ECSE 420 – Fall 2020

Group 8

ECSE 420 Parallel Computing

Final Project Report

Liang Zhao - 260781081

Zikun Lyu - 260681819

Charles Liu - 260765100

Hongshuo Zhou-260792817

1. Introduction

1.1. Project Description

For this project, we explored for quantum computing and reproduced our own version of quantum Fourier transform. First, we had a basic understanding of the theory and methodology of quantum computing by reading relative books and research papers, as quantum computing was a completely new area for all of us. We then split our team into two sub-teams, one team handling the code implementation of quantum Fourier transform while the other team handling the presentation preparation. In this report, the algorithm of discrete Fourier transform, Fast Fourier transform and quantum Fourier transform will be discussed, as well as our implementation of quantum Fourier transform.

1.2. Fourier Transform

Fourier transform is a mathematical transform that decomposes a function or a signal into its constituent frequencies, it occurs in many different versions throughout classical computing, in areas ranging from signal processing to data compression to complexity theory. It is closely related to the Fourier Series.There are three main ways to do fourier transform: discrete fourier transform (DFT), fast fourier transform (FFT), and quantum fourier transform (QFT). All of which are explained below. Fourier transform can be used in network and signal, image compression, image editing, and much more.

1.3. Quantum Fourier Transform

The quantum Fourier transform (QFT) is the quantum implementation of the discrete Fourier transform over the amplitudes of a wavefunction. It is part of many quantum algorithms, most notably Shor's factoring algorithm and quantum phase estimation[1]. Quantum Fourier transform (QFT) transforms between two bases, the computational (Z) basis, and the Fourier basis. The H-gate is the single-qubit QFT, and it transforms between the Z-basis states $|0\rangle$ and $|1\rangle$ to the X-basis states $|+\rangle$ and $|-\rangle$. In the same way, all multi-qubit states in the computational basis have corresponding states in the Fourier basis. The QFT is simply the function that transforms between these bases[1].

$$|\text{State in Computational Basis}\rangle \xrightarrow{\text{QFT}} |\text{State in Fourier Basis}\rangle$$

$$\text{QFT}|x\rangle = |\tilde{x}\rangle$$

2. Program Analysis

2.1. Algorithms and Background

2.1.1. Discrete Fourier Transform (DFT)

The discrete Fourier transform (DFT) converts a finite sequence of equally-spaced samples of a function into a same-length sequence of equally-spaced samples of the discrete-time Fourier transform (DTFT). The discrete Fourier transform (DFT) is an important tool in digital signal processing. There are three common ways that it is used:
(1): The DFT can calculate a signal's frequency spectrum. This is a direct examination of information encoded in the frequency, phase, and amplitude of the component sinusoids. For example, human speech and hearing use signals with this type of encoding.
(2): The DFT can find a system's frequency response from the system's impulse response, and vice versa. This allows systems to be analyzed in the frequency domain, just as convolution allows systems to be analyzed in the time domain.
(3): The DFT can be used as an intermediate step in more elaborate signal processing techniques. The classic example of this is FFT convolution, an algorithm for convolving signals that is hundreds of times faster than conventional methods.
The discrete Fourier transform transforms a sequence of N complex numbers into another sequence of complex numbers. The expression of DFT is:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}} \qquad k = 0, \ldots, N-1.$$

Figure 2.1

2.1.2. Fast Fourier Transform (FFT)

Fast Fourier Transform (FFT) is a method to quickly calculate Discrete Fourier Transform (DFT) and its inverse transform (IDFT). The time complexity for DFT and FFT are:

- Time Complexity (DFT): O(N^2)
- Time Complexity (FFT): O(N*log N)

The reason why FFT can speed up DFT is that it uses the symmetry property of Fourier transform. According to the DFT definition in figure 2.1, replacing k with N k, we get the following:

$$
\begin{aligned}
X_{N+k} &= \sum_{n=0}^{N-1} x_n \cdot e^{-i\,2\pi\,(N+k)\,n\,/\,N} \\
&= \sum_{n=0}^{N-1} x_n \cdot e^{-i\,2\pi\,n} \cdot e^{-i\,2\pi\,k\,n\,/\,N} \\
&= \sum_{n=0}^{N-1} x_n \cdot e^{-i\,2\pi\,k\,n\,/\,N}
\end{aligned}
$$

Figure 2.2

This means that x is repeated for every N elements,

$$X_{k+iN} = X_k$$

We can use Cooley-turkey to split DFT to two parts:

$$
\begin{aligned}
X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-i\,2\pi\,k\,n\,/\,N} \\
&= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i\,2\pi\,k\,(2m)\,/\,N} + \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i\,2\pi\,k\,(2m+1)\,/\,N} \\
&= \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-i\,2\pi\,k\,m\,/\,(N/2)} + e^{-i\,2\pi\,k\,/\,N} \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-i\,2\pi\,k\,m\,/\,(N/2)}
\end{aligned}
$$

Figure 2.3

If we keep splitting, the time complexity is reduced to O(N*log N).

