**A Survey on Quantum Computing**

**Operators, Protocols, Algorithms, and Applications.**

**A Project Report submitted in partial fulfillment of the requirements for the award of the degree of**

**BACHELOR OF TECHNOLOGY IN**

**COMPUTER SCIENCE AND ENGINEERING**

**Submitted by**

### Anurag K S V, 121810303007

### Surya N, 121810303061

### Kamala Sree, 121810303062

**Under the esteemed guidance of**

**Dr. Bhaskara Rao B**

**Associate Professor**



### DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING GITAM

**(Deemed to be University)**

**VISAKHAPATNAM**

**April 2022**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING GITAM INSTITUTE OF TECHNOLOGY**

**GITAM**

### (Deemed to be University)



**DECLARATION**

I/We, hereby declare that the project report entitled “**A Survey on Quantum Computing Operators, Protocols, Algorithms, and Applications.**” is an original work done in the Department of Computer Science and Engineering, GITAM Institute of Technology, GITAM (Deemed to be University) submitted in partial fulfillment of the requirements for the award of the degree of B.Tech. in Computer Science and Engineering. The work has not been submitted to any other college or university for the award of any degree or diploma.

Date: 13 April 2022

### Registration No(s) Name(s) Signature(s)

### 121810303007 Anurag K S V

### 121810303061 Surya N

### 121810303062 Kamala Sri Marepalli

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING GITAM INSTITUTE OF TECHNOLOGY**

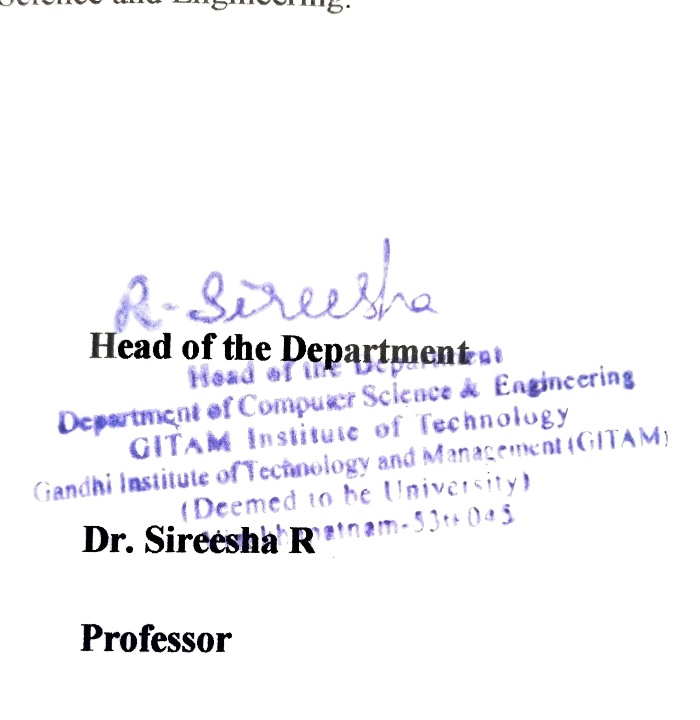
**GITAM**

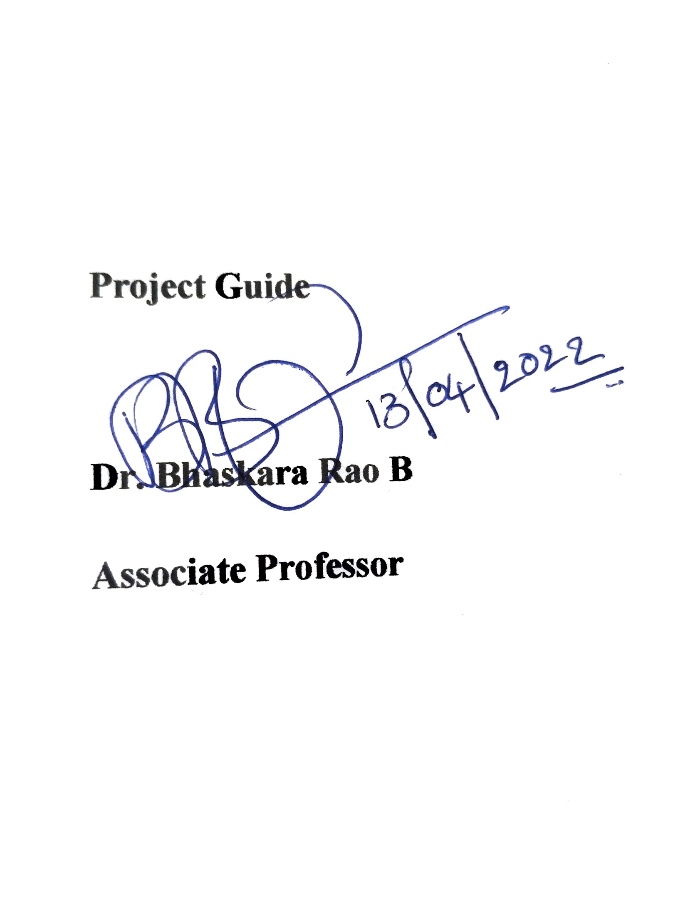
**(Deemed to be University)**



**CERTIFICATE**

This is to certify that the project report entitled “**A Survey on Quantum Computing Operators, Protocols, Algorithms, and Applications.**” is a bonafide record of work carried out by **Anurag K S V (121810303007), Surya N (121810303061), Kamala Sri Marepalli (121810303062)** students submitted in partial fulfillment of the requirement for the award of the degree of Bachelors of Technology in Computer Science and Engineering.





|  |  |
| --- | --- |
| **Associate Professor Professor** |  |
|  |  |
|  |  |

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| 1. | Abstract | 1 |
| 2. | Introduction | 2 |
| 3. | Literature Review | 3 |
| 4. | Objectives | 6 |
| 5. | Overview of Technologies | 7 |
|  | 5.1. Qiskit [IBM] | 7 |
|  | 5.2. Wheel & PyTest | 7 |
|  | 5.3. Bloch Sphere | 7 |
| 6. | Methodology | 8 |
|  | 6.1. Design of the Quantum Library | 8 |
|  | 6.2. Structure of a Quantum Program | 9 |
| 7. | Implementation & Results | 10 |
|  | 7.1. Building the Package | 10 |
|  | 7.2. Quantum Operators | 10 |
|  | 7.3. Quantum Teleportation | 17 |
|  | 7.4. Superdense Coding | 21 |
|  | 7.5. Grover’s Search | 25 |
|  | 7.6. Truly Random Byte Generator | 28 |
|  | 7.7. Quantum Walks | 30 |
|  | 7.8. setup.py | 33 |
| 8. | Conclusion & Future Scope | 34 |
| 9. | References | 35 |
|  |  |  |

### ABSTRACT

Quantum computers are expected to surpass the classical computational capabilities in due time and have a revolutionary impact on many industries. The implementation and simulations of Quantum Algorithms in the current Noisy Intermediate-Scale Quantum (NISQ) era have promising results for us to contribute to this field, which has many famous algorithms such as Grover’s Search Algorithm, which promises a quadratic boost to search technique over classical counterparts, Shor’s Algorithm for factoring huge numbers; when implemented has the potential to break the fundamentals of current day cryptography, and fundamental applications such as Generation of Truly Random Numbers which work on principles drawn from the nature of Quantum Physics Phenomenon of Superposition and Entanglement. This project focuses on reviewing the fundaments of Quantum Computing via implementation of Quantum Operators, simulation of Quantum Algorithms on Quantum Simulators using available Quantum Programming development libraries such as Qiskit, Cirq, and Applications based on Gate-Based Quantum Computers, which trend towards better optimization and time complexity compared to their classical computing counterparts.

### INTRODUCTION

Quantum Computing refers to the usage of Quantum Mechanics phenomena such as Superposition and Entanglement for computation, providing us with a significant boost in computational capabilities compared to classical approaches to specific problems [10]. The field of quantum computing takes its origin in Quantum Information Science which consists of Quantum Computation, Quantum Communication, and Quantum Sensing. This work focuses on Quantum Computation, whose roots are embedded in Quantum Mechanics, Quantum Information Theory, and Computer Science [9].

