UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

Quantum Computing

MARÍA GABRIELA ZUMÁRRAGA DÁVILA

Ingeniería en Ciencias de la Computación

Trabajo de fin de carrera presentado como requisito
para la obtención del título de
Ingeniera en Ciencias de la Computación

Quito, 17 de diciembre de 2021

UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Ciencias e Ingenierías

HOJA DE CALIFICACIÓN
DE TRABAJO DE FIN DE CARRERA

Quantum Computing

MARÍA GABRIELA ZUMÁRRAGA DÁVILA

Nombre del profesor, Título académico **Daniel Riofrío, PhD**

Quito, 17 de diciembre de 2021

© DERECHOS DE AUTOR

Por medio del presente documento certifico que he leído todas las Políticas y Manuales de la Universidad San Francisco de Quito USFQ, incluyendo la Política de Propiedad Intelectual USFQ, y estoy de acuerdo con su contenido, por lo que los derechos de propiedad intelectual del presente trabajo quedan sujetos a lo dispuesto en esas Políticas.

Asimismo, autorizo a la USFQ para que realice la digitalización y publicación de este trabajo en el repositorio virtual, de conformidad a lo dispuesto en la Ley Orgánica de Educación Superior del Ecuador.

Nombres y apellidos: María Gabriela Zumárraga Dávila

Código: 200457

Cédula de identidad: 1717210742

Lugar y fecha: Quito, 17 de diciembre de 2021

ACLARACIÓN PARA PUBLICACIÓN

Nota: El presente trabajo, en su totalidad o cualquiera de sus partes, no debe ser considerado como una publicación, incluso a pesar de estar disponible sin restricciones a través de un repositorio institucional. Esta declaración se alinea con las prácticas y recomendaciones presentadas por el Committee on Publication Ethics COPE descritas por Barbour et al. (2017) Discussion document on best practice for issues around theses publishing, disponible en http://bit.ly/COPETheses.

UNPUBLISHED DOCUMENT

Note: The following capstone project is available through Universidad San Francisco de Quito USFQ institutional repository. Nonetheless, this project – in whole or in part – should not be considered a publication. This statement follows the recommendations presented by the Committee on Publication Ethics COPE described by Barbour et al. (2017) Discussion document on best practice for issues around theses publishing available on http://bit.ly/COPETheses.

RESUMEN

La Computación cuántica es un campo relativamente nuevo dentro de las ciencias computacionales el cual se ha venido explorando desde hace algunos años por empresas grandes tales como Google e IBM. Sin embargo su potencial es bastante prometedor; posee áreas emergentes en estudio tales como la teletransportación cuántica, la criptografía, machine learning, entre otros. Esta, al basar su procesamiento en el uso de propiedades físicas como el entrelazamiento cuántico y la superposición, nos permite desarrollar algoritmos y sistemas que sean capaces de solucionar problemas complejos que no han podido ser resueltos aún con las computadoras actuales. Este estudio explora áreas tanto de la computación clásica como de la computación cuántica mediante una aproximación tanto teórica como práctica utilizando los recursos que IBM y Google ponen a disposición del público.

Palabras clave: Computación Clásica, Computación Cuántica, Circuitos Lógicos, Circuitos Cuánticos, Compuertas Lógicas, Compuertas Cuánticas, Sistema Cuántico IBM, Sistema Cuántico Google, Qiskit, Cirq, Teletransportación Cuántica, Deutsch-Jozsa.

ABSTRACT

Quantum Computing is a relatively new field in computing which has been explored for ears by large companies such as Google and IBM. However, its potential is quite promising and any emerging areas are under study such as quantum *teleportation*, quantum cryptography, quantum machine learning, amongst others. Quantum Computing is based on principles from physics such as quantum *entanglement* and *superposition*. This allows us to develop algorithms and systems capable of solving complex problems that have not been able to be solved even with current computers. This study explores areas of both classic and quantum computing with the use of theoretical and practical assumptions by using IBM and Google public resources.

Key words: Classic Computing, Quantum Computing, Logic Circuits, Quantum Circuits, Logic Gates, Quantum Gates, Quantum IBM, Quantum Google, Qiskit, Cirq, Quantum Teleportation, Deutsch-Jozsa.

CONTENTS

RESUMEN **5**
ABSTRACT **6**
CONTENTS **7**
ÍNDICE DE TABLAS **8**
ÍNDICE DE FIGURAS **9**
INTRODUCTION **10**
CLASSICAL COMPUTING **12**
Programming Language 12
Compilers and Interpreters: 12
Logic Gates: 13
QUANTUM COMPUTING **14**
Qubit 14
Quantum Gates: 15
Quantum Computers: 18
Qiskit and Cirq 18
ALGORITHMS **19**
Quantum Teleportation 20
Implementation 21
Deutsch-Jozsa 26
Implementation 26
RESULTS AND DISCUSSIONS **32**
REFERENCES **35**

ÍNDICE DE TABLAS

Table 1: Gates Comparison....................17

Table 2: Tools....................19

Table 3: Teleportation Algorithm: Step #1....................22

Table 4: Teleportation Algorithm: Step #2....................22

Table 5: Teleportation Algorithm: Step #3....................23

Table 6: Teleportation Algorithm: Step #4....................23

Table 7: Teleportation Algorithm: Step #5....................24

Table 8: Teleportation Algorithm: Step #6....................25

Table 9: Teleportation Algorithm: Step #7....................26

Table 10: Deutsch-Jozsa Algorithm: Step #1....................28

Table 11: Deutsch-Jozsa Algorithm: Step #2....................28

Table 12: Deutsch-Jozsa Algorithm: Step #2....................29

Table 13: Deutsch-Jozsa Algorithm: Step #2....................30

Table 14: Deutsch-Jozsa Algorithm: Step #3....................30

Table 15: Deutsch-Jozsa Algorithm: Step #4....................31

Table 16: Deutsch-Jozsa Algorithm: Step #5....................31

Table 17: Deutsch-Jozsa Algorithm: Step #6....................32

ÍNDICE DE FIGURAS

Figure 1: Bloch Sphere 15

INTRODUCTION

Nowadays, computers play an important role in our daily lives. Almost everybody uses a smartphone or a computer to get or send information. Yet, do you know how that information is processed? Computers process information every second by using *logic gates* that work with a binary system i.e. 1's and 0's (Vogt, 2021). Likewise, the same happens in quantum computers. These computers process information by using quantum circuits and quantum gates which are equivalent to classic systems. In these circuits computers use *Qubits* to make operations based on probabilities.

Despite today's computers have great processing power there are algorithms that require large amounts of time to compute. But, since quantum computers depend on quantum properties, it is possible to solve some of these complex problems faster. Quantum Computing is a field which has been explored for years by large companies such as Google and IBM (Carrascal, del Barrio, & Botella, 2020). In fact, its potential is quite promising; it has emerging areas under study like cryptography, machine learning, amongst others where we can develop algorithms and systems capable of solving complex problems that have not been able to be solved even with current computers.

It is a relatively new field although their studies began half way through the 20th century because quantum computers did not appear until the end of the century. This allowed to begin some practical studies that continue to this day. In those studies, it was discovered that there exists physical elements that allows us to represent more than two states by using quantum *superposition* and also to share states by using *entanglement* between two *qubits*. These special characteristics allow machines to have greater capacity and processing power to simulate complex systems. Nevertheless, there are still a lot of challenges in this field because finding a practical application is hard. Current architectures do not have remarkable

computational power to solve on-demand problems due to memory and/or computing time limitations. And, although it is a field that still has many barriers to overcome, it is important to continue exploring it as it can help model and solve new problems. That is why this study analyzes how both classical and quantum computing work and focuses on two well-known algorithms, Quantum Teleportation and Deutsch-Jozsa, which are implemented in public available quantum computers from IBM and Google.

CLASSICAL COMPUTING

In order to understand how quantum computers work, we must first understand some concepts about classical computing that allows us to use them.

Programming Language

Classical computers use a binary system which is represented by 1's and 0's. These symbols are then interpreted by the computer. The computer uses *logic gates* and circuits to make basic operations with the symbols and a set of instructions for them to follow. Nonetheless, this system becomes complex to human understanding and even more so to program in it. That is why another language is needed, in which we can work in an less abstract upper layer and thus developing programs becomes easier (Patterson & Hennessy, 2014).

