Fourier Transform, IBM Quantum Lab, Qiskit, Invertible Operation, Shor Algorithm**

I. Introduction

Fourier Transform in math is transforming a function into a form that describes the frequency of the original function. [1] One of typical representations of the Fourier transform would look as follows.

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(x)\, e^{-2\pi i \xi x}\, dx.$$

This formula would convert any original function on the x axis into a function on a vector space with a cyclic nature.

There are a lot of variations of the original Fourier Transform, and both the original form and the variations are used heavily by engineering and other applied science field, as transforming of a function into a cyclic form would give a very strong tool for detecting and analyzing hidden properties, such as mixed frequencies in audiology.

Shor first noticed that this period finding nature of the Fourier Transform can be used in quantum computation to find hidden patterns, and suggested this as a step inside his Shor Algorithm, which needs to find a period out of the remainder function to factor an integer.[2] Later, Coppersmith wrote out the detailed math for this transformation.[3]

The Quantum Version of the Fourier Transform as introduced by Coppersmith can be written in the following format.[4]

$$|j\rangle \mapsto \frac{1}{2^{n/2}} \sum_{k=0}^{2^n-1} e^{2\pi i jk/2^n} |k\rangle$$

II. Implementation

In implementation, the circuit for Quantum Fourier Transform can be built in 3 parts:

1) The Hadamard gates to put the qubits to Quantum states.

2) The controlled phase rotations to shift the cyclic properties.

3) The swap gates at the end to rearrange the qubits.[5] [6]

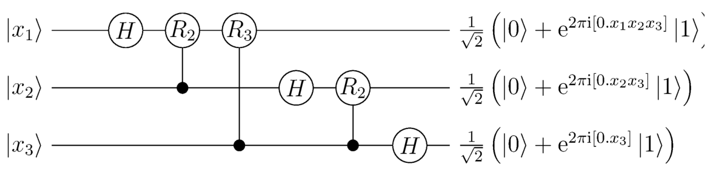
$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad \text{and} \quad R_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{i2\pi/2^k} \end{pmatrix}$$

A. Three Qubit Implementation

QFT with 3 qubit can be described as follows:

$$\mathrm{QFT} : |x\rangle \mapsto \frac{1}{\sqrt{8}} \sum_{k=0}^{7} \omega^{xk} |k\rangle,$$

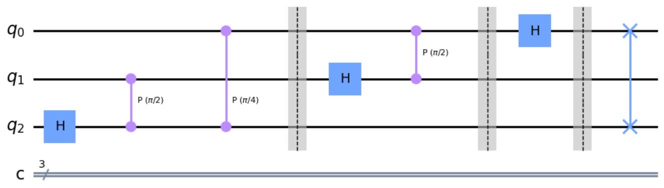
where w requires a rotation of the qubits. The resulting circuit diagram is


$|x_1\rangle$
$|x_2\rangle$
$|x_3\rangle$
H
R_2
R_3
$\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i[0.x_1x_2x_3]}|1\rangle\right)$
$\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i[0.x_2x_3]}|1\rangle\right)$
$\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i[0.x_3]}|1\rangle\right)$

With Qiskit, the rotations can be implemented with cp gates cp(), and hadamard gates can be implemented with h(). The actual write up of the code would be as follows.

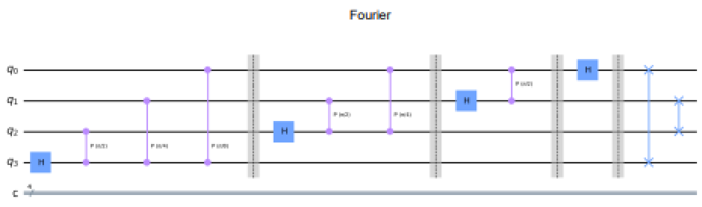
```
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
from qiskit import IBMQ, Aer, transpile
from qiskit.tools.monitor import job_monitor
from qiskit.visualization import plot_histogram
from qiskit.circuit.library import QFT
import numpy as np
from math import gcd

pi = np.pi


# build a Quantum Fourier Transformer for 3 qubits

# initialize the circuit with 3 qubits
q = QuantumRegister(3, 'q')
c = ClassicalRegister(3, 'c')
q3 = QuantumCircuit(q, c)

# the bottom qubit at qubit[2], hadamard, then controlled phase rotation on 0, 1
q3.h(2)
q3.cp(pi/2, 2, 1)
q3.cp(pi/4, 2, 0)
q3.barrier()

# qubit[1], hadamard, then controlled phase rotation on 0
q3.h(1)
q3.cp(pi/2, 1, 0)
q3.barrier()

# qubit[0], hadamard only, as it does not have any more qubits to control
q3.h(0)
q3.barrier()

# swap top and bottom till we reach the center
q3.swap(0,2)

# draw the circuit that worked so far
q3.draw()
```

The resulting circuit built by this code block is as follows.