2.1.3. Quantum Fourier Transform (QFT)

Quantum Fourier transform is the quantum solution of DFT,

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{\frac{2\pi ijk}{N}}$$

Where $x_0,...,x_{N-1}$ are vectors consisting of complex numbers. What quantum computing does is to first calculate the quantum state, and then map it to DFT. This can be expressed as,

$$|j> -> y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{\frac{2\pi ijk}{N}} |k>$$

Here is how to calculate QFT. Set $N=2^n$ where n is integer, $l_n>$...... $l_{2^n-1}>$ is the computational basis of one n qubit. Let's express a qubit $l_j>$ in binary: $j = j_1j_2...j_n$ which is $j = j_1*2^{n-1}+...+ j_n*2^0$; therefore,

$0,j_lj_{l+1}..j_m$ *is* $j_l/2 + j_{l+1}/4 + ... + j_m/(m-l+1)$.

Thus, the equation above can be written as:

$$\begin{aligned}
|j\rangle \to & \frac{1}{2^{n/2}} \sum_{k=0}^{2^n-1} e^{2\pi ijk/2^n} |k\rangle \\
= & \frac{1}{2^{n/2}} \sum_{k_1=0}^{1} \cdots \sum_{k_n=0}^{1} e^{2\pi ij\left(\sum_{l=1}^{n} k_l 2^{-l}\right)} |k_1 \ldots k_n\rangle \\
= & \frac{1}{2^{n/2}} \sum_{k_1=0}^{1} \cdots \sum_{k_n=0}^{1} \bigotimes_{l=1}^{n} e^{2\pi ijk_l 2^{-l}} |k_l\rangle \\
= & \frac{1}{2^{n/2}} \bigotimes_{l=1}^{n} \left[\sum_{k_l=0}^{1} e^{2\pi ijk_l 2^{-l}} |k_l\rangle\right] \\
= & \frac{1}{2^{n/2}} \bigotimes_{l=1}^{n} \left[|0\rangle + e^{2\pi ij2^{-l}} |1\rangle\right] \\
= & \frac{\left(|0\rangle + e^{2\pi i0.j_n}|1\rangle\right)\left(|0\rangle + e^{2\pi i0.j_{n-1}j_n}|1\rangle\right)\cdots\left(|0\rangle + e^{2\pi i0.j_1j_2\cdots j_n}|1\rangle\right)}{2^{n/2}}
\end{aligned}$$

2.2 Quantum Circuit for QFT

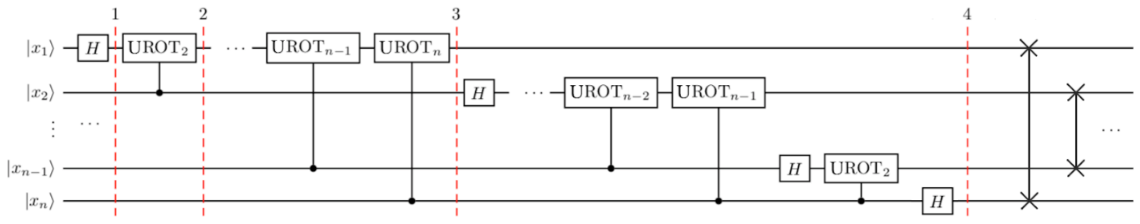
The circuit that implements QFT makes use of two gates. The first one is a single-qubit Hadamard gate, which is a fundamental circuit unit in quantum circuits. H on the single-qubit state $|x_k>$ can be expressed as:

$$H|x_k\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + \exp\left(\frac{2\pi i}{2}x_k\right)|1\rangle\right)$$

The second is a two-qubit controlled rotation CROT given in matrix form as
where

$$CROT_k = \begin{bmatrix} I & 0 \\ 0 & UROT_k \end{bmatrix} \qquad UROT_k = \begin{bmatrix} 1 & 0 \\ 0 & \exp\left(\frac{2\pi i}{2^k}\right) \end{bmatrix}$$

The following image is the Quantum circuit for QFT.



The circuit operates as follows. We start with an n-qubit input state $|x_1 x_2 \ldots x_n>$.