Quantum Computing is a rapidly evolving field that has taken significant strides in the 21st century, compelling us to review the fundamentals of this field from time to time. The current stage of quantum computing is called the Noisy Intermediate-Scale Quantum era [1], as we are yet to develop an error-free Quantum Computer scaled to produce significant results. The current quantum computers should be scaled up to at least >106 physical qubits for error-free resilient quantum computation for practical purposes.

Despite the fact that Quantum Information Science has significantly progressed theoretically than practically due to engineering challenges, we successfully created simulators that could simulate various quantum hardware to carry out our research findings in the field to understand the true nature of such a computing capability.

### LITERATURE REVIEW

Quantum Computation’s fundamental driving forces originate in Quantum Mechanics via Superposition, Entanglement, and Reversibility. However, they can be easily understood via the language of Mathematics using Linear Algebra in Vectors, Complex Numbers, Set Theory, Matrices, Linear Transformations, Hermitian Operators, and Hilbert Spaces. Due to this sole reason, we have successfully implemented the advantages of quantum computing to various types of problem classes theoretically to prove computational advantages via complexity theory.

While the discussions above talk about the representation of Quantum Computation theoretically for understanding, a physical device that aims to represent a Quantum Computer has to satisfy specific criteria outlined by David DiVincenzo in his work [2], which states, A Quantum Computer should:

1. Be a scalable physical system with well-characterized Qubits.
2. Have the ability to initialize the state of Qubits to a simple fiducial state.
3. Have Long appropriate decoherent time for the Qubits states.
4. Work on a Universal set of Quantum Gates.
5. Have a Qubit-Specific strong measurement capability.

Currently, we have multiple ways to build a Quantum Computer physically. Some of the implementations include Neutral Atoms, Superconducting Qubits, Topological Quantum Computation, Trapped Ion, etc. The current state of physical development of Quantum Computers is akin to 1940s and 1950s classical computation [9]. The current NISQ era of Quantum Computation needs Quantum Error Correction to implement important Quantum algorithms. The current error correction techniques require a thousand (1000) physical qubits to generate an error-free resilient logical qubit [9].

Now that we established a criterion for a system to be accepted as a Quantum Computer, let us look at various metrics on which a Quantum Computer is assessed for comparison [3]:

1. Universality – Is the system Turing-Complete?
2. Fidelity – A qubit can remain coherent throughout a computation cycle. Objectively it is ‘1 – Error Rate’.
3. Scalability – Does the architecture sustain a scalable upscaling of the system to 106 qubits and beyond?
4. Circuit Depth – Number of operations that can be performed before decoherence comes into the picture for the qubits.
5. Logical Connectivity – How are qubits connected? Can we implement any set of logical operators on any set of qubits?
6. Cloud Access – Is the system accessible to consumers via Cloud?
7. Qubits – How many error-free qubits does the system provide for computation?

The smallest unit of physical information in a Quantum Computer is a Quantum Bit, better known as a Qubit. A system of n Qubits can handle 2(n) states [9]. Each Qubit state can be represented in Vectors (in Dirac Notation) or the Block Diagram. To change the state of these Qubits, we use Unitary Quantum Operators that are reversible. Various types of Quantum Operators (Unitary, Binary, Ternary) and Quantum algorithms are represented in the forms of Quantum Circuit Diagrams.

Though we are yet to build a robust error-free Quantum Computer, researchers, with the help of corporations, are successful in building Quantum Computer Simulators [9] which follow the basic method of:

Where,

= Simulated Result State

= Quantum Circuit (Unitary Transformation)

= Wave Function

A concern with Quantum Simulation is that the classical system suffers from an Exponential Explosion of Memory as we increase the number of Qubits in the simulations [9]. Alternatively, applying Clifford Gates helps us negate this burden, but these gates are not universal.

All the current Quantum Algorithms fall under one of the five principal quantum algorithmic paradigms or are a hybrid of two or more of them [4], which include:

1. Quantum Fourier Transformation (QFT)
2. Grover Operator (GO)
3. Harrow – Hassidim – Lloyd (HHL) method for linear systems
4. Variational Quantum Eigen Value Solver (VQE)
5. Direct Hamiltonian Simulation (SIM)

Finally, to program a quantum algorithm on a quantum computer, we use various Quantum Programming Languages. Some of which are functional such as Qiskit, LIQUi|⟩, Q#, and Quipper, while others are imperative languages such as Cirq, Scaffold, and ProjectQ.

### OBJECTIVE

The rapid advancements in quantum computing are hard to keep track of and do not have a distinct learning path defined to enter the field. This survey attempts to bring together the theory, learning, and practical implementations being carried out in the field by developing a quantum computing library accessible to all via the python programming language. This library consists of various fundamentals of quantum computing such as qubit representation via Bloch Spheres, Quantum Logic-Gates or Operators, simulation of Protocols used for Quantum Computation, some algorithms, and various application simulations currently possible with classical computation. It gives us an overview of the field while focusing on the practical implementation of concepts to better understand Quantum Computation.

### OVERVIEW OF TECHNOLOGIES

### Qiskit [IBM]

Quantum Information Science KIT (Qiskit) is an open-source quantum computing library provided by IBM for all practical implementation of quantum systems [11]. We have four core modules under Qiskit, which include:

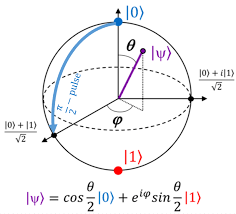
1. Qiskit Terra: For writing quantum programs at the level of circuits and pulses followed by optimizing them.
2. Qiskit Aer: A C++ simulator framework to perform noisy models’ simulations of errors during execution on real devices.
3. Qiskit Ignis: Framework to understand and mitigate noise in quantum circuits and devices.
4. Qiskit Aqua: Contains cross-domain quantum algorithms upon which apps for near-term quantum computers can be built.

### Wheel and PyTest

Wheel python package is used to build the codebase into a package for distribution and local deployment of the package for testing [12]. Post this; we can publish the package in PyPI, a python package index repository.

PyTest is a testing library for python which primarily focuses on testing various API or databases by enabling us to write scalable and straightforward test cases for it [13].

### Bloch Sphere



**Figure 5.1: Representation of Bloch Sphere in 3D Space. [5]**

A three-dimensional representation of a qubit in a superposition of states is called a Bloch Sphere, as shown in Fig. 5.1.

A set of gates that let us reach any point on the block sphere enable universal computation.

### METHODOLOGY

### Design of the Quantum Library

The ‘qulib’ package designed during the implementation of this project followed the following structure:

|

|----- qulib

| |

| |----- \_\_init\_\_.py

| |----- Fundamentals

| | |----- Operators.py

| |----- Protocols

| | |----- QuantumTeleportation.py

| | |----- SuperDenseCoding.py

| |----- Algorithms

| | |----- GroverSearch.py

| |----- Applications

| |----- TruelyRandomByte.py

| |----- QuantumWalk.py

|----- tests

| |----- testing.py

|-----README.md

|-----setup.py

Here, we have the package structure containing various classes of concepts being simulated which include:

1. Fundamentals – We work on Quantum Operators or Quantum Gates, a universal paradigm of Quantum Computing, understanding various unary, binary, and ternary qubit operators or gates.
2. Protocols - are the fundamental communication circuits built using operators, including Quantum Teleportation and Superdense Coding.
3. Algorithms – We will be mainly surveying and understanding one of the Oracular Paradigm Algorithm, Grover’s Search, in this class of concepts.
4. Applications – We simulate the generation of truly-random byte and quantum walks to understand the potential of quantum computing in real-world scenarios.

### Structure of a Quantum Program

We will be principally working with the Qiskit library from IBM, which follows a procedural programming language’s paradigm, emphasizing how we acquire results or approach a specific problem statement.

A generic pseudo-code for any quantum program in Qiskit is as follows:

1. Import Qiskit module/library
2. Initialize Quantum Register with a name and required number of qubits.
3. Initialize the Classical register with a name and required number of classical bits.
4. Initialize the Quantum Circuit using Quantum and Classical registers.
5. Perform operations via Quantum Operators on the qubits in Quantum registers.
6. Measure the final states of each Qubit and store them in the classical bits present in the classical register.
7. Acquire a backend from the Qiskit library to perform simulation.
8. Create a job to execute the circuit on the simulator.
9. Get and print the results of the job.