That language is known as a *programming language*, which is nothing more than a mixture of syntactic symbols that allows us to build any type of system, program or even an application. *High-level* programs ("the *high-level* term is used to distinguish these from *low-level* assembler languages, which are basically thin wrappers around machine code") (Raymond, 2010) use well-known and easy-to-understand syntax like English so that the programmer can easily give instructions to the machine. Therefore, those instructions have to be translated into machine code, using the binary system, so that the computer can execute them (Patterson & Hennessy, 2014).

Compilers and Interpreters:

Now, usually the programmer is the one who writes those instructions in the preferred high-level language that then will be translated by an *interpreter* or a *compiler* (depending on the language that has been chosen). After the instructions are written, the source file is then passed to the *compiler* or *interpreter* so it can be translated and then turn into an executable program.

The most traditional thing to use are compiled languages which translate source code into machine code before executing it. Only then the processor executes the software, obtaining all the instructions in a binary code before starting so it can determine the order of execution of each instruction.

On the other hand, there are interpreted languages which translate the source code during execution time by using system calls.

Logic Gates:

We have already discussed how programs are built and how they are translated for computers to understand. But what happens after they is translated? Once source code is translated by an *interpreter* or a *compiler*, every program executes the stream of bytes sent as instructions translated to computer's machine language in the steps before. Programming language translates a symbolic version of an instruction into the binary version" (Patterson & Hennessy, 2014). Now we can consider the process in a lower level which is the hardware.

At this level, computers understand only a binary language, so the processor receives a set of instructions to do the operations required. As we have seen, this binary language has 1's and 0's that will be interpreted by electronic devices known as transistors and semiconductors arranged in circuits. Those circuits use *logic gates* that will work with voltage letting the bit pass (1) or not (0) to perform calculations (Patterson & Hennessy, 2014).

These bits will go through the *logic gates* that implement basic logic functions to build logic blocks where computation of bits will take place (Stallings, 2013). In classical computers, we have three types of *logic gates* which are AND, OR and NOT gate. The first one, the AND gate, returns a 1 only when both of the entries are 1. The OR gate returns a 1 only when one of the entries has a high voltage, that is to say it has a 1. Finally, the NOT gate returns the inverse signal; in other words, when the entry is 1, it returns a 0. When the entry is 0, it returns a 1

(Vogt, 2021). One important property of classic *logic gates* is that some of them are not reversible, which will let us know what the initial entries were.

QUANTUM COMPUTING

Now we can explore more about the quantum field. Quantum computers appeared in 1981 after the physicist Richard Feynman studied the atoms in quantum mechanics and encouraged scientists to build one (Angara, Stege, & MacLean, 2020). In those studies, he discovered that in the quantum world, elements work differently and do not follow rules of traditional physics. Here, elements do not necessarily have defined states since they can exist in two or more places at once.

Then, we can define Quantum Computing as the act of performing computation by making use of quantum mechanics, where "quantum" refers to the atomic or subatomic units that the system uses to calculate different (Carrascal, del Barrio, & Botella, 2020). This means that we will be making quantum operations on quantum data using *qubits* but also some resources in real time of classical computation.

Qubit

In this quantum world, computers need *qubits* to process information. They are considered the elementary units of quantum computing information as the quantum generalization of a classic bit.

Representation of *qubits* is summarized in being an orthonormal basis, which describes a set of unitary elements (0 and 1). To denote them, we use a mathematical notation which is in form of matrices and space vectors. So, we have a vector $|q>$ where q represents the value of the quantum state. Those values can be either 1 or 0 and are represented by $|0>$ and $|1>$.

Classical bits use the binary system while qubits are a two-level system. Qubits based their calculations on probabilities as they have a “third” state called *superposition*. This “third” state can be seen as a set of states represented by a linear combination, in which they can hold both states at the same time before it is measured. This means that the qubit can have every state possible. But, those space vectors, also known as state vectors, will eventually “point to a specific direction in space that will correspond to a particular quantum state”. This can be seen using a Bloch Sphere (Qiskit Development Team, 2021). For example, in Figure 1 we can see a qubit in state $|0>$.

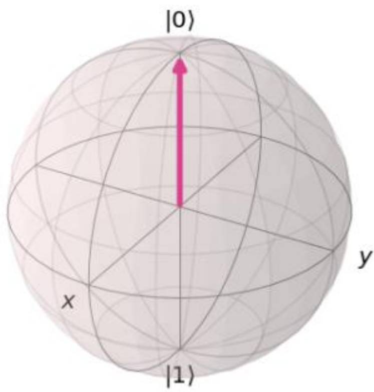


Figure 1: Bloch Sphere

Now, qubits have a second important property called *entanglement* that will allow them to work together as a system. This means that, when two qubits are entangled, they will form a relationship that will not only contain more information but will also keep their states linked. Then, no matter how far qubits are from each other, any change we make on one particle will trigger immediately a change of state in the other one (Carrascal, del Barrio, & Botella, 2020).

Quantum Gates:

As we have seen in classical computing, we use transistors in circuits to build logic gates that will represent bits so that we can make operations with the computer. In quantum computing, we cannot use the same system; therefore, quantum gates are required to create quantum circuits. This quantum circuit will manipulate qubits to perform operations. This

means that quantum gates, unlike logic gates, can take advantage of some of the qubit's properties that we have seen before like superposition and entanglement. In addition to this, the concept of reversibility must be taken into consideration because quantum gates will never lose information and laws of quantum physics will be reversible in time. The "qubits that are entangled on their way into the quantum gate remain entangled on the way out, keeping their information safely sealed throughout the transition" (Roell, 2018).

But how does this system works? A quantum circuit is made up of equal number of output and input wires that will carry the qubits between gates from outputs to inputs and gates where the qubit processing will take place. This system is based on a two level system where the qubits states are represented by vectors as a superposition shown in Equation 1.

$$|0> = \begin{pmatrix}1\\0\end{pmatrix} \qquad |1> = \begin{pmatrix}0\\1\end{pmatrix}$$

Equation 1: Qubit States

Now, to perform calculations over these states, unitary matrices are used. These matrices receive input vectors with a certain number of qubits (n qubit) and since each wire carries a two-state quantum bit, a circuit of n qubits performs a unit operation represented by a unit matrix of $2^n \times 2^2$ (Bengtsson, 2005).

Table 1 shows some of the quantum gates provides by Google and IBM in their quantum computers (Qiskit Development Team, 2021).

Type	Gate	Description	IBM	Google
Classic	NOT - X Gate	Pauli gate: flips states	x	x
	CNOT - CX Gate	Acts wits two qubits (one as control, and the other as target) performing a NOT-gate on target when control qubit is in state 1. Used to create an entanglement when control qubit is in superposition.	x	x
	Toffoli - CCX Gate	acts with 3 qubits (2 as control, 1 as target) to apply a not gate on target when controls are in state 1	x	x
	Swap	swap states of 2 qubits	x	x