1. After the first Hadamard gate on qubit 1, the state is transformed from the input state to

$$H_1|x_1x_2\ldots x_n\rangle = \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2}x_1\right)|1\rangle\right] \otimes |x_2x_3\ldots x_n\rangle$$

2. After the UROT2 gate on qubit 1 controlled by qubit 2, the state is transformed to

$$\frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^2}x_2 + \frac{2\pi i}{2}x_1\right)|1\rangle\right] \otimes |x_2x_3\ldots x_n\rangle$$

3. After the application of the last UROTn gate on qubit 1 controlled by qubit n, the state becomes

$$\frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^n}x_n + \frac{2\pi i}{2^{n-1}}x_{n-1} + \ldots + \frac{2\pi i}{2^2}x_2 + \frac{2\pi i}{2}x_1\right)|1\rangle\right] \otimes |x_2x_3\ldots x_n\rangle$$

Note that

$$x = 2^{n-1}x_1 + 2^{n-2}x_2 + \ldots + 2^1x_{n-1} + 2^0x_n$$

We can write the above state as

$$\frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^n}x\right)|1\rangle\right] \otimes |x_2x_3\ldots x_n\rangle$$

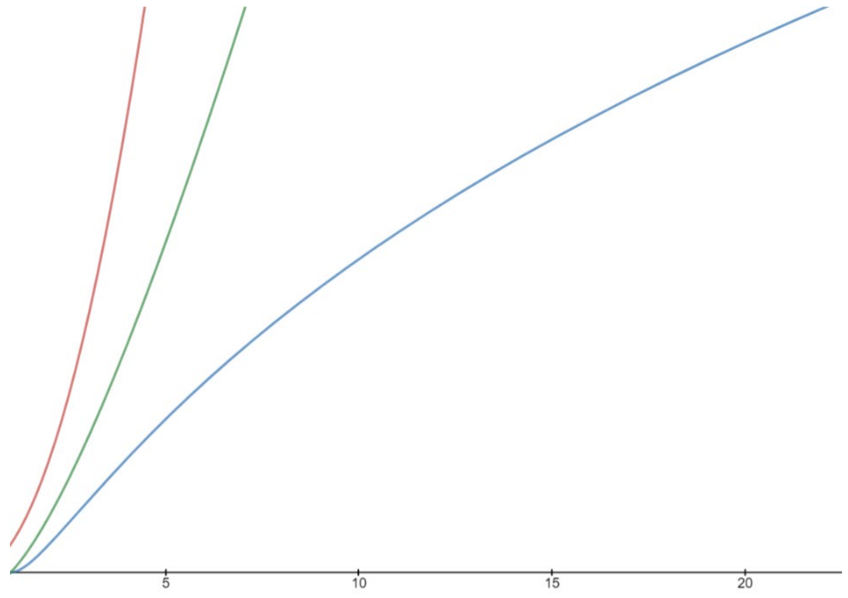
4. After the application of a similar sequence of gates for qubits 2…n, we find the final state to be

$$\frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^n}x\right)|1\rangle\right] \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^{n-1}}x\right)|1\rangle\right] \otimes \ldots \otimes \frac{1}{\sqrt{2}}\left[|0\rangle + \exp\left(\frac{2\pi i}{2^2}x\right)|1\rangle\right] \otimes$$

which is exactly the QFT of the input state with the caveat that the order of the qubits is reversed in the output state.

2.3. Algorithm Complexity Comparison

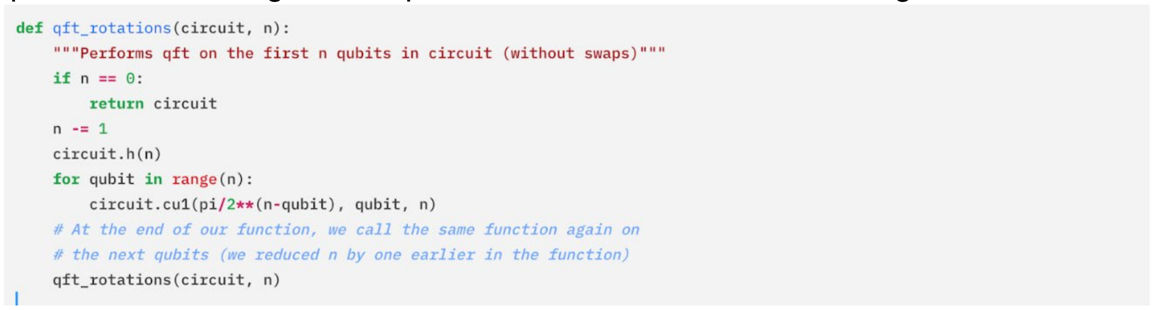
Since QFT uses the idea of DFT, the time complexity is n2. However, n is different in this case. Because of qubits, quantum computers can represent 2N amount of N bits integer at the same time. Therefore, it can do 2N calculations at the same time rather than 1. Therefore, the time complexity for QFT is actually O((log N)2) where N is the number of bits. Compared to DFT and FFT, QFT is a lot faster.