B. Four Qubit Implementation

Four qubit implementation is almost identical with three qubit implementation, with one more extra step. The details will be in the attachment. The circuit will yield following circuit diagram:

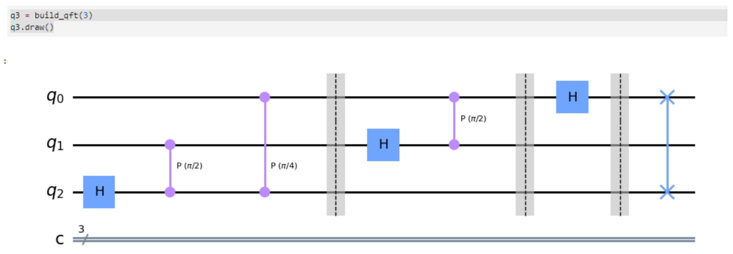


C. Building with Functions

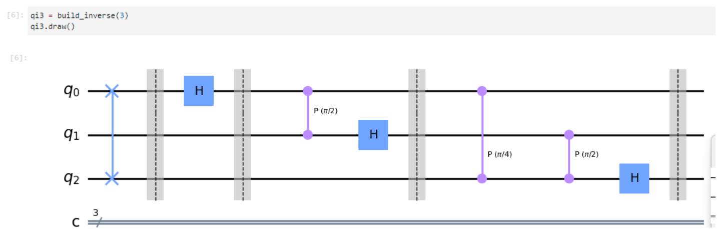
As the building process itself is repetitive and iterative operations in 3 steps, it can be built into a function. Also, the inverse function is just the opposite order of building, and it can be captured into another function. These 2 functions are as follows.

```
 # build the transformer
def build_qft(numbers):
    q = QuantumRegister(numbers, 'q')
    c = ClassicalRegister(numbers, 'c')
    circuit = QuantumCircuit(q, c)

    k = numbers
    while (k > 0):
        k = k -1

        # hadamard gate
        circuit.h(k)
        rotation = pi
        j = k

        while (j > 0):
            j = j -1
            rotation = rotation/2

            # controlled phase gate
            circuit.cp(rotation, k, j)
        circuit.barrier()

    top = 0
    bottom = numbers -1

    while (top < bottom):
        # swap the qubits
        circuit.swap(top, bottom)
        top = top + 1
        bottom = bottom -1


    return circuit

# build inverse version
def build_inverse(numbers):

    q = QuantumRegister(numbers, 'q')
    c = ClassicalRegister(numbers, 'c')
    circuit = QuantumCircuit(q, c)

    # swap first
    high = numbers -1
    low = 0

    while (high > low):
        circuit.swap(high, low)
        high = high -1
        low = low + 1

    circuit.barrier()

    k = 0

    while (k < numbers):
        j = 0
        divide = 2 ** k

        while (j < k):
            rotation = pi/divide

            # controlled phase rotation
            circuit.cp(rotation, k, j)

            j = j + 1
            divide = divide/2

        # hadamard gate
        circuit.h(k)

        k = k + 1

        circuit.barrier()

    return circuit
```

Testing these functions, 3 qubit circuit and 3 qubit inverse circuit are built as follows.



```
q3 = build_qft(3)
q3.draw()
```

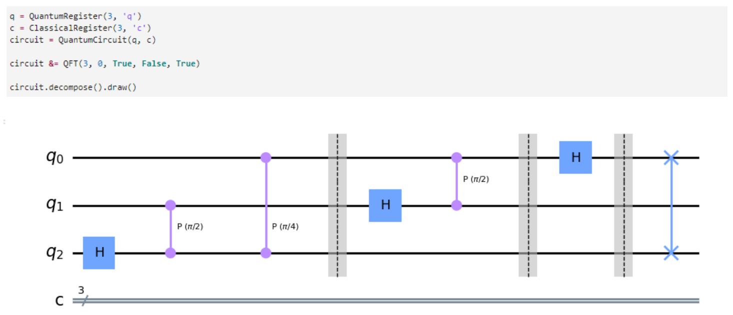


```
[6]: qi3 = build_inverse(3)
     qi3.draw()
```