<table>
<tr><td></td><td>Identity - I Gate / Wait Gate</td><td>Ensure that nothing is applied to a qubit. Does not change the qubit state. Known as the absence of a gate.</td><td>x</td><td>x</td></tr>
<tr><td rowspan="7">Phase</td><td>T - RZ (pi/4)</td><td>Rotates the qubit in pi/4</td><td>x</td><td>x</td></tr>
<tr><td>S - RZ (pi/2)</td><td>Rotates the qubit in pi/2. Applies this phase to 1 state</td><td>x</td><td>x</td></tr>
<tr><td>Z</td><td>Pauli gate: acts as identity on state 0 and multiplies the sign of the 1 state by -1 so the states are flipped.</td><td>x</td><td>x</td></tr>
<tr><td>T*</td><td>Inverse T</td><td>x</td><td>x</td></tr>
<tr><td>S*</td><td>Inverse S</td><td>x</td><td>x</td></tr>
<tr><td>Phase - P Gate</td><td>Applies an specific phase to |1> state to rotate the qubit. For certain values its equivalent to other phase gates.</td><td>x</td><td>x</td></tr>
<tr><td>RZ</td><td>Rotates the qubit around the z axis by the given phase</td><td>x</td><td>x</td></tr>
<tr><td rowspan="7">Non Unitary</td><td>Reset</td><td>Returns a qubit to state |0></td><td>x</td><td></td></tr>
<tr><td>Measurement</td><td>Used to make measurements in the circuit</td><td>x</td><td>x</td></tr>
<tr><td>Control Modifier</td><td>Yields a gate.</td><td>x</td><td></td></tr>
<tr><td>If</td><td>Applies a conditional on a gate depending on the state of the qubit</td><td>x</td><td></td></tr>
<tr><td>Barrier</td><td>Prevents combination of gates to gain efficiency. Useful for visualizing steps of the circuit.</td><td>x</td><td></td></tr>
<tr><td>Matrix</td><td>A gate defined by its unitary matrix in the form of a NumPy ND array.</td><td></td><td>x</td></tr>
<tr><td>Qubit Permutation</td><td>Performs a permutation on a given set of qubits</td><td></td><td>x</td></tr>
<tr><td>Hadamard</td><td>H</td><td>Rotates the states to make superpositions (considered as the universal gate of quantum computers)</td><td>x</td><td>x</td></tr>
<tr><td rowspan="8">Quantum</td><td>sqrt(x)</td><td>Implements a square. By applying this gate twice we implement the standard X gate. Creates a superposition with a pi/2 phase</td><td>x</td><td></td></tr>
<tr><td>sqrt(x)*</td><td>Inverse of sqrt(x)</td><td>x</td><td></td></tr>
<tr><td>Y</td><td>Pauli Gate: flips states</td><td>x</td><td>x</td></tr>
<tr><td>RX</td><td>Rotates the qubit around the x axis with the given angle</td><td>x</td><td>x</td></tr>
<tr><td>RY</td><td>Rotates the qubit around the y axis with the given angle</td><td>x</td><td>x</td></tr>
<tr><td>RXX</td><td>Ion -trap system. Implements exponent</td><td>x</td><td>x</td></tr>
<tr><td>RZZ</td><td>Requires a single parameter: an angle expressed in radians. This gate is symmetric; swapping the two qubits it acts on doesn’t change anything.</td><td>x</td><td>x</td></tr>
<tr><td>U</td><td>Allows construction of any single qubit gate</td><td>x</td><td></td></tr>
</table>

Table 1: Gates Comparison

Quantum Computers:

As we know, this field has been in studies for not so long, but since then some large companies, like Google, IBM, amongst others, have built quantum computers that are available to anyone that is interested in quantum computing. In the case of IBM, its Quantum Computer has public access since 2016 and Google's since 2018.

These computers take advantage of quantum physics to process information which they do by using quantum processors built with superconducting Josephson junction-based quantum chips and superconducting asymmetric oscillators (Carrascal, del Barrio, & Botella, 2020). IBM's quantum computer uses 65 qubits and Google's quantum computer uses 54.

Actual quantum architecture built by these companies allows us to explore, study the field and develop some basic algorithms with quantum circuits. To use quantum computers, we just have to create an account on Google or IBM platforms and get into their quantum section. These enterprises have also developed some quantum development kits that run in their computers, such as Cirq and Qiskit respectively. We will see them on the following section.

Qiskit and Cirq

These environments allow us to design quantum circuits where we can manipulate quantum gates and qubits to develop algorithms (Qiskit Development Team, 2020). Both Qiskit and Cirq are Python-based frameworks for quantum computing (Google, 2021). Both are open-source software used to simulate quantum circuits using the gates we have seen previously. In essence, they have the same gates; however, the way to use them syntactically is different even though the same programming language is used in both environments.

Qiskit is the IBM quantum framework, which works with noisy quantum computers at different levels (Qiskit Development Team, 2020). That is to say that it provides the necessary

tools for developing quantum algorithms at the lowest level which includes circuits and gates. At this level, Qiskit is “optimizing [circuits and pulses] for the constraints of a particular physical quantum processor, and managing the batched execution of experiments on remote-access backends” (Qiskit Development Team, 2020). This framework has 4 packages (Terra, Ignis, Aer, Ibmq) for working with the different levels (from high level to low level pulses). The description of the functionality of each one is in Table 2 (Carrascal, del Barrio, & Botella, 2020).

Package	Description
Terra	Imports translation functionality to be able to communicate with quantum computers
Ignis	Provides tools for noise characterization & parametrization
Aer	Implements a simulator
Ibmq	Allows remote access to IBM computer

Table 2: Tools

On the other hand, we have Cirq, which is Google's quantum framework. It also works with noisy quantum computers using Noisy Intermediate-Scale Quantum (NISQ) circuits at different levels. But, as IBM's framework, it provides a high-level library for working with quantum circuits.

ALGORITHMS

In this section, we discuss two different quantum algorithms: Quantum Teleportation Deutsch-Jozsa algorithm. We implemented these algorithms using both Google and IBM quantum computers and compared working in both environments. Both algorithms were implemented following the Qiskit documentation (Qiskit Development Team, 2020).

Quantum Teleportation

Quantum Teleportation is a process used to transfer information between two entangled qubits that are in different places. Because of this relationship the information will be shared by the particles (sender and receiver) regardless of the distance. An interesting thing to notice here is that none so the qubits need to know the location of its pair, but they still share their state. This has been demonstrated by a team of scientists at Fermi National Accelerator Laboratory in collaboration with Argonne National Laboratory, Caltech, Northwestern University and industry partners an some partner institutions (Chicago News, 2020). As stated in a paper published in PRX Quantum (Valivarthi, et al., 2020), researchers were able to use teleport information over a distance of 44 [km], demonstrating that long-distance teleportation is possible with fidelity greater than 90%.

In this case, we will be moving the information between qubits on the same computer. To demonstrate it, we will build the circuit and make some measurements on the qubits.
To build Quantum Teleportation algorithm we need three qubits: the message that will be send, the sender and the receiver. With both, sender, and receiver, we need to create the entanglement to establish the communication channel for receiving some classical bits of information. These classical bits, at the end of the experiment, will be used to recover the teleported state. So, first we must set the message which in this case means to initialize q0 in the state we want to send. Then we will apply some quantum gates to create the entanglement and send the information which are: H-Gate, CNOT-Gate, X-Gate and Z-Gate. By using them we will be able to copy the state of the message on q1 and send it through the channel. Finally, and once, we have sent the state, we apply some measurements on the receiver, and we will store its state in the classical bits to get the results. It’s important to emphasize that, after the message was sent, the receiver will have it but the sender won't; that’s why it is called “*Teleportation*”.

Implementation

In this section we will see how to build the Quantum Teleportation algorithm in Qiskit and Cirq. The steps will be the same but we will note some differences in the syntax.

To use Google's computer, we need to use Google Collab to implement the circuit. So the first thing we need to do is install Cirq and also qutip for plotting results. Then we will import, in both cases the necessary modules so we can run the code and initialize the circuit. For this, we will use 3 qubits were Q0 is the qubit that is going to be sent, q1 will send the information and q2 the one that will receive it. In Google's environment, we will use LineQubits because they will help use represent the circuit with qubits as in line, but in IBM's we just declare qubits as Quantum registers. We also need to add 2 classical bits to the circuit so we can get the results, by applying measurements later, and to check if teleportation worked correctly. This step is shown in Table 3.

<table>
<tr><th>Qiskit</th><th>Cirq</th></tr>
<tr><td><pre>
# Modules
import numpy as np
from qiskit import *
from qiskit.extensions import Initialize
from qiskit.quantum_info import Statevector
from qiskit.providers.aer import AerSimulator
from qiskit.quantum_info import random_statevector
from qiskit.ignis.verification import marginal_counts
from qiskit.visualization import plot_histogram,
    plot_bloch_multivector, array_to_latex

Step 1: Create Circuit
Quantum Circuit: 3 qubits & 2 classical bits. Q1 and Q2 will be
    entangled to send the info. Q0 will be send

qubits = QuantumRegister(3, 'q')
bit_1 = ClassicalRegister(1, 'b1')
bit_2 = ClassicalRegister(1, 'b2')

circuit = QuantumCircuit(qubits, bit_1, bit_2)
display(circuit.draw()) # Draw Circuit
</pre></td><td><pre>
# Cirq Installation
try:
    import cirq
except ImportError:
    print(" installing cirq ... ")
    !pip install --quiet cirq
    print(" installed cirq .")