Generation of Quantum Registers & Classical Registers

Performing Measurement from Q.Reg to C.Reg

Creation of Quantum Circuits on Q.Reg

Simulation of Circuit on the backend initialized

Generation of Results

**Figure 6.1: Flow Chart depicting the control flow and various steps involved in constructing a Quantum Program using the Qiskit library.**

### IMPLEMENTATION & RESULTS

### Building the Package

Upon creating the structure of the library and coding each of its .py files, we build the library locally using the Wheel package using the following commands in the command prompt.

> python setup.py bdist\_wheel

> pip install /path/to/wheelfile.whl

### Quantum Operators

### Concept

Quantum Operators are the fundamental building blocks for universal gate-based quantum computing, providing us with various unary, binary, and ternary operators who work on the qubits to perform various bit-level computations. Let us discuss the discrete classification and functions of the above mentions Quantum Operators in detail:

### Unary Operators: Work on single Qubit.

### Pauli Group -

### X – Quantum NOT Operator

### Y – Rotates state-vector about Y-Axis.

### Z – Rotates state-vector about Z-Axis.

### Hadamard Operator (H) - used to take a qubit from a definite computational state to the superposition of two states.

### General Phase Shift Operator (Rφ):

### S Operator if φ = π/2

### T Operator if φ = π/4

### Z Operator is a particular case of Rφ with φ = π

### Measurement (M) - used to measure the value of a qubit and record it.

### Binary Operators: Work on two Qubits.

### Controlled Not Operator (CNOT) – We apply the NOT operation in a controlled fashion. Based on the value of the control qubit, the value of the target qubit is switched, i.e.,

### | or |

### Controlled Z Operator (CZ) - We apply the Z Operation based on the value of the control qubit; the target qubit experienced a phase shift of π.

### SWAP Operator (SWAP) - Here we swap the values of both the qubits to which the operator is applied to i.e.,

### | or |

### Ternary Operators: Work on three Qubits.

### Toffoli Operator (CCNOT) - Based on the value of two control qubits, the target qubit is applied with a NOT operation i.e.,

### or |

### Fredkin Operator (CSWAP) - Based on the value of a single control qubit, the target qubits values are swapped i.e.,

### or |

### Algorithm

1. Create a parameterized function with the name of the quantum operator & initialize the parameters based on the operator’s requirement.
2. Initialize the Quantum Circuit with the required number of qubits and classical bits (1:1 for Unary, 2:2 for Binary, and 3:3 for Ternary).
3. Simulate the Circuit and Return the Circuit Diagram and the results produced post-simulation.

### Code

'''

## Unitary Operators

1. Pauli Group Operators: X, Y, Z

2. General Phase Shift Operator: R, P, S, T

3. Hadamard Operator: H

4. Identity Operator: I

---

## Binary Operators

1. Controlled NOT Operator (CNOT)

2. Control Operator (CZ)

3. SWAP Operator (SWAP)

---

## Ternary Operator

1. Toffoli Operator (CCNOT)

2. Fredkin Operator (CSWAP)

'''

**import** **qiskit**

**from** **qiskit.providers.aer** **import** AerSimulator

**from** **qiskit** **import** QuantumCircuit

**from** **math** **import** pi

#Quantum NOT Operator [X]

**def** **x**():

#prepare circuit

qc = QuantumCircuit(**1**,**1**)

qc.x(**0**)

qc.measure(**0**,**0**)

#simulaton & result

result = AerSimulator().run(qc).result()

**print**('**\n**result: ',result.get\_counts())

**return** qc.draw(output='mpl')

#Quanutm [Y] Operator

**def** **y**():

#prepare circuit

qc = QuantumCircuit(**1**,**1**)

qc.y(**0**)

qc.measure(**0**,**0**)

#simulaton & result

result = AerSimulator().run(qc).result()

**print**('**\n**result: ',result.get\_counts())

**return** qc.draw(output='mpl')

#Quanutm [Z] Operator

**def** **z**():

#prepare circuit

qc = QuantumCircuit(**1**,**1**)

qc.z(**0**)

qc.measure(**0**,**0**)

#simulaton & result

result = AerSimulator().run(qc).result()

**print**('**\n**result: ',result.get\_counts())

**return** qc.draw(output='mpl')

#Quanutm [R] Operator

**def** **r**(phase):

#prepare circuit

qc = QuantumCircuit(**1**,**1**)

qc.rz(phase, **0**)

qc.measure(**0**,**0**)

#simulaton & result

result = AerSimulator().run(qc).result()

**print**('**\n**result: ',result.get\_counts())

**return** qc.draw(output='mpl')

#Quanutm [P] Operator

**def** **p**(phase):

#prepare circuit

qc = QuantumCircuit(**1**,**1**)

qc.p(phase,**0**)

qc.measure(**0**,**0**)

#simulaton & result

result = AerSimulator().run(qc).result()

**print**('**\n**result: ',result.get\_counts())

**return** qc.draw(output='mpl')

#Quanutm [S] Operator: R-Gate with Phase = Pi/2

**def** **s**():

#prepare circuit

qc = QuantumCircuit(**1**,**1**)

qc.s(**0**)

qc.measure(**0**,**0**)

#simulaton & result

result = AerSimulator().run(qc).result()

**print**('**\n**result: ',result.get\_counts())

**return** qc.draw(output='mpl')

#Quanutm [T] Operator: R-Gate with Phase = Pi/4

**def** **t**():

#prepare circuit

qc = QuantumCircuit(**1**,**1**)

qc.t(**0**)

qc.measure(**0**,**0**)

#simulaton & result

result = AerSimulator().run(qc).result()

**print**('**\n**result: ',result.get\_counts())

**return** qc.draw(output='mpl')

#Quantum [H] Operator

**def** **h**():

#prepare circuit

qc = QuantumCircuit(**1**,**1**)

qc.h(**0**)

qc.measure(**0**,**0**)

#simulaton & result

result = AerSimulator().run(qc).result()

**print**('**\n**result: ',result.get\_counts())

**return** qc.draw(output='mpl')

#Quantum [I] Operator

**def** **i**():

#prepare circuit

qc = QuantumCircuit(**1**,**1**)

qc.i(**0**)

qc.measure(**0**,**0**)

#simulaton & result

result = AerSimulator().run(qc).result()

**print**('**\n**result: ',result.get\_counts())

**return** qc.draw(output='mpl')

#Controlled NOT [CNOT] Operator

**def** **cx**(a, b):

#preparing state-vectors

a = [**1**,**0**] **if** a == **0** **else** [**0**,**1**]

b = [**1**,**0**] **if** b == **0** **else** [**0**,**1**]

#prepare circuit

qc = QuantumCircuit(**2**,**2**)

qc.initialize(a,**0**)

qc.initialize(b,**1**)

qc.cx(**0**,**1**)

qc.measure(**0**,**0**)

qc.measure(**1**,**1**)

#simulaton & result

result = AerSimulator().run(qc).result()

**print**('**\n**result: ',result.get\_counts())

**return** qc.draw(output='mpl')

#Control [CZ] Operator

**def** **cz**(a, b):

#preparing state-vectors

a = [**1**,**0**] **if** a == **0** **else** [**0**,**1**]

b = [**1**,**0**] **if** b == **0** **else** [**0**,**1**]

#prepare circuit

qc = QuantumCircuit(**2**,**2**)

qc.initialize(a,**0**)

qc.initialize(b,**1**)

qc.cz(**0**,**1**)

qc.measure(**0**,**0**)

qc.measure(**1**,**1**)

#simulaton & result

result = AerSimulator().run(qc).result()

**print**('**\n**result: ',result.get\_counts())

**return** qc.draw(output='mpl')

#SWAP [SWAP] Operator

**def** **swap**(a, b):

#preparing state-vectors

a = [**1**,**0**] **if** a == **0** **else** [**0**,**1**]

b = [**1**,**0**] **if** b == **0** **else** [**0**,**1**]

#prepare circuit

qc = QuantumCircuit(**2**,**2**)

qc.initialize(a,**0**)

qc.initialize(b,**1**)

qc.swap(**0**,**1**)

qc.measure(**0**,**0**)

qc.measure(**1**,**1**)

#simulaton & result

result = AerSimulator().run(qc).result()

**print**('**\n**result: ',result.get\_counts())

**return** qc.draw(output='mpl')

#Toffoli [CCNOT] Operator

**def** **ccx**(a, b):

#preparing state-vectors

a = [**1**,**0**] **if** a == **0** **else** [**0**,**1**]

b = [**1**,**0**] **if** b == **0** **else** [**0**,**1**]

