Doctoral Dissertation Academic Year 2024

A Suite For Testing and Debugging Quantum Programs

Keio University
Graduate School of Media and Governance

Sara Ayman Fathi Mohamed Metwalli

Abstract

The forty-year history of quantum computers has taken us through initial curiosity, naive optimism, then dismay at the scale of proposed error-corrected systems, and into today's excitement over the availability of real, but still small and error-prone, systems [1, 2, 3]. Algorithms have followed a similar roller coaster, arriving at the point where a demonstration of the implementation of algorithms originally defined as abstract equations in theory papers are now commonly represented as circuit diagrams [4]. The challenge now is, for both the hardware and software is scalability, how can we have more qubits and build larger and more sophisticated programs.

This thesis introduces a collection of software functions (Cirquo), a packaged testing suite aiming to address the complex challenges of quantum circuit development as quantum programs scale up to the complexities of classical software. As the field of quantum software engineering evolves, the need for sophisticated tools that can handle the nuances of quantum computing becomes imperative. Cirquo stands out by offering tailored debugging strategies and comprehensive testing capabilities specifically designed to manage the unique aspects of quantum computation.

In this thesis we discuss the three main types of quantum circuit blocks: Amplitude Permutation, Phase Modulation, and Amplitude Redistribution. Each type presents specific challenges that we proposed different approaches to address. For Amplitude Permutation Circuits, we provide techniques to correct amplitude permutations, effectively mimicking classical operations and ensuring circuit accuracy. In the realm of Phase Modulation Circuits, the suite offers precise calibration tools for phase alterations, which are critical for the successful execution of quantum algorithms. The most complex Amplitude Redistribution Circuits benefit from advanced methods that adjust probability amplitudes to maintain the integrity of quantum states.

The suite enhances quantum circuit debugging by allowing developers to divide circuits into manageable slices, either vertically or horizontally, categorize these slices,

and perform targeted tests. This slicing mechanism is vital for isolating problematic sections and applying focused debugging strategies without disrupting the circuit's overall functionality. Additionally, Cirquo facilitates gate tracking within each circuit slice, providing developers with detailed insights into gate behavior and interactions.

One of the main features of Cirquo is its automated test generation, which supports six common subroutines used in quantum algorithms. This feature streamlines the testing process and enhances the efficiency of detecting and resolving bugs. Moreover, the suite's capability to simulate or execute tests on current quantum devices ensures that developers can optimize circuits for real-world applications.

This thesis also offers strategic advice on creating efficient test cases for different circuit categories. These test cases are instrumental in pinpointing the location of bugs, thus significantly reducing debugging time and improving circuit reliability.

In conclusion, this research significantly advances the field of quantum computing by providing a robust framework for debugging quantum circuits. The introduction of Cirquo is a pivotal step towards developing reliable quantum computing systems, which are expected to have profound implications across various domains. The empirical evidence confirms the effectiveness of Cirquo in optimizing quantum circuit performance, marking a significant milestone in the maturation of quantum software engineering.

Keywords: Quantum computation, testing, debugging, software, bug analysis.

Thesis Committee:

Supervisor:

Prof. Rodney Van Meter, Keio University

Co-Advisers:

Prof. Hiroyuki Kusumoto, Keio University
Assoc. Prof. Takahiko Satoh, Keio University
Project Assoc. Prof. Michal Hajdušek, Keio University
Assoc. Prof. Hideyuki Kawashima, Keio University

Acknowledgment

I stand at the closing of a profound journey that has been as enriching as it has been challenging. This journey would not have been possible without the support, guidance, and encouragement of several remarkable individuals and groups whose assistance has been invaluable. I would not be able to complete this thesis without them, and because of that, I would love to extend my deepest gratitude to each one of them.

First and foremost, I express my heartfelt appreciation to my supervisor, Professor Rodney Van Meter, whose expertise, understanding, and patience considerably enhanced my experience. Your guidance was much more than academic supervision; it was a beacon of wisdom in moments of doubt and confusion. Thank you for believing in my capabilities and pushing me to excel. Your mentorship has shaped this research and prepared me for the journey ahead in the scientific community.

I am incredibly fortunate to friends who have been more like family during long this journey. To Ruta, Kevin, Victor, Robin, Akina, and Rachel - your role as my support system, your unwavering support, endless humor, and timely distractions made this taxing journey seem lighter. Whether it was thesis writing sessions, brainstorming ideas, or just sharing a meal after a long day, your presence made all the difference in my professional and personal life. I cherish our bonds deeply and am grateful for your kindness and camaraderie. To Aaron, thank you for being my comfort person on many occasions.

To my family, your encouragement has been invaluable. Your love and belief in my dreams have driven my every endeavor. My mother, father, Saly, and Sif, thank you for your endless patience, for listening to my endless talks about my research, and for providing a haven of peace and encouragement. Your support has been the foundation upon which I built my aspirations.

All AQUA members, students, and faculty deserve special mention for creating an environment of collaboration and learning. Working alongside such talented and dedicated individuals has been a privilege. Our discussions, related to our research or the much-needed breaks we took together, have been a source of joy and learning. Thank you for the shared experiences, the insightful feedback on my work, and the constant encouragement.

In conclusion, this thesis is a testament to my efforts and the collective support and guidance of everyone mentioned above. The journey of a PhD is unique and challenging, yet it was made enjoyable and memorable through your support. I look forward to carrying forward the lessons learned and the friendships made as I step into the next phase of my career. To all of you, I extend my deepest gratitude and appreciation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Technology, in general, has become an essential part of today's world. It plays a vital role in our daily lives, from medicine to business and education. Every day, new technologies are created and researched. The more we depend on technology to ease our lives, the more data technology must process, which will require more computation power.

Technology has grown exponentially over the past decade, as has the size of data and the complexity of problems it needs to solve. However, the more we advance technology, the closer we approach the limits of Moore's law [5]. This growth led many scientists and industries to invest in quantum computing to answer the question, "What must we do once we reach the limit of Moore's Law?" Quantum computing does not only answer that question; quantum computers will be able to solve problems our computers today would take a very long time to solve [6]. Quantum computing, however, will not displace existing classical technology.

One example of a challenging problem for classical computers that quantum computers can solve is factoring of large numbers. Factoring large numbers is a fundamental problem in cryptography, and traditional encryption methods rely on the difficulty of factoring large numbers to provide security. For example, the RSA algorithm [7] is built on the difficulty of factoring large numbers. Classical computers use superpolynomial running time algorithms for factoring large numbers [8, 9, 10]. This means that as the size of the number to be factored increases, the time required to factor it grows almost exponentially. As a result, even relatively small numbers can take traditional computers a very long time to factor.

In contrast, quantum computers can use Shor's algorithm [11], which factors a large number N in $O(\log^3 N)$ time, unlike classical computers [12, 13]. Furthermore, Shor's algorithm relies on the principles of quantum mechanics to factor large numbers efficiently [14].

This poses a significant problem for classical computers because many encryption

methods commonly used today could be vulnerable to attack by quantum computers in the future. For example, a classical computer would require billions of years to factor a 2048-bit RSA key, which is the size commonly used for encryption today. However, using Shor's algorithm on a quantum computer could factor the same key in hours or days, depending on the number of qubits, the accuracy of the hardware, the clock speed, and the modular exponential algorithm.

1.1 The Need for Quantum Computing

Quantum computing is a rapidly advancing field that has the potential to revolutionize the way we process information. Classical computers rely on manipulating bits, which can be either a 0 or a 1. However, quantum computing is based on manipulating quantum bits or qubits, which can be 0 or 1 or a superposition of 0 and 1. This unique property allows quantum computers to process information differently and solve complex problems difficult for traditional computers. Because of the use of superposition, quantum computers solve problems differently than classical computers do.

One of the most significant advantages of quantum computing is its ability to process data much faster than traditional computers through quantum algorithms [15]. As a result, quantum computing is significant in fields such as finance [16, 17], logistics [18], and data analysis [19, 20]. We can also utilize quantum computing in optimizing supply chain management [21] or enable companies to manage their inventory and reduce waste more efficiently. Moreover, cryptography [22, 23, 24] is another area where quantum computing has the potential to make a significant impact.

Quantum computing also has the potential to revolutionize scientific research by enabling more complex simulations and modeling, which could lead to significant breakthroughs in fields such as chemistry [25, 26], physics [27, 28, 29], and materials science [30]. For example, quantum computing could simulate the behavior of molecules and materials [31], enabling scientists to develop new drugs [32], catalysts, and materials with high accuracy and speed [33].

Additionally, quantum computing could significantly affect artificial intelligence (AI). Traditional AI algorithms rely on large datasets and require significant computational resources to train, often necessitating extensive time and energy to achieve desired accuracies and efficiencies [34, 35]. The inherent properties of quantum systems, like entanglement and superposition, could enable the development of new types of AI algorithms that are not feasible with classical computers. These quantum algorithms could potentially create more nuanced and complex models that could predict outcomes more accurately and adapt to new data dynamically.

Many quantum algorithms are designed for different problems; in addition to the above, the Quantum Algorithm Zoo cites more than 300 papers on those algorithms when writing this dissertation [36].

1.1.1 Technical Advantage: Beyond Classical Computing

The most striking technical advantage of quantum computing lies in its potential to solve specific problems at speeds surpassing current classical computers, dramatically altering the computational complexity landscape for those problems.

For instance, integer factorization is crucial for encryption. Classical algorithms for this task, like the general number field sieve [37, 38, 39], have complexities that are superpolynomial, approximately $O\left(\exp\left(\left(\frac{64}{9}\right)^{\frac{1}{3}}(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}\right)\right)$, where N is the number to be factored and $n = \lceil \log_2(N) \rceil$ is the bit length. In contrast, Shor's algorithm [40] can factor integers in polynomial time, roughly $O(n^3)$, representing a speedup for large N.

Similarly, for searching an unsorted database, Grover's algorithm [41] showcases quantum advantage by reducing the complexity from $O(N)$ in classical settings [42, 43] to $O(\sqrt{N})$ in quantum computing, offering a quadratic speedup.

Quantum computing offers significant advantages over classical computing in terms of time complexity, particularly for specific computational tasks. One example is the use of quantum algorithms like the Quantum Fourier Transform (QFT), which calculates the Fourier transform of a vector size N with time complexity $O(n^2)$, where $n = \log_2(N)$, compared to the classical complexity of $O(N \log N)$ [44]. Furthermore, quantum computational models can achieve tasks such as Gaussian boson sampling in microseconds, which would otherwise take classical supercomputers thousands of years by offering *asymptotic* speedups [45].

Asymptotic Speedups

Asymptotic speedup refers to the comparison of the growth rates of the time complexity of algorithms as the size of the input n becomes very large. In the context of quantum computing, asymptotic speedup describes how quantum algorithms can solve specific problems significantly faster than classical algorithms as the problem size increases.

Classical Algorithms: The time complexity of an algorithm describes the number of basic operations (or steps) that an algorithm takes as a function of the input size n. Common complexities include polynomial time ($O(n^k)$), exponential time ($O(2^n)$), and others.

Quantum Algorithms: Similarly, quantum algorithms have their time complexity, often expressed in the number of quantum operations (gates) required.

When a quantum algorithm reduces the time complexity from $O(n^k)$ to $O(n^{k-1})$ or $O(\sqrt{n})$, it is called a polynomial speedup. For instance, Grover's algorithm provides

a polynomial speedup for searching unsorted databases, reducing the time complexity from $O(n)$ to $O(\sqrt{n})$.

If a quantum algorithm reduces the time complexity from $O(2^n)$ to $O(n^k)$, it is called an exponential speedup. Shor's algorithm for integer factorization is a prime example, reducing the time complexity from exponential ($O(2^n)$) in classical algorithms to polynomial ($O(n^3)$) in a quantum algorithm.

For example, classical algorithms for factoring large integers, such as the general number field sieve, have a sub-exponential time complexity of $O(\exp((\log n)^{1/3}(\log\log n)^{2/3}))$ as discussed earlier. Shor's algorithm can factor integers in polynomial time, specifically $O(n^3)$, providing an exponential speedup. Another example is searching an unsorted database. A classical algorithm requires $O(n)$ steps. Grover's algorithm reduces this to $O(\sqrt{n})$, providing a quadratic speedup.

Asymptotic speedup is particularly important when considering scalability. While a quantum algorithm might not be faster for small input sizes due to overhead in quantum operations, it can become significantly faster as the input size grows. Moreover, asymptotic speedup is not universal; it applies to specific problems where quantum algorithms have a proven advantage. Many problems still lack known quantum algorithms that outperform classical ones.

Asymptotic speedup highlights quantum algorithms' potential to solve certain problems more efficiently than classical algorithms as the problem size increases. It underscores the fundamental differences in computational power between classical and quantum computing, especially for problems with high computational complexity.

1.1.2 Societal Impacts

Quantum computing presents unique challenges and opportunities in cybersecurity, necessitating the development of robust, quantum-resistant cryptographic methods to protect sensitive data against potential quantum attacks. As quantum computers become capable of breaking traditional encryption systems such as RSA and ECC, which rely on the difficulty of factoring large numbers and elliptic curve discrete logarithms, respectively, the field of post-quantum cryptography (PQC) has gained significant attention. PQC aims to develop secure cryptographic systems against both classical and quantum computers, ensuring long-term data protection. Integrating these systems into current digital infrastructures is critical, requiring updates to protocols and software to support new cryptographic standards [46].

Access to quantum computing also raises essential considerations. The high cost and complexity of quantum hardware limit access primarily to well-funded corporations and research institutions, potentially creating a "quantum divide" where unequal access to this powerful technology could lead to imbalances in economic and technological advancements. Ensuring equitable access involves lowering entry barriers, developing more accessible quantum computing platforms, and addressing regulatory and policy

challenges to prevent monopolistic practices and encourage a competitive quantum industry [1].

Ethical considerations are equally paramount. The deployment of quantum computing in finance, military, and healthcare sectors must be governed by clear ethical guidelines to prevent misuse and ensure that quantum advancements are aligned with societal values and benefits. Privacy, consent, and transparency are crucial, especially as quantum computing may enable new surveillance or data analysis forms that could infringe on personal liberties. Developing a framework for ethical quantum computing involves stakeholders from various sectors, including ethics, law, technology, and public policy, to explore the implications of quantum technologies and establish guidelines that promote responsible use [47].

The implications of quantum computing extend beyond technical challenges to encompass broad societal impacts, making it essential to consider these aspects in tandem as the technology develops.

1.2 Problems With Quantum Computers

Though many useful quantum algorithms have been developed, we have yet to construct a quantum computer that can practically solve large problems. For example, no quantum bit (qubit) implementation can sustain a state with sufficient fidelity from the start of some computations to the end. 23 years ago, DiVincenzo summarized the physical conditions necessary for building a practical quantum system in [48],

1. A scalable physical system with reliable qubits.
2. The ability to initialize the state of the qubits to a simple state, such as "the zero states" $|000...\rangle$.
3. Decoherence times that are much longer than the gate operation time.
4. A hardware-independent set of quantum gates (a universal set).
5. A qubit-specific measurement capabilities.

These conditions provide a clear direction towards building a fault-tolerant quantum computer. We still can not build one for different reasons [1].

Some of the problems quantum computers face today are:

Decoherence and High Error Rates

One of the main challenges in quantum computing is decoherence [49]. When qubits interact with their environment, decoherence leads to a loss of information, increasing

the error rates in the computations. Therefore, maintaining coherence and minimizing errors is crucial for building reliable quantum computers [50, 51, 52]. Quantum error correction codes have been developed to counteract the effects of decoherence and other errors [53, 54, 55, 56]. However, these methods require a significant overhead regarding the number of qubits and computational resources.

Scalability

Scalability is another primary concern in quantum computing. While small-scale quantum computers with a few dozen qubits have been built and can be accurately simulated, scaling up to thousands or millions of qubits is required to solve practical problems [57, 58, 59, 60]. One of the main reasons is the increase in the system's complexity as more qubits are added. Aside from the number of qubits, the increasing number of connections and interactions between qubits adds to the difficulty in maintaining their coherence and controlling their states precisely. However, the biggest problem we face when scaling quantum computers is the exponential decline in the probability of successful execution of an algorithm when gates are imperfect.

Algorithm Development and Implementation

The development of quantum algorithms is another area where progress is needed, mainly on a software and practical level.

While some groundbreaking algorithms, such as Shor's and Grover's algorithms, have implementations for small problems on today's quantum computers, many quantum algorithms are still very theoretical and must be implemented practically.

Additionally, discovering new quantum algorithms that can offer significant advantages over classical algorithms for various problems remains an ongoing research goal. Future research will likely focus on algorithm optimization for NISQ devices, error correction to enhance algorithm stability, and the exploration of hybrid algorithms that combine classical and quantum computing elements to perform tasks more efficiently than purely classical or quantum approaches could alone. This holistic approach to algorithm development is crucial for advancing practical quantum computing applications.

Integration with Classical Systems

Quantum computers are not meant to replace classical computers entirely; they are expected to complement them by solving problems currently intractable for classical systems. As a result, seamless integration between quantum and classical computing systems is crucial for practically implementing quantum computing [61, 62]. Therefore, developing efficient methods for exchanging information and coordinating computations between quantum and classical systems is significant for advancing quantum systems.

Testing and Debugging Quantum Software

Finally, we discuss the reasons for and motivation behind this thesis. Testing and debugging quantum programs presents a unique and formidable challenge in the field of quantum computing, mainly due to the intrinsic properties of quantum mechanics, such as superposition and entanglement [63, 64, 65]. Unlike classical computing, where the system's state can be observed directly at any point without altering it, measuring a quantum state inherently changes it, making traditional debugging techniques ineffective [66, 67]. Moreover, the probabilistic nature of quantum computation means that outcomes can only be predicted in terms of probabilities rather than certainties, complicating the verification of program correctness [68, 69, 70, 71]. As quantum algorithms increase in complexity, ensuring their accuracy and reliability demands innovative testing and debugging methodologies that can accommodate the non-deterministic behavior of quantum systems. This need represents a significant hurdle in developing and deploying quantum technologies, necessitating new tools and approaches that can handle the nuanced complexities of quantum programming.

1.3 Contribution of This Dissertation

As quantum computing progresses toward more extensive, scalable systems, significant gaps remain in the toolchain for quantum software development, particularly in-circuit testing and debugging. Practical quantum circuit debugging and testing tools must allow isolation and in-depth examination of sub-circuits to understand their functionality without incurring exponential costs in computational resources. Such tools are critical as they enable developers to interface parts of the circuit, prepare and execute tests, and analyze outcomes efficiently. This necessity becomes more pronounced as quantum circuits grow in size and complexity, making simulation on classical computers impractical.

The capacity to work with sub-circuits without simulating every possible quantum state—each with potentially exponential terms due to quantum superposition—is essential. As quantum circuit design advances and developers transition from simulating small to medium circuits to tackling larger quantum applications, the limitations of existing simulators become apparent. This underscores the importance of developing robust, scalable testing and debugging tools that can handle the increasing complexity without relying on classical simulation methods.

The dissertation addresses these challenges by introducing practical tools to enhance quantum circuit comprehension, testing, and debugging. These tools are designed to be hardware-independent and provide a foundational step towards more sophisticated quantum software development capabilities. Contributions include a circuit slicer for managing large circuits, a categorizer to assess circuit parts' impacts, and various testing

strategies tailored to different quantum circuits. This approach not only aids developers in understanding and debugging their circuits but also facilitates the broader adoption and optimization of quantum computing technologies.

Currently, most circuits being developed are small to medium-medium-sized; the developers can simulate them after performing some optimization techniques. However, as we move on to more extensive applications, simulators will fail to keep up with the increased size of circuits. Making tools such as the ones presented in this work is essential for the progress of quantum software.

The primary contribution of this dissertation is a practical quantum software tool that allows developers and quantum enthusiasts to better understand their circuits. When they know their circuits better, they can debug them whenever a problem occurs. This work focuses on the software shortage in quantum computing today and aims to be the first step towards a hardware-independent quantum circuit testing and debugging tool.

The structure of this dissertation is as follows:

Chapter 2: This chapter provides an overview of the fundamentals of quantum computation, the current state of quantum hardware and software, and the challenges quantum software faces, particularly in debugging.

Chapter 3: This chapter details the process of implementing a quantum algorithm, from theoretical foundations to code development and hardware analysis. The implementation of the clique-finding problem using Grover's algorithm is used as a case study to illustrate this process step-by-step.

Chapter 4: This chapter first shows the common bugs in quantum programs and then presents the proposed suite, explaining its core components, including circuit categorization, testing methodologies for various types of quantum circuits, and debugging strategies.

Chapter 5: This chapter demonstrates the application of the proposed suite and debugging strategies through various examples.

Chapter 6: This chapter reviews the existing literature and related work on quantum debugging, testing, and software analysis. It also discusses the results and current limitations of debugging quantum computers.

Chapter 7: This chapter concludes the dissertation, summarizing the proposed work and outlining potential directions for future research.

Chapter 2

Quantum Information Science

This chapter will discuss the basis of quantum information, its characteristics, and its operations. Then, we will address the information the reader needs to know to understand the rest of the thesis, including quantum superposition, interference, entanglement, and gates.

Before discussing the details of a quantum computer, we need to answer one crucial question: "What is a quantum computer?" A quantum computer is a computing device that harnesses the principles of quantum mechanics to perform calculations and solve problems. Unlike classical computers, which use bits to represent data in binary form (0 or 1), quantum computers use quantum bits or qubits. Due to quantum mechanics, qubits possess unique properties, such as superposition and entanglement, which I will address shortly. Superposition allows a qubit to exist in a combination of 0 and 1 states simultaneously, while entanglement creates strong correlations between qubits that classical physics cannot describe. These properties enable quantum computers to perform specific calculations much more efficiently than classical computers. In addition, by manipulating qubits in superposition and leveraging entanglement, quantum computers can process vast amounts of information in parallel and solve problems considered intractable for classical computers.

To understand the basics of quantum computing, we first need to cover the behind-the-scenes mathematics.

2.1 The Math Used in This Thesis

Quantum information science represents an intersection of mathematics, physics, and computer science, built on the foundation of quantum mechanics principles. The mathematical framework of quantum information is primarily centered around linear algebra,

which is fundamental to describing the states and transformations of quantum systems.

2.1.1 Complex Numbers

Complex numbers are integral to the framework of quantum mechanics. Unlike real numbers, which can be visualized as points on a straight line, complex numbers require a two-dimensional plane for their representation. A complex number is typically expressed in the form $z = a + bi$, where a and b are real numbers, and i is the imaginary number $\sqrt{(-1)}$, defined by the property $i^2 = -1$. This seemingly simple extension has profound implications in various fields of mathematics and physics.

The fundamental component a of the complex number z represents the real number, whereas the imaginary component b introduces a new dimension, allowing for exploring phenomena that cannot be described by real numbers alone. The introduction of the imaginary unit i was initially met with skepticism, as it challenged the existing norms of mathematics; however, it has since become indispensable in solving equations with no real solutions, such as $x^2 + 1 = 0$.

Complex numbers have several features:

1. **The complex plane representation of numbers:** Each complex number is composed of a real part and an imaginary part, where the horizontal axis is for the real part, and the vertical axis is for the imaginary part.

2. **Arithmetic operations:**

   - **Addition and Subtraction:** For $z_1 = a + bi$ and $z_2 = c + di$, then:

   $$z_1 + z_2 = (a + c) + (b + d)i. \tag{2.1}$$

   - **Multiplication:** Using the distributive property and $i^2 = -1$:

   $$z_1 \times z_2 = (ac - bd) + (ad + bc)i. \tag{2.2}$$

   - **Division:** To divide by a complex number, multiply by the conjugate of the denominator:

   $$\frac{z_1}{z_2} = \frac{(ac + bd) + (bc - ad)i}{c^2 + d^2}. \tag{2.3}$$

3. **Conjugates:** The complex conjugate of $z = a + bi$ is $z^* = a - bi$.