! pip install qutip

# Modules
import cirq
import sympy
import qutip
import random
import cirq_google
import numpy as np
import matplotlib.pyplot as plt

Step 1: Create Circuit
Quantum Circuit with 3 qubits & 2 classical bits. Q1 and Q2
    will be entangled to send the info. Q0 will be send

circuit = cirq.Circuit()
q0, q1, q2 = cirq.LineQubit.range(3) # q0 -> msg, q1 -> sender,
    q1 -> receiver
</pre></td></tr>
</table>

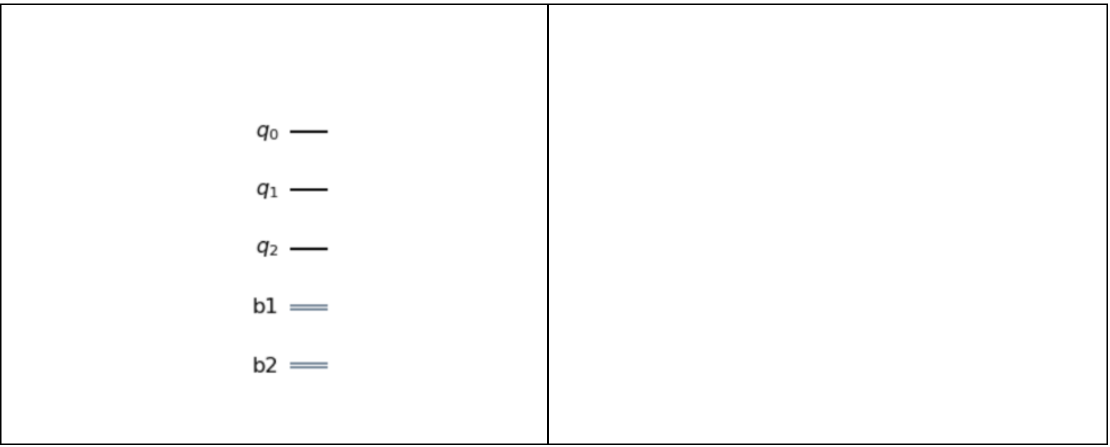


Table 3: Teleportation Algorithm: Step #1

The next thing to do, shown in Table 4 is to initialize the qubit that is going to be sent with the message we want. In this case, we initialize q0 in the state $|0>$.

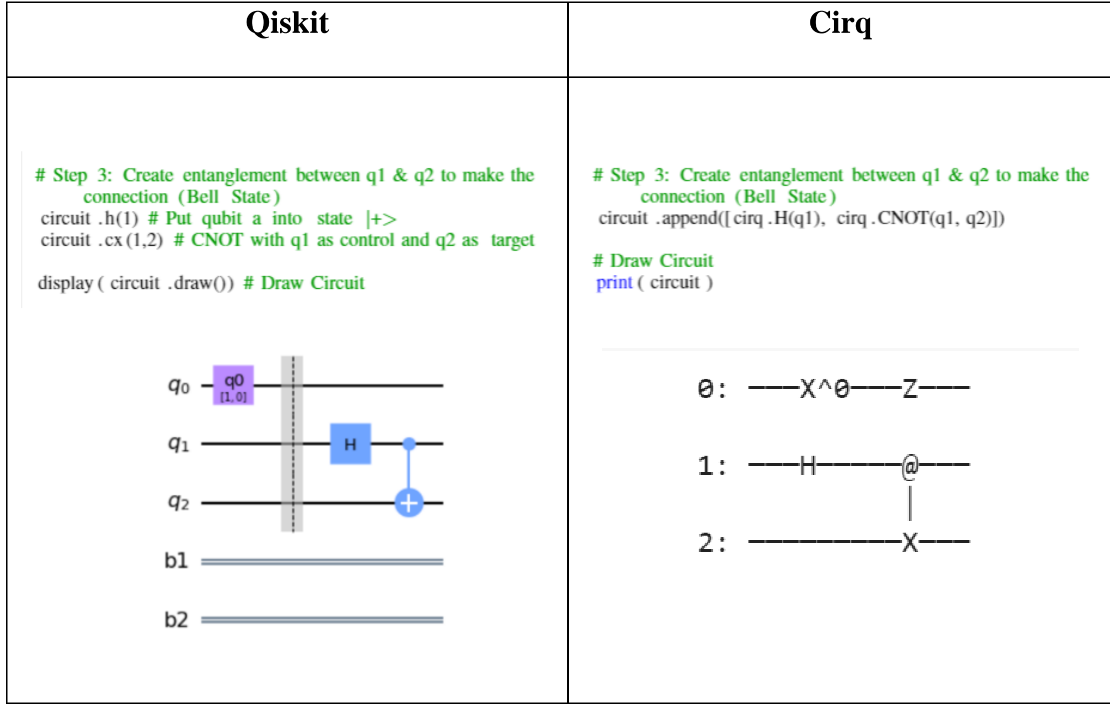
Qiskit	**Cirq**

# Step 2: Inicialization of q0 that is going to be send 	
# Creates the state 	
qubitState = Statevector . from_label ('0')
 	
# Display statevector 	
display (array_to_latex (qubitState , prefix ="|q0\\\\rangle ="))
 	
# Initialize qubit 	
init_gate = Initialize (qubitState)
init_gate . label = "q0"
 	
# Append it to the circuit 	
circuit .append(init_gate , [0])
circuit . barrier ()
display (circuit .draw()) # Draw Circuit
 	
$\|q0\rangle = [1 \quad 0]$
 	
q0 ─[q0 [1, 0]]─┆─
q1 ─────────┆─
q2 ─────────┆─
b1 ═════════
b2 ═════════
	# Step 2: Inicialization of q0 that is going to be send
	# Creates the state
	stateVectorX = 1
	stateVectorY = 0
	
	# Append it to the circuit
	circuit .append([cirq .X(q0) ** stateVectorY , cirq .Z(q0) ** stateVectorX])
	
	# Draw Circuit
	print (circuit)
	
	0: ───X^0───Z───

Table 4: Teleportation Algorithm: Step #2

Now, in Table 5 we will create an entanglement between q1 and q2 applying a Bell State so we can create the channel to send the information through it. After

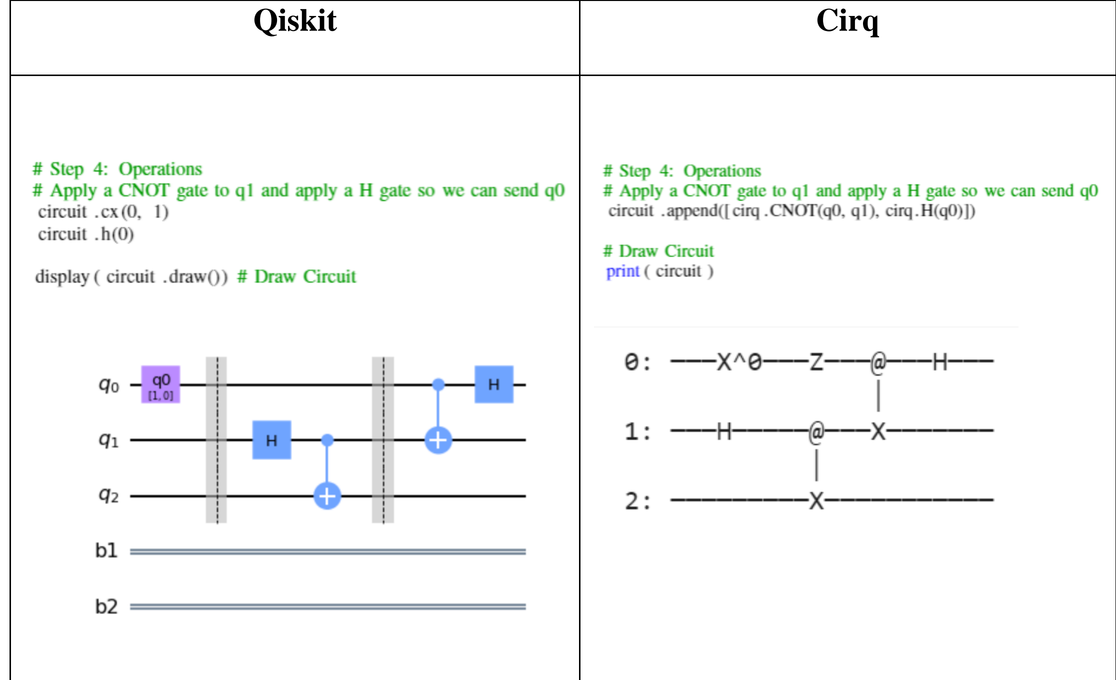
establishing the teleportation channel, both qubits can be physically separated and no matter how far the are they will still share the connection.