D. Build with an operator

While building a QFT and inverse QFT can be written as above, there is already a pre-packaged QFT operator in Qiskit.[7]

Building with this operator shows an identical result with the building with manual steps, or building with build functions, except that we should apply "decompose()" to see inside the operator.



```
q = QuantumRegister(3, 'q')
c = ClassicalRegister(3, 'c')
circuit = QuantumCircuit(q, c)

circuit &= QFT(3, 0, True, False, True)

circuit.decompose().draw()
```

E. QFT Class

The attached source code includes a class object that would include all the operations mentioned above: building QFT and inverse QFT using Qiskit's operator, and alternatively building them using the functions introduced above. This class is to be used for all the illustrations below.

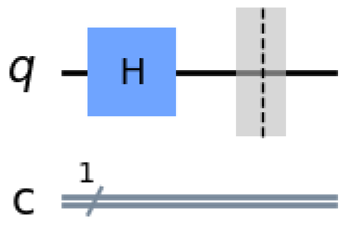
III. Properties of QFT

These QFT circuits show a few interesting properties. Firstly, a single qubit QFT would be simply identical to a hadamard gate. As there is only a single qubit, there would not be any controlled rotation, neither will there be any swapping. So the only element in the circuit building should be just a hadamard gate.

It is demonstrated as follows.

```
q1Class = FourierClass(1)
q1 = q1Class.build_manually()
q1.draw()
```

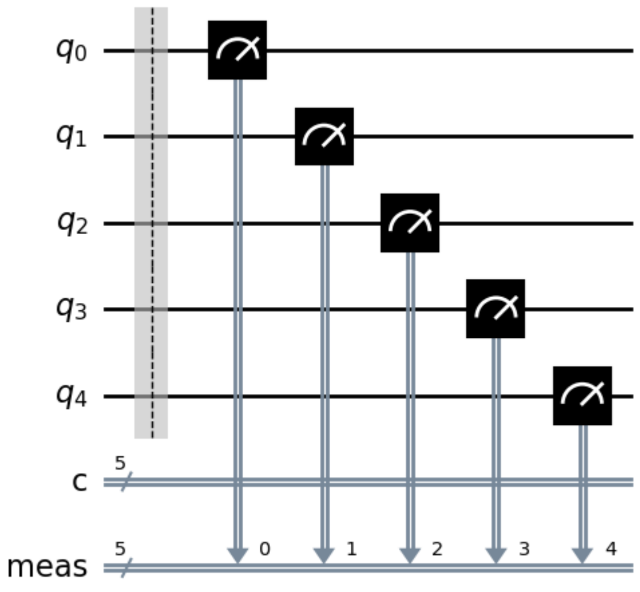
]:



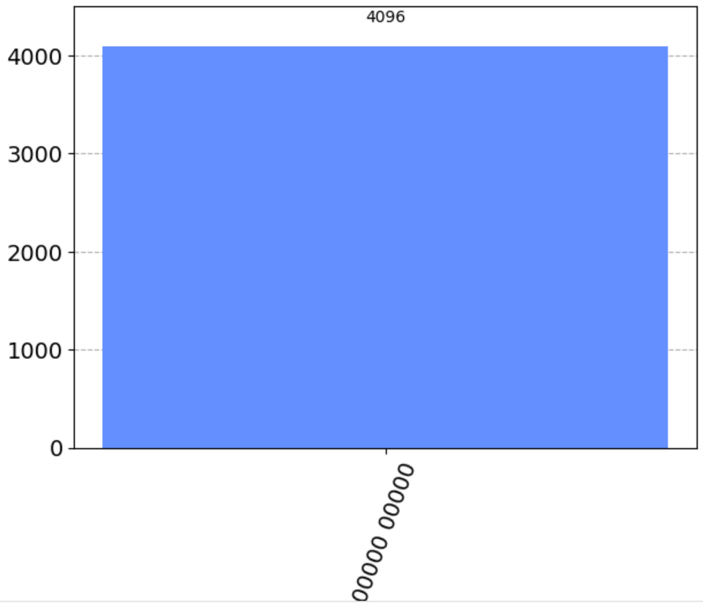
On the other hand, QFT as shown above has an inverse operation, and when a QFT is combined with an inverse QFT with the same number of qubits, it would result in an identity matrix.

First, using the QFT class and getting the qubit right after the initializing would get a circuit that would not do anything.