#prepare circuit

qc = QuantumCircuit(**3**, **3**)

qc.initialize(a, **0**)

qc.initialize(b, **1**)

qc.ccx(**0**, **1**, **2**)

qc.measure(**0**, **0**)

qc.measure(**1**, **1**)

qc.measure(**2**, **2**)

#simulaton & result

result = AerSimulator().run(qc).result()

**print**('**\n**result: ',result.get\_counts())

**return** qc.draw(output='mpl')

#Fredkin [CSWAP] Operator

**def** **cswap**(a, b, c):

#preparing state-vectors

a = [**1**,**0**] **if** a == **0** **else** [**0**,**1**]

b = [**1**,**0**] **if** b == **0** **else** [**0**,**1**]

c = [**1**,**0**] **if** c == **0** **else** [**0**,**1**]

#prepare circuit

qc = QuantumCircuit(**3**, **3**)

qc.initialize(a, **0**)

qc.initialize(b, **1**)

qc.initialize(c, **2**)

qc.cswap(**0**, **1**, **2**)

qc.measure(**0**, **0**)

qc.measure(**1**, **1**)

qc.measure(**2**, **2**)

#simulaton & result

result = AerSimulator().run(qc).result()

**print**('**\n**result: ',result.get\_counts())

**return** qc.draw(output='mpl')

### Result

The following code imports the qulib library and calls five operators for which the outputs are shown below with the NOT gate and H gate in Fig 8.1, whose result states that upon simulating each of the circuits for 1024 times, we get an output of '1' for NOT (X) Gate and '1': 519, '0': 505 times. Fig 8.2. shows Controlled NOT and SWAP Gates where CX has 1,0 as input giving 1,1 as output. Fig 8.3. shows a ternary operator Controlled SWAP with one as the control bit and the value of target qubits being 1,0, we get the output 1 for control Qubit, while the target qubit values being swapped to 0,1.

**from** **qulib** **import** Operators **as** op

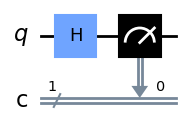
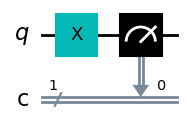
op.x()

op.h()

op.cx(**1**, **0**)

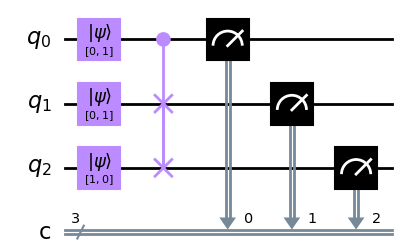
op.swap(**1**, **1**)

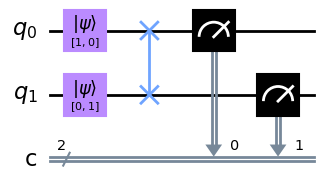
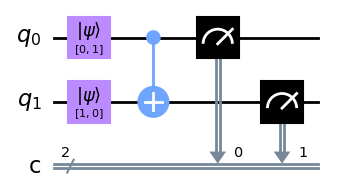
op.cswap(**1**, **1**, **0**)



Result: {‘1’: 1024} Result: {‘1’: 519, ‘0’:505}

**Fig 8.1: Unary Operators (X, H)**





Result: {‘1’: 1024} Result: {‘1’: 519, ’0’: 505}

**Fig 8.1: Unary Operators (X, H)**

Result: {‘11’: 1024} Result: {‘11’: 1024}

**Fig 8.2: Binary Operators (CX, SWAP)**

Result: {‘101’: 1024}

**Fig 8.3: Ternary Operator (CSWAP)**

### Quantum Teleportation

### Concept

A process by which the state of a single qubit is transmitted to another person using two classical bits of classical computation and a bell pair [14]. Quantum Teleportation was discovered after 70 years since the inception of Quantum Mechanics. It was developed in 1993 and verified in 1997. Entangled states / EPR / Bell Pairs are the principal resource in Quantum Teleportation.

### Algorithm

Let us consider a Person A, who has a qubit with state, who wants to transmit it securely to Person B. To accomplish this, we do the follows:

### Setup the three-qubit system with Person A having the required qubit Q with state along with two additional qubits, say, R and S.

### Person A prepares a Bell state with qubits R and S. Done by applying Hadamard Operator to R then applying CNOT Operator between R and S qubits with R as the control qubit. Then, Person A sends S to Person B.

### Person A performs a Bell measurement on the qubit Q and half of the EPR pair R by performing CNOT Operator on qubits Q and R, having Q as the control-qubit. Then perform Hadamard Operator on Q followed up by measuring both the Qubits.

### Person A now has two bits of classical information after measurement. The value (00 or 01 or 10 or 11) is sent to Person B over a classical communication channel.

### Based on the value received by Person B, he/she performs a set of operations on the qubit Q received as part of the Bell Pair, which are shown in Table 7.1 below.

|  |  |
| --- | --- |
| Person A Classical String | Person B Operations |
| 0 0 | I (Correct State) |
| 0 1 | Z |
| 1 0 | X |
| 1 1 | X Z (apply Z first, then X) |

**Table 7.1: Classical Bit String transmitted by A and corresponding Operator Applied by B**

### Code

#For Circuit Creation, Measurement, and Simulation

**from** **qiskit** **import** Aer, assemble

**from** **qiskit** **import** QuantumCircuit, QuantumRegister, ClassicalRegister

#Bloch Sphere Visualization

**from** **qiskit.visualization** **import** plot\_bloch\_multivector, plot\_histogram

#To intailize the state of Qubit for Quantum Teleportation

**from** **qiskit.extensions** **import** Initialize

**from** **qiskit.quantum\_info** **import** random\_statevector

**def** **QuantumTeleportation**():

qq = QuantumRegister(**1**, name='qq')

qr = QuantumRegister(**1**, name='qr')

qs = QuantumRegister(**1**, name='qs')

cq = ClassicalRegister(**1**, name='cq')

cr = ClassicalRegister(**1**, name='cr')

qc = QuantumCircuit(qq, qr, qs, cq, cr)

psi = random\_statevector(**2**)

qq\_init= Initialize(psi)

qq\_init.label = "init\_state"

**def** **initialize\_psi**():

qc.append(qq\_init, qq)

qc.barrier()

**return** plot\_bloch\_multivector(psi)

**def** **bell\_pair\_generation**():

qc.h(qr)

qc.cx(qr,qs)

qc.barrier()

#return qc.draw(output='mpl')

**def** **encode**():

qc.cx(qq,qr)

qc.h(qq)

qc.barrier()

#return qc.draw(output='mpl')

**def** **measure**():

qc.measure(qq,cq)

qc.measure(qr,cr)

qc.barrier()

#return qc.draw(output='mpl')

**def** **decode**():

qc.x(qs).c\_if(cr, **1**)

qc.z(qs).c\_if(cq, **1**)

#return qc.draw(output='mpl')

**def** **final\_states**():

sim = Aer.get\_backend('aer\_simulator')

qc.save\_statevector()

out\_vector = sim.run(qc).result().get\_statevector()

**return** plot\_bloch\_multivector(out\_vector)

**def** **disentangle**():

inverse\_qq\_init = qq\_init.gates\_to\_uncompute()

qc.append(inverse\_qq\_init, qs)

initialize\_psi()

bell\_pair\_generation()

encode()

measure()

decode()

final\_states()

disentangle()

qc.draw(output='mpl')

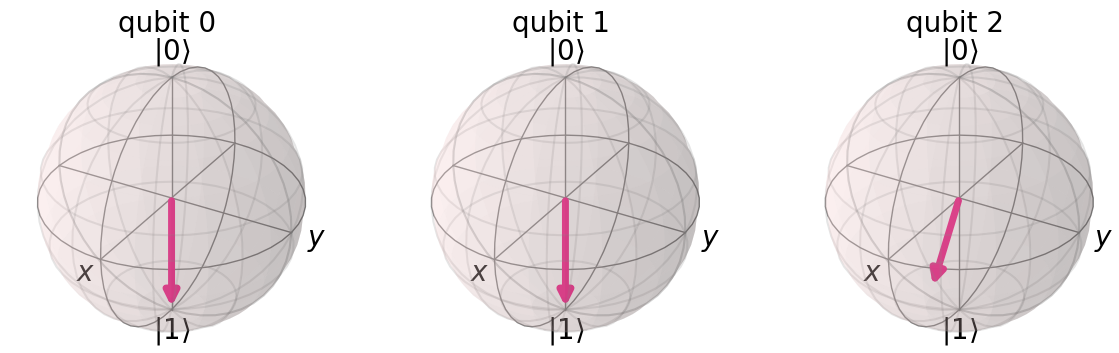
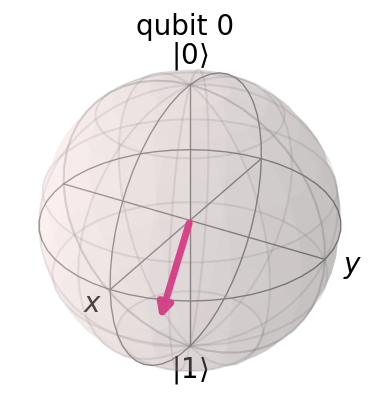
### Result

The qulib library has the function QuntaumTeleportation() to implement the protocol directly. We consider a random start state initialized with qubit qq. The following code represents the implementation of Quantum Teleportation using the qulib library.