4. **Magnitude and Argument:**

- The magnitude (or modulus) of $z = a + bi$:

$$|z| = \sqrt{a^2 + b^2}. \tag{2.4}$$

- The argument (or phase) of z, denoted θ, where z can be expressed in polar coordinates as:

$$z = |z|e^{i\theta}. \tag{2.5}$$

5. **Euler's Formula:** This formula connects exponential and trigonometric functions:

$$e^{i\theta} = \cos(\theta) + i\sin(\theta). \tag{2.6}$$

For example, the complex number $Z = e^{\frac{\pi i}{4}}$ and its conjugate Z^* can be drawn as in Figure 2.1.

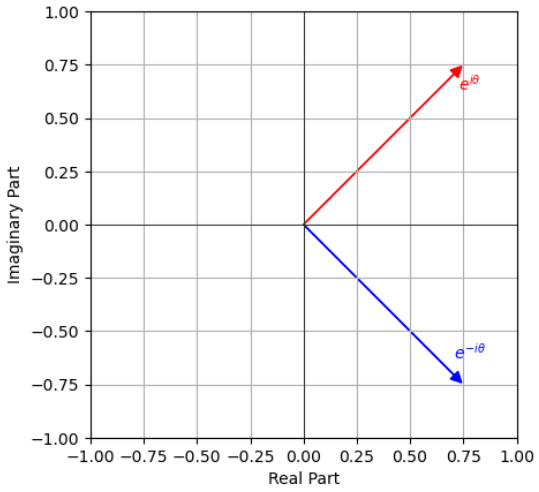


Figure 2.1: Complex Plane Representation of $e^{i\theta}$ and $e^{-i\theta}$.

2.1.2 Vectors and Matrices

Vectors and matrices are fundamental concepts in mathematics and are essential tools in various fields such as physics, engineering, computer science, and economics. They provide a systematic way of organizing data and performing calculations on sets of numbers, enabling the efficient representation and manipulation of linear equations.

A vector is a mathematical object with both a magnitude and a direction. Vectors can be considered arrows pointing from one point to another in space. In mathematics, vectors are often represented as ordered lists of numbers; these numbers indicate the vector's components along various axes. For example, a three-dimensional vector $\mathbf{v}$ might be written as $\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$, where v_1, v_2, and v_3 are scalar quantities representing the vector's magnitude along the x, y, and z axes, respectively.

Operations and Properties of Vectors

- **Vector Addition:** The addition of two vectors $\mathbf{a}$ and $\mathbf{b}$ results in a vector $\mathbf{c}$ that combines the magnitude and direction of both.

$$\mathbf{c} = \mathbf{a} + \mathbf{b} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{bmatrix}. \tag{2.7}$$

- **Scalar Multiplication:** Multiplying a vector by a scalar k scales the magnitude of the vector without altering its direction.

$$k\mathbf{a} = \begin{bmatrix} ka_1 \\ ka_2 \\ ka_3 \end{bmatrix}. \tag{2.8}$$

- **Inner Product (Dot Product):** The inner product of two vectors is a scalar value that quantifies the extent to which two vectors point in the same direction. For two vectors $\mathbf{u}$ and $\mathbf{v}$ in $\mathbb{R}^n$, where

$$\mathbf{u} = [u_1, u_2, \ldots, u_n] \quad \text{and} \quad \mathbf{v} = [v_1, v_2, \ldots, v_n], \tag{2.9}$$

  the inner product (dot product) is defined as:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^{n} u_i v_i. \tag{2.10}$$

- **Outer Product:** For two vectors $\mathbf{u}$ and $\mathbf{v}$ in $\mathbb{R}^n$ and $\mathbb{R}^m$ respectively, where

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix}, \quad \mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix}, \tag{2.11}$$

  the outer product $\mathbf{u} \otimes \mathbf{v}$ is defined as the $n \times m$ matrix:

$$\mathbf{u} \otimes \mathbf{v} = \begin{bmatrix} u_1v_1 & u_1v_2 & \cdots & u_1v_m \\ u_2v_1 & u_2v_2 & \cdots & u_2v_m \\ \vdots & \vdots & \ddots & \vdots \\ u_nv_1 & u_nv_2 & \cdots & u_nv_m \end{bmatrix}. \tag{2.12}$$

- **Tensor Product:** The tensor product denoted $\mathbf{a} \otimes \mathbf{b}$, results in a higher-dimensional tensor. For vectors $\mathbf{a} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$ and $\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$, their tensor product is:

$$\mathbf{a} \otimes \mathbf{b} = \begin{bmatrix} a_1b_1 \\ a_1b_2 \\ a_2b_1 \\ a_2b_2 \end{bmatrix}. \tag{2.13}$$

- **Vector Norm:** The norm (or magnitude) of a vector $\mathbf{a}$ is given by:

$$\|\mathbf{a}\| = \sqrt{a_1^2 + a_2^2 + a_3^2}. \tag{2.14}$$

- **Unit Vector:** A unit vector $\mathbf{u}$ in the direction of $\mathbf{a}$ is obtained by dividing $\mathbf{a}$ by its norm.

$$\mathbf{u} = \frac{1}{\|\mathbf{a}\|}\mathbf{a} = \begin{bmatrix} \frac{a_1}{\sqrt{a_1^2+a_2^2+a_3^2}} \\ \frac{a_2}{\sqrt{a_1^2+a_2^2+a_3^2}} \\ \frac{a_3}{\sqrt{a_1^2+a_2^2+a_3^2}} \end{bmatrix}. \tag{2.15}$$

- **Projection:** The projection of vector $\mathbf{a}$ onto vector $\mathbf{b}$ is a vector $\mathbf{p}$ that lies along

b and represents the component of **a** in the direction of **b**.

$$\mathbf{p} = \frac{\mathbf{a} \cdot \mathbf{b}}{\mathbf{b} \cdot \mathbf{b}} \mathbf{b}. \tag{2.16}$$

A matrix is a rectangular array of numbers arranged in rows and columns. It is a highly organized grid representing a system of linear equations, transformations in space, or any data that can be structured into rows and columns. For example, a matrix A with two rows and three columns is represented as:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}, \tag{2.17}$$

where a_{ij} denotes the element in the i-th row and j-th column.

Operations and Properties of Matrices

- **Matrix Addition:** Matrix addition is performed element-wise. Two matrices of the same order can be added by adding their corresponding elements:

$$A + B = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix}. \tag{2.18}$$

- **Scalar Multiplication of Matrices:** Multiplying a matrix by a scalar involves multiplying each element of the matrix by the scalar:

$$cA = c \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} ca_{11} & ca_{12} \\ ca_{21} & ca_{22} \end{bmatrix}. \tag{2.19}$$

- **Multiplication of Matrices:** Matrix multiplication is more complex and is defined when the number of columns in the first matrix matches the number of rows in the second matrix. The element at the i-th row and j-th column of the product is obtained by taking the dot product of the i-th row of the first matrix and the j-th

column of the second matrix.

$$AB = \begin{bmatrix} \sum_{k=1}^{n} a_{1k}b_{k1} & \sum_{k=1}^{n} a_{1k}b_{k2} & \cdots & \sum_{k=1}^{n} a_{1k}b_{kp} \\ \sum_{k=1}^{n} a_{2k}b_{k1} & \sum_{k=1}^{n} a_{2k}b_{k2} & \cdots & \sum_{k=1}^{n} a_{2k}b_{kp} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{k=1}^{n} a_{mk}b_{k1} & \sum_{k=1}^{n} a_{mk}b_{k2} & \cdots & \sum_{k=1}^{n} a_{mk}b_{kp} \end{bmatrix}. \tag{2.20}$$

Matrix addition is commutative and associative. Matrix multiplication is associative and distributive over addition but generally not commutative.

- **Transpose of a Matrix:** The transpose of a matrix is obtained by swapping its rows with its columns.

$$A^T = \begin{bmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{bmatrix}. \tag{2.21}$$

- **Trace of a Matrix:** The trace of a matrix is the sum of the elements on the main diagonal. It is defined only for square matrices.

$$\mathrm{Tr}(A) = a_{11} + a_{22} + \ldots + a_{nn}. \tag{2.22}$$

- **Types of Matrices:** There are various types of matrices, such as square matrices, diagonal matrices, identity matrices, symmetric matrices, and orthogonal matrices.
- **Determinant of a Matrix:** The determinant is a scalar value that can be computed from the elements of a square matrix and encodes certain matrix properties.

$$\det(A) = a_{11}a_{22} - a_{12}a_{21}. \tag{2.23}$$

- **Inverse of a Matrix:** The inverse of a matrix A is another matrix, denoted A^{-1}, such that $AA^{-1} = I$ where I is the identity matrix.
- **Eigenvalues and Eigenvectors of a Matrix:** An eigenvector of a matrix A is a non-zero vector v such that $Av = \lambda v$ where λ is a scalar known as an eigenvalue.

2.1.3 Probability Theory

Probability theory is a branch of mathematics concerned with analyzing random phenomena. The fundamental object of study in probability theory is the probability measure, which assigns a numerical probability to events within a certain framework. Here are some key concepts and operations in probability theory:

- **Sample Space and Events:** The sample space is the set of all possible outcomes of a random experiment, and an event is a subset of the sample space. For example, in a dice roll, the sample space is $\{1, 2, 3, 4, 5, 6\}$, and an event could be rolling an even number, $\{2, 4, 6\}$.

- **Probability of an Event:** The probability of an event is a measure of the likelihood that the event will occur, denoted as $P(E)$. It is a number between 0 and 1, where 0 indicates impossibility and 1 indicates certainty.

- **Conditional Probability:** The probability of an event given that another event has occurred is called conditional probability and is denoted by:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}, \tag{2.24}$$

  where $P(A \cap B)$ is the probability that both events A and B occur simultaneously and $P(B)$ is not zero.

- **Independence:** Two events are independent if one occurrence does not affect the occurrence of the other. Mathematically, events A and B are independent if and only if:

$$P(A \cap B) = P(A)P(B). \tag{2.25}$$

- **Random Variables:** A random variable is a function that assigns a real number to each outcome in the sample space. For example, a random variable could assign the number rolled to each outcome when rolling a dice.

- **Probability Distributions:** The probability distribution of a random variable describes how probabilities are assigned to each possible value. Common distributions include the binomial, normal, and Poisson distributions.

- **Expected Value:** The expected value of a random variable is the sum of all possible values, each multiplied by the probability of its occurrence, and is calculated as:

$$E(X) = \sum_{x} xP(X = x). \tag{2.26}$$

  For discrete variables or an integral for continuous variables.

- **Variance and Standard Deviation:** The variance of a random variable is a measure of the dispersion of its values around the mean, and the standard deviation

is the square root of the variance. They are calculated as follows:

$$\mathrm{Var}(X) = E[(X - E(X))^2], \quad \sigma_X = \sqrt{\mathrm{Var}(X)}. \tag{2.27}$$

- **Law of Large Numbers:** This law states that as the number of trials in a random experiment increases, the average of the results obtained from the experiment is likely to get closer to the expected value.
- **Central Limit Theorem:** The central limit theorem states that the distribution of the sum (or average) of a large number of independent, identically distributed variables will be approximately normal, regardless of the underlying distribution.

2.1.4 Hilbert Spaces

A Hilbert space is a key concept in mathematics and physics, particularly useful in quantum mechanics, signal processing, and functional analysis. Below are the core attributes that define a Hilbert space:

1. **Vector Space:** A Hilbert space is a vector space equipped with vector addition and scalar multiplication operations that satisfy standard vector space axioms.
2. **Inner Product:** The defining feature of a Hilbert space is the inner product, a function that assigns a scalar to each pair of vectors. This inner product is denoted as $\langle x, y \rangle$, linear in the first argument, conjugate symmetric, and positive definite.
3. **Norm Derived from Inner Product:** The inner product induces a norm on the space, defined by $\|x\| = \sqrt{\langle x, x \rangle}$. This norm measures the magnitude or length of vectors.
4. **Completeness:** A Hilbert space is complete, meaning every Cauchy sequence of vectors converges to a limit within the space. This property ensures that the space is closed under the limit of sequence operation.
5. **Orthogonality and Orthonormal Basis:** Vectors can be orthogonal in a Hilbert space, with two vectors x and y being orthogonal if $\langle x, y \rangle = 0$. An orthonormal basis is a basis where all vectors are orthogonal to each other, and each has a unit norm. Every vector in the space can be uniquely expressed as a sum of scalars times these basis vectors.
6. **Examples of Hilbert Spaces:**
   - **Euclidean Space:** $\mathbb{R}^n$ or $\mathbb{C}^n$ with the standard inner product $\langle x, y \rangle = \sum_{i=1}^{n} x_i \overline{y_i}$.

- **Sequence Spaces:** l^2, the space of all square-summable sequences of real or complex numbers.
- **Function Spaces:** L^2, the space of square-integrable functions over a given interval or domain.

Hilbert spaces provide the framework for the mathematical formulation of quantum mechanics, where states are vectors and observables are operators on these spaces.

In addition to the above concepts (which we will use in this thesis), group theory helps us understand the symmetries and invariants in quantum systems, which is crucial for tasks like quantum error correction and entanglement characterization. The concept of tensor products allows for the description of composite systems, where individual quantum systems are combined into larger configurations, crucial for the scalability of quantum computing. Operator theory, particularly the study of unitary and Hermitian operators, enables the rigorous description of quantum dynamics and quantum observables.

The rest of this chapter will go through quantum computing basics, including:

- All About Qubits.
- Is it 0, 1, or both? (Superposition).
- Spooky Action At A Distance (Entanglement).
- No Clones Allowed! (The No-clone Theorem).
- Waves and Interference.
- Decoherence.
- Quantum Gates.
- Measurements.
- Quantum Circuits.

2.2 All About Qubits

A qubit, short for "quantum bit," is the fundamental unit of quantum information and the basic building block of quantum computers. It is the quantum analog of a classical bit. Mathematically, a qubit is represented as a vector using the bra-ket notation (Which will be explained in depth shortly).

2.2.1 Qubits Mathematically

The bra-ket notation, also known as Dirac notation, is used in quantum mechanics to represent quantum states and operators. It was introduced by the physicist Paul Dirac to differentiate the state of a qubit from that of the classical binary 0 or 1. So, when speaking of qubits, we say the qubit state is either $|0\rangle$ or $|1\rangle$ (read as ket 0 and ket 1, respectively).

Ket vectors represent quantum states and are typically written as $|\psi\rangle$, where $|\psi\rangle$ is a column vector in a complex vector space. For example:

- $|0\rangle$ represents the quantum state corresponding to the classical bit 0 as a column vector $\begin{bmatrix}1\\0\end{bmatrix}$.
- $|1\rangle$ represents the quantum state corresponding to the classical bit 1 d as a column vector $\begin{bmatrix}0\\1\end{bmatrix}$.
- $|\psi\rangle$ represents an arbitrary quantum state as a column vector $\begin{bmatrix}\alpha\\\beta\end{bmatrix}$, where α and β are complex numbers. These coefficients must satisfy the normalization condition:

$$|\alpha|^2 + |\beta|^2 = 1, \tag{2.28}$$

Bra vectors are the complex conjugate transposes of ket vectors and are typically written as $\langle\psi|$, where $\langle\psi|$ is a row vector. The bras for the kets we just mentioned are:

- $\langle 0|$ represents the bra vector corresponding to the quantum state $|0\rangle$ and can be represented as a row vector $\begin{bmatrix}1 & 0\end{bmatrix}$.
- $\langle 1|$ represents the bra vector corresponding to the quantum state $|1\rangle$ and can be represented as a row vector $\begin{bmatrix}0 & 1\end{bmatrix}$.
- $\langle\psi|$ represents the bra vector corresponding to the quantum state $|\psi\rangle$ and can be represented as a row vector $\begin{bmatrix}\alpha^* & \beta^*\end{bmatrix}$, where α^* and β^* are the complex conjugates of α and β.

We can also use the bra-ket notation to represent vectors' inner and outer products (qubits). The inner product of two vectors, $\langle a|b\rangle$, can be calculated as the dot product of their corresponding column vectors and will result in a scaler. For example, the inner product of $\langle 0|$ and $|1\rangle$ can be written as:

$$\langle 0|1\rangle = \begin{bmatrix}0 & 1\end{bmatrix} \cdot \begin{bmatrix}1\\0\end{bmatrix} = 0 \tag{2.29}$$

The result of the inner product of $|0\rangle$ and $|1\rangle$ is zero because they are orthogonal states. In quantum computing, we refer to them as the *computational basis* (Discussed in depth in the measurement section).

We can use the bra-ket notation to describe the outer product of $|a\rangle$ and $|b\rangle$, resulting in a matrix. For example, the outer product of $|0\rangle\langle 0|$ corresponds to the matrix $\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$.

We also use the bra-ket notation to describe a qubit's arbitrary state as a linear combination of the basis states $|0\rangle$ and $|1\rangle$. Using the format: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ Here, α and β are complex numbers determining the probability amplitudes of the qubit in the 0 or 1 state.

The probabilities of measuring the qubit in either state are given by the squared magnitudes of the coefficients, i.e., $|\alpha|^2$ and $|\beta|^2$. The sum of these probabilities is always 1, ensuring that the qubit will be measured in either the 0 or 1 state.

2.2.2 Qubits Physically

Conceptually, qubits can be defined using a quantum system with two distinguishable states. For example, the energy levels are one type of observable of an electron. Two levels of the energy levels can be used to describe a qubit, such as the ground state $|g\rangle$ for $|0\rangle$ and the first excited state $|e\rangle$ for $|1\rangle$. Another example is the polarization of a photon, where we can denote horizontal polarization as $|H\rangle$ and the vertical polarization as $|V\rangle$.

There are various approaches to construct qubits physically speaking, including:

- Superconducting qubits: These qubits are based on superconducting circuits, where electrical current can flow without resistance. Superconducting qubits utilize Josephson junctions to create nonlinear circuits that store and manipulate quantum information. Companies like IBM, Google, and Rigetti Computing have built quantum processors using superconducting qubits [72, 73, 74].

- Trapped ion qubits: In this approach, individual ions are trapped using electromagnetic fields and manipulated using lasers or microwave radiation. The internal energy levels of the ions are used to represent the qubit states. Trapped ion qubits offer long coherence times and high-fidelity operations. Companies such as IonQ and Quantinuum are developing trapped ion quantum computers [75, 76, 77].

- Topological qubits: Topological qubits rely on anyons particles (particles less restricted than the two kinds of standard elementary particles, fermions, and bosons as they acquire any phase when the particles are swapped) in two-dimensional systems. These qubits store quantum information in the topological properties of the system, making them inherently more robust against local errors and noise. Microsoft is one of the companies researching this type of qubits [78, 79, 80].

- Photonic qubits: These qubits utilize the quantum properties of photons, such as their polarization or path, to encode quantum information. Photonic qubits can be manipulated using linear optical elements, such as beam splitters and phase shifters. We can measure these qubits using single-photon detectors. Xanadu and PsiQuantum are companies developing photonic quantum computing platforms [81, 82, 83].

- Quantum dots: Quantum dots are semiconductor nanostructures that can trap individual electrons. The electron spin can be used to represent the 0 and 1 states. Quantum dots can be manipulated using electric and magnetic fields [84, 85, 86].

- Atomic spin qubits: These are based on atoms' nuclear or spin states, often in solid-state systems like diamond or silicon. Researchers can control and read out the spin states by manipulating the magnetic environment and applying microwave pulses [87, 88, 89].

Though all these techniques are used to construct qubits physically, it is essential to understand the difference between what we refer to as logical qubits and physical qubits.

- *Physical qubits* are implemented using a specific physical system using one of the approaches above. Physical qubits store and manipulate quantum information and are subject to noise and errors. The performance and reliability of a quantum computer depend on the quality of the physical qubits, such as their coherence time, gate fidelity, and error rates.

- *Logical qubits* are an abstract representation of qubits in fault-tolerant quantum computing. They are constructed by encoding the quantum information across multiple physical qubits using quantum error correction (QEC) [56, 90, 91]. QEC protects the quantum information from errors and noise by distributing the information redundantly and allowing the detection and correction of errors without directly measuring the qubit's state.

The ratio of physical qubits to logical qubits can vary depending on the specific QEC code and the selected error threshold. Generally, the more physical qubits used to encode a single logical qubit, the more robust the quantum computer will be against errors. Developing practical, large-scale quantum computers requires successfully implementing logical qubits and quantum error correction techniques.

2.2.3 Qubits Graphically

Bloch Sphere

To have a better understanding of qubits, we needed to create a visual way to understand them better. The visualization approach we use to represent qubits is Bloch

spheres. A Bloch sphere is a geometrical representation of the state of a single qubit in quantum mechanics. It is a three-dimensional sphere with a radius of 1 on which the pure states of a qubit can be visualized as points on the sphere's surface. A qubit can be represented as a linear combination of the basis states $|0\rangle$ and $|1\rangle$: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. The complex numbers α and β are the probability amplitudes of the qubit being in the 0 or 1 state. The probabilities of measuring the qubit in either state are given by the squared magnitudes of the coefficients, i.e., $|\alpha|^2$ and $|\beta|^2$, with the constraint that $|\alpha|^2 + |\beta|^2 = 1$.

A Bloch sphere represents a qubit's state using coordinates (θ, ψ) as follows: $|\psi\rangle = \cos(\frac{\theta}{2})|0\rangle + e^{i\phi}\sin(\frac{\theta}{2})|1\rangle$ Here, θ (theta) ranges from 0 to π, and $\phi(phi)$ ranges from 0 to 2π. The north pole of the Bloch sphere corresponds to the $|0\rangle$ state, and the south pole corresponds to the $|1\rangle$ state, as can be seen in Figure 2.2. The superposition states lie on the surface between these poles, with the angles θ and ϕ determining the specific state.

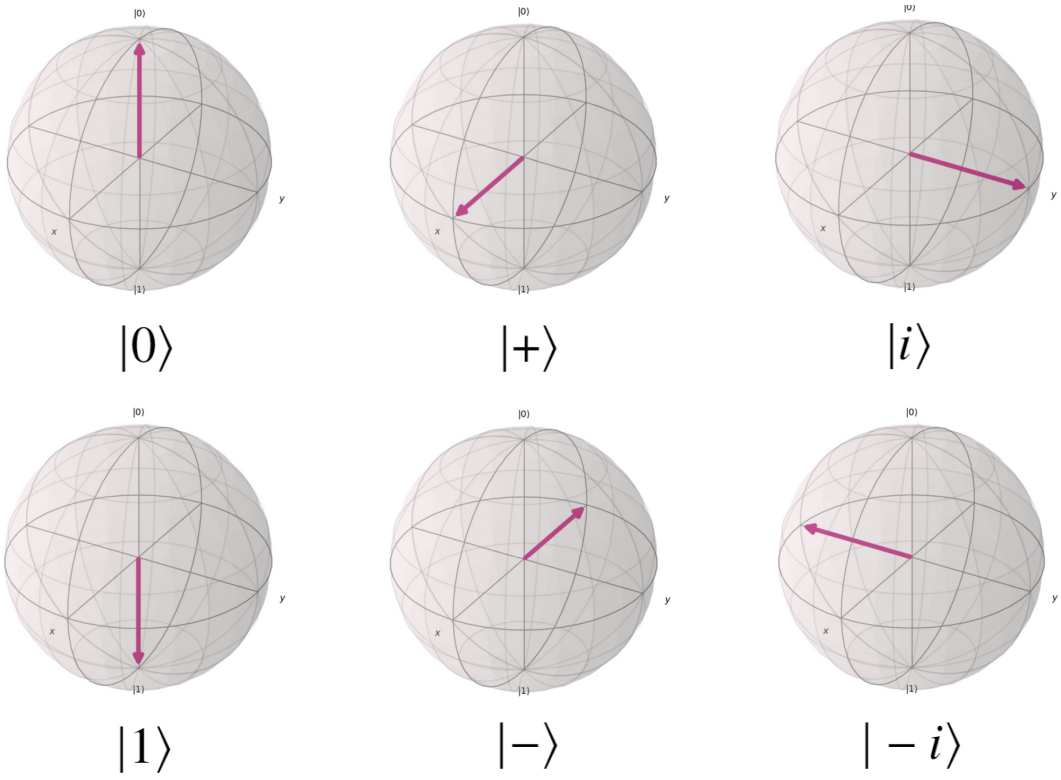


Figure 2.2: Bloch sphere representation of states $|0\rangle$, $|1\rangle$, $|+\rangle$, $|-\rangle$,$|i\rangle$, and$| - i\rangle$.