Qiskit	Cirq
# Step 3: Create entanglement between q1 & q2 to make the connection (Bell State) circuit .h(1) # Put qubit a into state |+> circuit .cx (1,2) # CNOT with q1 as control and q2 as target display (circuit .draw()) # Draw Circuit	# Step 3: Create entanglement between q1 & q2 to make the connection (Bell State) circuit .append([cirq .H(q1), cirq .CNOT(q1, q2)]) # Draw Circuit print (circuit)

Table 5: Teleportation Algorithm: Step #3

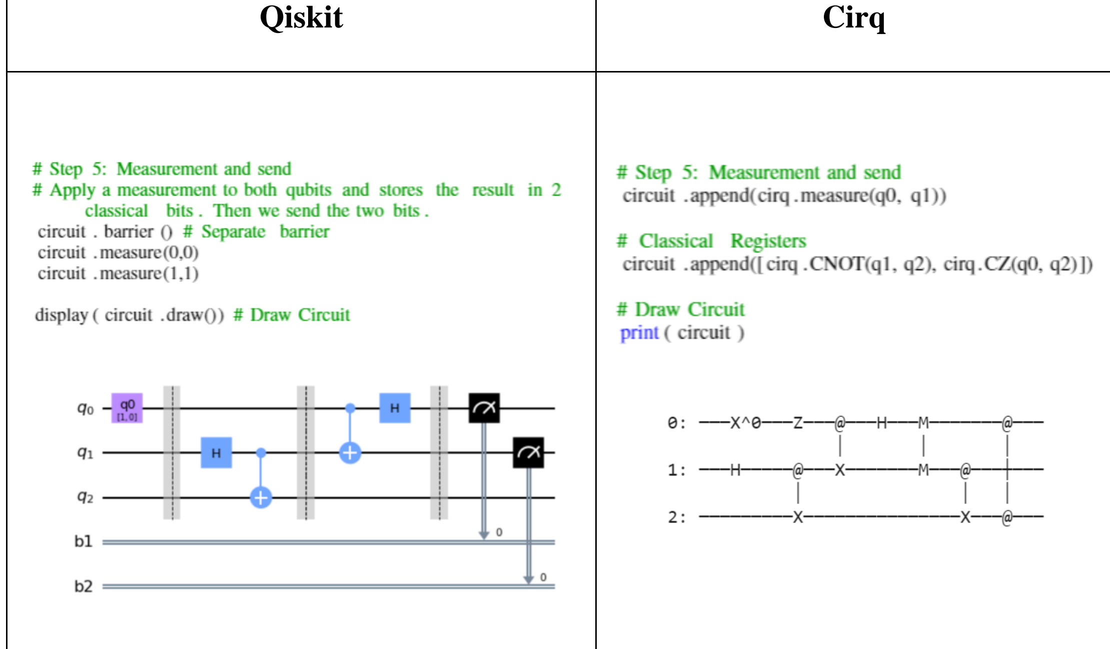
Then, for the teleportation of the state, we will apply a CNOT gate on q1 that will be controlled by q0 (qubit that is going to be sent) and we also apply an H gate on this same qubit shown in Table 6.



Qiskit	Cirq
# Step 4: Operations # Apply a CNOT gate to q1 and apply a H gate so we can send q0 circuit .cx (0, 1) circuit .h(0) display (circuit .draw()) # Draw Circuit	# Step 4: Operations # Apply a CNOT gate to q1 and apply a H gate so we can send q0 circuit .append([cirq .CNOT(q0, q1), cirq .H(q0)]) # Draw Circuit print (circuit)

Table 6: Teleportation Algorithm: Step #4

Now teleportation is done, we make measurements on q0 and q1 and store the results in the classical bits. But we must consider that since q1 and q2 are entangled, the measurement will also affect the state of q2 Table 7 shows this.



Qiskit	Cirq

# Step 5: Measurement and send 	
# Apply a measurement to both qubits and stores the result in 2 classical bits. Then we send the two bits. 	
circuit.barrier() # Separate barrier
circuit.measure(0,0)
circuit.measure(1,1)
 	
display(circuit.draw()) # Draw Circuit
	# Step 5: Measurement and send
	circuit.append(cirq.measure(q0, q1))
	
	# Classical Registers
	circuit.append([cirq.CNOT(q1, q2), cirq.CZ(q0, q2)])
	
	# Draw Circuit
	print(circuit)

Table 7: Teleportation Algorithm: Step #5

Since the transfer of the information was in classical bits, in Table 8 we need to apply some gates depending on the state that is being received. At this point q0 and q1 can be in one of the following states: $|00>, |01>, |10>, |11>$, so the gates will encode the possible phases and bit flip errors in q2. The state can then be established in qubit 2 by applying an X operation (to correct a bit flip) and/or a Z operation (to correct a phase flip).

Qiskit	Cirq

# Step 6: Receive the qubit 	
# Depending on the bits that are received we change the type of gate on q2 (received) 	
# Use c_if to control our gates with a classical bit instead of a qubit 	
circuit.x(2).c_if(bit_2, 1) # Apply gates if the registers
circuit.z(2).c_if(bit_1, 1) # are in the state '1'
 	
display(circuit.draw()) # Draw Circuit
	# Step 6: Receive the qubit
	# Depending on the bits that are received we change the type of gate on q2 (received) to get the message
	sim = cirq.Simulator()
	
	q0 = cirq.LineQubit(0)
	message = sim.simulate(cirq.Circuit([cirq.X(q0) ** stateVectorY, cirq.Z(q0) ** stateVectorX]))
	
	print(circuit) # Draw Circuit

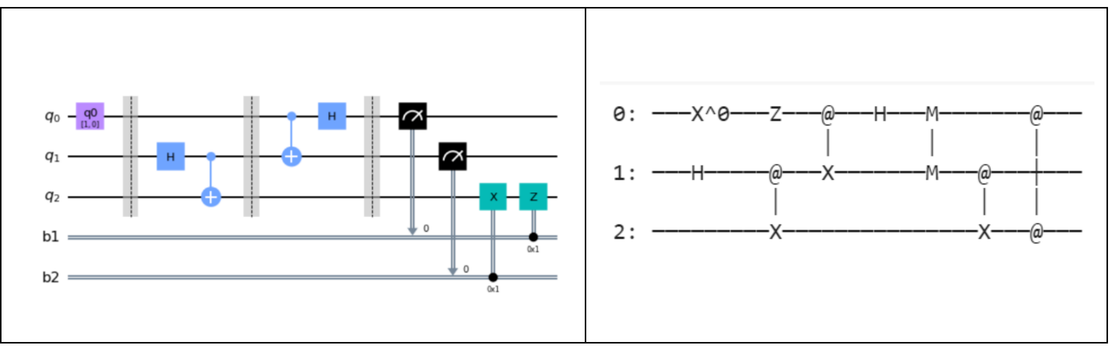


Table 8: Teleportation Algorithm: Step #6

Finally, we can get and see the results by plotting them in Table 9. In case of Cirq, we get the results as a state vector of the three qubits. Since we imported the library qutip, we can plot the Bloch spheres to visualize the states. We can note that q0 and q1 will not change but q2 will. The state at this point of the experiment, the receiver, q2, must be identical to the state we sent previously on the message, q0.

