## Quantum Computing – EN.605.728

# FINAL RESEARCH PROJECT

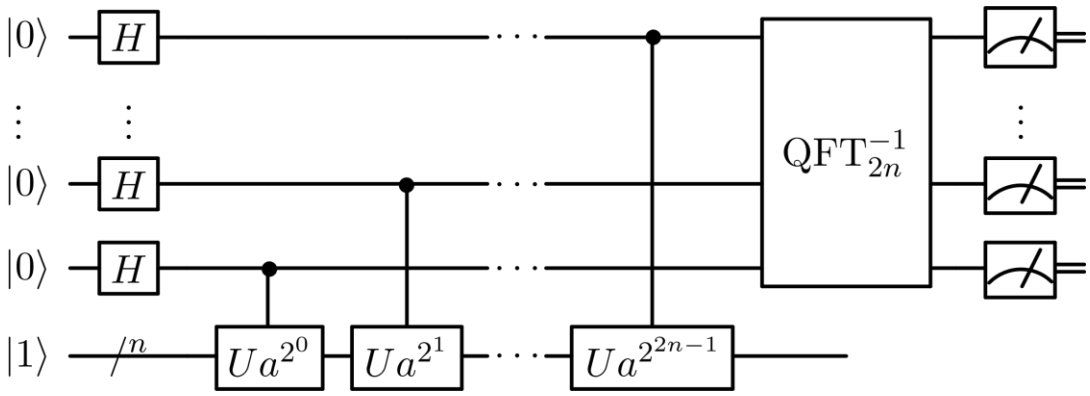



***Figure 1:*** *Quantum Fourier transform subroutine in Shor's Algorithm*[1]

*[12/09/2021]*

**Author:**

→ *Sabneet Bains*

[1] Modified from Shor's algorithm - Shor's algorithm - Wikipedia

# Contents

# 1. Introduction

## 1.1. Purpose

This document will serve as an expository piece in conjunction with the main quantum computing code provided in *"Bains-ResearchProject.py".* With that said, the general purpose of this project is to simulate a specific quantum Fourier transform subroutine in Shor's integer factoring algorithm.

## 1.2. Scope

This document will contain information regarding the background & theory behind Shor's algorithm and the specifications for the quantum Fourier transform that is the heart of Shor's algorithm. This document will provide the details of the programming implementation of the quantum Fourier transform and a sample execution tutorial. This document will not delve into the simulation of the entire Shor's algorithm as that is outside the scope of this semester-based course.

## 1.3. Definitions, Acronyms & Abbreviations

- **DFT** – Discrete Fourier transform
- **QFT** – Quantum Fourier transform
- **U** – Unitary function
- **QFT†** – Hermitian adjoint of quantum Fourier transform
- **QFT$^{-1}$** – Inverse of quantum Fourier transform
- **H** – Hadamard gate
- **$R_m$** – Controlled phase gate, where m is the rotation phase
- **NISQ** – Noisy intermediate-scale quantum computers
- **Cirq** – Google's open-source framework for NISQ
- **RSA** – Rivest–Shamir–Adleman public-key cryptosystem
- **BQP** – Bounded-error quantum polynomial time

## 1.4. Overview

This Final Research Project document contains the overview of Shor's algorithm and the specifications for a Cirq implementation of QFT and QFT†.

The document is organized into three sections: Introduction, Background & Theory, and Cirq Implementation. The latter section contains the Cirq code, and the sample runs to simulate both QFT and QFT†.

# 2. Background & Theory

## 2.1. Brief Background

Modern public-key cryptography schemes like RSA assume that factoring large integers cannot be done in polynomial time; however, that is where Peter Shor's algorithm comes in and shakes the core of this assumption. In fact, given a large quantum computer, Shor's algorithm can indeed find integer factors in polynomial time in the complexity class of BQP. More meticulously, the quantum algorithm efficiently factors integers in $O\big((\log N)^2(\log\log N)(\log\log\log N)\big) \approx \log N$, where $N$ is the integer to factor.[2] The key to its success is the quantum analog of inverse DFT, known as the Quantum Fourier Transform or QFT for short.