While invaluable for visualizing the state of one qubit, the Bloch sphere encounters limitations with multi-qubit systems. It cannot directly represent states involving more than one qubit, essential for understanding quantum entanglement and other multi-qubit

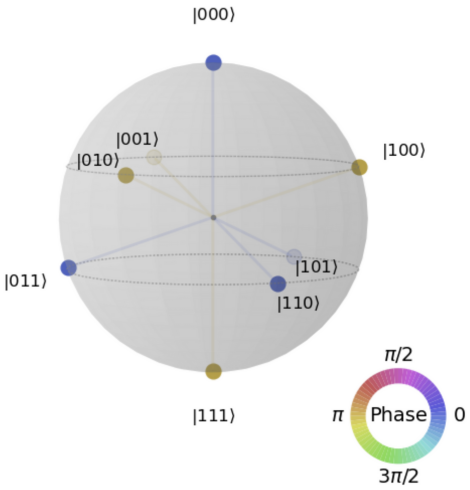


Figure 2.3: The Q-sphere representation of $|000\rangle - |001\rangle - |010\rangle + |011\rangle - |100\rangle + |101\rangle + |110\rangle - |111\rangle$.

phenomena. The sphere's framework is restricted to two levels (or states), and thus, it cannot accommodate the visualization of higher-dimensional quantum states in complex quantum computing scenarios [92].

Q-Sphere

The Q-sphere is an approach used to represent the state of a system of one or more qubits by associating each computational basis state with a point on the surface of a sphere. Each node's radius is proportional to the probability of its basis state. In contrast, the node color indicates its phase according to the phase color circle in the bottom right corner of the Q-sphere Figure 2.3.

In a Q-sphere, we place the state where all qubits are 0 at the sphere's top (north pole) and where all qubits are 1 at the bottom (south pole). Other states are arranged in between, forming circles of latitude based on their Hamming distance from the 0 states. For example, if we have a circuit constructing a system of 3 qubits in superposition, except that states $|001\rangle$, $|010\rangle$, $|100\rangle$, and $|111\rangle$ have phase equal to π, the visualization of the Q-sphere of that system can be seen in Figure 2.3.

The Circle Notation

Circle notation is a graphical method for representing quantum states, particularly useful in quantum computing and information science. This visualization technique simplifies the understanding of complex quantum states by depicting them in a more intuitive and accessible manner, bridging the gap between abstract mathematical concepts and their practical applications.

In an n-qubit system, there are 2^n possible basis states, each represented by a circle. These circles visualize the complex numbers that describe the quantum state's amplitude and phase. The magnitude of the amplitude is shown as the filled area within a circle,

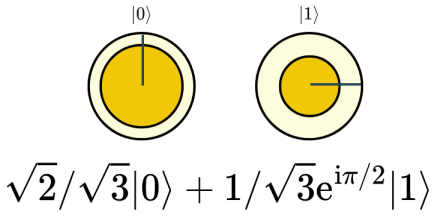


Figure 2.4: The circle notation representation of $\sqrt{2}/\sqrt{3}|0\rangle + 1/\sqrt{3}e^{i\pi/2}|1\rangle$.

while the phase is indicated by the angle of a radial line within the circle relative to a vertical line. This approach allows learners to grasp quantum algorithms' basic ideas and mechanisms without delving deeply into linear algebra and matrix operations.

For example, a one qubit state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ can be visualized using two circles. The magnitude of α and β are represented by the radii of the inner circles, and their phases are shown as the angles of lines inside these circles. When extending this notation to multi-qubit systems, each basis state is represented by a corresponding circle, and the overall state is depicted as a superposition of these circles.

The dimensional circle notation (DCN) is an extension of the circle notation [93]. The DCN builds upon the circle notation, which graphically represents quantum states using circles to depict complex numbers, with magnitudes shown as filled areas within the circles and phases as angles relative to a vertical line, as seen in the example in Figure 2.4.

While the circle notation makes the visualization of quantum states more accessible, it has some limitations. For instance, the action of quantum gates on the states is not always intuitive and may require additional effort to understand. Moreover, distinguishing between separable and entangled states directly from the circle notation can be challenging. Another limitation of the circle notation is the ability to visualize the state of a large number of qubits effectively.

Despite these limitations, circle notation is a powerful tool for introducing quantum computing concepts. It aids in education and research by providing a more intuitive way to visualize and comprehend quantum states and their dynamics.

2.3 The Fundamentals

2.3.1 Is It 0, 1, Or Both? (Superposition)

One of the fundamental principles of quantum mechanics is superposition. Quantum superposition states that a quantum system can exist in multiple states simultaneously

until a measurement is made. In classical physics, a system can only be in one well-defined state at any given time. However, particles such as electrons, photons, or qubits can exist in a linear combination of multiple states in quantum mechanics. This linear combination is described mathematically as the "wave function" or "state vector" of the qubit(s). For example, as mentioned in Subsection 2.2, a qubit can be in state $|0\rangle$ or $|1\rangle$, but it can also be in a superposition of both the $|0\rangle$ and $|1\rangle$ states, represented by the following state vector: $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ Here, α and β are complex numbers that determine the probability amplitudes of the qubit in the $|0\rangle$ and $|1\rangle$ states, respectively.

The squared magnitudes of the corresponding amplitudes give the probabilities of measuring the qubit in either state: $P(0) = |\alpha|^2$ and $P(1) = |\beta|^2$. Superposition allows quantum systems to explore multiple possibilities simultaneously, so quantum computers outperform classical computers in specific computational tasks. Qubits can be in superposition during the computations until a measurement is done. Once a measurement is performed on a quantum system in a superposition, the system *collapses* into one of the possible states with a probability determined by the amplitudes.

2.3.2 Spooky Action At A Distance (Entanglement)

Entanglement is a unique quantum phenomenon that occurs when two or more particles become correlated so that the state of one particle cannot be described independently of the other particles, even when large distances separate them. Einstein has described this correlation as "Spooky action at a distance." Because these particles are entangled, they somehow "know" the other particle's state *without* communication. This correlation is stronger than any possible classical correlation, fundamental to quantum mechanics, and a basis for many quantum algorithms, such as teleportation.

Entanglement has been experimentally verified in many quantum systems, including photons, electrons, and atoms. These systems can be used to construct qubits as discussed in Subsection 2.2.

Mathematically, we can describe one joint state of two entangled qubits as $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

Entanglement is a key resource in quantum information processing, quantum computing, quantum communication, and quantum cryptography.

A multi-qubit state that can be described as a product of the states of one qubit is called *separable state*.

Separable States

Consider a composite quantum system composed of two subsystems, A and B. The state of this system is separable if it can be expressed as:

$$|\psi\rangle = |\phi_A\rangle \otimes |\phi_B\rangle, \tag{2.30}$$

where:

- $|\psi\rangle$ is the entire system's state.
- $|\phi_A\rangle$ is the state of subsystem A.
- $|\phi_B\rangle$ is the state of subsystem B.
- $\otimes$ denotes the tensor product, which combines the states of A and B into a single state vector for the composite system.

Properties:

- **No Entanglement**: In separable states, measurements on one subsystem do not affect the outcomes of measurements on another subsystem, reflecting a lack of entanglement.
- **Independent Subsystems**: Each subsystem can be described completely by its state vector.

For two qubits, consider states $|0\rangle$ and $|1\rangle$ as the basis states for each qubit. A separable state of this two-qubit system could be:

$$|\psi\rangle = |0\rangle_A \otimes |1\rangle_B \tag{2.31}$$

This state indicates that qubit A is definitively in the state $|0\rangle$ and qubit B in $|1\rangle$, independently of each other.

A separable state can also involve superpositions if those superpositions pertain to individual subsystems. For instance:

$$|\psi\rangle = \left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)_A\right) \otimes \left(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)_B\right) \tag{2.32}$$

Each qubit is in a superposition, but the overall state is still separable as there is no entanglement between the qubits.

Entangled States

There are many ways we can construct entangled states. Some famous and widely used in quantum algorithms entangled states include:

- Bell states (EPR pairs) are maximally entangled two-qubit states, meaning the qubits are as strongly correlated as possible. There are four possible Bell states:

$$\left|\Phi^+\right\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle_A \otimes |0\rangle_B + |1\rangle_A \otimes |1\rangle_B\right)$$
$$|\Phi^-\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle_A \otimes |0\rangle_B - |1\rangle_A \otimes |1\rangle_B\right)$$
$$\left|\Psi^+\right\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle_A \otimes |1\rangle_B + |1\rangle_A \otimes |0\rangle_B\right)$$
$$|\Psi^-\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle_A \otimes |1\rangle_B - |1\rangle_A \otimes |0\rangle_B\right)$$

- W states [94] are entangled states involving three or more qubits. W states exhibit a more robust form of entanglement by maintaining some entanglement even if one qubit is measured. The general equation of an N-qubit W state can be written as: $|W_N\rangle = \frac{1}{\sqrt{N}}(|100\ldots0\rangle + |010\ldots0\rangle + \cdots + |000\ldots1\rangle)$

- GHZ state (Greenberger-Horne-Zeilinger state) [95] is an entangled quantum state that involves three or more qubits. It is named after its discoverers, Daniel Greenberger, Michael Horne, and Anton Zeilinger. An N-qubit GHZ state can be written as:

$$|GHZ_N\rangle = \frac{1}{\sqrt{2}}(|0\ldots0\rangle + |1\ldots1\rangle) \tag{2.33}$$

- Cluster states are entangled states that can be generated in a lattice-like structure of qubits [96].

These are just some examples of entangled states. However, we can create different entangled states based on the system we decide to use and the target application.

The Basis States

Basis states provide the framework within which quantum superpositions and entanglements are defined. A basis set is a set of linearly independent vectors that span the vector space associated with a quantum system. Basis states are the individual vectors of this set. The most commonly used basis for qubits, the quantum analogs of classical binary bits, is the computational basis.

The computational basis, also known as the standard basis, consists of all possible states in which a set of qubits can be, each represented by a different combination of 0s and 1s.

For example, the computational basis for a single qubit is $\{|0\rangle, |1\rangle\}$, where:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

2.3.3 No-cloning Theorem

The no-cloning theorem states that creating an exact copy of an arbitrary unknown quantum state is impossible. Meaning there is no unitary operator U such that for an arbitrary quantum state $|\psi\rangle$ and a fixed state $|e\rangle$:

$$U(|\psi\rangle \otimes |e\rangle) = |\psi\rangle \otimes |\psi\rangle \tag{2.34}$$

The proof of the no-cloning theorem relies on the linearity of quantum mechanics. Assume there exists a unitary operator U that can clone any arbitrary quantum state $|\psi\rangle$. If we have two arbitrary quantum states $|\psi_1\rangle$ and $|\psi_2\rangle$ and U can clone these states, we must have:

$$U(|\psi_1\rangle \otimes |e\rangle) = |\psi_1\rangle \otimes |\psi_1\rangle. \tag{2.35}$$

$$U(|\psi_2\rangle \otimes |e\rangle) = |\psi_2\rangle \otimes |\psi_2\rangle. \tag{2.36}$$

Now consider a superposition of $|\psi_1\rangle$ and $|\psi_2\rangle$:

$$|\psi\rangle = a|\psi_1\rangle + b|\psi_2\rangle. \tag{2.37}$$

If we apply U to the superposition state it gives:

$$U((a|\psi_1\rangle + b|\psi_2\rangle) \otimes |e\rangle) = aU(|\psi_1\rangle \otimes |e\rangle) + bU(|\psi_2\rangle \otimes |e\rangle). \tag{2.38}$$

$$= a(|\psi_1\rangle \otimes |\psi_1\rangle) + b(|\psi_2\rangle \otimes |\psi_2\rangle). \tag{2.39}$$

However, if U were a perfect cloning operator, it should produce:

$$U((a|\psi_1\rangle + b|\psi_2\rangle) \otimes |e\rangle) = (a|\psi_1\rangle + b|\psi_2\rangle) \otimes (a|\psi_1\rangle + b|\psi_2\rangle). \tag{2.40}$$

Since the two expressions for the application of U to the superposition are not equal unless $|\psi_1\rangle$ and $|\psi_2\rangle$ are identical or orthogonal, leading to a contradiction. This

concludes that there is no such unitary operator U, proving the no-cloning theorem.

2.3.4 Interference

Interference is a concept existing in both classical and quantum physics. It describes the phenomenon where two or more overlapping waves combine to produce a new wave pattern from the superposition principle. Interference can be constructive, where the amplitudes of the waves add up, or destructive, where the amplitudes cancel each other out. Quantum particles, such as electrons and photons, have a dual nature; they exhibit wave-like and particle-like properties. As a result, these particles can interfere with themselves and each other, as demonstrated by the double-slit experiment.

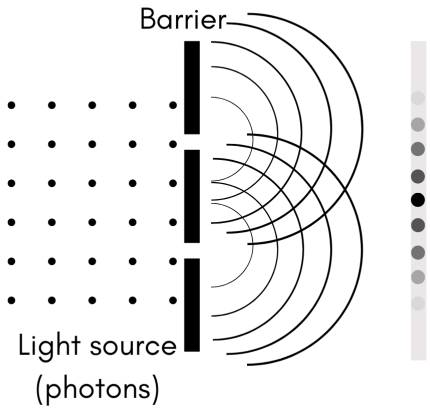


Figure 2.5: A simple demonstration of the two-slit experiment.

The two-slit experiment was initially observed and explained using classical coherent waves. However, the paradox of a single particle interfering with itself illustrates wave-particle duality. When many single-photon events are accumulated, they produce the same interference pattern as seen with classical coherent waves (Figure 2.5).

Mathematically, the wave function describing the particles can be represented as $\psi(x,t)$ where x is the position on the screen and t is time.

The interference pattern that appears on the screen is because the wave function from Slit A and the one from Slit B overlap and interfere with each other.

The interference pattern is described mathematically by the superposition of the wave functions from both slits as $\psi(x,t) = \psi_A(x,t) + \psi_B(x,t)$.

After passing through the slits, the two parts of the wave function interfere with each other. The interference pattern depends on the relative phases of the wave functions from

each slit, which evolve. The resulting wave function at the screen, $\psi(x,t)$, determines the probability distribution of detecting particles at different positions x.

Now, let's consider what happens when we try to detect which slit the particle goes through. To do this, we place detectors near each slit to determine which path the particle takes.

When we introduce detectors, something interesting happens. The interference pattern disappears, and we see a pattern that corresponds to particles passing through either Slit A or Slit B, which is what we would expect to see if particles did not have a dual nature.

In quantum algorithms, qubits are manipulated to create superposition states. Then, interference patterns are designed to amplify the probability of obtaining the correct solution while canceling the probability of incorrect solutions. We will discuss how quantum algorithms work further in Chapter 3. For the remainder of this thesis, we will only address $\psi(x,t)$ as $\psi(x)$ since we are focusing on the computational aspects only.

2.4 Building Quantum Circuits

2.4.1 Quantum gates

Quantum gates are operators we use to manipulate the state of a qubit. There are many quantum gates; you can even create custom gates once you fully understand the basic ones. Quantum gates are the building blocks of quantum circuits. They are operators that perform a specific function on one qubit or more. They are the quantum equivalent of classical logic gates in all technology today. However, unlike most classical logic gates, quantum gates are reversible, which means you can undo their effect by reapplying on the same qubits.

Mathematically, quantum gates are represented using unitary operations. Unitary operations are then described using square matrices. On the other hand, the state of qubits is represented as a vector. Geometrically speaking, vectors usually have an amplitude and a phase representing their location in space. In higher-dimensional spaces, the phase can be described by a set of angles corresponding to the spherical coordinates of the vector.

To find the results of applying a gate to a qubit in any state, we multiply the gate's matrix by the qubit's state vector (Figure 2.6).

Before we go further, we will discuss unitaries a bit. A unitary is a type of linear transformation that preserves the inner product of vectors. Unitaries are invertible if U is a unitary operator, then there exists a unitary operator $U^\dagger$ such that $U^\dagger U = UU^\dagger = I$, where I is the identity operator.

Any unitary operator has the following characteristics:

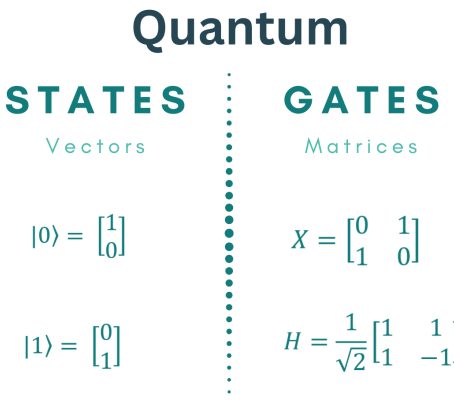


Figure 2.6: A mathematical representation of qubits and gates.

1. **Preservation of Norm:** A unitary operator U acting on a vector $|\psi\rangle$ preserves the norm of the vector, which means that $\|U|\psi\rangle\| = \||\psi\rangle\|$, where $\|\cdot\|$ is the norm of a vector.
2. **Preservation of Inner Product:** If $|\psi\rangle$ and $|\phi\rangle$ are two quantum states, then the inner product is preserved under the action of a unitary operator: $\langle U\psi|U\phi\rangle = \langle\psi|\phi\rangle$.
3. **Hermitian (Self-Adjoint) Operators:** a unitary operator is also Hermitian in finite-dimensional vector spaces. This means that $U^\dagger = U$.
4. **Orthonormal Basis Transformation:** Unitary operators can perform transformations between different orthonormal bases in vector spaces.

One qubit Gates

Firstly, we will address gates that operate on single qubits. As we will see, those gates operate on only one qubit at a time and can be extended to run on more qubits.

There are different one qubit gates; the basic ones are:

- Pauli gates:

- X gate (a bit-flip or NOT gate) flips the state of a single qubit:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\text{For example, applying the X gate flips the 0 state to } 1|0\rangle : \quad X|0\rangle = |1\rangle$$

$$\text{and vice versa} |1\rangle : \quad X|1\rangle = |0\rangle$$

The rotation form of the X gate is:

$$R_x(\theta) = \begin{pmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ -i\sin(\theta/2) & \cos(\theta/2) \end{pmatrix}$$

For example, for $\theta = \pi$:

$$R_x(\pi) = \begin{pmatrix} 0 & -i \\ -i & 0 \end{pmatrix} = -iX$$

Since the difference here is only in the global phase, the X gate and the $R_x(\pi)$ are equivalent.

- Z gate (also called a phase-flip gate) flips the phase of the qubit.

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

$$\text{The Z gate does not change the 0 state:} |0\rangle : \quad Z|0\rangle = |0\rangle$$

$$\text{However, it flips the phase of the 1 state} |1\rangle : \quad Z|1\rangle = -|1\rangle$$

The rotation form of the Z gate is:

$$R_z(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}$$

For example, for $\theta = \pi$:

$$R_z(\pi) = \begin{pmatrix} -i & 0 \\ 0 & i \end{pmatrix}$$

- Y gate combines the X and Z effects and introduces a complex phase.

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

Applying the Y gate to the 0 state adds a complex phase $|0\rangle : \quad Y|0\rangle = i|1\rangle$

while it adds a negative complex phase to the 1 state $|1\rangle : \quad Y|1\rangle = -i|0\rangle$

The rotation form of the Y gate is:

$$R_y(\theta) = \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix}$$

For example, for $\theta = \pi$:

$$R_y(\pi) = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}$$

- Hadamard gate (H gate) creates superposition by transforming the basis states $|0\rangle$ and $|1\rangle$ to states that are an equal combination of $|0\rangle$ and $|1\rangle$ with different phases, which are states $|+\rangle = H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.
- Phase gates are used to manipulate the relative phases of qubit states in quantum algorithms.
  - The S gate adds a $\pi/2$ phase to the $|1\rangle$ state.
  - The T gate adds a $\pi/4$ phase to the $|1\rangle$ state.

We can describe these gates using matrices or using Dirac notation. Table 2.1 shows the different gates, their matrix representation, and their effect on the computational basis.

Multi-qubit Gates

When we work with qubits, we often need to use more than one qubit at a time to perform computations. Hence, the one-qubit gates can be used to build multi-qubit gates. Using different approaches, we can create a controlled version of the one qubit gates. Two of the most commonly used multi-qubit gates are the two-qubit and three-qubit extensions of the NOT gate.

- CNOT gate (Controlled-NOT gate) is a two-qubit gate that flips the state of the second qubit (target) if and only if the first qubit (control) is in the $|1\rangle$ state. The truth table for the CNOT gate is 2.2.

Gate	Matrix Representation	Gate Applied to Basis States
X	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	X$\|0\rangle$ = $\|1\rangle$, X$\|1\rangle$ = $\|0\rangle$
Y	$\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$	Y$\|0\rangle$ = -i$\|1\rangle$, Y$\|1\rangle$ = i$\|0\rangle$
Z	$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$	Z$\|0\rangle$ = $\|0\rangle$, Z$\|1\rangle$ = -$\|1\rangle$
H	$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$	$H\|0\rangle = \frac{1}{\sqrt{2}}(\|0\rangle + \|1\rangle)$, H$\|1\rangle = \frac{1}{\sqrt{2}}(\|0\rangle$ - $\|1\rangle)$
S	$\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$	S$\|0\rangle$ = $\|0\rangle$, S$\|1\rangle$ = i$\|1\rangle$
T	$\begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$	T$\|0\rangle$ = $\|0\rangle$, T$\|1\rangle$ = $e^{i\pi/4}\|1\rangle$

Table 2.1: Some one qubit gates, their matrix representation, and their effect on the computational basis.

- Toffoli gate (CCNOT gate) is a three-qubit gate that performs a controlled-controlled-NOT operation. It flips the state of the third qubit (target) only if both the first and second qubits (controls) are in the $|1\rangle$ state. The truth table for the CCNOT gate is in Table 2.2.

2.4.2 Measurements

Measurement in quantum mechanics represents the process of extracting information from a quantum system, which leads to the collapse of the system's wave function. Mathematically, measurements are described by projection operators, and the outcomes are probabilistic.

Because of the probabilistic nature of quantum systems, we cannot predict the outcome we will obtain when we measure a quantum system. Instead, we can only compute probabilities for various possible outcomes.

Now let's clarify some of the concepts in that definition:

- Wavefunction Collapse: When a measurement is made, it collapses the wavefunction to one of the possible eigenstates of the observable being measured. This collapse is random, and the probability of collapsing to a particular state is determined by the coefficients of that state in the superposition.