Qiskit	**Cirq**

# Step 7: Result 	
# By runnung this a few times you may notice that the qubits 0 & 1 change states, but qubit 2 is always in the state 	
 	
sim = Aer.get_backend('aer_simulator')
circuit . save_statevector ()
out_vector = sim.run(circuit). result (). get_statevector ()
plot_bloch_multivector (out_vector)
 	
qubit 0 |0⟩ |1⟩ x y
qubit 1 |0⟩ |1⟩ x y
qubit 2 |0⟩ |1⟩ x y
	# Step 7: Result
	# By runnung this a few times you may notice that the qubits 0 & 1 change states, but qubit 2 is always in the state
	print ("\nBloch Sphere q0")
	bloch_q0 =
	cirq . bloch_vector_from_state_vector (message. final_state_vector 0)
	print ("x: ", np.around(bloch_q0[0], 4),
	"y: ", np.around(bloch_q0[1], 4),
	"z: ", np.around(bloch_q0[2], 4),)
	
	blochSphere_q0 = qutip.Bloch()
	vec = [np.around(bloch_q0[0], 4), np.around(bloch_q0[1], 4),
	np.around(bloch_q0[2], 4)]
	blochSphere_q0.add_vectors(vec)
	blochSphere_q0.show()
	
	# Records the final state of the simulation.
	final_results = sim.simulate(circuit)
	
	print ("\nBloch Sphere of q1")
	bloch_q1 = cirq . bloch_vector_from_state_vector (
	final_results . final_state_vector , 1)
	print ("x: ", np.around(bloch_q1[0], 4),
	"y: ", np.around(bloch_q1[1], 4),
	"z: ", np.around(bloch_q1[2], 4),)
	
	blochSphere_q1 = qutip.Bloch()
	vec = [np.around(bloch_q1[0], 4), np.around(bloch_q1[1], 4),
	np.around(bloch_q1[2], 4)]
	blochSphere_q1.add_vectors(vec)
	blochSphere_q1.show()
	
	print ("\nBloch Sphere of q2")

	`bloch_q2 = cirq . bloch_vector_from_state_vector ( final_results . final_state_vector , 2)`
	`print ( "x: ", np.around(bloch_q2[0], 4), "y: ", np.around(bloch_q2[1], 4), "z: ", np.around(bloch_q2[2], 4), )`
	
	`blochSphere_q2 = qutip.Bloch()`
	`vec = [np.around(bloch_q2[0], 4), np.around(bloch_q2[1], 4), np.around(bloch_q2[2], 4)]`
	`blochSphere_q2.add_vectors(vec)`
	`blochSphere_q2.show()`

Table 9: Teleportation Algorithm: Step #7

Deutsch-Jozsa

Deutsch-Jozsa algorithm was the first algorithm to demonstrate that quantum computers can provide better performance than classical computers. This algorithm is used to determine if a function is balance or constant. The algorithm receives a function containing an n-bit string and after evaluating it, the algorithm should return a string of 0 or 1 that will let us know the result. For a balanced function will return 0's at least for the half of the inputs and 1's for the rest. On the other hand, for a constant function, the algorithm will return a string with only just 1's or just 0's.

Usually for solving this problem we must consider the worst case in a classical computer, so the time it will take to solve will increase as entries do. This means that in the worst case, at least half of the inputs plus one will have to be evaluated in order to know whether or not it is constant. That is to say that the cost of solving the algorithm will be $2^{n-1} + 1$ in order to have 100% confidence in the algorithm. This, however, can be solved faster by taking advantage of the resources of a quantum computer by making a single call to the function.

Implementation

In this section we will see how to build the Deutsch-Jozsa algorithm in Qiskit and Cirq. For this, we will need two quantum registers initialized in $|0>$ and $|1>$.

For the next step we will have to apply an H-Gate to each qubit and then create an oracle that will evaluate the function. When the oracle is constant, it will make no changes on qubits and since H-gate is its inverse we can get the initial state. On the other hand, if the oracle is balanced, will change the qubits by adding a negative phase to them. After the oracle has made the evaluation, we will ignore the first qubit and we will apply an H-gate to each qubit on the first register. Finally, we can make measurements to the circuit and we will get the probability of the function being balanced or constant.

Table 10 describes the steps to build the Deutsch-Jozsa algorithm in Qiskit and Cirq. The first thing we need to do is to initialize the modules that we are going to use. Then we will initialize the circuit with 2 oracles: balanced and constant. Both will receive 3 bit as an input. For the constant one, as the input has no effect on the output we will set a random output between 0 and 1.

Qiskit	**Cirq**

`# Modules`
`import numpy as np`
`from qiskit import *`
`from qiskit.extensions import Initialize`
`from qiskit.quantum_info import Statevector`
`from qiskit.providers.ibmq import least_busy`
`from qiskit.providers.aer import AerSimulator`
`from qiskit.quantum_info import random_statevector`
`from qiskit.ignis.verification import marginal_counts`
`from qiskit.visualization import plot_histogram,`
`    plot_bloch_multivector, array_to_latex`
 	
`# Step 1: Create Constant Oracle`
`# Lenght input regiter string`
`n = 3`
	`# Cirq Installation`
	`try:`
	`    import cirq`
	`except ImportError:`
	`    print("installing cirq...")`
	`    !pip install --quiet cirq`
	`    print("installed cirq.")`
	
	`# Modules`
	`import cirq`
	`import random`
	`import cirq_google`
	`import numpy as np`
	`import matplotlib.pyplot as plt`

Qiskit	Cirq

`circuitConstOracle = QuantumCircuit(n + 1)`
 	
`# The input has no effect on the output so set a random output (0/1)`
`output = np.random.randint(2)`
`if output == 1:`
`    circuitConstOracle.x(n)`
 	
`display(circuitConstOracle.draw()) # Draw Circuit`
 	
q_0 ───
q_1 ───
q_2 ───
q_3 ─X─
	`# Step 1: Create Constant Oracle`
	`n = 3`
	`q0, q1, q2, q3 = cirq.LineQubit.range(n + 1)`
	`circuitConstOracle = cirq.Circuit()`
	
	`# The input has no effect on the output so set a random output (0/1)`
	`output = np.random.randint(2)`
	`print(output)`
	`if output == 1:`
	`    circuitConstOracle.append(X(q3))`

Table 10: *Deutsch-Jozsa Algorithm: Step #1*

For the balance oracle, in Table 11 we will use CNOT gates for each input qubit and X-Gates so we can vary the input. Also we are going to use a bit string of length 3 to be evaluated: 101. Now as being evaluated, when the function detects a 0 will do nothing, but if the function detects a 1, we will place an X-Gate to the corresponding qubit.

Qiskit	**Cirq**

`# Step 2: Create Balance Oracle`
`string = "101"`
`circuitBalanceOracle = QuantumCircuit(n + 1)`
 	
`# Place X-gates when input is 1`
`for qubit in range(len(string)):`
`    if string[qubit] == '1':`
`        circuitBalanceOracle.x(qubit)`
 	
`circuitBalanceOracle.draw()`
 	
q_0 ─X─
q_1 ───
q_2 ─X─
q_3 ───
	`# Step 2: Create Balance Oracle`
	`string = "101"`
	`circuitBalanceOracle = cirq.Circuit()`
	
	`# Place X-gates when input is 1`
	`for qubit in range(len(string)):`
	`    if string[qubit] == '1':`
	`        # Check what qubit is to append the gate`
	`        if qubit == 0:`
	`            circuitBalanceOracle.append(X(q0))`
	`        if qubit == 1:`
	`            circuitBalanceOracle.append(X(q1))`
	`        if qubit == 2:`
	`            circuitBalanceOracle.append(X(q2))`
	
	`print(circuitBalanceOracle)`
	
	`0: ───X───`
	
	`2: ───X───`

Table 11: *Deutsch-Jozsa Algorithm: Step #2*

Then, in Table 12 we will place the NOT gates using each input qubit as a control, and the output qubit as a target. This will guarantee that the circuit is balanced.

Qiskit	**Cirq**

# Step 2: Create Balance Oracle 	
circuitBalanceOracle . barrier ()
 	
# Controlled−NOT gates 	
for qubit in range(n):
    circuitBalanceOracle .cx(qubit , n)
 	
circuitBalanceOracle . barrier ()
circuitBalanceOracle .draw()
	# Step 2: Create Balance Oracle
	for qubit in range(n): # Controlled−NOT gates
	    # Check what qubit is to append the gate
	    if qubit == 0:
	        circuitBalanceOracle .append(CNOT(q3, q0))
	    if qubit == 1:
	        circuitBalanceOracle .append(CNOT(q3, q1))
	    if qubit == 2:
	        circuitBalanceOracle .append(CNOT(q3, q2))

Table 12: Deutsch-Jozsa Algorithm: Step #2

To finish the balance oracle we need to make the same evaluation that we made before to apply the X-Gates. This can be seen in Table 13.