red: N^2, green: N*log N, blue: $(\log N)^2$

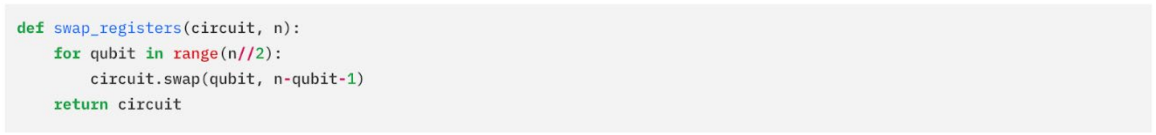
3. Implementation and Results

For our implementation, we chose to use Qiskit. Qiskit is an open-source framework for quantum computing which provides tools for creating and manipulating quantum programs and running them on prototype quantum devices on IBM Experience or on simulators on our local computer.
Firstly, we built a circuit that implements the QFT by using Qiskit. Based on our research, we preferred to build the circuit upside down and then swap them afterwards. Therefore, we created a function to rotate our qubits. In this qft_rotation, recursion was used to rotate the qubits from the most significant qubit to the least significant qubit. As can be seen from this image below:

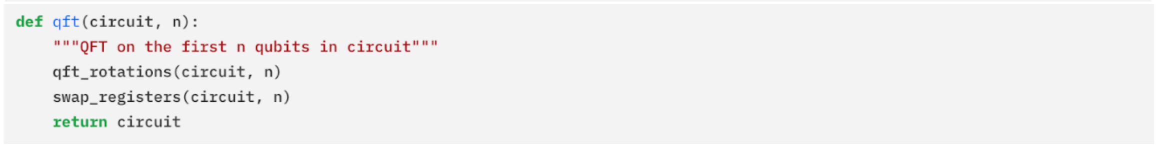


```python
def qft_rotations(circuit, n):
    """Performs qft on the first n qubits in circuit (without swaps)"""
    if n == 0:
        return circuit
    n -= 1
    circuit.h(n)
    for qubit in range(n):
        circuit.cu1(pi/2**(n-qubit), qubit, n)
    # At the end of our function, we call the same function again on
    # the next qubits (we reduced n by one earlier in the function)
    qft_rotations(circuit, n)
```

Then we had a swap function to swap the circuit and we defined our qft function.

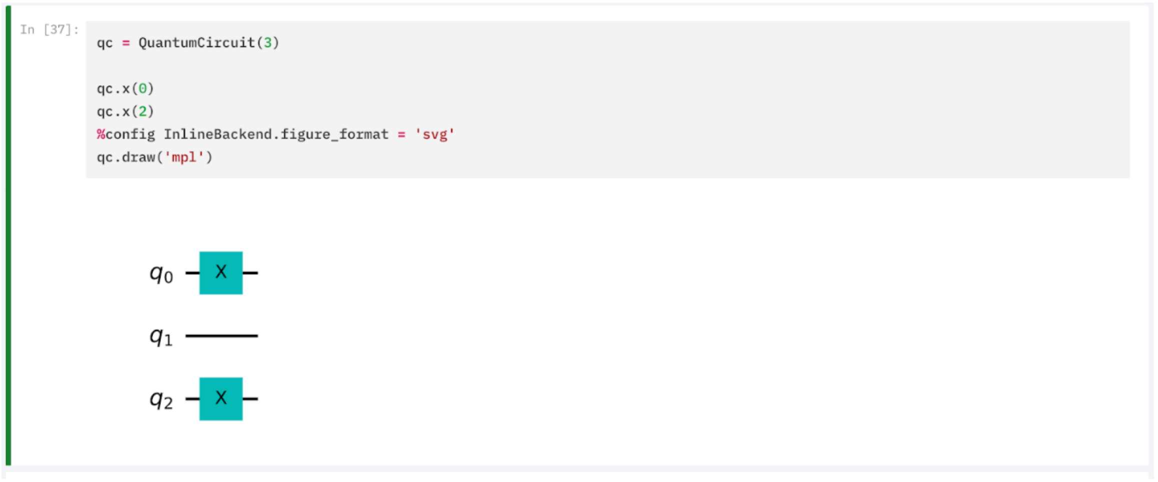


```python
def swap_registers(circuit, n):
    for qubit in range(n//2):
        circuit.swap(qubit, n-qubit-1)
    return circuit
```