- Projection Operators: These are operators that project the quantum state onto one of the eigenstates of the observable. The square of the absolute value of

Gate	Input	Output
CNOT	$\|00\rangle$	$\|00\rangle$
	$\|01\rangle$	$\|10\rangle$
	$\|10\rangle$	$\|11\rangle$
	$\|11\rangle$	$\|10\rangle$
CCNOT	$\|000\rangle$	$\|000\rangle$
	$\|001\rangle$	$\|001\rangle$
	$\|010\rangle$	$\|010\rangle$
	$\|011\rangle$	$\|011\rangle$
	$\|100\rangle$	$\|100\rangle$
	$\|101\rangle$	$\|101\rangle$
	$\|110\rangle$	$\|111\rangle$
	$\|111\rangle$	$\|110\rangle$

Table 2.2: Truth table for the CNOT and CCNOT gates

the projection of the quantum state onto the corresponding eigenstate gives the probability of obtaining a particular measurement outcome.

Projection operators play a pivotal role in the theory of quantum measurements, where they mathematically formalize the outcome of measuring quantum states. A projection operator P is associated with each possible outcome of a quantum measurement and is defined for a particular eigenstate $|\psi\rangle$ of an observable as:

$$P = |\psi\rangle\langle\psi| \tag{2.41}$$

The projection operator acts on the quantum state space to project any state $|\phi\rangle$ onto the subspace spanned by $|\psi\rangle$. This operation results in the component of $|\phi\rangle$ aligned with $|\psi\rangle$, effectively isolating this measurement outcome.

Two essential properties characterize projection operators:

- **Idempotence**: $P^2 = P$. This property signifies that once a state has been projected onto $|\psi\rangle$, further applications of P do not change the state, which aligns with the stable post-measurement state in the same projected subspace.
- **Hermiticity**: $P^\dagger = P$. This indicates that the projection operator is self-adjoint, a necessary condition to represent an observable in quantum mechanics.

In the quantum measurement process, the projection operator quantifies the state reduction or collapse mechanism. Upon measuring an observable and obtaining a specific outcome represented by $|\psi\rangle$, the state of the system instantaneously reduces to

the state $|\psi\rangle$, which is mathematically described by the action of P on the system's initial state:

$$|\phi\rangle \rightarrow P|\phi\rangle = |\psi\rangle\langle\psi|\phi\rangle \tag{2.42}$$

$$\text{Probability} = \|\langle\psi|\phi\rangle\|^2 \tag{2.43}$$

The use of projection operators underscores the probabilistic nature of quantum mechanics and the non-deterministic outcomes of measurements.

Quantum states can be represented in various bases, corresponding to the eigenstates of different observables. When a quantum state is measured, the basis of the measurement determines the possible outcomes and their associated probabilities.

`Projection Operators and Measurement Basis`

For an observable with eigenstates $\{|a_i\rangle\}$, the projection operator associated with each eigenstate $|a_i\rangle$ is defined as:

$$P_i = |a_i\rangle\langle a_i|$$

This operator projects any quantum state $|\psi\rangle$ onto the direction of $|a_i\rangle$.

The process of measuring a quantum state $|\psi\rangle$ with respect to an observable involves:

1. **Projection:** The state $|\psi\rangle$ is projected onto each eigenstate $|a_i\rangle$ using the projection operator P_i:
$$P_i|\psi\rangle = |a_i\rangle\langle a_i|\psi\rangle$$

2. **Probability Calculation:** The probability that the measurement results in the state $|a_i\rangle$ is given by:
$$\text{Probability}(a_i) = \|P_i|\psi\rangle\|^2 = |\langle a_i|\psi\rangle|^2$$

3. **Outcome:** The outcome of the measurement is one of the eigenstates $|a_i\rangle$, with the system collapsing to this state post-measurement.

Changing the measurement basis to another set of eigenstates $\{|b_j\rangle\}$ associated with a different observable changes the projection operators to:

$$Q_j = |b_j\rangle\langle b_j|$$

The measurement process similarly involves:

$$\text{Probability}(b_j) = \|Q_j|\psi\rangle\|^2 = |\langle b_j|\psi\rangle|^2$$

Now, let us take an example of performing measurements on a qubit on a different basis. Consider a qubit in the $|+\rangle$ state:

$$|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle. \tag{2.44}$$

Measuring $|\psi\rangle$ in the Computational Basis

The computational basis consists of the states $|0\rangle$ and $|1\rangle$. When measuring $|\psi\rangle$ in this basis, the probability of obtaining each outcome is given by the squared magnitude of the inner product between $|\psi\rangle$ and the basis states.

The probability of measuring $|0\rangle$ is:

$$P(0) = |\langle 0|\psi\rangle|^2 \tag{2.45}$$

The inner product $\langle 0|\psi\rangle$ is:

$$\langle 0|\psi\rangle = \langle 0|\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) = \frac{1}{\sqrt{2}}\langle 0|0\rangle + \frac{1}{\sqrt{2}}\langle 0|1\rangle = \frac{1}{\sqrt{2}}\cdot 1 + \frac{1}{\sqrt{2}}\cdot 0 = \frac{1}{\sqrt{2}} \tag{2.46}$$

Thus:

$$P(0) = \left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2} \tag{2.47}$$

Similarly, the probability of measuring $|1\rangle$ is:

$$P(1) = |\langle 1|\psi\rangle|^2 \tag{2.48}$$

The inner product $\langle 1|\psi\rangle$ is:

$$\langle 1|\psi\rangle = \langle 1|\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) = \frac{1}{\sqrt{2}}\langle 1|0\rangle + \frac{1}{\sqrt{2}}\langle 1|1\rangle = \frac{1}{\sqrt{2}}\cdot 0 + \frac{1}{\sqrt{2}}\cdot 1 = \frac{1}{\sqrt{2}} \tag{2.49}$$

Thus:

$$P(1) = \left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2} \tag{2.50}$$

Therefore, the output probabilities for $|\psi\rangle$ in the computational basis are:

$$P(0) = \frac{1}{2} \tag{2.51}$$

$$P(1) = \frac{1}{2} \tag{2.52}$$

Measuring $|\psi\rangle$ in the $|+,-\rangle$ Basis

If we measure $|\psi\rangle$ in the basis formed by $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$, the probability of obtaining each outcome is given by the squared magnitude of the inner product between $|\psi\rangle$ and the new basis states.

The probability of measuring $|+\rangle$ is:

$$P(+) = |\langle +|\psi\rangle|^2 \tag{2.53}$$

The inner product $\langle +|\psi\rangle$ is:

$$\langle +|\psi\rangle = \langle +| \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) = \left(\frac{1}{\sqrt{2}}(\langle 0| + \langle 1|)\right)\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) \tag{2.54}$$

$$= \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\langle 0|0\rangle + \frac{1}{\sqrt{2}}\langle 0|1\rangle + \frac{1}{\sqrt{2}}\langle 1|0\rangle + \frac{1}{\sqrt{2}}\langle 1|1\rangle\right) \tag{2.55}$$

$$= \frac{1}{2}(1 + 0 + 0 + 1) = \frac{1}{2}(2) = 1 \tag{2.56}$$

Thus:

$$P(+) = 1 \tag{2.57}$$

Similarly, the probability of measuring $|-\rangle$ is:

$$P(-) = |\langle -|\psi\rangle|^2 \tag{2.58}$$

The inner product $\langle -|\psi\rangle$ is:

$$\langle -|\psi\rangle = \langle -| \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) = \left(\frac{1}{\sqrt{2}}(\langle 0| - \langle 1|)\right)\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) \tag{2.59}$$

$$= \frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}}\langle 0|0\rangle - \frac{1}{\sqrt{2}}\langle 0|1\rangle + \frac{1}{\sqrt{2}}\langle 1|0\rangle - \frac{1}{\sqrt{2}}\langle 1|1\rangle\right) \tag{2.60}$$

$$= \frac{1}{2}\left(1 - 0 + 0 - 1\right) = \frac{1}{2}(0) = 0 \tag{2.61}$$

Thus:

$$P(-) = 0 \tag{2.62}$$

Therefore, the output probabilities for $|\psi\rangle$ in the $|+,-\rangle$ basis are:

$$P(+) = 1 \tag{2.63}$$

$$P(-) = 0 \tag{2.64}$$

What happens when we try to measure $|0\rangle$ or $|1\rangle$ in the $|+,-\rangle$ basis?
`Measurement of` $|0\rangle$

- Probability of measuring $|0\rangle$ as $|+\rangle$:

$$P(|0\rangle \text{ in } |+\rangle) = |\langle +|0\rangle|^2 = \left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}$$

- Probability of measuring $|0\rangle$ as $|-\rangle$:

$$P(|0\rangle \text{ in } |-\rangle) = |\langle -|0\rangle|^2 = \left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}$$

`Measurement of` $|1\rangle$

- Probability of measuring $|1\rangle$ as $|+\rangle$:

$$P(|1\rangle \text{ in } |+\rangle) = |\langle +|1\rangle|^2 = \left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}$$

- Probability of measuring $|1\rangle$ as $|-\rangle$:

$$P(|1\rangle \text{ in } |-\rangle) = |\langle -|1\rangle|^2 = \left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}$$

Measurement in the $|i, -i\rangle$ Basis

The basis states $|i\rangle$ and $|-i\rangle$ are defined as:

$$|i\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i|1\rangle) \tag{2.65}$$

$$|-i\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i|1\rangle) \tag{2.66}$$

When measuring $|\psi\rangle$ in this basis, the probabilities can be calculated using the inner product of $|\psi\rangle$ with $|i\rangle$ and $|-i\rangle$.

2.4.3 Quantum Circuits

A quantum circuit is a sequence of quantum gates applied to a set of qubits in a specific order to perform quantum computation. Quantum circuits can be represented as a series of time-ordered operations where time flows from left to right in the circuit diagram, and the gates are applied in that order. Quantum circuits typically begin with qubits initialized in the $|0\rangle$ state. Then, various quantum gates, such as one-qubit gates (e.g., Pauli-X, Pauli-Y, Pauli-Z, Hadamard) and multi-qubit gates (e.g., CNOT, Toffoli), are applied to manipulate the qubits's states.

After quantum operations are applied, the final state of the qubits represents the computation's result. Then, measurements are performed on the qubits, collapsing their state to a classical value (0 or 1) based on the probability distribution determined by the sequence of gates used. The current circuit model is used to implement algorithms on various NISQ devices. One of the platforms that can be used to create quantum circuits is Qiskit [97]. To create the different circuits for the four Bell states ($|\Psi+\rangle$, $|\Psi-\rangle$, $|\Phi+\rangle$, and $|\Phi-\rangle$), we can follow the code in Listing 2.1.

```
import qiskit as qk
from qiskit.visualization import plot_circuit_layout, plot_histogram,
    plot_bloch_multivector
from qiskit import Aer, transpile, assemble

# Function to create Bell state circuit
def create_bell_state(qc, qubit1, qubit2, state):
    qc.h(qubit1)  # Apply Hadamard to the first qubit
    qc.cx(qubit1, qubit2)  # Apply CNOT with first qubit as control
```

```
    and second as target

    if state == 'Phi-':
        qc.z(qubit1)
    elif state == 'Psi+':
        qc.x(qubit2)
    elif state == 'Psi-':
        qc.x(qubit2)
        qc.z(qubit1)

# Create a Quantum Circuit for 2 qubits
bell_states = ['Phi+', 'Phi-', 'Psi+', 'Psi-']
circuits = []

for state in bell_states:
    qc = qk.QuantumCircuit(2, 2)
    create_bell_state(qc, 0, 1, state)
    qc.measure([0, 1], [0, 1])
    circuits.append(qc)

# Display the circuits
for i, qc in enumerate(circuits):
    print(f"Bell State {bell_states[i]}:")
    display(qc.draw('mpl'))
```

Listing 2.1: Qiskit code to generate the different Bell states.

Running this code block will display the circuits corresponding to the different states, as seen in Figure 2.7.

2.5 Decoherence

Before we address decoherence, let us first define what is meant when we say "quantum coherence." Quantum coherence refers to the ability of a quantum state to maintain

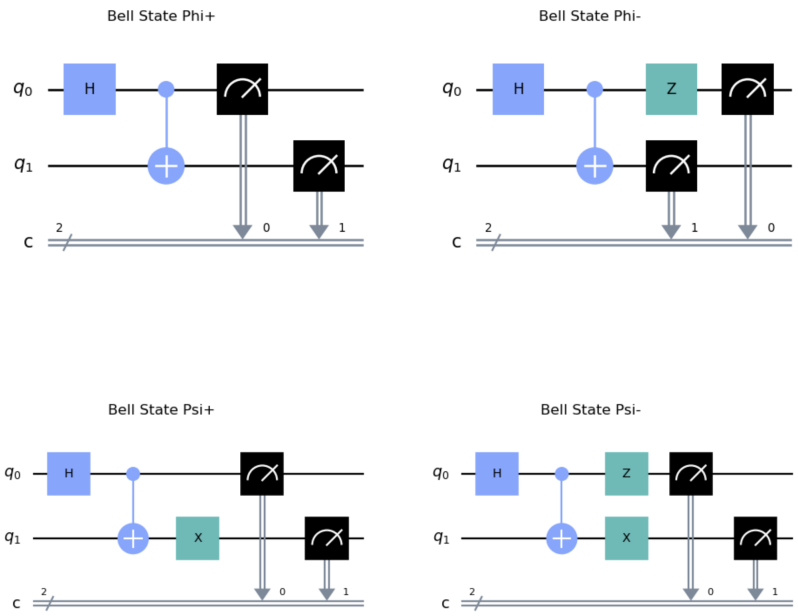


Figure 2.7: The different circuits constructing and measuring the four Bell states.

its superposition and entanglement over time despite interactions with the surrounding environment and the effects of thermalization (which will be explained shortly). It is a fundamental property that underpins many of quantum systems' unique behaviors and applications.

Decoherence occurs when a quantum system interacts with its surrounding environment, including other particles, electromagnetic fields, or other external influences. These interactions introduce entanglement between the quantum system and the environment.

Due to the interactions with the environment, the quantum system becomes entangled with it, causing the quantum coherence to dissipate.

When a quantum system interacts with its environment, its state becomes intertwined with the environment's state. This entanglement means that information about the system's state spreads into the environment, effectively causing the system to lose its distinct quantum properties.

To make it easier to understand, imagine a quantum system as a perfectly balanced coin on its edge, representing a superposition of heads and tails. If this coin interacts with the environment (like being nudged by the wind), it starts to tilt towards heads or tails, losing its balanced superposition. The interaction with the environment makes the coin fall to one side, similar to how quantum states lose their superposition and become more like classical states.

As a result of these interactions, the system no longer behaves like a quantum system in a superposition of states but rather like a classical system in a definite state. We observe

this transition from quantum to classical behavior as decoherence. The unique quantum interference effects disappear, and the system's behavior becomes more predictable and classical.

Decoherence is generally an irreversible process. Once a quantum system becomes entangled with its environment, restoring the original quantum coherence is challenging (if not impossible).

To explain the effects of decoherence mathematically, we must first talk about density matrices.

A density matrix often denoted as ρ, is a mathematical representation used in quantum mechanics to describe the statistical state of a quantum system, particularly in situations where the system may be in a mixed state. It is a Hermitian, positive semi-definite matrix (a symmetric matrix with non-negative eigenvalues) that encapsulates the information about the quantum state, including pure and mixed states. Pure states represent quantum systems in definite, well-defined states, while mixed states describe probabilistic combinations of different pure states.

A density matrix ρ can be expressed as:

$$\rho = \sum_i p_i |\Psi_i\rangle\langle\Psi_i| \tag{2.67}$$

Where:

- ρ is the density matrix.
- p_i represents the probability of the system being in state $|\Psi_i\rangle$.
- $|\Psi_i\rangle$ are the pure states that compose the mixed state.

The density matrix ρ is a Hermitian matrix ($\rho = \rho^\dagger$), which has only non-negative eigenvalues because it represents a valid physical state. Moreover, the trace of ρ is equal to 1 because it is normalized.

Density matrices are helpful when dealing with mixed states, where a quantum system is not in a pure state but in a statistical ensemble of pure states.

Going back to decoherence, let's consider a quantum system described by a density matrix ρ. The density matrix ρ represents the quantum system's state, which may initially be pure or mixed. The evolution of ρ is described by the Schrödinger equation, which describes the system's evolution when it is isolated from the environment.

However, it becomes entangled when the quantum system interacts with its environment. This interaction causes the density matrix to become non-unitary due to the entanglement between the system and the environment, leading to decoherence.

The evolution of a closed quantum system is unitary and described by $\rho' = U\rho U^\dagger$, where U is a unitary operator. However, open systems interact with their environment, leading to non-unitary evolution.

The Kraus operator formalism describes the evolution of an open quantum system. Given a system with density matrix ρ representing its initial state and interacting with an environment, its evolution is described by a positive trace-preserving map:

$$\rho' = \sum_k E_k \rho E_k^\dagger,$$

where $\{E_k\}$ are the Kraus operators, satisfying $\sum_k E_k^\dagger E_k = I$, ensuring trace preservation.

We can model decoherence (phase damping and amplitude damping) using Kraus operators as follows:

Amplitude Damping

Amplitude damping represents the loss of energy to the environment. The Kraus operators are:

$$E_0 = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{1-\gamma} \end{pmatrix}, \quad E_1 = \begin{pmatrix} 0 & \sqrt{\gamma} \\ 0 & 0 \end{pmatrix},$$

Where γ is the damping parameter.

Each Kraus operator E_k represents a potential error due to environmental interaction. The formalism does not require a specific environment model but fully describes the environment's effect on the system. It is crucial for designing quantum error-correcting codes and achieving noise-resilient quantum computing.

Coherence loss due to amplitude damping refers to the reduction in the off-diagonal elements of the density matrix, which represent quantum superpositions. Under amplitude damping, the off-diagonal elements decay, leading to a loss of coherence. This effect is due to the interaction with the environment, causing the system to lose information about the relative phases between the quantum states.

In addition to coherence loss, amplitude damping also causes loss of energy. This means the probability of the system being in the excited state ($|1\rangle$) decreases over time as energy is dissipated to the environment.

Phase Damping

Phase damping describes the loss of quantum coherence without energy loss. Its Kraus operators are:

$$E_0 = \sqrt{1-\lambda} I, \quad E_1 = \sqrt{\lambda} Z,$$

where I is the identity operator, Z is the Pauli-Z operator, and λ is the damping parameter.

The loss in coherence due to phase damping refers to the reduction in the off-diagonal elements of the density matrix. Unlike amplitude damping, phase damping does not involve energy loss but still leads to decoherence due to the disruption of the phase relationships between quantum states.

Partial Trace

Another important concept to consider is the partial trace. If a composite system is described by a density matrix representing the state of two or more subsystems, the partial trace over one of the subsystems gives us the state of the remaining system.

The partial trace of the subsystems is the average of their states, which gives information about the other subsystems.

Given a composite quantum system consisting of two subsystems A and B, described by the density matrix ρ_{AB} in the combined Hilbert space $\mathcal{H}_A \otimes \mathcal{H}_B$, the partial trace over subsystem B is denoted as $\mathrm{Tr}_B(\rho_{AB})$. This operation traces out (removes) the degrees of freedom associated with subsystem B, leaving the reduced density matrix ρ_A for subsystem A.

Formally, if ρ_{AB} is the density matrix of the composite system, the reduced density matrix ρ_A is given by:

$$\rho_A = \mathrm{Tr}_B(\rho_{AB}). \tag{2.68}$$

If $\{|i\rangle_B\}$ is an orthonormal basis for the Hilbert space $\mathcal{H}_B$, the partial trace over B is defined as:

$$\rho_A = \mathrm{Tr}_B(\rho_{AB}) = \sum_i \langle i|_B \rho_{AB} |i\rangle_B, \tag{2.69}$$

where $\langle i|_B$ and $|i\rangle_B$ denote the bra and ket vectors in the basis of $\mathcal{H}_B$, respectively.

The partial trace operation extracts the state of a subsystem from the state of the entire system. This is particularly useful when we are interested in the behavior of only a specific part of the system.

Tracing out one of the qubits from a Bell pair is a great way to demonstrate how a pure entangled state can produce a mixed state for one of its subsystems. Let's consider the Bell state, an EPR pair, and perform the partial trace operation to see the effect.

For example, consider the Bell pair:

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle). \tag{2.70}$$

The density matrix ρ_{AB} of $|\Phi^+\rangle$ is:

$$\rho_{AB} = |\Phi^+\rangle\langle\Phi^+| = \frac{1}{2}(|00\rangle + |11\rangle)(\langle 00| + \langle 11|). \tag{2.71}$$

Which we can expand to:

$$\rho_{AB} = \frac{1}{2}(|00\rangle\langle 00| + |00\rangle\langle 11| + |11\rangle\langle 00| + |11\rangle\langle 11|). \tag{2.72}$$

Now, assume we want to trace out $|0\rangle_B$:

$$\langle 0|_B \rho_{AB} |0\rangle_B = \frac{1}{2}(\langle 0|_B(|00\rangle\langle 00| + |00\rangle\langle 11| + |11\rangle\langle 00| + |11\rangle\langle 11|)|0\rangle_B). \tag{2.73}$$

Which leads to:

$$\langle 0|_B \rho_{AB} |0\rangle_B = \frac{1}{2}(|0\rangle_A\langle 0|_A). \tag{2.74}$$

Similarly, if we want to trace out $|1\rangle_B$

$$\langle 1|_B \rho_{AB} |1\rangle_B = \frac{1}{2}(\langle 1|_B(|00\rangle\langle 00| + |00\rangle\langle 11| + |11\rangle\langle 00| + |11\rangle\langle 11|)|1\rangle_B). \tag{2.75}$$

We get:

$$\langle 1|_B \rho_{AB} |1\rangle_B = \frac{1}{2}(|1\rangle_A\langle 1|_A). \tag{2.76}$$

Summing the contributions from tracing out $|0\rangle_B$ and $|1\rangle_B$:

$$\rho_A = \frac{1}{2}(|0\rangle_A\langle 0|_A) + \frac{1}{2}(|1\rangle_A\langle 1|_A) = \frac{1}{2}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{2.77}$$

This is the reduced density matrix for subsystem A, which is a mixed state. Which shows that tracing out one qubit of a pure entangled state results in a mixed state for the remaining subsystem.

The diagonal elements of ρ_A give the probabilities of finding subsystem An in its respective states. The off-diagonal elements represent the coherences (quantum correlations) between these states.

It is important to differentiate between superposition and mixed states. Unlike a superposition, a mixed state is a statistical mixture of different states. It represents a situation where the system is in one of several possible states, but we do not know which one. In a mixed state, probabilities describe the likelihood of each state. Experimentally, distinguishing between an equal superposition involves measuring the qubit in different bases and analyzing the results.

For example, consider the density matrix of the pure state $|\psi\rangle$ in an equal superposi-

tion of $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle). \tag{2.78}$$

$$\rho_{\text{superposition}} = |\psi\rangle\langle\psi| = \frac{1}{2}\begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}. \tag{2.79}$$

A maximal mixture is represented by the density matrix:

$$\rho_{\text{mixture}} = \frac{1}{2}I = \frac{1}{2}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{2.80}$$

To distinguish between those two states, we will perform measurements in different bases:

1. Measurement in the Computational Basis: - For state $|\psi\rangle$, measuring in the computational basis will give $|0\rangle$ and $|1\rangle$ with equal probabilities (0.5 each). - For the maximal mixture, measuring in the computational basis will also give $|0\rangle$ and $|1\rangle$ with equal probabilities (0.5 each).