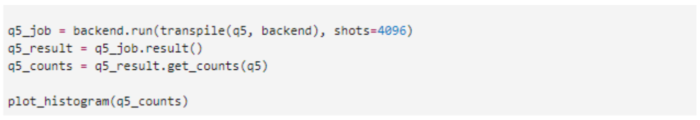
```
raw = FourierClass(5).get_circuit()
raw.measure_all()
raw.draw()
```



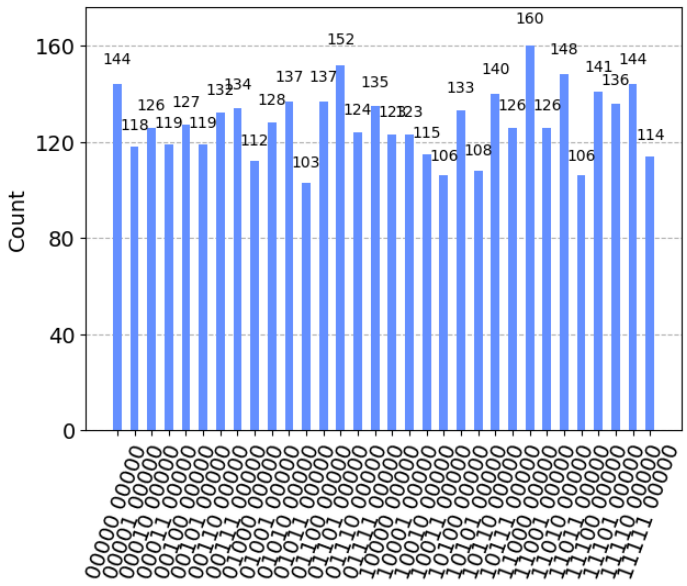
Running a simulation on this would result in a flat, no-operation result.



Then once we build a 5 qubit QFT, and run another simulation, the result is as follows. It shows a cyclic result that is different from the flat result above.

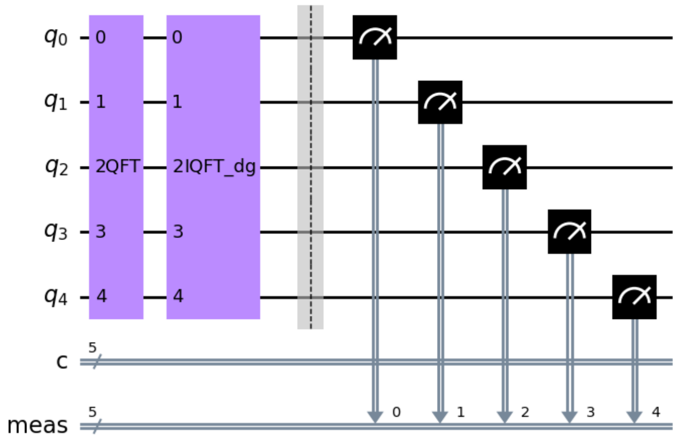


```
q5_job = backend.run(transpile(q5, backend), shots=4096)
q5_result = q5_job.result()
q5_counts = q5_result.get_counts(q5)

plot_histogram(q5_counts)
```

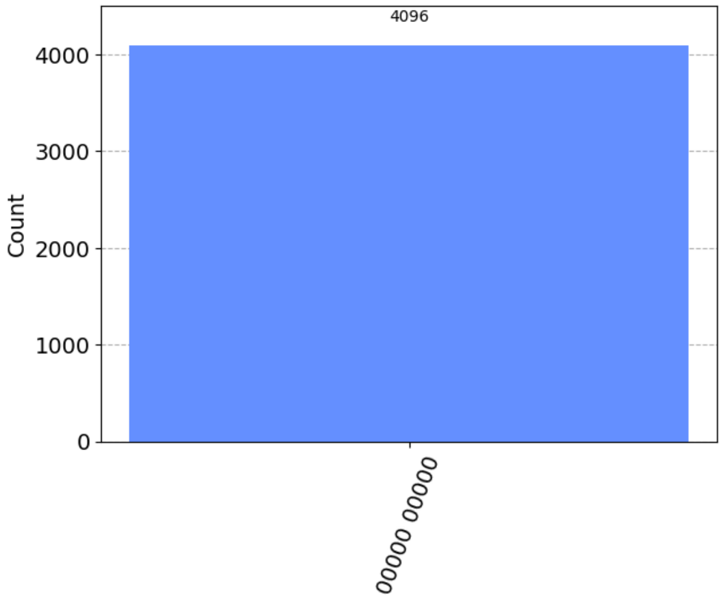


Now as the final step, we combine the QFT with an inverse QFT, then the circuit would look like this.

```
q5Class = FourierClass(5)
combined = q5Class.build()
reverse = FourierClass(5, True)
qr5 = reverse.build()

combined &= qr5
combined.measure_all()
combined.draw()
```



Measuring on this circuit with another round of simulation shows the following result.