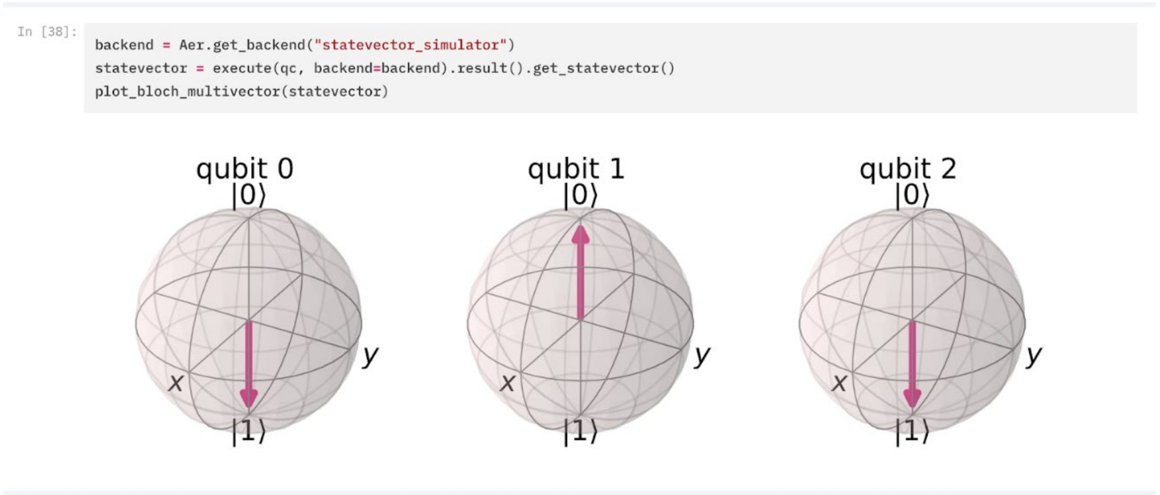


```python
def qft(circuit, n):
    """QFT on the first n qubits in circuit"""
    qft_rotations(circuit, n)
    swap_registers(circuit, n)
    return circuit
```

Then we tested if this circuit worked correctly. We chose number 5 which is 101 in binary as our example. Then, we encoded state 5 into our qubits. Then, we used state vector simulator to check qubit's states.

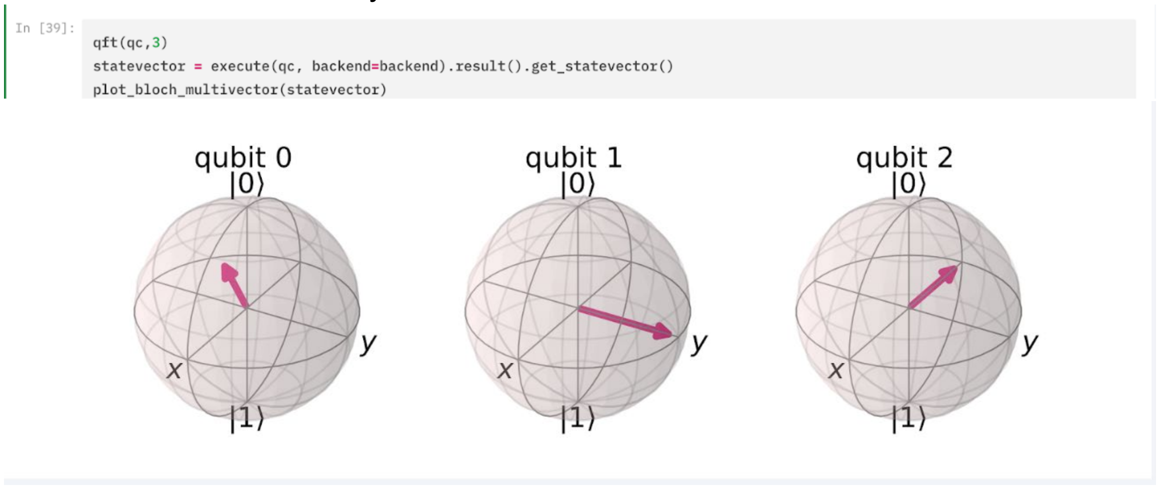


```
In [37]:
qc = QuantumCircuit(3)

qc.x(0)
qc.x(2)
%config InlineBackend.figure_format = 'svg'
qc.draw('mpl')
```