## 2.2. How does Shor's algorithm work?

Baring the initial classical reduction to an order-finding problem, in most general cases, the quantum subroutine for Shor's algorithm can be deciphered down into the following steps:

**Step 0**. *Initial state:*

$$|\psi_0\rangle = |0\cdots 0\rangle_n |0\cdots 0\rangle_{n_0}$$

**Step 1**. *Apply $H^{\otimes n}$ to the first register of $|\psi_0\rangle$:*

$$|\psi_1\rangle = \frac{1}{\sqrt{q}}\sum_{x=0}^{q-1}|x\rangle|0\rangle$$

[2] [quant-ph/9602016] Efficient Networks for Quantum Factoring (arxiv.org)

**Step 2**. *Apply $U_f$ to $|\psi_1\rangle$:*

$$|\psi_2\rangle \;=\; U_f|\psi_1\rangle = \frac{1}{\sqrt{q}}\sum_{x=0}^{q-1}|x\rangle|f(x)\rangle$$

**Step 3**. *Measure the second register in $|\psi_2\rangle$, obtain value $f(x_0)$:*

$$|\psi_3\rangle \;=\; \frac{1}{\sqrt{m}}\sum_{k=0}^{m-1}|x_0 + kr\rangle$$

**Step 4**. *Now apply the QFT to $|\psi_3\rangle$:*

$$|\psi_4\rangle \;=\; QFT|\psi_3\rangle = \frac{1}{\sqrt{m}}\sum_{k=0}^{m-1} QFT|x_0 + kr\rangle$$

$$\rightarrow \sum_{y=0}^{q-1} e^{\frac{2\pi i y x_0}{q}} \frac{1}{\sqrt{qm}}\left(\sum_{k=0}^{m-1} e^{\frac{2\pi i y k r}{q}}\right)|y\rangle$$

**Step 5.** *Use continued fraction expansion on $\frac{y}{q}$ to find a candidate period:*

- $d < N$
- $\left|\frac{a}{d} - \frac{y}{q}\right| < \frac{1}{2q}$

**Step 6.** *Check $f(x) = f(x+d) \Leftrightarrow a^d \equiv 1 \; mod \; N$*

**Step 7.** *If step 6 fails, try again with more candidates near $\frac{y}{q}$*

**Step 8.** *If step 7 fails, start over!* [3]

[3] Zaret, Lecture 9A

# 3. Cirq Implementation

## 3.1. QFT Simulation Code

Given the complexity and time constraints of this project, it was deemed appropriate to simulate $QFT$ and its conjugate $QFT^{\dagger}$ rather than simulating the entirety of Shor's factoring algorithm. As this proves an essential yet subtle characteristic of Shor's algorithm! – that being during $QFT$, unitary gates alter period phases, and thus, $QFT$ itself must be unitary. In other words, $QFT \times QFT^{\dagger} = I$ and on the same token, $QFT^{-1} = QFT^{\dagger}$.

Therefore, to set up the simulation, I initialized four qubits: $\{ q_0, q_1, q_2, q_3 \}$ in the following corresponding states: $|0\rangle, |0\rangle, |0\rangle, |0\rangle$

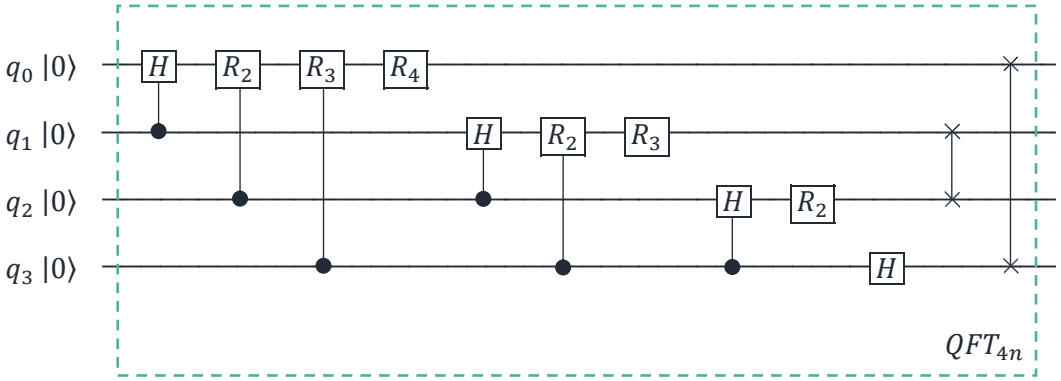
$$
\begin{aligned}
q_0\ |0\rangle &\text{ ———— } |0\rangle \\
q_1\ |0\rangle &\text{ ———— } |0\rangle \\
q_2\ |0\rangle &\text{ ———— } |0\rangle \\
q_3\ |0\rangle &\text{ ———— } |0\rangle
\end{aligned}
$$