Qiskit	**Cirq**

# Step 2: Create Balance Oracle 	
# Place X−gates 	
for qubit in range(len(string)):
    if string [qubit] == '1':
        circuitBalanceOracle .x(qubit)
 	
# Show oracle 	
circuitBalanceOracle .draw()
	# Step 2: Create Balance Oracle
	# Place X−gates
	for qubit in range(len(string)):
	    if string [qubit] == '1':
	        # Check what qubit is to append the gate
	        if qubit == 0:
	            circuitBalanceOracle .append(X(q0))
	        if qubit == 1:
	            circuitBalanceOracle .append(X(q1))
	        if qubit == 2:
	            circuitBalanceOracle .append(X(q2))

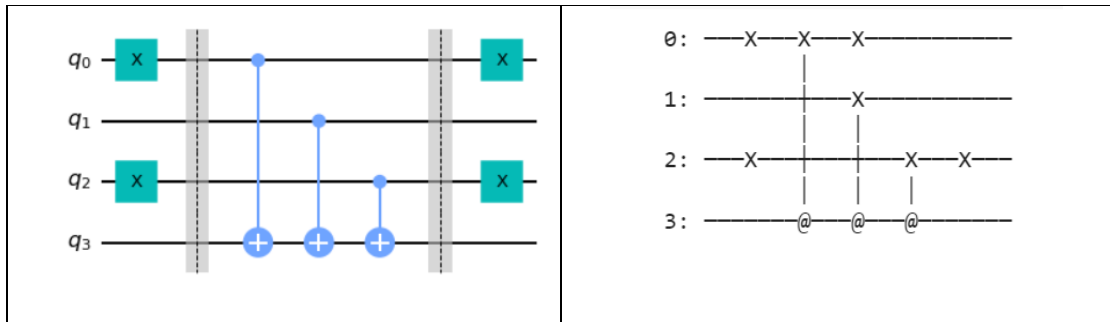


Table 13: Deutsch-Jozsa Algorithm: Step #2

Once we have the two oracles that will predict the result. We can build the complete circuit in Table 14. For this we need to initialize input qubits in a |+> state and output qubits in |−> state.

<table>
<tr><th>Qiskit</th><th>Cirq</th></tr>
<tr>
<td><pre>
# Step 3: Create complete circuit
circuit = QuantumCircuit(n + 1, n)

# Apply H−gates to initialize qubits
for qubit in range(n):
    circuit .h(qubit)

Put qubit in state |−>
circuit .x(n)
circuit .h(n)
circuit .draw()
</pre>
q0 ─H─
q1 ─H─
q2 ─H─
q3 ─X─H─
c 3
</td>
<td><pre>
# Step 3: Create complete circuit
n = 3
q0, q1, q2, q3 = cirq .LineQubit.range(n + 1)
circuit = cirq . Circuit ()

Apply H−gates to initialize qubits
circuit .append([X(q3), H(q0), H(q1), H(q2), H(q3)])

print (circuit)
</pre>
<pre>
0: ───H───────

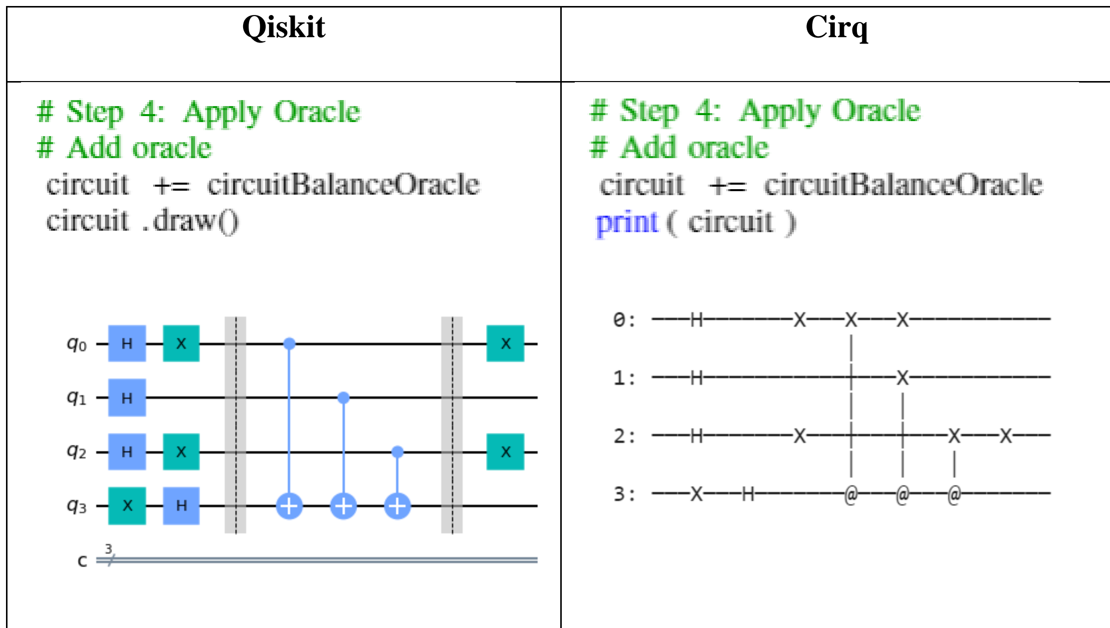
1: ───H───────

2: ───H───────

3: ───X───H───
</pre>
</td>
</tr>
</table>

Table 14: Deutsch-Jozsa Algorithm: Step #3

Then, we will apply the balance oracle we just created. Table 15.



Qiskit	Cirq
# Step 4: Apply Oracle # Add oracle circuit += circuitBalanceOracle circuit .draw()	# Step 4: Apply Oracle # Add oracle circuit += circuitBalanceOracle print (circuit)

Table 15: Deutsch-Jozsa Algorithm: Step #4

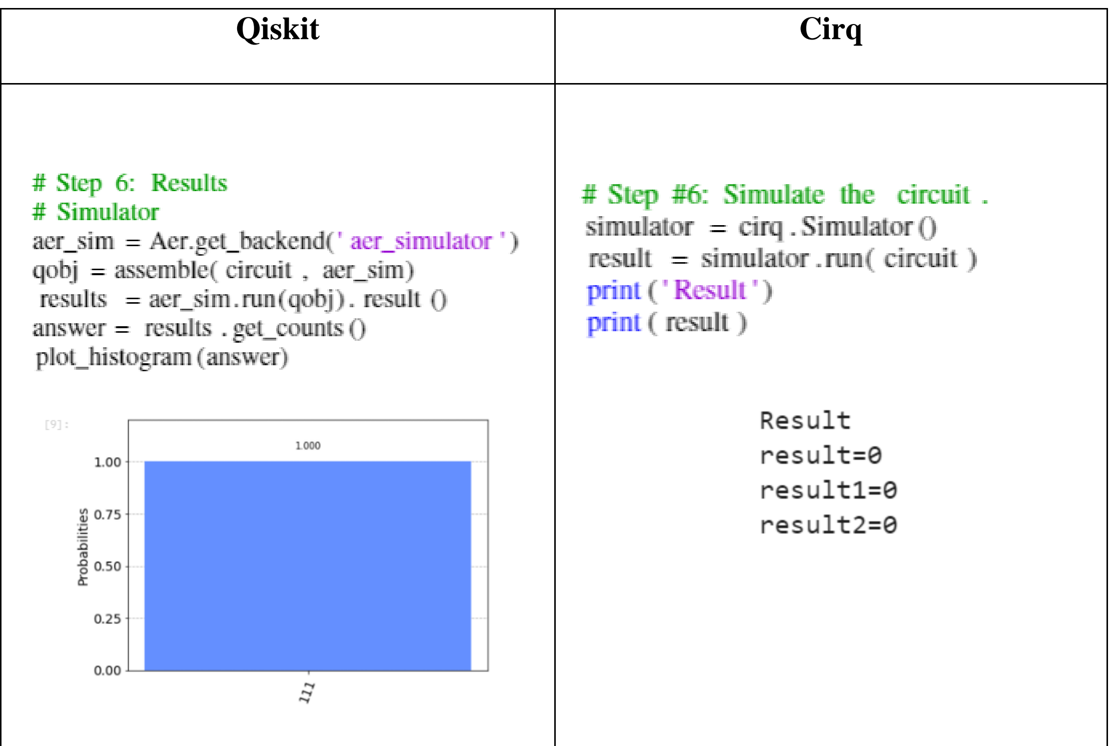
And, in Table 16 we will apply H-Gates on the n-input qubits to finally makes measurements on the input register.