2. Measurement in the Superposition Basis $|+, -\rangle$: - For the equal superposition state, measuring in the $|+, -\rangle$ basis will yield:

$$P(|+\rangle) = |\langle +|\psi\rangle|^2 = \left|\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}\right)\right|^2 = 1, \tag{2.81}$$

$$P(|-\rangle) = |\langle -|\psi\rangle|^2 = \left|\frac{1}{\sqrt{2}}\left(\frac{1}{\sqrt{2}} - \frac{1}{\sqrt{2}}\right)\right|^2 = 0. \tag{2.82}$$

- For the maximal mixture, measuring in the $|+, -\rangle$ basis will give: We use the formula:

$$P(|\phi\rangle) = \langle\phi|\rho|\phi\rangle. \tag{2.83}$$

Probability of Measuring $|+\rangle$

First, we calculate $\langle +|\rho_{\text{mixture}}|+\rangle$:

$$\langle +| = \left(\frac{1}{\sqrt{2}}(\langle 0| + \langle 1|)\right) \tag{2.84}$$

$$|+\rangle = \left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right) \tag{2.85}$$

Now,

$$\langle +|\rho_{\text{mixture}}|+\rangle = \left(\frac{1}{\sqrt{2}}(\langle 0| + \langle 1|)\right)\frac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}\left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right) \tag{2.86}$$

Simplifying this:

$$\langle +|\rho_{\text{mixture}} = \left(\frac{1}{\sqrt{2}}(\langle 0| + \langle 1|)\right)\frac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} = \frac{1}{2\sqrt{2}}(\langle 0| + \langle 1|) \tag{2.87}$$

$$\left(\frac{1}{2\sqrt{2}}(\langle 0| + \langle 1|)\right)\left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right) = \frac{1}{4}\left(\langle 0| + \langle 1|\right)\left(|0\rangle + |1\rangle\right) \tag{2.88}$$

$$\langle 0|0\rangle = 1, \quad \langle 1|1\rangle = 1, \quad \langle 0|1\rangle = 0, \quad \langle 1|0\rangle = 0 \tag{2.89}$$

$$= \frac{1}{4}(1 + 0 + 0 + 1) = \frac{1}{4}\cdot 2 = \frac{1}{2} \tag{2.90}$$

So,

$$P(|+\rangle) = \langle +|\rho_{\text{mixture}}|+\rangle = \frac{1}{2} \tag{2.91}$$

Probability of Measuring $|-\rangle$

Similarly, we calculate $\langle -|\rho_{\text{mixture}}|-\rangle$:

$$\langle -| = \left(\frac{1}{\sqrt{2}}(\langle 0| - \langle 1|)\right) \tag{2.92}$$

$$|-\rangle = \left(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right) \tag{2.93}$$

$$\langle -|\rho_{\text{mixture}}|-\rangle = \left(\frac{1}{\sqrt{2}}(\langle 0| - \langle 1|)\right)\frac{1}{2}\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}\left(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right) \tag{2.94}$$

Which leads to:

$$P(|-\rangle) = \langle -|\rho_{\text{mixture}}|-\rangle = \frac{1}{2} \tag{2.95}$$

We can experimentally distinguish an equal superposition from a maximal mixture by performing these measurements and analyzing the outcomes.

Mixed states reflect a lack of knowledge about which pure state the system is in. They often result from decoherence, where interaction with the environment causes a loss of quantum coherence. Mixed states are also described by density matrices, which are a weighted sum of the outer products of the state vectors with their complex conjugates.

This contrasts with a pure state's density matrix, which is simply the outer product of its state vector with itself.

2.6 Current Status of Quantum Hardware

The current era of quantum computers is called the NISQ (Noisy Intermediate-Scale Quantum) era [98]. NISQ devices are called "intermediate-scale" because they have a relatively small number of qubits (typically a few dozen to a few hundred) compared to the larger-scale quantum computers that we aim to build in the future with the ability to use millions of qubits and run real-life applications. They are also called "noisy" because the qubits in these devices are prone to errors and noise, affecting the accuracy of their computations. Though NISQ devices are faulty and cannot run general-purpose applications, they provide a stepping stone in developing quantum computing. Moreover, they have considerable importance because they allow us to:

- Develop and refine quantum algorithms on real hardware, providing insights into their performance and potential improvements. Some algorithms are designed to be less sensitive to errors and noise, making them more suitable for NISQ devices.
- With NISQ devices, researchers can study various error mitigation strategies to minimize the impact of noise on the computation results. These techniques can help improve the performance and reliability of NISQ devices, even without full-fledged quantum error correction.
- They offer a testbed for researchers to explore new quantum computing applications across various domains, such as chemistry, optimization, and machine learning.
- By building and operating NISQ devices, researchers can gain insights into the challenges and potential improvements in qubit technology, control systems, and hardware architectures.

2.6.1 Performance Metrics for Quantum Computers

IBM has introduced several new metrics and benchmarks to measure the performance of quantum computers. Some of these metrics include:

Quantum Volume

IBM has proposed a single number indicator to describe the quantum processing capabilities of any NISQ device. IBM introduced the concept of Quantum Volume (QV) in [99] and it is defined as in Definition 1.

Definition 1. *Quantum volume (QV) is the ability to run a square circuit with at least a 2/3 success probability. This success depends on the number of qubits n and the depth of the circuit d [100]. they also laid out a prediction for the future of their quantum devices Their proposed roadmap for the advancement of quantum processor power aims to double the performance every year in order to achieve Quantum Advantage in the near future [101]. Quantum volume is a machine metric that by definition uses "square" circuits; an nqubit computer capable of running a depth d circuit is said to have:*

$$QV \approx 2^{\sqrt{nd}} \tag{2.96}$$

CLOPS (Circuit Layer Operations per Second)

CLOPS measures the speed at which a quantum computer can execute quantum circuits [102]. It takes into account the classical computation needed to manage the quantum operations and the latency of the quantum operations themselves. Higher CLOPS values indicate a faster system capable of executing more quantum operations in a given time frame.

Quantum Processor Performance (QPP)

QPP is a comprehensive metric that combines multiple performance factors, including gate fidelity, qubit coherence times, and the overall reliability of the quantum processor [103]. This metric aims to provide a holistic view of the processor's capabilities.

These new metrics and benchmarks help provide a more comprehensive understanding of a quantum computer's capabilities, addressing various aspects of performance.

2.6.2 Common Errors In Quantum Hardware

Decoherence is particularly detrimental to quantum computing because it directly undermines the core principles of quantum mechanics that quantum computers rely on—superposition and entanglement. When a quantum system decoheres, the superposition states collapse into a mixture of classical states, and the entanglement between

qubits is lost. This transition from a coherent quantum state to an incoherent classical state reduces the computational power of the quantum system.

Quantum computing devices are prone to various types of errors due to their interaction with the environment, imperfections in the hardware, and the limitations of control mechanisms. These errors can significantly affect the performance and reliability of quantum computations. In this section, we will describe some of the most prevalent errors encountered in quantum hardware and how they relate to the phenomenon of decoherence.

Thermal-relaxation Error

Thermal relaxation needs two main parameters defined, T_1 and T_2, together called decoherence times. T_1 is known as the relaxation time constant; it is defined as the time needed for the system to go from state $|1\rangle$ to $|0\rangle$ with probability $\frac{1}{e}$. For example, the probability of state $|\psi\rangle$ remaining in its state for some time t is given by $P(|\psi\rangle) = e^{-\frac{t}{T_1}}$.

The time constant T_2 represents the timescale over which the phase coherence of a qubit decays. Specifically, it is the time it takes for the off-diagonal elements of the qubit's density matrix (which represent the coherence terms) to decay to $1/e$ of their initial value [104].

In order for the values of T_1 and T_2 to be valid, they have to satisfy the relation $T_2 \leq 2T_1$. Since both T_2 and T_1 are, in a sense, measures of qubit stability, larger values mean a more stable qubit. The qubit will maintain its state without decaying for a longer time, hence making it easy to measure and use in complex algorithms.

Gate Error

Gate errors occur due to imperfections in the execution of quantum gates, which are fundamental operations that alter the states of qubits. These errors may be caused by:

1. Imprecise control signals that fail to accurately implement the intended quantum gates.
2. Unintended interactions between qubits that lead to erroneous gate operations.
3. Environmental disturbances such as thermal fluctuations or electromagnetic fields.

Readout Error

Readout errors occur during the measurement phase of quantum computing, where the state of a qubit is incorrectly determined. Causes include:

1. Measurement apparatus inaccuracies or failure to accurately capture the state due to quantum superposition.
2. Environmental noise affecting the qubits at the measurement stage.

Crosstalk Error

Crosstalk errors describe unwanted interactions between qubits, typically in densely packed quantum systems, leading to erroneous operations. This error can occur due to:

1. Electromagnetic interactions or quantum information leakage between physically proximate qubits.

2. Increased complexity and qubit density in quantum circuits heightening interaction risks.

Minimizing crosstalk involves careful layout and timing adjustments in quantum circuit design.

In summary, understanding and mitigating the various types of errors, especially decoherence, is crucial for the development of robust and reliable quantum computing devices. Techniques such as error correction, qubit isolation, and advanced control mechanisms are essential to preserve quantum coherence and ensure accurate quantum computations.

2.7 Current Status of Quantum Software

The forty-year history of quantum computers has taken us through initial curiosity, naive optimism, then dismay at the scale of proposed error-corrected systems, and into today's excitement over the availability of real, but still small and error-prone, systems [1, 2, 3]. Algorithms have followed a similar roller coaster, arriving at the point where modest demonstration implementations of algorithms originally defined as abstract equations in theory papers are now common [4]. The challenge on both hardware and software now is scalability: more qubits and larger, more sophisticated programs where, unlike today's demonstrations, the results are not *a priori* known. Working at a large scale implies the need for a mature software engineering (SE) approach, including tools for all phases of the life cycle.

Software engineering follows a specific cycle and is a reasonably mature process. Two essential phases are designing and developing the key conceptual elements and testing and fixing bugs Figure 2.8). The conceptual elements may be supported with formal specifications, pseudocode, libraries, modeling tools, languages, etc. Bugs arise from mistakes in the specification of a program or in translating the specification into code (or, sometimes, from bugs in the tools themselves). A variety of methods, both formal and informal, are used to find such bugs and to prevent their recurrence once isolated and fixed. Unit testing, regression testing and continuous integration, path coverage testing, and the many types of test cases that software testers construct all contribute to locating and eliminating the different types of bugs. Using these techniques and tools, it is now possible to build and support systems as complex as tens of millions of lines of code, as in the Linux kernel and other similar systems.

Considering the quantum software development cycle described in [105] as shown in Figure 2.8, we can find general similarities to the classical software development

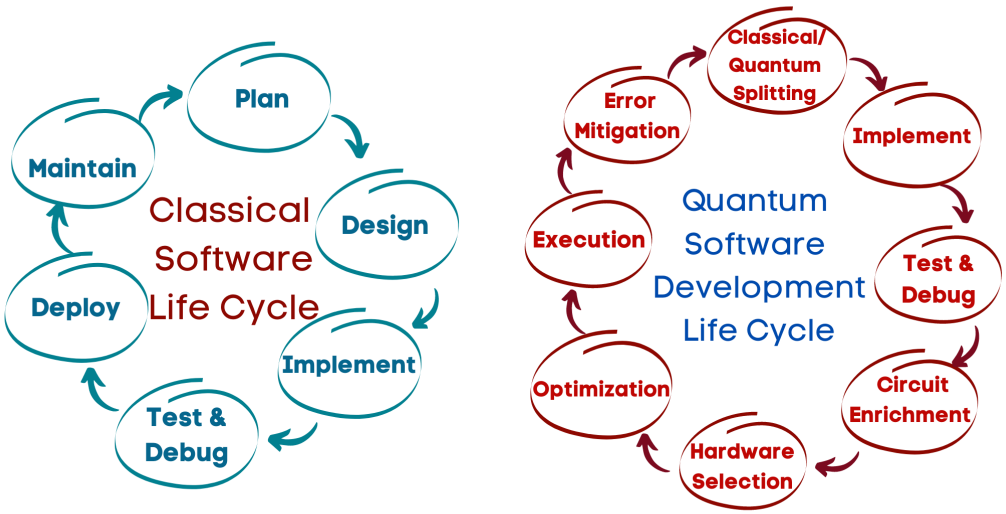


Figure 2.8: A general life cycle of classical software vs quantum software as described in [105].

cycle. There are two significant differences in debugging quantum programs. Quantum computers can operate on a *superposition* of states, each with a *complex amplitude* [106, 107]. The exponential growth in the state space poses a fundamental problem for debugging and testing quantum programs. In classical computers n bits can carry any of 2^n values, however, we do not worry about testing all 2^n input values for a program, let alone all of the astronomical numbers of possible states, when we include temporary variables. After all, a system with 1GB of RAM has $2^{8,589,934,592}$ possible states, almost all of which will never be reached. Instead, we focus on key paths and expected input values and work to build robust error handling for the vast majority of unwanted states. With quantum computers, though, the process is different, and we often have to consider the behavior of all possible inputs.

Today, there are different approaches developers could use to transform their ideas and algorithms into quantum programs [108, 109]. If they can develop and implement their algorithms efficiently, leading to small/medium-scale circuits, they can try executing them on actual quantum hardware. Otherwise, they can implement a smaller version of the algorithm and try to extrapolate its behavior for larger instances of the problem. The currently available approaches differ based on the core programming model into four categories.

- High-level quantum programming language supporting the developer's quantum intuition such as Silq [110] and Quipper [111].

- Gate-level programming. In this option, the developer translates his idea into a sequence of gates and then simulates this circuit, visualizes it, or runs it on a hardware device. Developers can use this approach in different ways:
  - Building the circuit using code, often using a classical-language-supported library or package, such as: Qiskit [112], Cirq [113], and PyQuil [114] which are Python Packages.
  - Using a drag-and-drop tool to build the circuit, simulate the results, and view them visually. These tools include, QUI [115], the IBM Circuit Composer [116], and Quirk [117].
  - Using the Quantum Assembly language, or QASM [118].
- Building the circuit using other compilation paradigms.
  - Using a low-level approach, for example, using pulses and signals to control the quantum hardware directly, the main example is OpenPulse [112].
  - Using a more quantum physics and mechanics approach like using the ZX-calculus [119].
  - Circuit optimizer, back-end compilers, and interpreters. For example Tket [120], TriQ [121], and Qbsolv [122].

A summary of the most used quantum software tools can be seen in Table 2.3. All of these tools mainly focus on the current generation of hardware, small programs, and the important problems of optimization and mapping to specific processors [98, 123, 124], as well as on designing and implementing programs for hybrid or adaptive algorithms [125, 126, 127, 128, 129].

The difference between these tools is not significant, at least, syntactically. Figure 2.9 shows different approaches to apply a Hadamard gate to the state $|0\rangle$.

All of these tools leave it to the developer to mentally and manually plan the algorithm and its implementation and examine its outputs. However, we need to start building tools for the future of quantum because as we move toward large scale, including a quantum debugger and tools for automated program testing such as unit tests. Both of these depend on the ability to isolate a portion of a quantum program and examine and understand its inner functionality. More specifically, developers need to understand the interactions between the different elements of the program, prepare input vectors, and check their corresponding outputs without paying exponential costs in state or time spaces. This, however, will not be possible as the circuit size increases, the current hardware fails to run them error-free, and classical devices fail to simulate them. As quantum debugging becomes a focus and an essential skill for the current and next generations of quantum circuits and developers, the value of a tool that enables understanding the circuit and the error reasons will only increase. Solving the challenge of debugging quantum circuits is

SILQ

```
//in Silq we don't need to
initialize the circuit, and qubits
are treated as variables
x:=H(x);
y:=H(y);
```

QUIPPER

```
"Create 2 qubits circuit"
ourCir :: Circ (Qubit, Qubit)
"apply hadamard to qubit 1,2"
"a,b are our qubits"
a,b = do
a <- hadamard a
b <- hadamard b
```

Q#

```
//Create a circuit and apply H
gate to both qubits
operation setCir(q: Qubit[2]) :
results{
  H(q[0]);
  H(q[1]);}
```

QX SIMULATOR

```
#Create two qubits
qubits 2
#qubits in QX are numbered by
default from 0 to number of
qubits-1
#Apply h gate to two qubits
h q0
h q1
```

QISKIT

```
from qiskit import
QuantumCircuit
#Create a quantum circuit with
2 qubits
qcir = QuantumCircuit(2)
#apply hadamard to qubit 1
qcir.h(0)
#apply hadamard to qubit 2
qcir.h(1)
```

CIRQ

```
import cirq
#Create 2 qubits
q0, q1 = cirq.LineQubit.range(2)
#apply hadamard to qubit 1
h0 = cirq.H(q0)
#apply hadamard to qubit 2
h1 = cirq.H(q1)
```

SCAFFOLD

```
int main () {
//Create 2 qubits
qbit reg[2];
  for ( i = 0; i < 2; i++ ) {
// apply hadamard to qubit 1,2
⊠H(reg[i]); }
return 0;}
```

PYQUIL

```
from pyquil import
get_qc, Program
from pyquil.gates import H
#Create 2 qubits circuit
qvm = get_qc('2q-qvm')
p = Program()
#apply hadamard to qubit 1,2
p += H(0)
p += H(1)
```

STRANGE

```
import com.gluonhq.strange.*;
import
com.gluonhq.strange.gate.*;
public static void main(String[]
args) {
    Program p = new Program(2);
Gate hGate0 = new Hadamard(0);
Gate hGate1 = new Hadamard(1);
    Step step1 = new Step()
step1.addGates(hGate0, hGate1);
    p.addStep(step1); }
```

INTEL SDK

```
#include <clang/Quantum/
quintrinsics.h>
const int total_qubits = 2;
//create a 2-qubit circuit
qbit qubit_register[total_qubits];
//apply hadamard to qubit 1,2
H(qubit_register[0]);
H(qubit_register[1]);
```

Figure 2.9: The code creating the $|+\rangle$ state using different platforms.

Table 2.3: A summary of some widely used current quantum software tools.

Name	Level	Year Released	Developer	Available For
QuTIP [130]	Gate-level	2012	J. R. Johansson, P. D. Nation, and F. Nori	Python Package
Quipper [111]	High-level	2013	Dalhousie University	Quantum PL
Quirk [117]	Drag and Drop	2016	Craig Gidney	JavaScript Package
Qiskit [112]	Gate-level	2017	IBM	Python Package
Q# [131]	Gate-level	2017	Microsoft	Quantum PL
QX Simulator [132]	QASM-based	2017	QuTech	QASM
Ocean SDK [133]	Gate-level	2018	D:Wave	Python Package
Cirq [113]	Gate-level	2018	Google (not an official product)	Python Package
Pyquil [114]	Gate-level	2018	Rigetti	Python Package
QUI [115]	Drag and Drop	2018	Hollenberg Group at the University of Melbourne	Quantum PL
PennyLane [134]	Gate-level	2018	Xanadu	Python Package
Q.js [135]	Drag and Drop	2020	Stewart Smith	JavaScript Package
Silq [110]	High-level	2020	ETH Zurich	Quantum PL
Amazon-Braket [136]	Gate-level	2020	Amazon	Python Package
TKET [120]	Gate-level	2021	Quantinuum	Python Package
Intel SDK [137]	Gate-level	2023	Intel	C++ Library

not going to be a simple task, but it is a challenge that we need to address and attempt to solve as best we can, which is the target of this thesis.

Chapter 3

How Are Quantum Algorithms Implemented?

3.1 Introduction

An important aspect of developing tools for debugging and testing quantum algorithms is a solid understanding of the inner workings of quantum algorithms.

Quantum algorithms leverage the principles of quantum mechanics to perform computations that would be infeasible or highly time-consuming to classical computers. One of the well-known quantum algorithms is Shor's algorithm [11], developed by mathematician Peter Shor in 1994.

Shor's algorithm poses a significant threat to traditional cryptographic systems, such as RSA and ECC, which are foundational to many authentication mechanisms. By efficiently factorizing large integers and solving discrete logarithms, Shor's algorithm can potentially break these encryption methods, undermining the security of digital signatures, key exchange protocols, and other authentication processes. This vulnerability necessitates a transition to quantum-safe or post-quantum cryptographic algorithms that can withstand quantum attacks.

To mitigate the risks posed by quantum computing, organizations must adopt post-quantum algorithms and consider hybrid cryptographic solutions that combine classical and quantum-resistant techniques. Continuous monitoring of advancements in quantum computing and post-quantum cryptography, along with education and training of cybersecurity professionals, is essential for maintaining secure authentication systems. Implementing quantum key distribution (QKD) can also enhance security by providing a method for secure key exchange that is immune to quantum attacks, ensuring robust

authentication in the quantum era.

Another important quantum algorithm is Grover's algorithm [41], formulated by Lov Grover in 1996, designed to search unsorted databases and solve black-box computational problems. It provides a quadratic speedup over classical algorithms for these types of tasks.

Quantum algorithms are also being developed for simulations of quantum systems, a task inherently suited to quantum computers.

Algorithms such as the Variational Quantum Eigensolver (VQE) and Quantum Phase Estimation algorithms are central in this domain and offer promising avenues for advancements in chemistry and material science by enabling the simulation of molecular structures and reactions with high precision. As research in the field progresses, we are likely to see the emergence of new quantum algorithms that harness the unique properties of quantum mechanics to solve a broader array of complex problems, potentially revolutionizing computing as we know it.

Understanding how quantum algorithms work and how we move them from theory to implementation is essential to developing efficient testing and debugging tools. In this chapter, we review one of the popular algorithms in quantum computing (Grover's Algorithm) and see how we can implement it from theory to implementation, even analyzing the hardware performance.

3.2 Grover's Algorithm

Grover's Search Algorithm answers the question "Given a function $f(x)$, what values of x cause $f(x)$ to evaluate to **True**?". The algorithm presents a framework for tackling the search problem in an unsorted database with complexity $O(\sqrt{N})$, where N is the number of possible inputs. It mainly consists of three sections, state preparation, the oracle, and the diffusion operator. The oracle and diffusion operators will be repeated depending on N and m, where m is the number of answers.

The algorithmic steps of Grover's search are (Figure 5.1):

1. Prepare the input in a symmetric-superposition state.
2. Apply Grover's Oracle to the prepared state.
3. Apply the diffusion operator to the oracle's results.
4. Iterate over steps 2 and 3 until the answer is reached.

The first step of Grover's algorithm is preparing the initial state. In the simplest version, the initial state is prepared in an equal superposition over the entire Hilbert space.

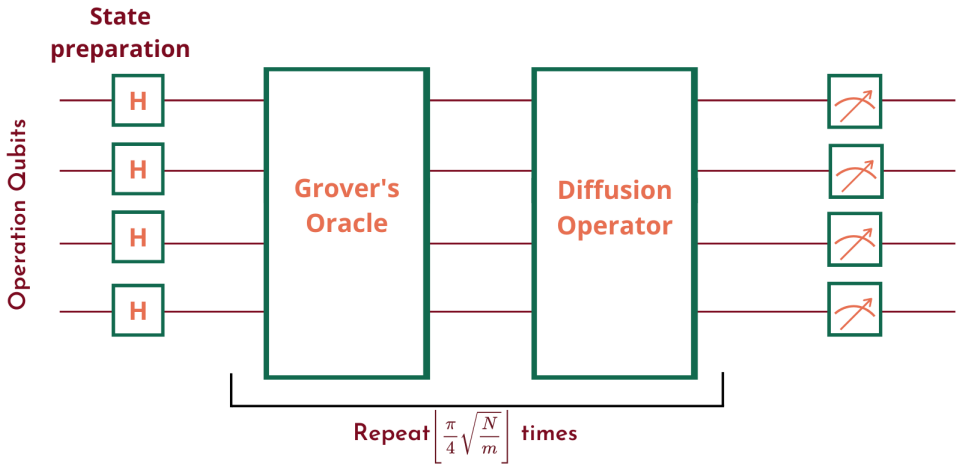


Figure 3.1: An overview of Grover's algorithm's steps.