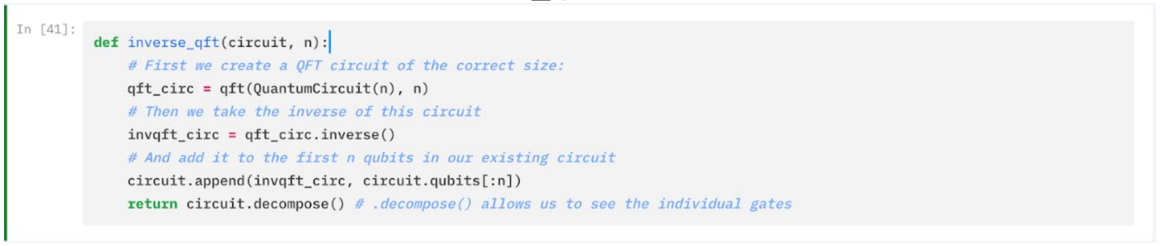
```
In [38]:
backend = Aer.get_backend("statevector_simulator")
statevector = execute(qc, backend=backend).result().get_statevector()
plot_bloch_multivector(statevector)
```

Next, we applied our QFT function and we use state vector simulator again to check qubit's states. By comparing these two states, we could figure out that our circuit worked correctly.



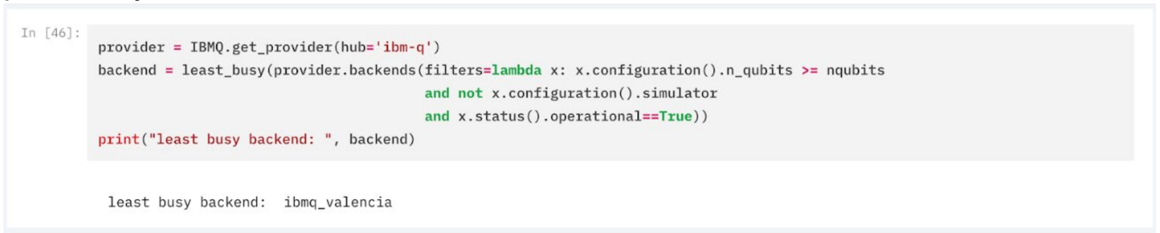
```
In [39]:
qft(qc,3)
statevector = execute(qc, backend=backend).result().get_statevector()
plot_bloch_multivector(statevector)
```

After proving this circuit function worked correctly theoretically, we moved forward to prove that QFT works correctly in real hardware. Based on what we learned, we could run our QFT function in reverse and verify if its result is correct. Then, we defined our inverse_qft function.

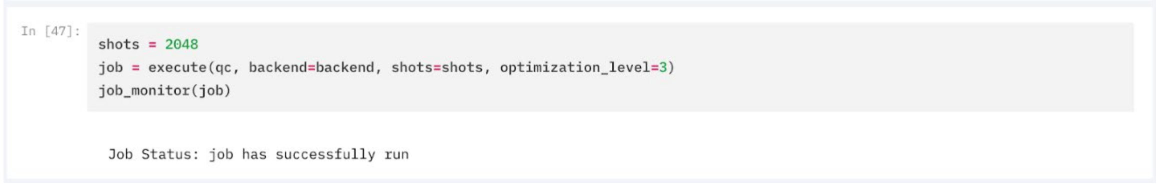


```
In [41]:  def inverse_qft(circuit, n):
              # First we create a QFT circuit of the correct size:
              qft_circ = qft(QuantumCircuit(n), n)
              # Then we take the inverse of this circuit
              invqft_circ = qft_circ.inverse()
              # And add it to the first n qubits in our existing circuit
              circuit.append(invqft_circ, circuit.qubits[:n])
              return circuit.decompose() # .decompose() allows us to see the individual gates
```

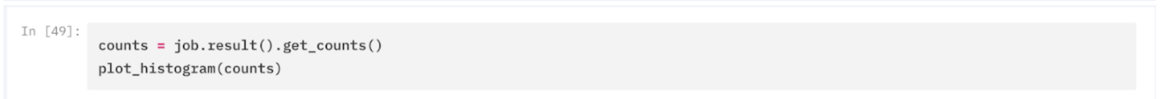
Then, we load our IBMQ account and find the least busy backend device with less than or equal to n qubits. We found that the outcome with the highest probability is 101.