**from** **qulib** **import** QuantumTeleportation **as** qt

qt.QunatumTeleportation()

The initial state vector of the qubit qq (qubit 0) is represented in fig. 7.4(a) in the form of a Bloch Sphere. Fig 7.4(b) shows the resultant state vectors of all the qubits' qq, qr, and qs'. The state of qubit qs (qubit 2) is the same as qq (qubit 0) from fig. 7.4(a) verifying the protocol of teleportation of qubit 'qq' state via quantum entanglement between 'qr' and 'qs'. Fig 7.5 shows the whole teleportation circuit diagram in action, starting from the initialization of qubit 'qq' followed by bell pair generation, encoding, measurement, transportation, decoding, and finally checking the final states of qubits.

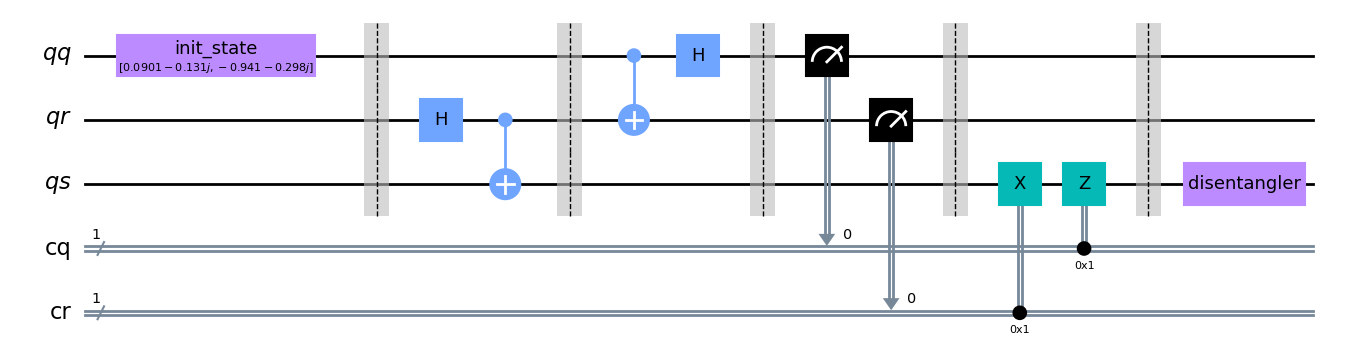


**7.4 (a) 7.4 (b)**

**Fig 7.4: (a) Initial State of qubit 0 (qq) in the form of Bloch Sphere**

**(b) Final States of qubits 0,1, 2 (qq, qr, qs) in the form of Bloch Sphere**

### Superdense Coding



**Fig 7.5: Quantum Circuit for Quantum Teleportation**

### Concept

Superdense Coding is a procedure used to transmit two classical bits of information using a single Qubit for communication. It is essentially the opposite process of Quantum Teleportation. The protocol had its origins in the hands of Bennett and Wiesner [6]. Later it was standardized as a secure form of communication [7]. In the year 1995, Superdense Coding was practically demonstrated by Zeilinger [8].

### Algorithm

Let us consider Alpha, who wishes to transmit two classical bits of information (00 or 01 or 10 or 11) to Beta using one qubit. The algorithm we follow is as follows:

1. A third-party Gamma creates an EPR pair between two qubits via applying Hadamard Operator and then a CNOT Operator to the pair of qubits. Gamma then sends one qubit to Alpha and another to Beta.
2. Alpha then encodes his/her qubit by performing a set of operations on the entangled qubit based on the message to be sent, detailed in Table 7.2 below.

|  |  |
| --- | --- |
| Message | Alpha’s Operations |
| 0 0 | I |
| 0 1 | X |
| 1 0 | Z |
| 1 1 | Z X (applying X first, then Z) |

**Table 7.2: Message to be transmitted by Alpha and corresponding Operator Applied**

1. Alpha sends his/her entangled qubit to Beta via an entanglement preserving quantum communication channel.
2. Here, Beta decodes the qubit received by applying the CNOT Operator first between qubit received from Alpha and his/her own qubit, followed by a Hadamard Operator on Alpha qubit.
3. Post decoding Beta performs measurement resulting in two classical bits of information, the required message from Alpha.

### Code

#For Circuit Creation, Measurement, and Simulation

**from** **qiskit** **import** BasicAer, Aer, assemble, execute

**from** **qiskit** **import** QuantumCircuit, QuantumRegister, ClassicalRegister

#Bloch Sphere Visualization

**from** **qiskit.visualization** **import** plot\_bloch\_multivector, plot\_histogram

**def** **SuperdenseCoding**():

epr = QuantumRegister(**1**, name='epr')

qubit = QuantumRegister(**1**, name='qubit')

c\_epr = ClassicalRegister(**1**, name='c\_epr')

c\_qubit = ClassicalRegister(**1**, name='c\_qubit')

qc = QuantumCircuit(epr, qubit, c\_epr, c\_qubit)

**def** **msg\_generator**():

msg\_circ = QuantumCircuit(**2**,**2**)

msg\_circ.h(**0**)

msg\_circ.h(**1**)

msg\_circ.measure(**0**,**0**)

msg\_circ.measure(**1**,**1**)

backend = BasicAer.get\_backend('qasm\_simulator')

result = execute(msg\_circ, backend, shots=**1**).result()

**for** i **in** result.get\_counts().keys():

**return** str(i)

msg = msg\_generator()

**def** **epr\_pair\_generation**():

qc.h(qubit)

qc.cx(qubit, epr)

qc.barrier()

**def** **encode**():

**if** msg[**1**] == '1':

qc.x(qubit)

**if** msg[**0**] == '1':

qc.z(qubit)

**if** msg == '00':

qc.i(qubit)

qc.barrier()

**def** **decode**():

qc.cx(qubit, epr)

qc.h(qubit)

**def** **simulate**():

qc.measure(qubit, c\_qubit)

qc.measure(epr, c\_epr)

backend = BasicAer.get\_backend('qasm\_simulator')

result = execute(qc, backend, shots=**1**).result().get\_counts()

**print**('Message: {0}**\n**Result: {1}'.format(msg, result))

**return** result

epr\_pair\_generation()

encode()

decode()

simulate()

qc.draw(output='mpl')

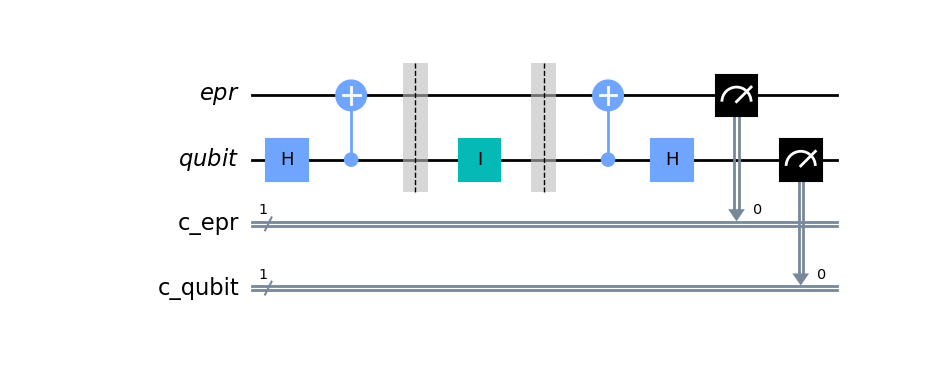
### Result

The qulib library has the function SuperdenseCoding() to implement the protocol.