After the state preparation comes to the oracle. The oracle is a black box function that inverts the answer by flipping its sign. Following that, the diffusion operator will magnify the amplitude of the correct state while damping the amplitude of other states until the amplitude of the answer is significantly larger than the rest of the states.

Grover's oracle is a special function that is problem-based. Each problem will require a different oracle to mark the correct answer/s.

```
import qiskit as qk
from qiskit.visualization import plot_histogram

def grover_oracle(qc, qubits):
    # Oracle for |111>
    # Apply multi-controlled Z gate
    qc.h(qubits[-1])
    qc.mct(qubits[:-1], qubits[-1])  # Multi-controlled Toffoli gate
    qc.h(qubits[-1])

def grover_diffuser(qc, qubits):
    # Apply H-gates
    for qubit in qubits:
        qc.h(qubit)
```

```
    # Apply X-gates
    for qubit in qubits:
        qc.x(qubit)
    # Apply multi-controlled Z gate
    qc.h(qubits[-1])
    qc.mct(qubits[:-1], qubits[-1])
    qc.h(qubits[-1])
    # Apply X-gates
    for qubit in qubits:
        qc.x(qubit)
    # Apply H-gates
    for qubit in qubits:
        qc.h(qubit)

# Number of qubits
n = 3
qubits = qk.QuantumRegister(n)
cbits = qk.ClassicalRegister(n)
qc = qk.QuantumCircuit(qubits, cbits)

# Apply Hadamard gates to initialize the superposition
for qubit in qubits:
    qc.h(qubit)

# Number of iterations
iterations = 2  # Adjusted to 2 for 3 qubits

# Apply Grover's algorithm
for _ in range(iterations):
    grover_oracle(qc, qubits)
    grover_diffuser(qc, qubits)

# Measurement
```

```
48 qc.measure(qubits, cbits)
49
50 # Execute the circuit
51 backend = qk.Aer.get_backend('qasm_simulator')
52 job = qk.execute(qc, backend, shots=1024)
53 result = job.result()
54
55 # Plot the results
56 counts = result.get_counts(qc)
57 plot_histogram(counts)
```

Listing 3.1: The output of the gateLoc function when querying the NOT gate in the diffusion operator

A simple example of Grover's algorithm assumes that we are looking for a state $|111\rangle$ in a 3-qubit system. We can implement that system using Qiskit as shown in Listing 3.1. When we run that code and display the histogram of the results, we can see that the probability of measuring state $|111\rangle$ is 1 (ideally) (Figure 3.2).

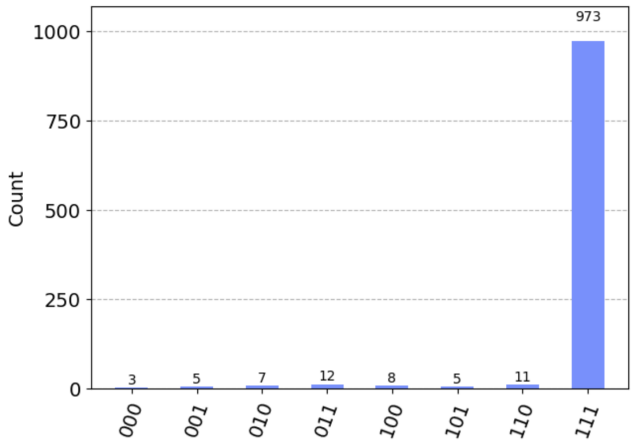


Figure 3.2: A simple implementation of Grover's algorithm with an oracle marking $|111\rangle$

It is important here to be able to distinguish between marking the correct answer and knowing the correct answer. Marking the answer means that the oracle modifies the phase of the correct state, but it does not reveal which state is the correct one. The algorithm doesn't gain classical knowledge about the correct item at this stage. Instead, it subtly alters the quantum state, setting up for further processing. Knowing the answer,

on the other hand, would imply that the algorithm explicitly identifies the correct item. In classical terms, it would mean pinpointing the exact position of the item in the database.

Grover's algorithm, the process involves repeatedly applying the oracle and a diffusion operator to amplify the probability amplitude of the correct state.

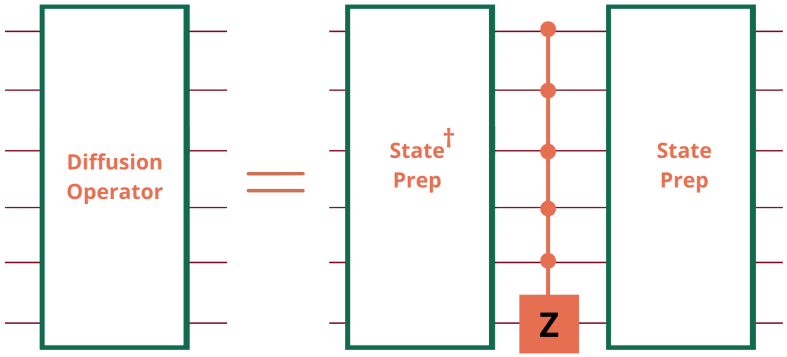


Figure 3.3: A general implementation of the diffusion operator.

The diffusion operator is formed by: the inverse of state preparation, $C^{\otimes n}Z$ gate, state preparation, as shown in Figure 5.5. The $C^{\otimes n}Z$ gate cost is $2n-1$ gates, divided into 1 CZ gate and $2n-2$ CCX gates when $n > 2$. When $n = 2$, however, we only need 2 Hadamard gates and a CZ gate to form a CCZ. Starting from $n = 3$ to form the $C^{\otimes n}Z$ gate, we will need some ancillary qubits. A construction of the CCZ, $C^{\otimes 3}Z$ and $C^{\otimes 4}Z$ can be seen in Figure 3.4.

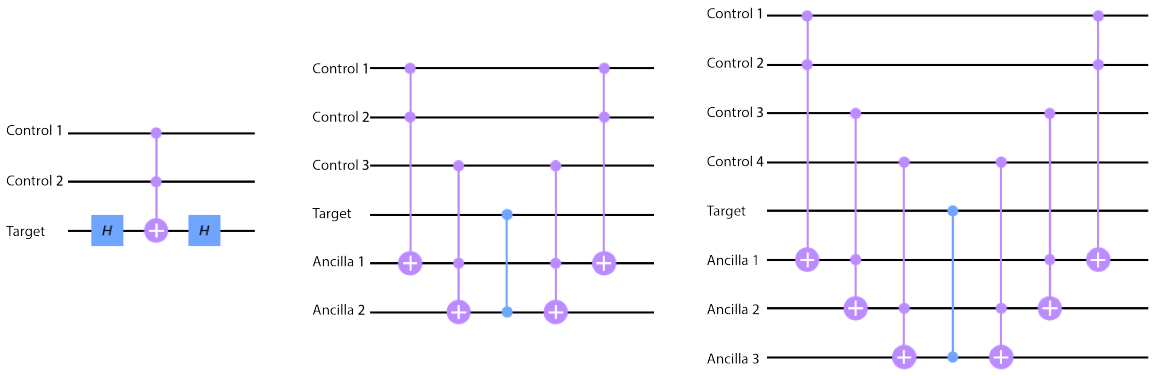


Figure 3.4: Different number of controlled Z gates. From the left, CCZ, $C^{\otimes 3}Z$ and $C^{\otimes 4}Z$

The answer's amplitude will grow to a maximum and then decline after the optimal number of iterations `opt_iter` repeating cyclically. Therefore, we need to measure the answer at the right time, leading to the first high amplitude of the answer. The

optimal number of repeating the oracle and the diffusion depends on two factors, the size of the search space N and the number of answers for our search query m as seen in Equation 3.1 [138].

$$opt\_iter = \left\lfloor \frac{\pi}{4}\sqrt{\frac{N}{m}} \right\rfloor \tag{3.1}$$

3.3 The Clique Finding Problem

A *clique* is a complete subgraph over a subset of vertices in an undirected graph. Several computational problems address finding cliques in a given graph. These problems vary based on what information about the clique needs to be found. One such is the k-clique problem, which answers the question, "Given an undirected graph and a positive integer k, does a clique with size k exist?" The k-clique problem is NP-Complete for large values of k, as shown by Karp [139] and Cook [140]. Probably one of the most studied versions of the k-clique problem is the triangle finding problem (the 3-clique problem), which has been addressed both classically [141, 142] and quantumly [143, 144]. The best-known classical algorithm has time complexity $O(n^{2.38})$, while the most famous quantum algorithm has time complexity $O(n^{1.5})$, where n is the number of nodes in the graph. Several quantum algorithms have also been proposed for the k-clique problem with $k > 3$ [144, 145, 146].

Given an undirected graph (G), if there exists a subset of k vertices that are connected to form a complete graph, then it is said that G contains a k-clique — for example, Figure 3.5 represents a graph of 6 vertices, which includes a 4-clique between vertices 1, 2, 3, and 4.

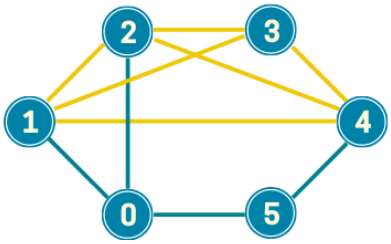


Figure 3.5: 6-node graph with 4-clique on nodes 1, 2, 3, and 4. The output of Grover's Algorithm will be $|011110\rangle$, with 1 for every node in the clique and 0 otherwise.

The k-clique problem asks us to determine if the input graph G contains a k-clique, and if it does, output the vertices forming the clique [147]. A popular variant of this problem only asks us to determine if G contains a k-clique [148].

Clique-finding algorithms have many practical applications. One of the main fields they can be used in is chemistry, to find chemicals matching a specific structure [149], to

model molecular docking, and to find the binding sites of chemical reactions [150]. They can also be applied to find similar structures within different molecules [151]. Another field for the clique-finding algorithms is automatic test pattern generation. Finding cliques helps to confine the size of the test sets [152]. The clique-finding problems are also used for Proof-of-Work (PoW) in cryptocurrencies [153].

3.4 Using Grover to Solve the Clique Finding Problem

The efficient execution of Grover is a two-fold problem: reducing the number of iterations and finding a practical implementation of each iteration. In this section, we present two approaches to implementing the oracle circuit; we will call them the checking oracle and the incremental oracle, respectively. The remainder of this section will discuss both implementations in detail, starting with the checking oracle. Although either implementation can be used to find any k-clique in any given undirected graph, while explaining how both implementations work, we will consider the simplest case possible, which is when $k = 3$ or a triangle. In all explanations, the graph in Figure 3.6 will be used.

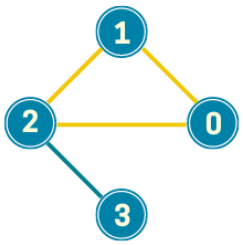


Figure 3.6: 4-node graph containing a triangle (3-clique) on nodes 0, 1 and 2. This graph is used in Tables 3.1,3.3

3.4.1 State Preparation

State preparation is the first step of the implementation. Usually, when implementing Grover's algorithm, the states are prepared in an equal superposition of the whole Hilbert space using the Hadamard Gate (H gate). Initializing into full superposition needs only n H gates and time complexity $O(1)$ since all H gates can be run simultaneously. This state preparation leads to a search of the entire Hilbert space, which is not always necessary.

If we wish to search for a 3-clique, it makes no sense to look for a subgraph with one, two, or even four nodes. Instead, we should consider only subgraphs with 3 nodes and then assess whether the induced subgraph contains $\binom{3}{2}$ edges (the number of edges

in a complete graph of 3 vertices) – three edges in case of a triangle. Searching over a limited space should be faster. However, it will cause a significant increase in the state preparation gate count.

For this example, we will use two approaches to limit the search space, using Dicke states, or W state followed by X gates.

Dicke State

A Dicke state $|D^n_k\rangle$ [154] is a fully symmetric entangled state over the n-qubit Hilbert space with Hamming weight k. For example, given a Hilbert space of 4 qubits, the Dicke state $|D^4_3\rangle$ will be the superposition of $\frac{1}{2}$ ($|1110\rangle + |1101\rangle + |1011\rangle + |0111\rangle$).

The number of basis states with k Hamming weight in a Hilbert space of n qubits is $\binom{n}{k}$.

Definition 2. *Dicke state $|D^n_k\rangle$ is an entangled superposition of all n-states $|s\rangle$ with Hamming weight k:*

$$\left|D^n_k\right\rangle = \left(\begin{array}{c} n \\ k \end{array}\right)^{-\frac{1}{2}} \sum_{s\in\{0,1\}^n s.t. hw(s)=k} |s\rangle. \tag{3.2}$$

Dicke states represent an essential class of entangled quantum states for their applications in quantum game theory [155], quantum networking [156] and quantum metrology [157]. Dicke states can be implemented in several different ways; we followed the approach proposed in [158] to prepare our Dicke states deterministically. The proposed method computes the Dicke state for any Hamming weight k and n qubits with $O(kn)$ gates and $O(n)$ depth [158].

W State

W states are a class of entangled quantum states that are a special case of the Dicke State. W state followed by a simple bit flip is a Dicke state with Hamming weight 1, such as $|W\rangle = \frac{1}{\sqrt{n}}(|100...0\rangle + ... + |01...0\rangle + |00...01\rangle)$. The implementation of the W-state preparation we used in this work is the algorithm proposed in [159].

W states as a state preparation approach works only for clique size $k = n - 1$; otherwise, W states cannot be used, and Dicke states have to be used instead. When using Dicke state, we can search for any size clique in any size graph. For example if we have a 10 node graph and we want to look for 4-cliques in it, we can limit the search space to $\binom{10}{4}$.

Figure 3.7 shows the change in the search space size for different clique sizes and approaches as the number of nodes grows. In the figure, the x-axis represents the number of nodes in the graph, and the y-axis represents the size of the search space. The worst-case search space for subsets of n nodes is 2^n (upper dotted line). For fixed-size cliques, simple search methods are polynomial (lower dashed lines, $k = 3$ and $k = 5$). When the clique size is a function of n, the search space is superpolynomial, and classical search becomes impractical. Constrained-Hamming-weight quantum searches using Dicke states extend the range of problems that can be addressed using Grover's algorithm (solid lines, $k = n/4$ and $k = n/2$).

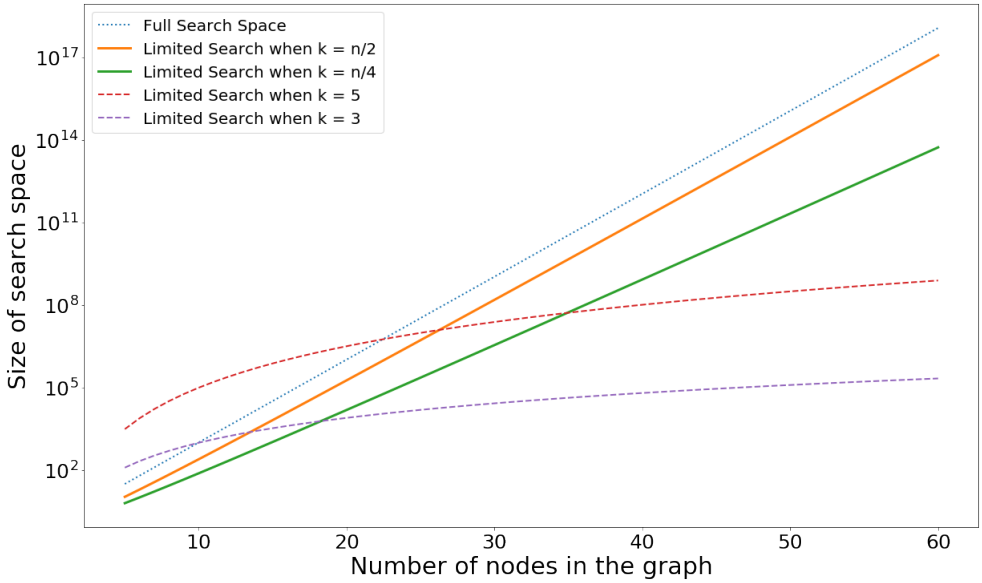


Figure 3.7: The difference in search space concerning the number of nodes in the graph. A limited search space (the two solid lines) is produced using state preparation (Dicke/W states) in the cases $k = n/2$ and $k = n/4$.

3.4.2 The Oracle

To determine that a triangle exists, we must confirm that 3 nodes are connected with 3 edges. This is exactly what both the oracle implementations do.

Checking-based Oracle

In the checking-based oracle, each node in the graph is represented as a qubit, and the edges between them are expressed using one or more multiple-Toffoli $C^{\otimes n}NOT$ gates

connecting specific qubits. After all edges have been counted, the results are checked. The sequence of $C^{\otimes n}NOT$ gates forms a simple adder that adds one every time an edge is encountered. In the case of a triangle, after the $C^{\otimes n}NOT$ gates, we need to check that we have precisely three edges (11_2). To check for 11_2, we need two qubits that we will call `edges_counter`.

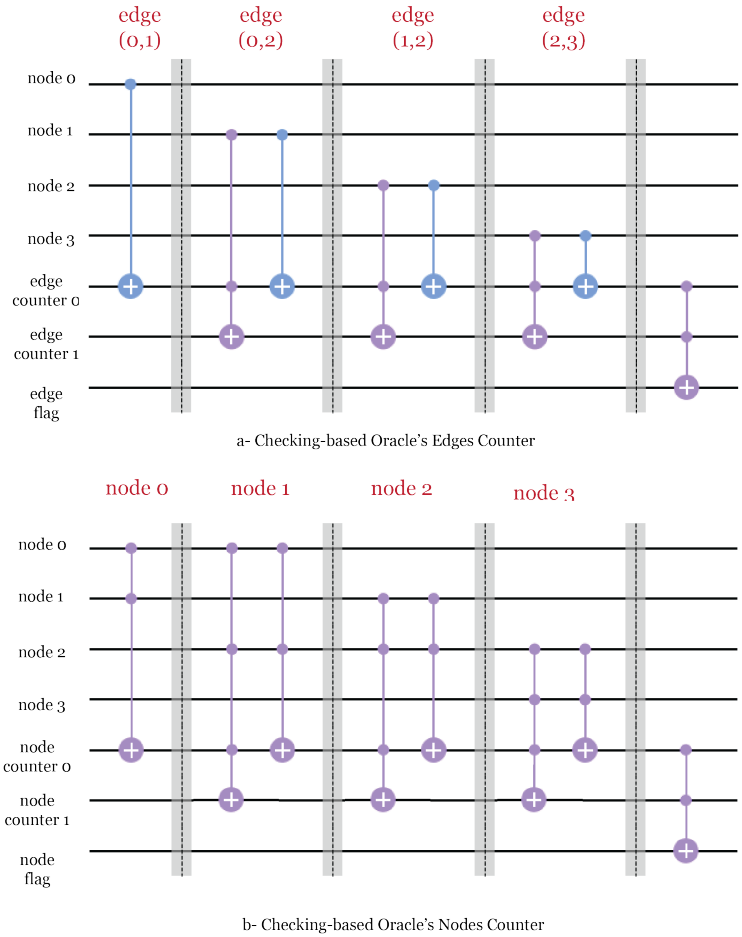


Figure 3.8: Checking-based oracle for the graph in Figure 3.6.

In general, we need $\lceil \log \binom{k}{2} \rceil$ qubits to represent the `edges_counter`. For example, for a 4-clique, the `edges_counter` will be a 3-qubit counter that can count up to 7 (111_2), and for a 5-clique which can count up to 15 (1111_2), the `edges_counter` will need four qubits and so on. For example, if we want to construct the oracle for the graph in Figure 3.5, we will need six node qubits, a 3-qubit edge counter, and one qubit edge flag. The connection of the edges in the graph is then made, as shown in Figure 3.9.

Finally, to check if the `edges_counter` contains the correct value, another $C^{\otimes n}NOT$ gate needs to be applied, the result of which will be saved in another qubit, `edge_flag`

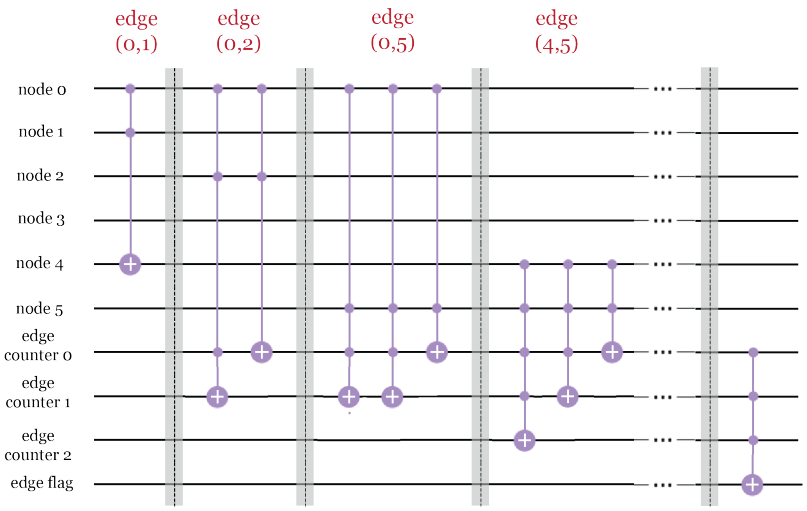


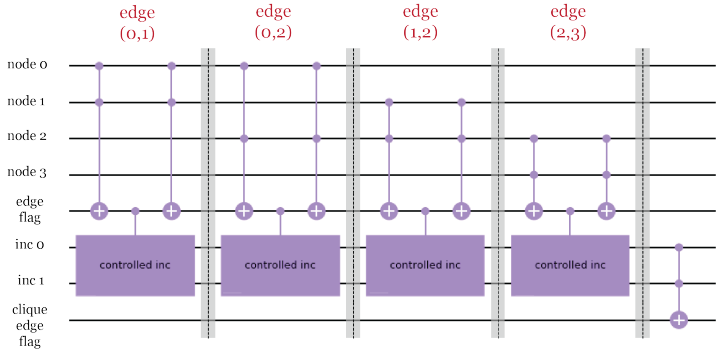
Figure 3.9: Checking-based oracle for the graph in Figure 3.5. The ten edges in the graph are expressed as ten groups of gates, giving the oracle cost $O(|E|)$.

(Figure 3.8-a). A similar circuit is then applied to count nodes; a k-clique should have k nodes. The `node_counter` needs $\lceil \log k \rceil$ qubits with $C^{\otimes n}NOT$ between them. If the `node_counter` contains the correct number of nodes (k), the qubit `node_flag` will become 1. Figure 3.8-b shows the node counting section of the oracle. Finally, after checking for both edges and nodes, a CCX is applied to `edge_flag` and `node_flag` and stored in another qubit `clique_exists`. If we have the correct number of edges and nodes, then a clique of size k exists; otherwise, no clique exists.