Qiskit	Cirq

# Step 5: Apply H–Gates and measurements 	
# Repeat H–gates 	
for qubit in range(n):
    circuit .h(qubit)
circuit . barrier ()
 	
# Measure 	
for i in range(n):
    circuit .measure(i, i)
 	
# Display circuit 	
circuit .draw()
	# Step 5: Apply H–Gates and measurements
	# Repeat H–gates
	circuit .append([H(q0), H(q1), H(q2)])
	
	# Measure
	circuit .append([measure(q0, key=' result '), measure(q1,
	    key=' result1 '), measure(q2, key=' result2 ')])
	
	# Display circuit
	print (circuit)

Table 16: Deutsch-Jozsa Algorithm: Step #5

Finally, in Table 17 we can get the results by using the simulator. Here we can observe that the model predicted that the function is balanced.



Qiskit	Cirq

# Step 6: Results 	
# Simulator 	
aer_sim = Aer.get_backend(' aer_simulator ')
qobj = assemble(circuit , aer_sim)
results = aer_sim.run(qobj). result ()
answer = results . get_counts ()
plot_histogram (answer)
	# Step #6: Simulate the circuit .
	simulator = cirq . Simulator ()
	result = simulator .run(circuit)
	print (' Result ')
	print (result)
	
	Result
	result=0
	result1=0
	result2=0

Table 17: Deutsch-Jozsa Algorithm: Step #6

RESULTS AND DISCUSSIONS

In this study, we have seen how both, classical and quantum computers work. Based on quantum theory, we have also built the circuits for two algorithms in different environments that not only demonstrated the potential of this technology but also allowed us to see the differences between them.

By analyzing both environments we could see that in a general view, they are good tools for creating and using quantum circuits. However, if you are about to start exploring this field, I personally recommend using Qiskit as IBM has more documentation that guides you through, which makes it easier to understand. It also has libraries included that help to visualize the states of the qubits and results. Even if you still do not know how to create the circuits using code, IBM has a section (Quantum Composer) with visual tools in which you build the circuit only by dragging the gates and you can easily visualize the results. On the other hand, if you

are an experienced user in this field, using Cirq should not be a difficult challenge. Even with Google platform you could take advantage of some hardware details that allows to maximize the effective utilization of processors but if you want to visualize the results you will have to import some additional libraries. As we have seen, both are a Python based environment, so the syntax is and easy to learn and understand. But, in comparison between them, Cirq and Qiskit differ in the use of quantum gates, measures and variables as we saw in the previous table.

In addition, quantum computing has a great advantage over classical computing. I will mention a few examples. They can solve problems that grow exponentially so they have a better performance (Carrascal, del Barrio, & Botella, 2020). They compute difficult factoring large numbers in few times so it not only simplifies mechanisms used in cryptography, but they could be more secure. Also, and if appropriate networks for quantum computers are built, they could be able to share information at incredible fast speeds. With a quantum network we could also distribute easily qubits between different locations (despite the distance), which is a an essential thing for quantum cryptography, distributed quantum computing, communications and sensing (Valivarthi, et al., 2020). “But keeping this information flow stable over long distances has proven extremely difficult due to changes in the environment including noise. Researchers are now hoping to scale up such a system, using both entanglements to send information and quantum memory to store it as well” (Banafa, 2021). This will make communications faster than they are known and may reach greater distances.

However, we could see that quantum computing is still very limited and they are not yet at the level of classical computing. We still need the use of classical computers to make some simulations and get the results but, this will change over time. If quantum computers keep developing steadily, they will eventually take over classical computers (Qiskit Development Team, 2020), we expect that quantum hardware and software will keep

improving as it is still under study and research. For now, this field is under study and is basically used for testing, random sampling. Nevertheless, scientists are still looking for better techniques to control quantum states and building architectures that will let use this technology in an easier way. But this is not the only thing necessary to continue developing this area. First, environments, compilers and programming languages that allow us to manage the gates, prepare the states and make measurements must be made so it can be easier to understand, and developers could make programs from a higher level.

With this, and because of its potential, it will lead us to new things that will probably make a huge difference in society. Just as before they did not imagine the influence of the classical computer in our lives, so it is with quantum computing. The technological advances that can be made with it are unimaginable. We are moving towards this little by little, and we must prepare for this change. We have seen how vapor machines led us to the first big change in the first industrial revolution, electricity to the second, energy to the next one and now, that we are in the fourth revolution, we cannot live without computing. Will quantum technology become the next big technological revolution in human history?

REFERENCES

Angara, P. P., Stege, U., & MacLean, A. (2020). Quantum Computing for High-School Students. *International Conference on Quantum Computing and Engineering*.

Banafa, A. (2021, 02 01). Quantum Teleportation: Facts and Myths. *Open Mind BBVA*. Retrieved from https://www.bbvaopenmind.com/en/technology/digital-world/quantum-teleportation-facts-and-myths/

Bengtsson, A. K. (2005). *Quantum Computation: A Computer Science Perspective.*

Carrascal, G., del Barrio, A. A., & Botella, G. (2020, 07 06). First experiences of teaching quantum computing. *The Journal of Supercomputing*.

Chicago News. (2020, 12 28). Fermilab and partners achieve sustained, high-fidelity quantum teleportation. *Chicago News*.

Google. (2021). Quantum AI. *Explore the possibilities of quantum*. Retrieved from https://quantumai.google/

Holton, W. C. (2021, 08 17). Quantum Computer. (E. Britannica, Ed.) Retrieved from https://www.britannica.com/technology/quantum-computer

IBM Corporation. (2016). Operations Glossary. *IBM Quantum Composer*. Retrieved from https://quantum-computing.ibm.com/composer/docs/iqx/operations_glossary

LaRose, R. (2021). Review of the Cirq Quantum Software Framework. *Quantum Computin Report*. Retrieved from https://quantumcomputingreport.com/review-of-the-cirq-quantum-software-framework/

Marchenkova, A. (2019, 10 7). Programming for Quantum Computing: What language should you learn? Retrieved from https://www.linkedin.com/pulse/programming-quantum-computing-what-language-should-you-marchenkova/

Microsoft. (2021, 11 30). Understanding quantum computing. *Azure Quantum Documentation*. Retrieved from https://docs.microsoft.com/en-us/azure/quantum/overview-understanding-quantum-computing

Patterson, D. A., & Hennessy, J. L. (2014). *Computer Organization and Design* (5 ed.). Morgan Kaufmann.

Qiskit Development Team. (2020). Introduction to Quantum Computing. Retrieved from https://qiskit.org/textbook-beta/course/introduction-course/

Qiskit Development Team. (2021, 10 08). Learn Quantum Computation using Qiskit. *Qiskit*. Retrieved from https://qiskit.org/textbook/ch-states/single-qubit-gates.html

QuTech. (2021). Qubit basis states. *Quantum Inspire*. Retrieved from https://www.quantum-inspire.com/kbase/qubit-basis-states/

Raymond, E. (2010, 07 31). The Unix and Internet Fundamentals HOWTO. Retrieved 09 15, 2021, from https://tldp.org/HOWTO/Unix-and-Internet-Fundamentals-HOWTO/index.html

Roell, J. (2018, 02 18). Demystifying Quantum Gates — One Qubit At A Time. *Towards Data Science*. Retrieved from https://towardsdatascience.com/demystifying-quantum-gates-one-qubit-at-a-time-54404ed80640

Shaik, E. h. (2020). Implementation of Quantum Gates based Logic. *Pondicherry University*.

Stallings, W. (2013). *Computer Organization and Architecture.* New Jersey: Pearson.

Sutter, P. (2021, 05 26). What is quantum entanglement? *Live Science*. Retrieved from https://www.livescience.com/what-is-quantum-entanglement.html

Valivarthi, R., Davis, S. I., Peña, C., Xie, S., Lauk, N., Narváez, L., . . . Ginn, Y. (2020, 12 4). Teleportation Systems Toward a Quantum Internet. *PRX QUANTUM*.

Vogt, J. A. (2021). An Introduction to Quantum Computing. *University of Applied Sciences*.

Yanosfsky, N., & Mannucci, M. (2012). Quantum Computing for Computer Scientists. *Cambridge University Press*.