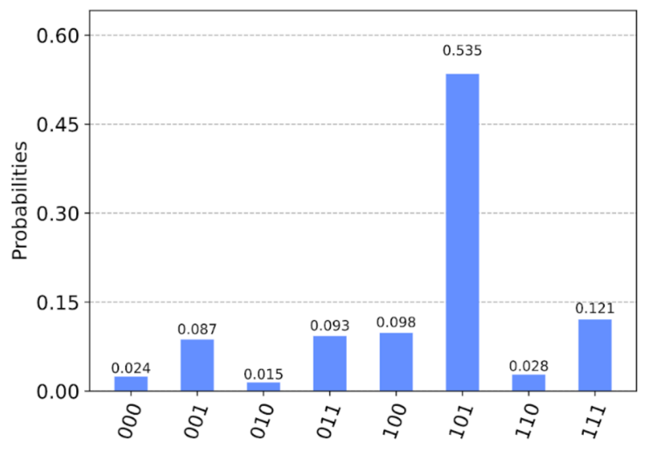
```
In [46]:  provider = IBMQ.get_provider(hub='ibm-q')
          backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits >= nqubits
                                                 and not x.configuration().simulator
                                                 and x.status().operational==True))
          print("least busy backend: ", backend)

           least busy backend:  ibmq_valencia
```



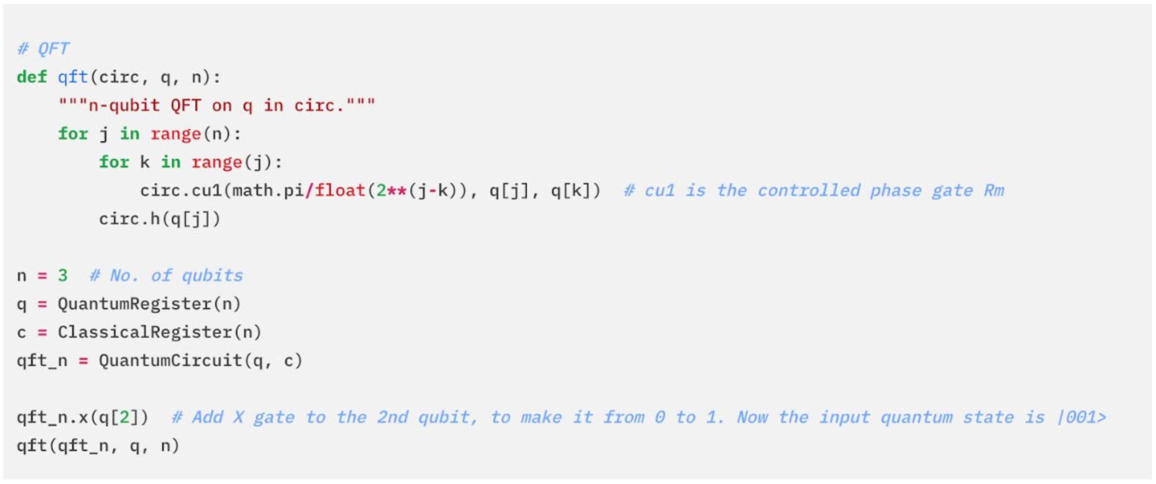
```
In [47]:  shots = 2048
          job = execute(qc, backend=backend, shots=shots, optimization_level=3)
          job_monitor(job)

           Job Status: job has successfully run
```



```
In [49]:  counts = job.result().get_counts()
          plot_histogram(counts)
```

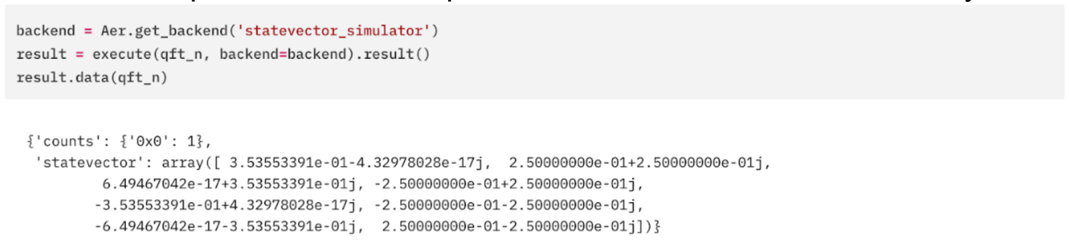


We also learned another way to prove QFT works correctly which is applying QFT to a vector corresponds to applying inverse FFT to it. Wavefunction is an object that stores a quantum state as a list of amplitudes and we used wavefunction to check if our QFT was the same to that of the inverse of FFT. Since we need to test bit string instead of a single bit, we refactored our previous QFT function to make sure it could do QFT on a bit string. And we use 01000000 as an example bit string.