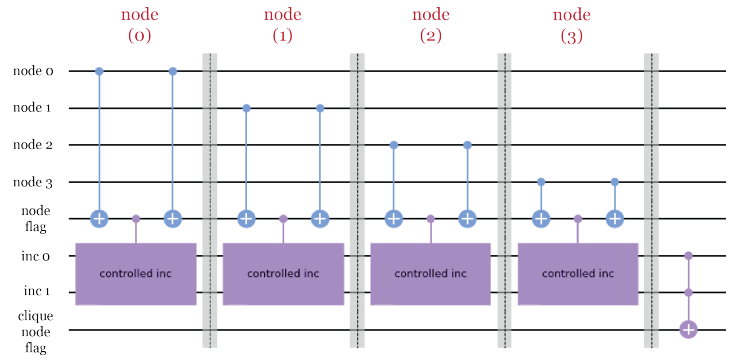
Incremental-based Oracle

For incremental-based oracle, each node in the graph is represented with a qubit, and the edges are expressed using $C^{\otimes n}NOT$ gates. The difference between this and the checking-based oracle is in the `edges_counter` and `clique_flag`. In this implementation, the `edges_counter` is replaced with a one qubit `edge_flag`, and the `edge_flag` becomes 1 if and only if an edge exists between two nodes. That flag is then used as a control qubit controlling an increment circuit that adds one every time it encounters an edge (Figure 3.10). For the `edge_flag` to function correctly, we need to uncompute it (reset to $|0\rangle$ state) after each increment.

The increment circuit size depends on the size of the clique; it will need $\lceil \log \binom{k}{2} \rceil$ qubits. For example, when applying the oracle for a triangle ($k = 3$), we will need a 2-qubit increment circuit to count up to 3 or 11_2. Figure 3.11 shows different sizes of the increment circuit. The circuit finding the triangle in Figure 3.6 needs two qubits for the increment circuit and some ancillary qubits to implement the control functionality.



a- Incremental-based Oracle's Edge Counter



b- Incremental-based Oracle's Node Counter

Figure 3.10: Incremental-based oracle for the graph in Figure 3.6.

After counting the edges in each subgraph Figure 3.10-a, the qubit `clique_edge_flag` will be 1 only if the number of edges is correct $\binom{k}{2}$. When applying this oracle on the entire Hilbert Space, another circuit to count nodes must be added to the oracle 3.10-b. The number of qubits needed in the increment circuits when counting nodes will be $\lceil \log k \rceil$. The number of nodes will be stored in a register `inc`, once the number of nodes in a specific subgraph reaches k, the `clique_node_flag` will turn into 1. Once both the edge counter and the node counter sections of the oracle are executed, the `clique_edge_flag` and `clique_node_flag` are used in a ccx to generate the `clique_flag` which will indicate if a clique of size k exists in the graph or not.

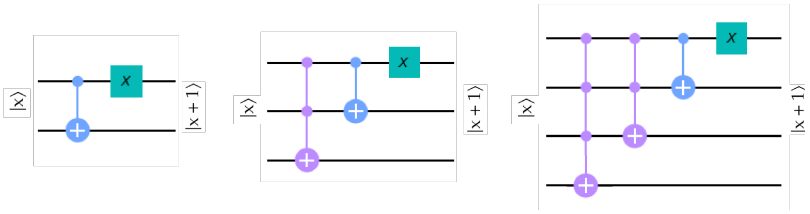


Figure 3.11: Different size increment circuits. From right to left, 2-qubit increment, 3-qubit increment, and 4-qubit increment circuits.

3.5 Results and Analysis

To test the efficiency of our implementation, we compared various combinations of the problem variables. To be consistent, the comparison is based on the smallest instance of the problem, i.e., the triangle finding problem, more precisely, finding the triangle in Figure 3.6. The combinations in the comparison are:

- Grover's algorithm with checking-based oracle over the entire Hilbert space.
- Grover's algorithm with checking-based oracle over limited search space using W state preparation (W state followed by n NOT gates).
- Grover's algorithm with checking-based oracle over limited search space using Dicke state preparation.
- Grover's algorithm with incremental-based oracle over the entire Hilbert space.
- Grover's algorithm with incremental-based oracle over limited search space using W state preparation (W state followed by n NOT gates).
- Grover's algorithm with incremental-based oracle over limited search space using Dicke state preparation.

Table 3.1: Circuit size, depth (length of critical path), and number of qubits needed for all approaches of Checking-based oracle and Incremental-based oracle for the optimal number of iterations for the triangle finding problem in all 3 edges or more 4-node graph shapes

Full search space										
Graph shape	Checking-based Oracle					Incremental-based Oracle				
	# of qubits	depth	size	# of 1-qubit gates	# of 2-qubit gates	# of qubits	depth	size	# of 1-qubit gates	# of 2-qubit gates
Star	13	942	1464	855	609	15	1041	2064	1263	801
Line	13	930	1464	855	609	15	1041	2064	1263	801
Loop	13	1194	1812	1047	765	15	1236	2340	1431	909
One triangle (Figure 3.6)	13	1173	1812	1047	765	15	1284	2640	1731	909
One diagonal square	13	1416	2160	1239	921	15	1551	2616	1599	1017
Complete graph	13	1659	2508	1431	1077	15	1674	2892	1767	1125
W state prep										
Graph shape	Checking-based Oracle					Incremental-based Oracle				
	# of qubits	depth	size	# of 1-qubit gates	# of 2-qubit gates	# of qubits	depth	size	# of 1-qubit gates	# of 2-qubit gates
Star	9	267	409	252	157	10	289	415	258	157
Line	9	251	409	252	157	10	289	415	258	157
Loop	9	345	529	324	205	10	354	507	314	193
One triangle (Figure 3.6)	9	331	529	324	205	10	354	507	314	193
One diagonal square	9	391	649	396	253	10	419	599	370	229
Complete graph	9	471	769	468	301	10	484	691	426	265
Dicke state prep										
Graph shape	Checking-based Oracle					Incremental-based Oracle				
	# of qubits	depth	size	# of 1-qubit gates	# of 2-qubit gates	# of qubits	depth	size	# of 1-qubit gates	# of 2-qubit gates
Star	9	518	719	410	309	10	528	733	432	301
Line	9	514	719	410	309	10	528	733	432	301
Loop	9	602	835	474	361	10	593	825	488	337
One triangle (Figure 3.6)	9	595	835	474	361	10	593	825	488	337
One diagonal square	9	676	951	538	413	10	658	917	544	373
Complete graph	9	757	1067	602	465	10	723	1009	600	409

We will address the analysis from two perspectives, complexity, and practicality, comparing the type of gates and depth of the resultant circuit. In addition, we will also discuss how different state preparations affect the amplitude of the correct answer, using both the ideal-case and gate-error simulations.

3.5.1 Gate Count Analysis

In quantum circuits, the more gates that involve multiple qubits, the more unreliable and difficult it will be to get the circuit to work on actual quantum hardware. First, we will discuss the different circuit sizes for other Oracle implementations and various state preparations. Again, as a base case, we will compare the different approaches in the case of finding a 3-clique (triangle) in a 4-node graph. Table 3.1 shows the other operation counts from checking- and incremental-based oracle for the optimal oracle iteration count. The table shows the variation of the circuit size for all 4-node graph shapes with 3 or more edges at optimal iteration. The size and depth used here are when decomposing the circuit to only single-qubit gates and CNOT gates.

To better understand the numbers in Table 3.1, we need to consider how often the oracle is repeated. Since Grover's Algorithm is periodic, the optimal number of repetitions of the oracle and diffusion is calculated based on the number of input qubits (number of nodes in the graph) and the number of solutions we want. For the sake of

this analysis, we will focus on the case where $m = 1$. If we are using the entire search space, then $N = 2^n$, and so the optimal number of iterations here will be three iterations. However, if we are using Dicke/W states to limit our search space, $N = \binom{n}{k}$, $m = 1$, giving an optimal iteration number of one. Although the number of iterations is smaller with state preparation (Dicke/W state), the circuit may increase in size, based on the state preparation approach followed. While it is relatively easy to prepare initial states in full superposition (only n H gates are needed), preparing an initial state using Dicke/W states is a costly operation, with the Dicke state being the most expensive in gate count. A detailed layout of the gates used in every state preparation is found in Table 3.2.

Table 3.2: Gate type and count for each state preparation approach

State Preparation Method	Gate Count	Gate Type
Full search space	4	(Hadamard, 4)
W State	17	(U3, 6), (CNOT, 6), (NOT, 5)
Dicke State	30	('CX', 12), ('u3', 6), ('CCNOT', 6), ('x', 3), ('CRY', 3)

Circuit size is crucial, but it is more important to check the complete list of gates used. More particularly, NOT, CNOT, CCNOT, $C^{\otimes n}$NOT gate counts play an essential factor in whether the circuit can be applied to an actual hardware device. Table 3.3 lists the number of NOT, CNOT, $C^{\otimes n}NOT$ gates in every approach proposed for the optimal number of iterations for each. Figure 3.12 shows a construction of the CCNOT gate.

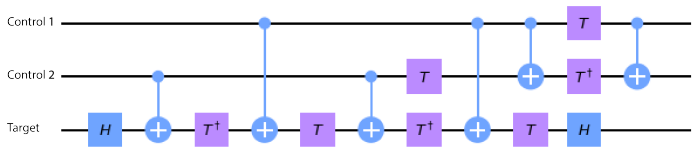


Figure 3.12: A construction of the CCNOT gate.

Another factor affecting whether a circuit is implementable is the circuit depth or length of the critical path of the circuit. Unfortunately, the circuit depth is highly dependent on the hardware layout of qubits and the connections between them. On the bright side, many works have focused on optimizing and generalizing circuit's depth and size for any hardware qubit layout [160] [161] [124] [162] [163].

3.5.2 Simulation Results

This subsection discusses how the change in state preparation affects the amplitude of the correct answer (probability of success). To observe this change, we will simulate the circuit twice, once using the ideal-case simulator (QASM Simulator) and another

Table 3.3: The number of NOT, CNOT, and CCNOT gates in Checking-based and Incremental-based approaches for the triangle finding problem in Figure 3.6

Checking-based Oracle			
	Full Search Space	W state Prep	Dicke state Prep
NOT	25	15	9
CNOT	24	205	115
CCNOT	63	0	18
Incremental-based Oracle			
	Full Search Space	W state Prep	Dicke state Prep
NOT	0	15	9
CNOT	621	145	163
CCNOT	48	8	26

Table 3.4: Average values of T_1, T_2, readout error, and single-qubit error for six different IBMQ Devices

Device name	T1 (in μs)	T2 (in μs)	Readout error	Single-qubit gate error
ibmq_16_melbourne	55	59	0.118	0.0022
imbq_poughkeepsie	64	65	0.0517	0.00179
ibmq_singapore	83	89	0.0389	0.000875
ibmq_paris	76	67	0.032	0.000573
ibmq_cambridge	81	39	0.0959	0.00124
ibmq_rochester	55	59	0.118	0.0022

simulation with added gate error. The Qiskit Aer module provides the pure-state QASM simulator. Aer is a high-performance Qiskit simulation framework for quantum circuits. It offers various backends to meet different simulation ends. QASM simulates any given circuit, assuming ideal qubits and gates with no errors. The results of using the QASM simulation are not realistic for current hardware and represent the goal of future advancements in quantum computers. However, for now, ideal simulators are used. For more realistic results, Aer also provides a way to add noise to the gates while assuming perfect qubits. In real life, both qubits and gates are faulty and noisy, but adding gate noise produces more realistic simulation results.

Several types of errors can be applied to the QASM simulator that correlate with various quantum errors [164] [165]; Qiskit Aer offers ten standard error models, including Depolarization Error, Reset Error, and Thermal Error with an option to create user-customized error models [112]. In addition, the user can choose whether to apply the error to all qubits or a specific set of qubits. In our gate-error simulation, we decided on a realistic thermal-error model (thermal relaxation) (explained in 2.6.2) and applied it to all algorithms' qubits.

Thermal Relaxation Error

To understand better how T_1, and T_2 affect the amplitude of the correct answer, we applied our two proposed oracle structures (for the graph in Figure 3.6) to six different IBMQ devices with different T_1, T_2. Since the value of T_1, T_2 depends on the specific qubits, we took the average T_1, T_2 of the devices when we applied our different circuits. Table 3.6 shows the average values of T_1, T_2, and the names of the six devices used. The table also shows the average readout error and single-qubit gate error rate. We should note that the error rates are determined by gate execution times and the qubit T_1 and T_2 values. The values chosen for the gate execution times are averages based on actual devices as follows, U2 gates take 50 nanoseconds, U3 gates take 100 nanoseconds, CNOT gates take 300 nanoseconds, and finally, the readout will take 1000 nanoseconds[1]. Figure 3.13 shows the results of all proposed approaches on each of the six devices. Various observations can be made by looking at the bar chart. It can be seen that the W state preparation approach mainly retains the correct answer better than other methods, followed by the incremental-based Dicke state preparation approach. It can also be seen that the `ibmq_singapore` device has the lowest error among this set of devices, followed by `ibmq_paris` due to these devices having the highest T_1, T_2 among the devices used. In addition, we added another simulation where T_1, T_2 = 200 μs, and 500 μs. These values were chosen to be noticeably larger than all 6 IBMQ devices we considered while remaining practical. As can be seen from the figure, these simulations have the lowest error rate among all simulations performed. Hence, increasing T_1, T_2 by 60% reduced the error rate and the damping in the amplitude of the correct answer by nearly 42%.

Device-specific Error

The above case incorporates only memory errors; gates are assumed to be perfect. Hence, to provide a more realistic effect of noise models in NISQ devices, we applied the device-specific noise models to three of our implementations. The three implementations we chose to apply device-specific models are Checking-based Oracle with W state Preparation, Incremental-based Oracle with W state Preparation, and Incremental-based Oracle with Dicke state Preparation. We chose these three approaches because they have the highest error tolerance among the six strategies. All three implementations have nine qubit circuits and an ideal (QASM simulator) amplitude of 1.

Considering Figure 3.14, we can observe that when executing the Checking-based Oracle with W state Preparation, Incremental-based Oracle with W state Preparation, and Incremental-based Oracle with Dicke state preparation on real IBMQ devices, the error rate increases sharply. Even the implementations with high error tolerance for changes in T_1, T_2, show a significant drop in the amplitude of the correct answer, with an error rate ranging from 93% to 96%. We can also see that `ibmq_singapore` and `ibmq_paris`

[1]U2, and U3 are basic single-qubit unitary gates presented by Qiskit [166]

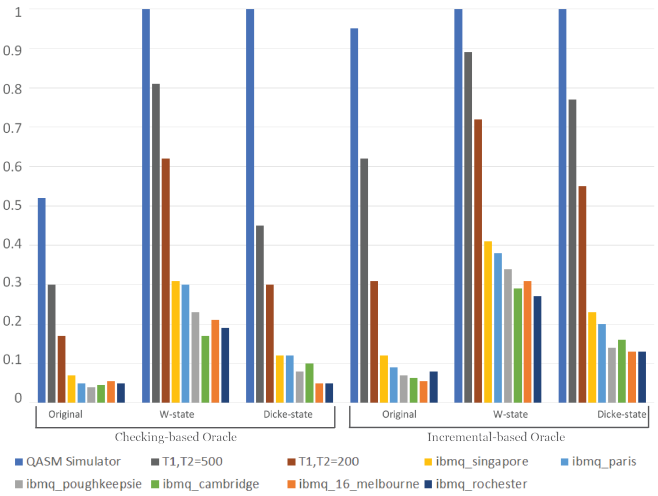


Figure 3.13: The amplitude damping effect of memory decoherence, assuming perfect gates. The bars are the probability of finding the correct answer after simulating a perfect machine (leftmost bar in each group), $T_1 = T_2 = 500$, 200 (next two bars) as well as T_1 and T_2 based on the simulation of six different IBMQ devices in table 3.6 (last six bars). The figure is sorted based on the average error rate from lowest to highest.

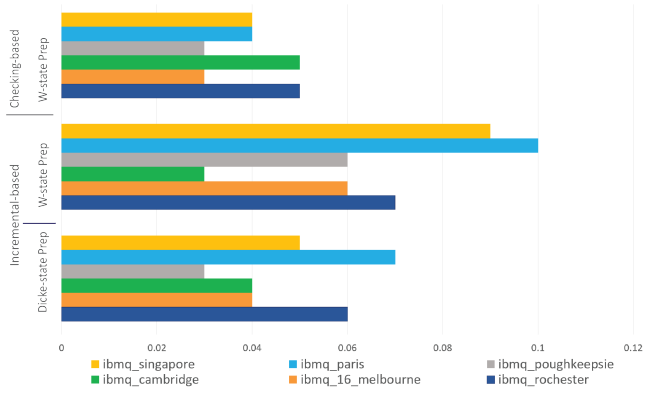


Figure 3.14: Probability of finding the correct answer using the Checking-based Oracle with W state Preparation, Incremental-based Oracle with W state Preparation, and Incremental-based Oracle with Dicke state Preparation. Data is taken from executing the different approaches on the six different IBMQ devices in Table 3.6.

maintained the best performance among the six devices used. We compared running the approaches on the actual devices to simulating the devices' error models on the QASM simulator and found that the results are incredibly close, with negligible differences. That gave us confidence that using the error models of the devices provides a valid representation of the performance of the actual devices.

3.5.3 Time Complexity Analysis

We can split the time complexity analysis into four main parts: analyzing the number of iterations in Grover's algorithm, the initial state preparation (in case of limited Hilbert space search) complexity, the different oracles and diffusion operators complexities, and finally analyzing the total complexity of the algorithm.

Number of Iterations in Grover's Algorithm

The oracle and the diffusion operator are repeated $\lfloor \frac{\pi}{4}\sqrt{\frac{N}{m}} \rfloor = O(\sqrt{\frac{N}{m}})$ times, which depends on the size of the search space and the expected number of answers. Assuming the simplest case, where $m = 1$, such as the case in Figure 3.6, the complexity then becomes $O(\sqrt{N})$. Notice that this applies to the case when the entire Hilbert space is used. However, if we limit the search space using initial state preparation, the number of iterations also depends on the size of clique k and becomes $O(\sqrt{\binom{n}{k}})$.

State Preparation Complexity

We used two different state preparation techniques to limit the search space and using either W state preparation in case $k = n - 1$ or Dicke state preparation otherwise. We followed the algorithm in [159] to prepare the nodes qubits in a W state superposition; the algorithm produces a circuit with complexity $O(\log n)$ and depth of $O(n)$. Here, n represents the number of qubits involved in the W state preparation, which is, in our case, the number of nodes $|V|$. Hence, the cost of preparing W states becomes $O(|V|)$. On the other hand, when using the Dicke state preparation proposed in [158], we get a circuit with depth $O(kn)$ and complexity $O(n)$, where k is the clique size, and n is the number of qubits. Therefore, the cost of preparing the Dicke state becomes $O(k|V|)$.

Oracle and Diffusion Operator Complexities

First, we will discuss the complexity of the diffusion operator. As seen in Figure 5.5, the diffusion operator consists of the state preparation, a $C^{\otimes n}Z$ gate, and the adjoint of

Table 3.5: Complexities for the different steps of the algorithm with and without the initial state preparation.

Algorithmic Step	Time Complexity	Gate Count
W state preparation	$O(\log \|V\|)$	$O(\|V\|)$
Dicke state Preparation	$O(\|V\|)$	$O(k\|V\|)$
Diffusion Operator	$O(\text{state\_prep}))$	$O(\text{state\_prep} + \|V\|)$
Oracle without state prep	$O(\|E\| + log(k))$	$O(\|E\| + \|V\| + log(k))$
Oracle with state prep	$O(\|E\| + log(k))$	$O(\|E\| + log(k))$

state preparation, respectively. Hence, we can generalize the complexity of the diffusion operator as $O(\text{state\_prep}) + O(C^{\otimes n}Z) + O(\text{state\_prep})$. The cost of the state preparation depends on which approach is used; hence, it will be $O(\log |V|)$ in case of W state preparation or $O(|V|)$ in case of Dicke state preparation, as can be seen in Table 3.5. However, the complexity of the $C^{\otimes n}Z$ gate depends on the number of nodes $|V|$; therefore, the complexity of the gate will be $O(|V|)$. Consequently, the cost of the diffusion operator will become $O(\text{state\_prep}) + O(|V|)$, while its complexity will be $O(\text{state\_prep})$.
The complexity of the oracle also depends on whether an initial state preparation is used. Regardless of the oracle implementation (checking-based or incremental-based), the primary function of the oracle counts the number of edges and nodes needed to compose a clique of size k. So, the complexity of the oracle for the entire Hilbert space is $O(\log(k) + |E| + |V|)$. When we use state preparation, we eliminate the need to count nodes; that is because we only allow states with the specific k nodes activated at any time to be included in the search space. Hence, the complexity of the oracle when using initial state preparation to limit the search space is $O(\log(k) + |E|)$.

Algorithm Total Complexity

The total complexity of Grover's algorithm can be expressed as the number of iterations times the cost of one iteration. The number of iterations, as discussed in previous subsections, can be presented as $O(\sqrt{\frac{\binom{n}{k}}{m}})$. Each iteration's cost can be divided into two parts: the oracle's cost and the diffusion operator's cost. Hence the total complexity becomes $O(\sqrt{\frac{\binom{n}{k}}{m}}) \times (O(\text{oracle}) + O(\text{diffusion operator}))$. This complexity assumes the initial state preparation of states in the entire Hilbert space. However, suppose we used W state or Dicke-state as initial state preparation. In that case, the complexity becomes $O(\text{state\_prep}) + O(\sqrt{\frac{\binom{n}{k}}{m}}) \times (O(\text{oracle}) + O(\text{diffusion operator}))$.

To estimate when our proposed schemes of Grover's algorithm to solve the clique

Table 3.6: Average values of T_1, T_2, readout error, and single-qubit error for six different IBMQ Devices

Device name	T1 (in μs)	T2 (in μs)	Readout error	Single-qubit gate error
ibmq_16_melbourne	55	59	0.118	0.0022
imbq_poughkeepsie	64	65	0.0517	0.00179
ibmq_singapore	83	89	0.0389	0.000875
ibmq_paris	76	67	0.032	0.000573
ibmq_cambridge	81	39	0.0959	0.00124
ibmq_rochester	55	59	0.118	0.0022

finding problem can be implemented on an actual device with minimal error, we need to address two factors: the quantum volume and the device performance.

3.5.4 Device Performance

Even among machines with similar QV, their performance depends on more than just the number of qubits in the machine. It also depends on the device noise model, as discussed in 3.5.2, and the coupling map (the connectivity between the qubits). That's why different devices with the same number of qubits perform differently and have different error rates. Each machine has a different error profile, which makes it challenging to estimate the ability to implement any algorithm on an actual device based solely on its QV. We analyzed the performance of our top three error-resistant approaches (Checking-based Oracle with W-state Preparation, Incremental-based Oracle with W-state Preparation, Incremental-based Oracle with Dicke state Preparation) on the two machines with the overall best performance, `ibmq_singapore` and `ibmq_paris`. We obtained the error model of both these devices and modified it in three ways to understand which factor affects the overall error most. We changed the thermal relaxation error by modifying T_1, and T_2 while keeping all other errors untouched, then did the same but with the gate error, and finally, edited both the thermal relaxation error and the gate error together.

To understand better how T_1, and T_2 affect the amplitude of the correct answer, we applied our two proposed oracle structures (for the graph in Figure 3.6) to six different IBMQ devices with different T_1, T_2. Since the value of T_1, T_2 depends on the specific qubits, we took the average T_1, T_2 of the devices when we applied our different circuits. Table 3.6 shows the average values of T_1, T_2, and the names of the six devices used. The table also shows the average readout error and single-qubit gate error rate. We should note that the error rates are determined by gate execution times and the qubit T_1 and T_2 values. The values chosen for the gate execution times are averages based on actual devices as follows, U2 gates take 50 nanoseconds, U3 gates take 100 nanoseconds, CNOT gates take 300 nanoseconds, and finally, the readout will take 1000 nanoseconds[2].