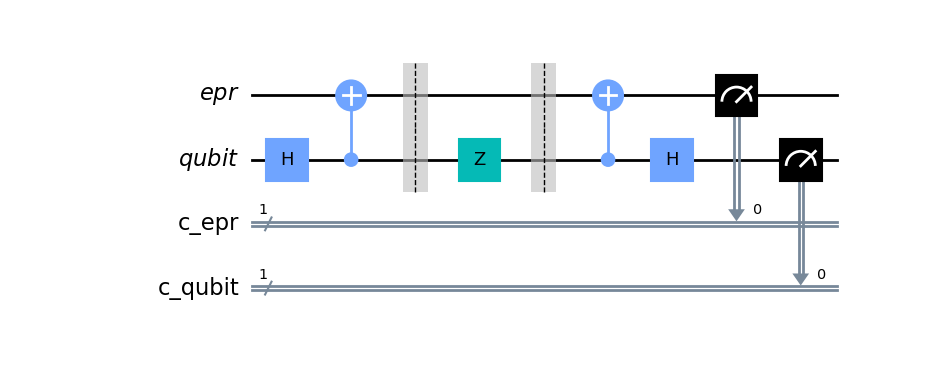
**from** **qulib** **import** SuperdenseCoding **as** sc

sc.SuperdenseCoding()

Upon executing the code snippet shown above, we will observe one of four different output cases based on the input message string. We look at each one in the circuit diagrams shown below.

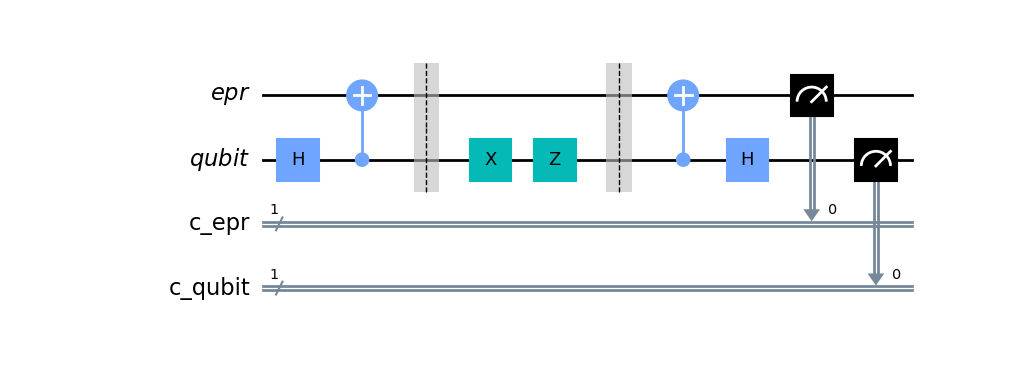
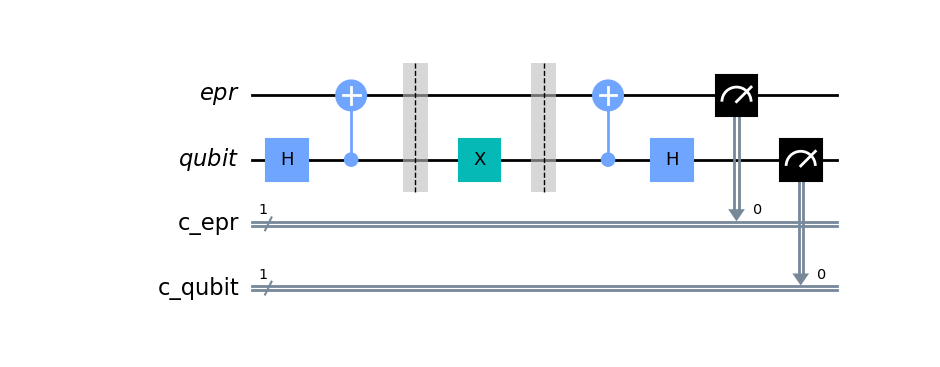


**Fig 7.6: Quantum Circuit for Message 00 using Superdense Coding**



**Fig 7.8: Quantum Circuit for Message 10 using Superdense Coding**

**Fig 7.7: Quantum Circuit for Message 01 using Superdense Coding**



**Fig 7.8: Quantum Circuit for Message 11 using Superdense Coding**

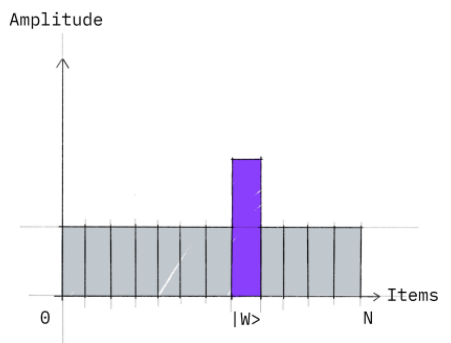
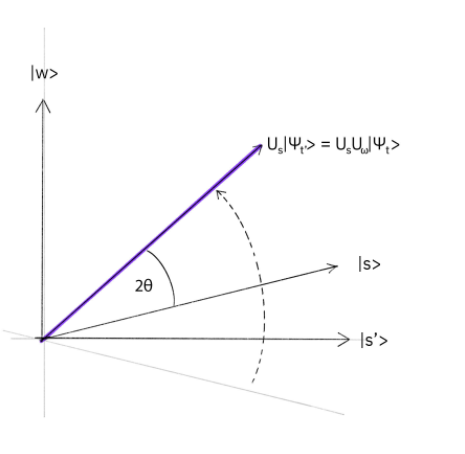
### Grover’s Search

### Concept

Grover's Search is named after the computer scientist Lov Grover who showed a quadratic speed-up in search algorithm using quantum computing in 1996 [15]. This algorithm finds a single element among N unordered list of elements. This algorithm follows the Oracular algorithm paradigm, where the function behind computation is unknown. However, we have an Oracle, which we can query to understand the inputs and outputs, mapping them to a function to find the output.

### Algorithm

Grover's Search has three main implementation steps: state preparation, Grover's oracle, and the diffusion operator [16]. Let us look at the sequential implementation of these steps:



**Fig 7.9: Vector Representation of Reflection and Graph representing amplitude amplification. [16]**

1. We apply Hadamard gates to each input qubits to form the search space, which defines all possible outcomes or answers.
2. Based on the search space, we construct Grover's oracle, where we flip the sign of the target value (reflect over orthogonal) using various quantum operators.
3. We then do amplitude amplification by using the reflection operator, where we reflect the target value on the original search space to get an amplified probability, as shown in figure 7.9.
4. Grover's Oracle + Reflection Operator together are called the Grover's Diffusion Operator. We perform Grover's Diffusion for N times to get the Search's desired output.

### Code

**from** **qiskit** **import** QuantumCircuit, Aer, execute

**import** **numpy** **as** **np**

**def** **ClassicalSearch**():

search\_list = ['00', '01', '10', '11']

**def** **classical\_oracle**(ip\_element):

winner = '11'

**if** ip\_element **is** winner:

response = True

**else**:

response = False

**return** response

**for** i,x **in** enumerate(search\_list):

**if** classical\_oracle(x) **is** True:

**print**('Winner found at index: {0}'.format(i))

**print**('Oracle calls: {0}'.format(i+**1**))

**break**

**def** **GroverSearch**():

#define oracle

oracle = QuantumCircuit(**2**, name = 'Oracle')

oracle.cz(**0**,**1**)

oracle.to\_gate()

oracle.draw(output = 'mpl')

#Checking StateVectors After Oracle

sv\_back = Aer.get\_backend('statevector\_simulator')

grover\_part = QuantumCircuit(**2**,**2**)

grover\_part.h([**0**,**1**])

grover\_part.append(oracle, [**0**,**1**])

job\_part = execute(grover\_part, sv\_back)

result\_part = job\_part.result()

sv = result\_part.get\_statevector()

np.around(sv,**2**)

**print**(sv)

#Reflection Operator

reflection = QuantumCircuit(**2**, name = 'reflection')

reflection.h([**0**,**1**])

reflection.z([**0**,**1**])

reflection.cz(**0**,**1**)

reflection.h([**0**,**1**])

reflection.to\_gate()

reflection.draw(output = 'mpl')

#Complete Circuit Post Oracle + Reflection

backend = Aer.get\_backend('qasm\_simulator')

grover\_circ = QuantumCircuit(**2**,**2**)

grover\_circ.h([**0**,**1**])

grover\_circ.append(oracle, [**0**,**1**])

grover\_circ.append(reflection, [**0**,**1**])

grover\_circ.measure([**0**,**1**],[**0**,**1**])

grover\_circ.draw(output = 'mpl')

job = execute(grover\_circ, backend, shots=**8**)

result = job.result()

**print**(result.get\_counts())

### Result

To implement Grover's Search using the qulib library, we use the contrast between the classical approach and the quantum approach, where we search for the string '11' in the search space (00, 01, 10, 11).