In Cirq, the analogous is the following:

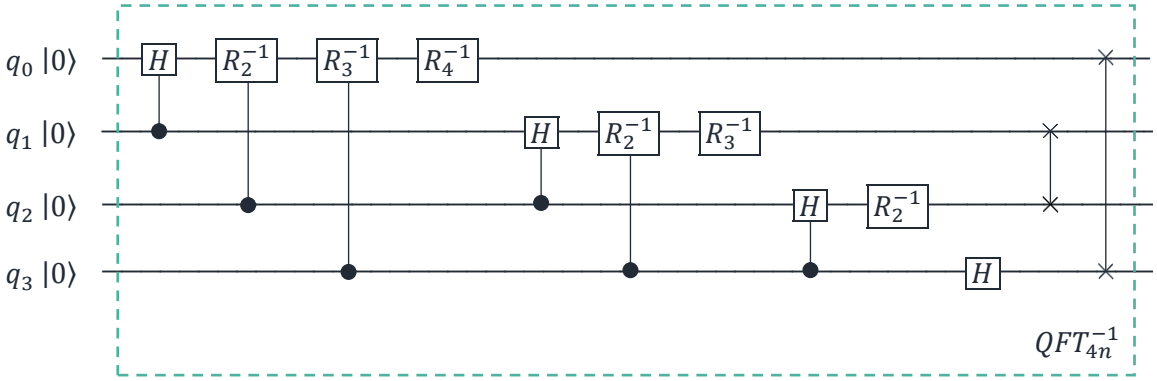
```
'''Quantum Fourier Transform Simulator for Shor's Algorithm'''

import cirq
import numpy as np

# define the initial four qubits
q0 = cirq.GridQubit(0, 0) # |0⟩
q1 = cirq.GridQubit(0, 1) # |0⟩
q2 = cirq.GridQubit(1, 0) # |0⟩
q3 = cirq.GridQubit(1, 1) # |0⟩
```

***Code Snippet 2:*** *Four-qubit QFT simulator initial state*

Next, recursively applying $QFT$ on this four-qubit circuit yields:




Likewise, since $QFT^{\dagger}$ is the inverse $QFT^{-1}$, it can be implemented as:




Whereas, in Cirq, I utilized the fact that controlled phase gates are related to controlled Z gates, given the said rotation phase. I.e., $R_3 = \sqrt[4]{Z}$ and $R_3^{-1} = \sqrt[-4]{Z}$

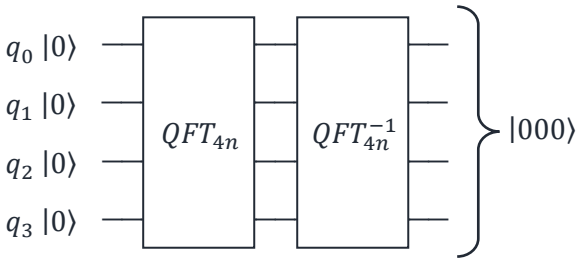
Therefore, I was able to write a modular function that represented the four-qubit $QFT$ and $QFT^{\dagger}$ with just a single boolean call for either inverse or not inverse.

Or, as in Cirq, the code is presented as follows:

```
def initialize_QFT(inv=False):
    '''Initializes the Four-qubit QFT Circuit'''

    if inv is True:
        sign = -1 # i.e., QFT†
    else:
        sign = 1 # i.e., QFT

    # GATE 1: apply Hadamard to q0
    yield cirq.H(q0)

    # GATE 2: apply controlled R₂ between q0 and q1
    yield cirq.CZPowGate(exponent=sign*0.5)(q0,q1)

    # GATE 3: apply controlled R₃ between q0 and q2
    yield cirq.CZPowGate(exponent=sign*0.25)(q0,q2)

    # GATE 4: apply controlled R₄ between q0 and q3
    yield cirq.CZPowGate(exponent=sign*0.125)(q0,q3)

    # GATE 5: apply Hadamard to q1
    yield cirq.H(q1)

    # GATE 6: apply controlled R₂ between q1 and q2
    yield cirq.CZPowGate(exponent=sign*0.5)(q1,q2)

    # GATE 7: apply controlled R₃ between q1 and q3
    yield cirq.CZPowGate(exponent=sign*0.25)(q1,q3)

    # GATE 8: apply Hadamard to q2
    yield cirq.H(q2)

    # GATE 9: apply controlled R₂ between q2 and q3
    yield cirq.CZPowGate(exponent=sign*0.5)(q2,q3)

    # GATE 10: apply Hadamard to q3
    yield cirq.H(q3)

    # GATE 11: apply swap between q1 and q2
    yield cirq.SWAP(q1,q2)

    # GATE 12: apply final swap between q0 and q3
    yield cirq.SWAP(q0,q3)
```