Which is back to the identity no-operation result at the beginning. With this simulation, this implementation of QFT shows a clear unitarity property.

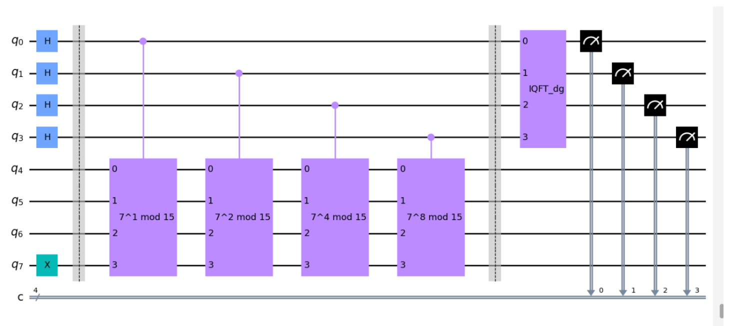
IV. Application to Shor Algorithm

As the QFT's motivation and first description was part of the Shor Algorithm,[8] it would be an interesting application to build a Shor Algorithm with the building blocks obtained above.

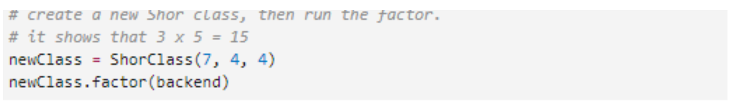
Shor Algorithm consists of 2 main parts, the Quantum Phase Estimate and the Inverse QFT. [9] This paper would not go into the details of either part, but it would just illustrate that the QFT builder we introduced above can be used as a valid element in building a Shor Algorithm circuit.

The Shor Algorithm calculation would also involve interpretation of the result. The interpretation code is also given here, but it will not be covered here either. The attached source code includes full code that would build a Shor circuit and calculate the factors from the measured results.

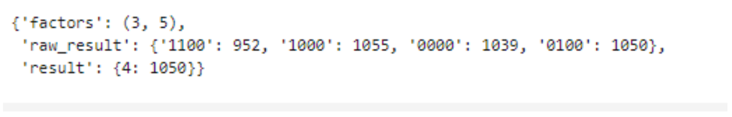
The circuit diagram of a Shor Algorithm using 8 qubits are as follows. Both the Quantum Phase Estimate part and the QFT part are represented as boxes.



The Shor class in the attachment will do actual factoring of the integer 15. Putting the class with a QASM simulator shows the following result, which shows that the factor of 15 is indeed 3 and 5.



```
# create a new Shor class, then run the factor.
# it shows that 3 x 5 = 15
newClass = ShorClass(7, 4, 4)
newClass.factor(backend)
```



```
{'factors': (3, 5),
 'raw_result': {'1100': 952, '1000': 1055, '0000': 1039, '0100': 1050},
 'result': {4: 1050}}
```

Conclusion

This implementation of Quantum Fourier Transform built a valid quantum circuit element that can be used in illustration of the QFT properties. The circuit can also be used as a building block to a Shor Algorithm and issued valid working results for factoring an integer.

References

[1] S. Lang, *The Fourier Integral*, In Undergraduate Analysis, Undergraduate Texts in Mathematics, Springer, New York, NY.(1997)

[2] P. W. Shor, *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*, SIAM Journal of Statistical Computation, 26 (1997)

[3] D. Coppersmith, *An Approximate Fourier Transform Useful in Quantum Factoring* IBM Research Report, RC 19642 (2002)

[4] https://qiskit.org/documentation/stable/0.24/stubs/qiskit.circuit.library.QFT.html

[5] D. Coppersmith, op. cit

[6] R. Asaka, K. Sakai, R. Yahagi, *Quantum Circuit for Fast Fourier Transform*, Quantum Information Processing, 19-277 (2020)

[7] https://qiskit.org/documentation/stubs/qiskit.circuit.library.QFT.html

[8] P. W. Shor, op cit

[9] H. Mohammadbagherpour, Y-H Oh, A. Singh, X. Yu, A. J. Rindos, *Experimental Challenges of Implementing Quantum Phase Estimation Algorithms on IBM Quantum Computer* arXiv: 1903.07605 [quant-ph] (20