**from** **qulib** **import** GroverSearch **as** gs

gs.ClassicalSearch()

gs.GroverSearch()

The ClassicalSearch() function returns the output:

Winner found at index: 3

Oracle calls: 4

Which states that the winner '11' is found at index 3 of the given search list [00, 01, 10, 11], and the number of calls required to the classical oracle is 4 (worst case scenario). Note that the best-case scenario for such an event would be one call to the oracle, meaning the Time Complexity of such a search would be in the order of O(n).

On the other hand, the GroverSearch() function returns the output:

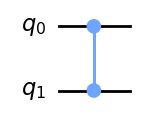
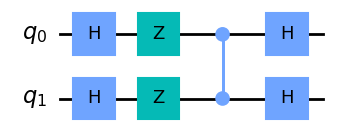
Statevector([ 0.5+0.j, 0.5+0.j, 0.5+0.j, -0.5+0.j],

dims=(2, 2))

{'11': 8}

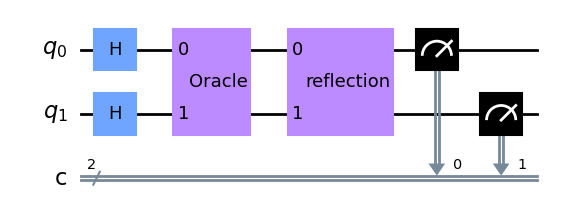
The above output shows the state vectors of the search space after reflection via oracle and the output of the search after simulating the quantum circuit shown in figure 7.9 for eight times with one call to the Grover’s oracle every time, giving us the output as ‘11’.

The quantum circuit and the Grover's Oracle for the above problem are shown in figure 7.10, and that of the reflection operator is shown in figure 7.11. Note that these quantum circuits change based on the search space and the outputs we seek using Grover's Search algorithm.



**Fig 7.11: Grover’s Oracle**

**Fig 7.12: Grover’s Reflection Operator**



**Fig 7.10: Quantum Circuit for Grover’s Search Algorithm**

### Truly Random Byte Generation

### Concept

The use of random numbers has extensively been helpful in various cryptographic techniques during the age of computation. However, most of our methods currently use pseudo-random number generation, where the series of numbers generated have a random source. Suppose we can decode the random source or understand the series pattern produced for a considerable amount of time. In that case, there is a possibility to predict the following pseudo-random number.

The nature of quantum mechanics allows us to generate truly random numbers using the Hadamard Gate, which puts a qubit in a superposition of 0 and 1 states which upon measurement may yield 0 or 1 with random possibilities every time it is measured.

### Algorithm

1. Initialize a quantum circuit consisting of eight qubits and eight classical bits of information on the quantum and classical registers.
2. Simultaneously apply Hadamard Gate to every qubit.
3. Measure the values of each qubit and record them to the classical register.
4. Simulate the circuit to generate a truly random byte of data.

### Code

#importing modules from Qiskit library

**from** **qiskit** **import** QuantumRegister, ClassicalRegister, QuantumCircuit, BasicAer, execute

**def** **TruelyRandomByte**():

#initializing quantum register

q = QuantumRegister(**8**)

#initializing classical register

c = ClassicalRegister(**8**)

#initializing quantum circuit

circuit = QuantumCircuit(q, c)

#applying H-Gate to qubits

**for** i **in** range(**8**):

circuit.h(q[i])

#measuring qubit values

circuit.measure(q, c)

#creating backend for simulation

b = BasicAer.get\_backend('qasm\_simulator')

#executing circuit on backend

j = execute(circuit, b, shots = **1**)

#calculating result

result = j.result()

#listing all results

ls = result.get\_counts().keys()

#returning result in decimal form

**for** i **in** ls:

**return** int(i,**2**)

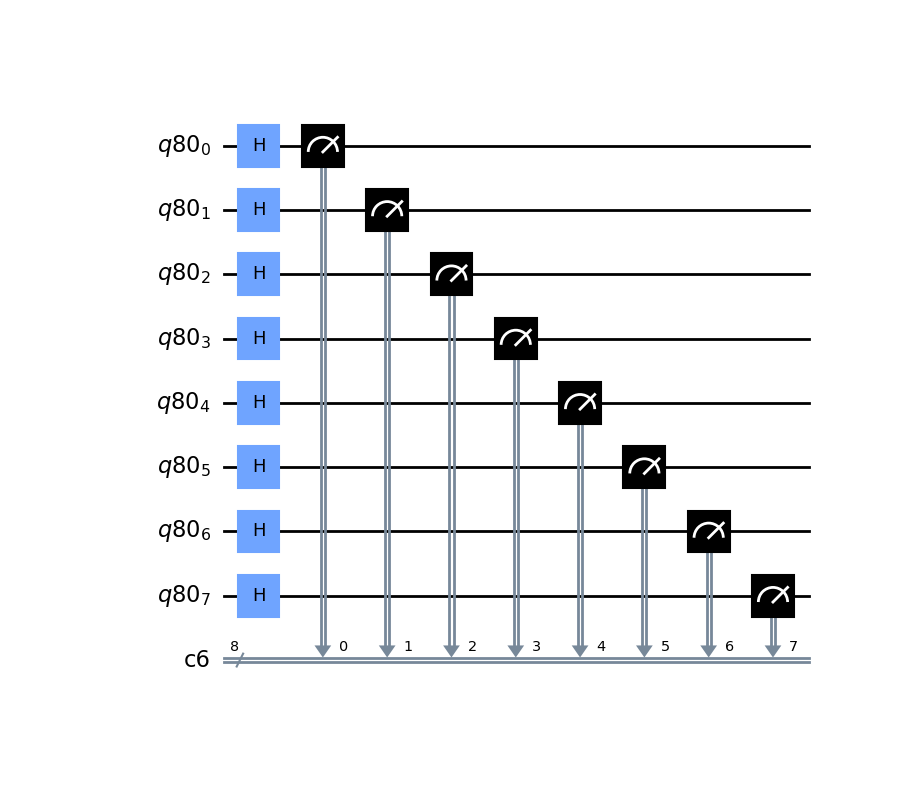
### Result

To implement trb generation via the qulib library, we use the TruelyRandomByte() function. We use a loop to print ten randomly generated bytes (numbers between 0-255) in the given code snippet below.

**from** **qulib** **import** TruelyRandomByte **as** trb

**for** i **in** range(**10**):

**print**(trb.TruelyRandomByte(), end = ' ')



**Fig 7.13: Quantum circuit for Truly Random Byte Generation**

The circuit diagram for TRB generation has eight Hadamard Gates followed by eight Measurement Gates applied on eight qubits, as shown in figure 7.13.

### Quantum Walks

### Concept

Quantum Walks are analogs to classical Markov Random walks but instead use complex amplitude functions to choose the next step rather than probability, replacing a random process with a quantum process in the concept of random walks [4].