***Code Snippet 2:*** *Four-qubit QFT and inverse QFT creation*

Needless to say, $QFT$ is indeed unitary as:



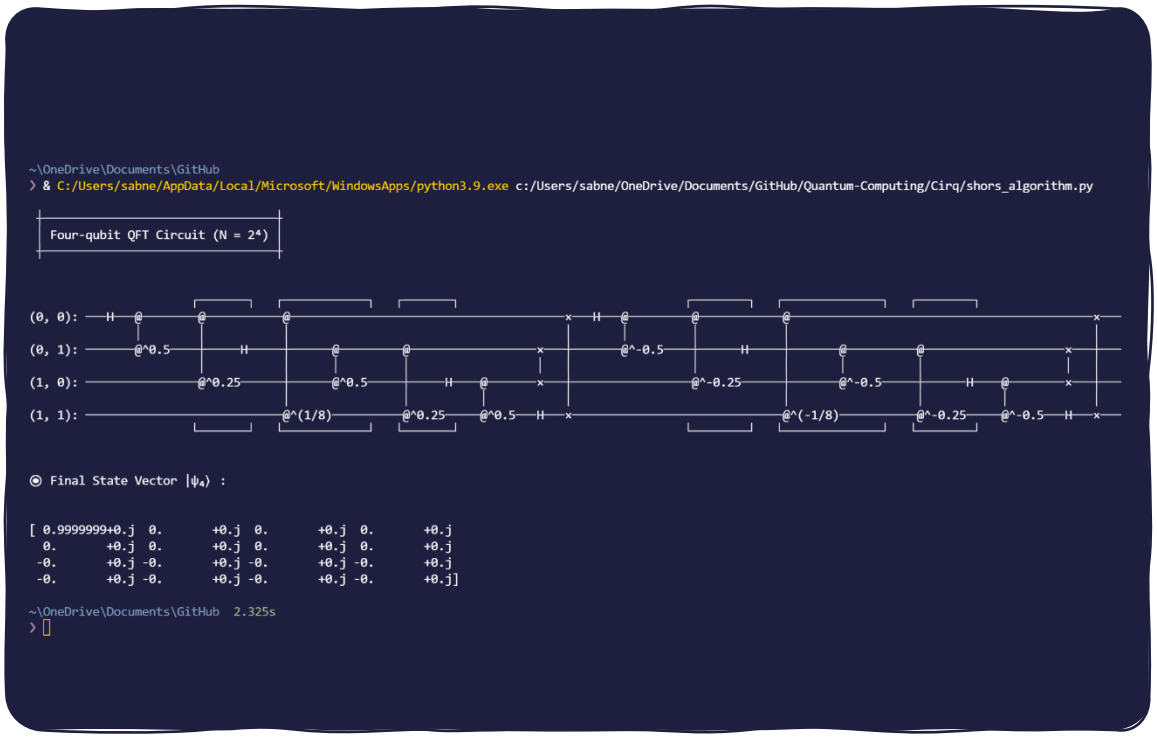

Thus, to test and achieve the original objective, I created the following main function to test the simulated QFT for unitary properties.

```
def main():
    '''Simulates the Four-qubit QFT Circuit'''

    # create and append the quantum circuit
    circuit = cirq.Circuit()
    circuit.append(initialize_QFT(False)) # i.e., QFT
    circuit.append(initialize_QFT(True)) # i.e., QFT†
    print("\n ┼──────────────────────────────────┼\n",
    "│ Four-qubit QFT Circuit (N = 2⁴) │\n",
    "┼──────────────────────────────────┼\n\n")
    print(circuit)

    # simulate the circuit and print the resulting state vector
    result = cirq.Simulator().simulate(circuit)
    print("\n\n◉  Final State Vector |ψ₄⟩:\n\n")
    print(np.around(result.final_state_vector, 7))
```

***Code Snippet 3:*** *Four-qubit simulator main function*

### *3.2. Sample Run*

As evident, the simulation proves the unitary assumption since the final state vector ends up in the same state as the initial qubits state: $|0\rangle, |0\rangle, |0\rangle, |0\rangle$




***Code Snippet 4:*** *Cirq Simulation execution result*