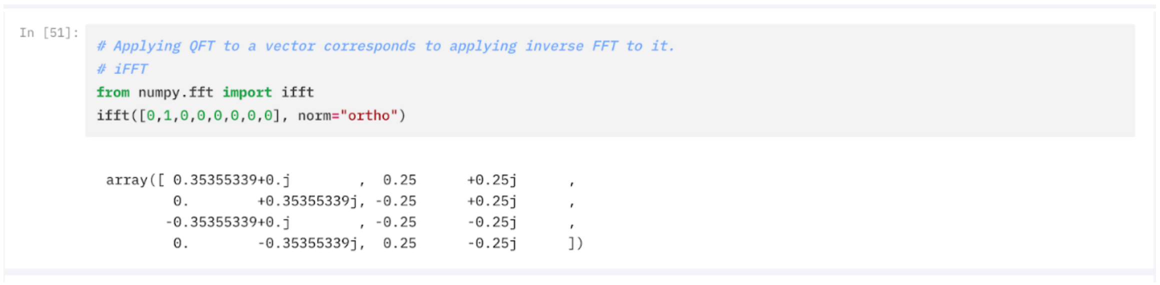
```
# QFT
def qft(circ, q, n):
    """n-qubit QFT on q in circ."""
    for j in range(n):
        for k in range(j):
            circ.cu1(math.pi/float(2**(j-k)), q[j], q[k])  # cu1 is the controlled phase gate Rm
        circ.h(q[j])

n = 3  # No. of qubits
q = QuantumRegister(n)
c = ClassicalRegister(n)
qft_n = QuantumCircuit(q, c)

qft_n.x(q[2])  # Add X gate to the 2nd qubit, to make it from 0 to 1. Now the input quantum state is |001>
qft(qft_n, q, n)
```

And we got the result array from our QFT function and the result array from iFFF we imported from the library. As we can see from these two results, they are identical. This proves that our implementation of QFT worked correctly.



```
backend = Aer.get_backend('statevector_simulator')
result = execute(qft_n, backend=backend).result()
result.data(qft_n)
```

```
{'counts': {'0x0': 1},
 'statevector': array([ 3.53553391e-01-4.32978028e-17j,  2.50000000e-01+2.50000000e-01j,
         6.49467042e-17+3.53553391e-01j, -2.50000000e-01+2.50000000e-01j,
        -3.53553391e-01+4.32978028e-17j, -2.50000000e-01-2.50000000e-01j,
        -6.49467042e-17-3.53553391e-01j,  2.50000000e-01-2.50000000e-01j])}
```



```
In [51]: # Applying QFT to a vector corresponds to applying inverse FFT to it.
         # iFFT
         from numpy.fft import ifft
         ifft([0,1,0,0,0,0,0,0], norm="ortho")
```

```
array([ 0.35355339+0.j        ,  0.25      +0.25j      ,
        0.        +0.35355339j, -0.25      +0.25j      ,
       -0.35355339+0.j        , -0.25      -0.25j      ,
        0.        -0.35355339j,  0.25      -0.25j      ])
```

4. Instructions

To run our sample code,
1. Go to https://quantum-computing.ibm.com/account.
2. Setup your own account and get your personal token in your account.
3. Go to https://quantum-computing.ibm.com/jupyter.
4. Import the ipynb file we submitted.
5. Open the project we imported.
6. Inside In[2], put your token into “IBMQ.save_account(‘token’)”.
7. Run the code from top to bottom.

5. Conclusion

5.1. Project Achievement

To summarise, in this project, we explored and learned background knowledge for quantum physics and quantum computing. Then we reproduced the quantum algorithm, which is QFT, and we verified the correctness of it. However, it is not possible to measure the real running time of QFT because of the long queuing time on remote devices. According to our reference, the better solution is to estimate it based on the circuit depth, just take the circuit, compile it, and then add up the delays on the longest path of the compiled circuit.

5.2. Further Improvement

As we mentioned before, this project reproduced the quantum algorithm QFT. It will also be meaningful to try more quantum algorithms on Qiskit, such as Grover’s algorithm and Bernstein-Vazirani algorithm. Many other teams with similar quantum computing topic chose to implement grover’s algorithm, this algorithm can speed up an unstructured search problem quadratically, but its uses extend beyond that; it can serve as a general trick or subroutine to obtain quadratic run time improvements for a variety of other algorithms.

6. References

[1] https://qiskit.org/textbook/ch-algorithms/quantum-fourier-transform.html

[2] https://www.math.mcgill.ca/darmon/courses/12-13/nt/projects/Fangxi-Lin.pdf

[3] https://courses.cs.washington.edu/courses/cse599d/06wi/lecturenotes9.pdf

[4] https://cs.uwaterloo.ca/~cleve/pubs/2004EfficientQft.pdf