### Algorithm

Let us consider a square with vertices (00, 01, 10, 11); we apply quantum walk to this graph via the following algorithm:

1. Initialize the quantum circuit with three qubits.
2. Input the number of steps required to execute the shift operator.
3. Apply the random flip operator to the first qubit of the circuit.
4. Apply the shift operator to the circuit.
5. Measure the final state of the system. (q0 – random flip, q1,q2 – position)

### Code

**from** **qiskit** **import** QuantumCircuit, QuantumRegister, ClassicalRegister

**from** **qiskit** **import** Aer, execute

**def** **QuantumWalk**():

#Number of steps

steps = **4**

#Defining the shift gate

qr = QuantumRegister(**3**)

#Circuit for shift operator

qc = QuantumCircuit (qr, name='shift\_circ')

#Toffoli Gate

qc.ccx (qr[**0**], qr[**1**], qr[**2**])

#CNOT Gate

qc.cx (qr[**0**], qr[**1**] )

qc.x (qr[**0**])

qc.x (qr[**1**])

qc.ccx (qr[**0**], qr[**1**], qr[**2**])

qc.x (qr[**1**])

qc.cx (qr[**0**], qr[**1**])

#Convert the circuit to Custom Gate

s\_gate = qc.to\_instruction()

q = QuantumRegister (**3**, name='q')

c = ClassicalRegister (**3**, name='c')

#Primary Circuit

circuit = QuantumCircuit (q,c)

**for** i **in** range(steps):

#Random Flip

circuit.h (q[**0**])

#Applying Shift

circuit.append (s\_gate, [q[**0**],q[**1**],q[**2**]])

circuit.measure ([q[**0**],q[**1**],q[**2**]], [c[**0**],c[**1**],c[**2**]])

circuit.draw(output='mpl')

backend = Aer.get\_backend('qasm\_simulator')

job = execute(circuit, backend, shots=**1**)

result = job.result()

**print**(result.get\_counts())

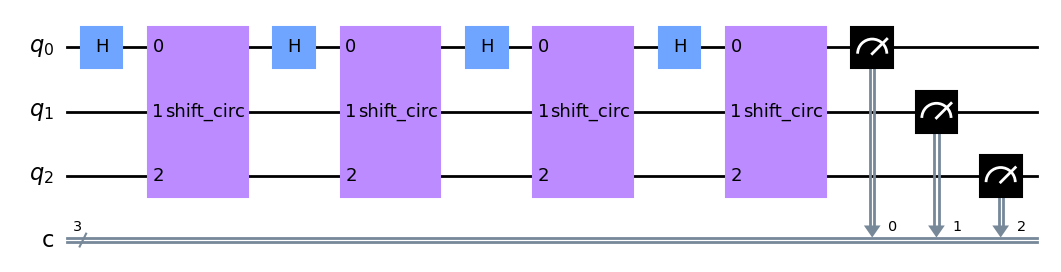
### Result

As shown in the code snippet below, we use the function QuantumWalk() in the qulib library to implement it.

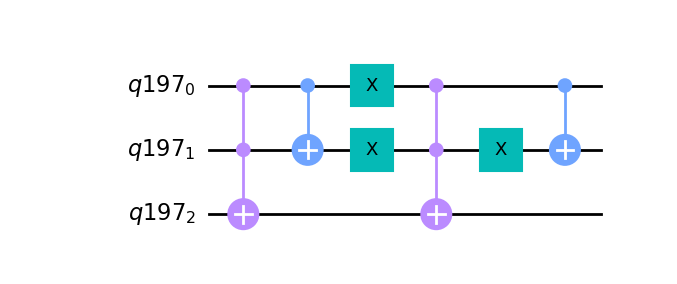
**from** **qulib** **import** QuantumWalk **as** qw

qw.QuantumWalk()

We execute the shift circuit four times as we take four steps using the quantum walk on the graph. The quantum circuit diagram for the shift operator in figure 7.14 and the quantum walk circuit is shown in figure 7.15.



**Fig 7.15: Quantum circuit for four step Quantum Walk**



**Fig 7.14: Quantum circuit for Shift Circuit in Quantum Walk**

The output for the following circuit gives us a random output each time we execute it. Giving us the value of the random flip qubit at q0 and the position using q1q2.

### setup.py

This python file is necessary for the directory of the library qulib to provide all essential information on the name, dependencies, versions, description, author, and license of the library required for installation of it on any required remote system—the contents of this file given the code snippet below.

**from** **setuptools** **import** find\_packages, setup

setup(

name='qulib',

packages=find\_packages(include=['qulib']),

version='0.1.0',

description='quantum library for operators, protocols, algorithms, and applications',

author='Anurag K S V',

license='MIT',

install\_requires=['qiskit','matplotlib'],

)

### CONCLUSION & FUTURE SCOPE

This project aimed to deliver a glimpse into the rapidly evolving field of quantum computing, focusing on multiple paradigms that dictate the direction this field will take over the years to come. We explored the building blocks of quantum computing via the Quantum Operators, moved to communication protocols that define the exchange of information between intra-quantum systems or inter-quantum systems in an experimental network, moved to work with one of the most famous quantum algorithms, Grover's Search which works on the Oracular paradigm of quantum algorithms. Finally, we looked at a few quantum computing applications in the form of the generation of truly random bytes with potential applications in quantum cryptography and simulation of quantum walks.

Due to very high noise rates, the current NISQ era of quantum computing restricts our simulation capabilities on real quantum systems. Despite this, low-depth quantum circuits produce consistently favourable results on real quantum computers even in today's era due to rapid improvement in Quantum Error Correction, which is still one of the most important and heavily researched topics in this domain. The current challenges compel us to work on engineering challenges to develop error-free resilient quantum computers with more than 106 logical, coherent qubits for computation and improved quantum error correction to facilitate the logical side of quantum computation. Upon moving past the obstacles present in the Quantum Computing industry, we can make the above-discussed quantum computing paradigms available for general public usage via looking at quantum-internet, quantum cryptography, and quantum information storage, all of which provide a significant increase in speed and conservation of space via large scale implementation of standard quantum paradigms.

### REFERENCES

### Journals/Articles

1. Preskill J. Quantum Computing in the NISQ era and beyond. Quantum. 2018 Aug 6; 2:79.
2. DiVincenzo D. Quantum computers: the first gate opens. Physics World. 1996 Mar;9(3):27–7.
3. Ladd TD, Jelezko F, Laflamme R, Nakamura Y, Monroe C, O’Brien JL. Quantum computers. Nature [Internet]. 2010 Mar [cited 2019 Nov 20];464(7285):45–53. Available from: https://www.nature.com/articles/nature08812.pdf
4. J. A, Adedoyin A, Ambrosiano J, Anisimov P, Casper W, Chennupati G, et al. Quantum Algorithm Implementations for Beginners. ACM Transactions on Quantum Computing. 2022 Mar 28;
5. Beckers, Arnout & Tajalli, Armin & Sallese, Jean-Michel. A Review on Quantum Computing: Qubits, Cryogenic Electronics and Cryogenic MOSFET Physics. Research Gate; 2019.
6. Bennett CH, Wiesner SJ. Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states. Physical Review Letters. 1992 Nov 16;69(20):2881–4.
7. Wang C, Deng F-G, Li Y-S, Liu X-S, Long GL. Quantum secure direct communication with high-dimension quantum superdense coding. Physical Review A. 2005 Apr 28;71(4).
8. Mattle K, Weinfurter H, Kwiat PG, Zeilinger A. Dense Coding in Experimental Quantum Communication. Physical Review Letters. 1996 Jun 17;76(25):4656–9.

### Books

1. Hidary JD. QUANTUM COMPUTING: an applied approach. S.L.: Springer Nature; 2019.

### E-websites / downloads

1. Fisher C. IBM | What is Quantum Computing? [Internet]. www.ibm.com. IBM; 2019. Available from: <https://www.ibm.com/quantum-computing/what-is-quantum-computing/>

1. Qiskit [Internet]. qiskit.org. Available from: <https://qiskit.org/>
2. Holth D. wheel: A built-package format for Python [Internet]. PyPI. [cited 2022 Apr 10]. Available from: <https://pypi.org/project/wheel/>
3. pytest: helps you write better programs — pytest documentation [Internet]. docs.pytest.org. Available from: <https://docs.pytest.org/en/7.1.x/>
4. Superdense Coding [Internet]. learn.qiskit.org. [cited 2022 Apr 10]. Available from: <https://learn.qiskit.org/course/ch-algorithms/superdense-coding>
5. Grover’s Search Algorithm and Amplitude Amplification — Grove 1.7.0 documentation [Internet]. grove-docs.readthedocs.io. [cited 2022 Apr 10]. Available from: <https://grove-docs.readthedocs.io/en/latest/grover.html>
6. Grover’s Algorithm [Internet]. learn.qiskit.org. [cited 2022 Apr 10]. Available from: <https://learn.qiskit.org/course/ch-algorithms/grovers-algorithm>