[2]U2, and U3 are basic single-qubit unitary gates presented by Qiskit [166]

Figure 3.15: The effect of manipulating the error model of the `ibmq_singapore` and `ibmq_paris` devices in decreasing or increasing the error percentage. Each color represents an approach, and the shades of the bars represent the type of modification as follows: dark shade modifies both T_1, T_2 and the gate error, the medium shade modifies T_1, T_2 only, and the light shade is modifying the gate error only. The x-axis is the percentage of modifying the errors.

As seen in Figure 3.15, changes in error rate depend on both the implementation of the circuit and the device used for execution. The changes applied to the noise model were increasing T_1, T_2 by 25, 50, 75, and 100% while decreasing the gate error by 25, 50, 75, and 100%. The difference in error rate due to modifications (changing T_1, T_2, and gate error) can increase the device error up to 7.5% and decrease down to 20.5%. It can also be seen that the incremental-based approach with W-state preparation has the largest decrease in error rate, significantly when modifying both T_1, T_2, and the gate error. Finally, we can see that changing T_1, T_2, only leads to better results than modifying the gate error only. Figure 3.15 also shows that in some cases, modifying T_1, T_2, and the gate error led to an increase in the overall error rate. The reason is that the overall error depends on many factors, such as other types of error (reset and readout errors, and the measurement—machine maintenance cycle) and the date on which experiments were conducted.

Chapter 4

Testing and Debugging Quantum Circuits

4.1 Introduction

In classical software, the development process follows a mature cycle. Two critical stages of the cycle are testing, debugging, and maintaining the application. The application's abstract aspects may be tested with formal specifications, pseudocode, modeling tools, etc. Bugs arise from errors in the specification of a program, in translating the specification into code, or, sometimes, from bugs in the tools themselves. Currently, there are many approaches to testing classical software, both formal and informal [167, 168, 169]. Approaches such as unit testing, regression testing, continuous integration, and path coverage testing make building and supporting systems as complex as tens of millions of lines of code, such as the Linux kernel, possible [170, 171].

Like the classical software development cycle, the quantum software development cycle shown in Figure 2.8 describes developing software for quantum computers as proposed in [105]. Since quantum computers can operate on the *superposition* of values (each with a *complex amplitude* [106, 107]), the exponential growth in the state space poses a fundamental problem in testing and debugging quantum programs.

When we want to test a quantum circuit, we often have to consider the behavior of all possible inputs *as a set*. That exponential growth in the input state space poses fundamental challenges during the testing and debugging.

The first and perhaps most critical challenge is the principle on which quantum algorithms operate. The goal of quantum algorithms is often to find a solution to a problem through building *interference patterns* that amplify the amplitude of correct

answers at the expense of the incorrect ones.

First, we must consider the steps of testing a quantum circuit to build a framework for testing and debugging quantum programs. Testing a quantum circuit involves several steps, each of which plays a vital role in ensuring the functionality and accuracy of the computations. We can set the needed steps as follows:

1. **Writing Code:** This step involves creating the quantum circuit using the specialized quantum programming languages or quantum packages supported by classical programming languages (Section 2.7).

2. **Writing Tests:** Creating test vectors for the circuit and specific parts. This step includes several sub-steps.

    - *Slicing the Circuit:* Dividing the circuit into smaller segments for efficient testing.
    - *Categorizing Slices:* Categorizing the slices based on their functionality within the circuit.
    - *Adjusting Slice Start/End Points:* Modifying the boundaries of a slice as necessary.
    - *Developing Test Vectors:* Creating sets of inputs and expected outputs for each slice type.
    - *Assessing Coverage:* Ensuring that the tests cover all (or most) possible scenarios the slices might encounter.

3. **Confidence Interval Selection:** Deciding on the statistical confidence level for the test results, which includes:

    - *Choosing Optimal Number of Shots:* Determining the number of circuit executions to balance accuracy and computational resource usage is discussed below.

4. **Integration:** Resolving issues arise when a slice, which works independently, fails upon integration.

5. **Running Tests:** Executing the developed test cases against the slices of the quantum circuit and comparing the actual behavior with the expected one. This step can be done on a simulator (if the slice size allows) or an actual device.

6. **Error Isolation and Additional Testing:** If an error is detected, the problematic slice is isolated and subjected to further focused testing to pinpoint and understand the bug.

These steps provide a systematic approach to testing and debugging quantum programs, which we will follow loosely in this chapter.

Before we demonstrate the proposed suite, we first need to look at the bugs commonly occurring when writing quantum programs today.

4.2 Bugs and Errors in Quantum Software

Most work done around the field of testing and debugging quantum software mainly focused on finding patterns in quantum bugs and collecting frequent bugs. Since understanding the flow of quantum programs and the causes of errors is essential for the ability to debug quantum circuits, researchers focused on reproducible bugs and categorizing the occurrences of bugs in quantum programs [172, 173, 174]. In these studies, researchers found that quantum program bugs can occur for multiple reasons. Based on their source, quantum bugs were categorized to make their understanding easier.

Bugs could occur due to either the package/library used or the developer's implementation of an algorithm. Library-related bugs occur because of a fault in the implementation of the API, such as deprecation and mistakes in the data handling within the API. On the other hand, bugs introduced by the developer cover incidents such as the wrong ordering of the gates or misuse of the functions offered by the API.

Because the API used to implement quantum programs can introduce a lot of bugs to the programs [175, 176] conducted work on these particular sources of bugs and how it can affect the following programs.

In [174], 35 of the bugs were related to a mistake in the implementation of the API (some provided functions do not behave as expected or described in the documentation), while 61 were due to some errors introduced by the developer. We collected an additional 73 bugs from the same sources (Stack Overflow, Stack Exchange, and GitHub) (a table with all 123 bugs can be seen in Appendix B). We then classified the bugs into four categories, summarized based on whether they are caused by the library/package or the user in Table 4.2:

- System Backend bugs include transpiler, decomposition, and simulator errors and those related to the API implementation. The programmer does not introduce these bugs, which may cause a correct implementation not to work correctly.

- Classical data post-processing bugs include misunderstanding/wrong interpretation of the data or a mistake in parameterizing the circuit. This category of bugs is about the different ways the programmer can mistakenly interpret the results. A common example of this bug is mistaking little Endian with big Endian representation [177] when reading the register variable of the working qubits. For example, if we have an algorithm that should output $|1101\rangle$, in Qiskit (which uses

Table 4.1: The count of bugs causing runtime exceptions based on whether the library/-package or the user causes them.

	Throws runtime exception?		
	Yes	No	Sum
Library	35	8	43
User	39	41	80
		Total	123

little Endian), it will be $|1011\rangle$ which often leads to confusion in interpreting the results.

- Classical semantics bugs include typos and mishandling of functions, such as missing attributes and confusing data types. This type of bug is not quantum-specific. These bugs also occur in classical programs, and although they may lead to false circuit results, fixing them does not require any quantum knowledge but rather an understanding of the software and programming language used.

- Quantum logic bugs are bugs that will cause difficulties in developing larger quantum software in the future. Quantum circuit bugs include missing or extra gates, applying a gate to the wrong qubit, wrong gate order, initialization error, wrong phase, or using the wrong gates.

Understanding the bugs in quantum programs is essential to debugging and testing quantum programs. A simple differentiation we can make here is: How many of these bugs cause a runtime exception? If a runtime exception also occurs in classical programming, fixing it is easier. Table 4.1 shows the distribution of the bugs based on their source and category.

Moreover, considering the categorization above, we can refer to the first three categories as non-quantum-specific bugs and the last as quantum bugs. Those categories include bugs that could occur in classical programming as well; hence, we can approach debugging them as we do classically. Another important note about such bugs is that they often cause a runtime exception, which makes it easier to locate and fix.

Let us consider the non-quantum-specific bugs first. These bugs are introduced by the package used, misuse of that package by the programmer, or misinterpretations of the results. An example of bugs introduced by the package includes errors due to deprecation or compatibility between the package and its dependencies.

```
from qiskit.providers.aer import QasmSimulator
# Create a quantum circuit
qc = QuantumCircuit(2, 2)
qc.h(0)
```

```
qc.cx(0, 1)
qc.measure([0, 1], [0, 1])
# Get the Qasm simulator and set the backend options
aer_qasm_simulator = Aer.get_backend('qasm_simulator')
# Set the backend options, method set to statevector
options = {'method': 'statevector', 'memory' : True, 'shots':10}
# Execute circuit using the backend options created
job = execute(qc, backend_simulator, backend_options=options)
result = job.result()
# Pull the memory slots for the circuit
memory = result.get_memory(qc)
# Print the results from the memory slots
print('Memory results: ', memory)'''
```

Listing 4.1: An example of non-quantum-specific bug due to misuse of the functions offered by the API by the programmer.

In Listing 4.2, we can see an example of a piece of code that produced an error due to the deprecation of the function *Shor* in the recent versions of Qiskit. Another example of a non-quantum-specific bug can be seen in Listing 4.1, where this code produces an error because the programmer used the wrong parameters when choosing a backend to execute the circuit. Finally, the programmer can implement an algorithm correctly and have no issues due to the API, yet misinterprets the results and thinks they are incorrect; that is often due to confusing little and big Endian representations of the qubits.

```
from qiskit import Aer
from qiskit.utils import QuantumInstance
from qiskit.algorithms import Shor

N = 15
backend = Aer.get_backend('aer_simulator')
quantum_instance = QuantumInstance(backend, shots=1024)
shor = Shor(quantum_instance=quantum_instance)
result = shor.factor(N)
print(f"The list of factors of {N} as computed by the Shor's
```

Table 4.2: Categorizing the bugs collected from StackExchange and StackOverflow based on their type and cause.

Bug Type	Bug Source		
	Due to package	Due to User	
Classical Data Post-processing	11	15	
Classical Semantics Bugs	4	24	
Quantum Logic	1	37	
System Backend	27	4	
Total	43	80	123

```
    algorithm is {result.factors[0]}.")
```

Listing 4.2: A simple example of a bug due to the library (Qiskit) deprecation of the Shor function.

If we focus on the quantum circuit bugs introduced by the programmer, the impact of bugs can often depend on the specific application and the type of algorithm used. These bugs can sometimes cause syntax or compile-time errors (which can be caught and fixed quickly), while others only evidence themselves at runtime. We can further categorize these as:

- Initialization bugs.
- Gate-order bugs include extra gates or gates applied to the wrong qubit.
- Quantum-logic bugs, such as using the wrong phase or misinterpreting an algorithmic step.

An initialization error indicates that the programmer made a mistake in the initial state required to form a specific state. An example would be if the programmer "forgot" to set all qubits in state $|+\rangle$ when constructing a cluster state [178], which would lead to incorrect results Listing 4.3. We can also think of state preparation when implementing different algorithms, such as Grover and Shor algorithms, as initialization bugs.

```
from qiskit import QuantumCircuit
def create_cluster_state(n_qubits: int) -> QuantumCircuit:
    """Create a cluster state for the given number of qubits."""
    # Initialize a quantum circuit with the specified number of
    qubits
    qc = QuantumCircuit(n_qubits)
```

```
    # Apply CZ gates between neighboring qubits to create the cluster state
    for i in range(n_qubits - 1):
        qc.cz(i, i+1)
    return qc

n_qubits = 4
cluster_circuit = create_cluster_state(n_qubits)
```

Listing 4.3: A Python and Qiskit implementation of cluster states missing the initialization of the qubits to the $|+\rangle$ state by applying H gate to all qubits before the CZ gates.

The final category is what we refer to as quantum-logic bugs. This includes applying gates to the wrong qubits, placing them in the wrong order, and missing or adding an extra gate. Section 4.5 will discuss an example of that. In addition to phase errors, they are caused by using the wrong phase or making a mistake in translating an algorithm into code. Section 4.5 will also discuss an example of such a bug.

4.3 Quantum Circuit Slicer

The current state of classical debuggers results from decades of research [179], development, and experiments [180]. One of the most fundamental concepts used in classical debugging is the concept of program slicing [181]. A programming tool divides a big body of code into smaller, easy-to-test and manage chunks. Each of these chunks is called a *slice*.

Slices are formed in two ways, either manually using breakpoints [182] or using a form of automatic/semi-automatic slicing. Using breakpoints, the debugger can divide the code so the user can observe its behavior and the variables' contents within each slice. There are different types of automated program slicing; the basic two are static and dynamic. Static slicing works by slicing the program based on a variable or set of variables by eliminating the lines of code that do not include or affect that variable directly or indirectly. In dynamic slicing, on the other hand, the slice is formed using variable(s) and condition(s). The variable(s) and shape used to create the slices are called the *slicing criteria*.

As the available interest and the current size of the available systems continue to increase, the circuits implemented will also increase. Hence, we implemented a quantum circuit debugging tool with a circuit slicer based on manual slicing and breakpoints. A

quantum circuit slicer will divide a large circuit into smaller, simulatable sub-circuits to prove the use of the circuit both in the current NISQ (Noisy Intermediate-scale Quantum Computers) era [1] and the future era of fault-tolerant quantum computing.

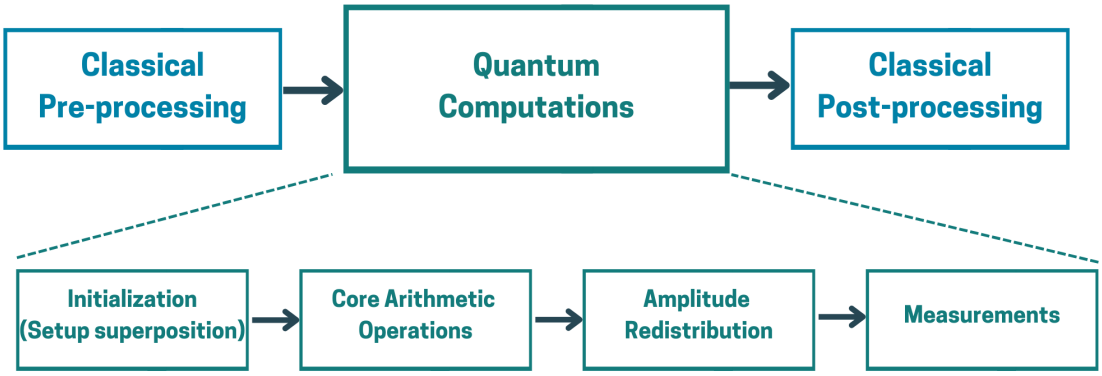


Figure 4.1: The different steps needed to implement and execute most quantum algorithms.

Generally, quantum algorithms follow a set of steps to solve a problem. All quantum algorithms start with preparing the qubits in a specific state or a uniform superposition, then perform some arithmetic and calculations, followed by redistribution of the amplitudes. Depending on the algorithm and the problem being solved, some may include classical pre-processing or post-processing after the measurement procedure Figure 4.1. For example, let us consider Grover's algorithm, which consists of three algorithm steps: preparing the qubits in a uniform superposition, followed by a problem-specific oracle, and then a diffusion operator. In the algorithm, the oracle and diffusion will repeat multiple times until the answer is reached.

We implemented a manual slicer, where the user inserts breakpoints (in a quantum context, breakbarriers) in the circuit and then simulates the resultant slices or runs them on an actual device to observe their behavior. To make the valuable tool for all sizes of circuits, it has to be able to slice the circuits on two axes, the gate axis (vertically) and the register axis (horizontally) Figure 4.3. That is, the user can insert breakbarriers vertically in the circuit to divide it into smaller circuits and horizontally remove any qubits that are unused in any of the slices.

Vertical Slicing

To explain the methodology and concept of slicing, let us think of a circuit corresponding to Grover's algorithm [41]. Based on each algorithmic step, we can use breakbarriers to divide the circuit into slices. Grover's algorithm consists of three main algorithmic steps: initial state preparation, an oracle, and the diffusion operator. To keep things simple,

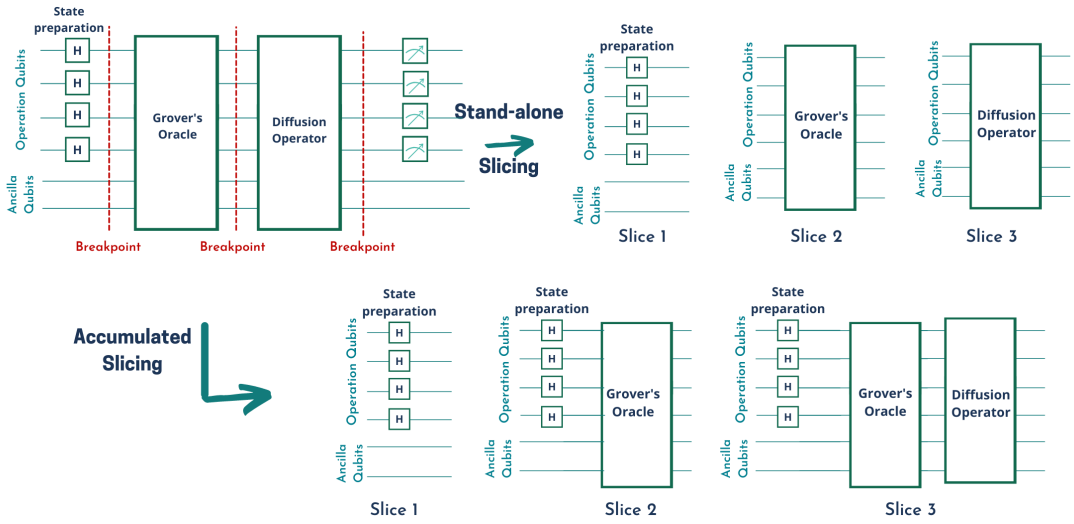


Figure 4.2: A generic Grover's algorithm circuit sliced using both stand-alone and accumulated slicing.

assume Grover's algorithm we are slicing consists of one iteration of the algorithm. We will insert two breakbarriers to slice this circuit, one after the state preparation and one after the oracle. This will result in three sub-circuits, each performing a specific step in the overall algorithm.

The circuit slicer offers two options for vertical slicing as in Figure 4.2:

- Stand-alone slices: The slices are defined by the breakpoints.
- Accumulated slices: Each slice is added to the slice before it to create a new slice.

Horizontal Slicing

Sometimes, after slicing the circuit vertically, we may end up with a slice that contains some unused qubits. Since our goal of slicing the circuit is creating smaller, simulatable, executable circuits, having unused qubits is redundant. Hence, we can do horizontal slicing to remove these unused qubits from the slice. The current version of the tool only allows for the automatic slicing of unused qubits. Future expansion will allow users to manually insert horizontal breakbarriers in slices with two independent registers or a set of qubits. One main challenge of slicing quantum circuits horizontally is cross-register entanglement before the slice, which becomes more prominent if the horizontal slicer is manual and allows the programmer to choose the slicing location, such as CutQC [183]. CutQC uses the Kronecker product to overcome the challenge of considering the effect of the slice. For example, if we cut only one wire, we get a 4^k Kronecher product, where k is the number of qubit wires cut. The math used to develop CutQC [184], indicates that

the probability of the measurement of an input state $|\psi\rangle$ for the unsliced circuit must be equal to the sum of possibilities of the same state for the slices following Equation 4.1, where N is the number of subcircuits resultant from the slicing.

$$p(|\psi\rangle) = \frac{1}{N} \sum_{i=1}^{4^k} p_{1,i} \otimes ..... \otimes p_{N,i}, \tag{4.1}$$

Where, $p_{1,i}$ is the probability of obtaining a particular measurement outcome i from the first subcircuit and the tensor product $p_{1,i} \otimes \cdots \otimes p_{N,i}$ combines the probabilities of all subcircuits to reconstruct the probability distribution of the original, unsliced circuit.

After slicing a large quantum circuit into smaller subcircuits, CutQC handles the entanglement between qubits that span different subcircuits through a classical postprocessing step. The key steps are:

- At the cut points, the qubit states are measured in different bases, and corresponding states are initialized in the next subcircuit. This process generates several combinations of measurement and initialization pairs.
- Each subcircuit is executed independently on a quantum computer, generating probability distributions for each possible measurement outcome.
- The measurement outcomes and corresponding probabilities from each subcircuit are combined using tensor products. The final probability distribution of the entire circuit is reconstructed by summing the tensor products over all combinations of measurement outcomes from the subcircuits.

This approach allows the effects of entanglement between qubits in different subcircuits to be captured and recombined classically, effectively reconstructing the behavior of the original, unsliced circuit.

Lastly, if we assume that the programmer is using a NISQ machine to execute both the slices and the original circuit they need to use an efficient number of shots to achieve good coverage of the possibilities of measuring the different states.

4.4 The Different Types of Quantum Circuits

Generally speaking, we can say that quantum algorithms consist of blocks (Algorithmic steps) that perform either classical computations and blocks that create constructive and destructive interference. Based on that, this thesis proposes a categorization of quantum circuits according to their behavior into three categories:

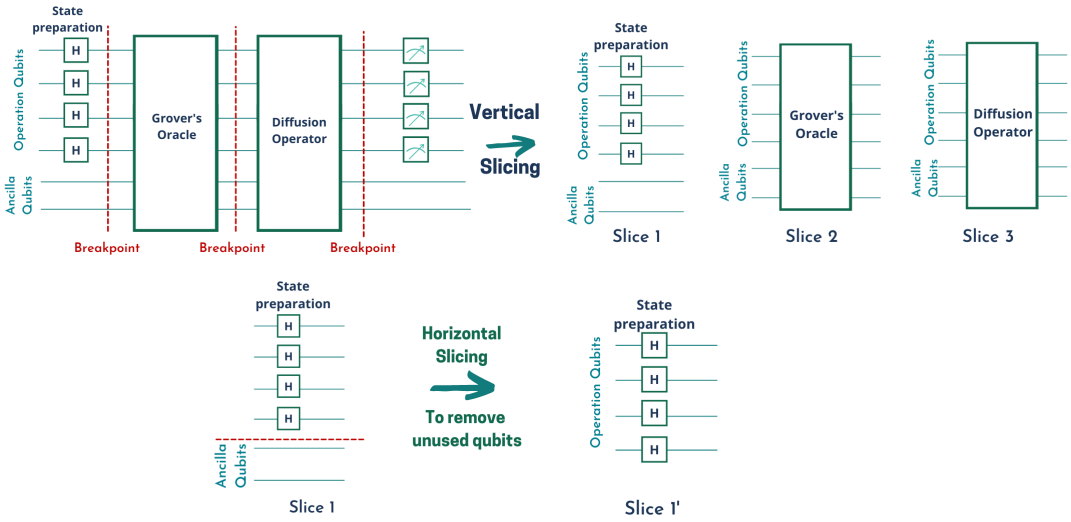


Figure 4.3: A generic circuit for Grover's algorithm is sliced into 3 vertical slices, then the first slice is horizontally re-sliced to remove unused qubits.

- **Amplitude-Permutation (AP) Blocks** A type of quantum circuit primarily focuses on permuting the amplitudes of quantum states. These circuits operate like classical logic gates within the quantum realm, rearranging the probabilities (amplitudes) associated with the quantum states without altering their phases. An example of that is a quantum adder or Grover's oracle. Those blocks are essentially classical reversible logic [185, 186]. Mathematically, for set of states $\alpha_j|j\rangle$, an AP block can be defined as:

$$\sum_j \alpha_j|j\rangle \rightarrow \sum_j \alpha_{\Pi(j)}|j\rangle \tag{4.2}$$

  Where $\Pi(j)$ is a permutation function. In terms of unitary operators, an example 2-qubit AP block unitary (a) might be a permutation matrix with exactly one 1 in each row and column, such as

$$a = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

  An AP circuits can be tested and executed using single amplitudes.

- **Phase-Modulation (PM) Blocks** Quantum circuits that focus exclusively on alter-

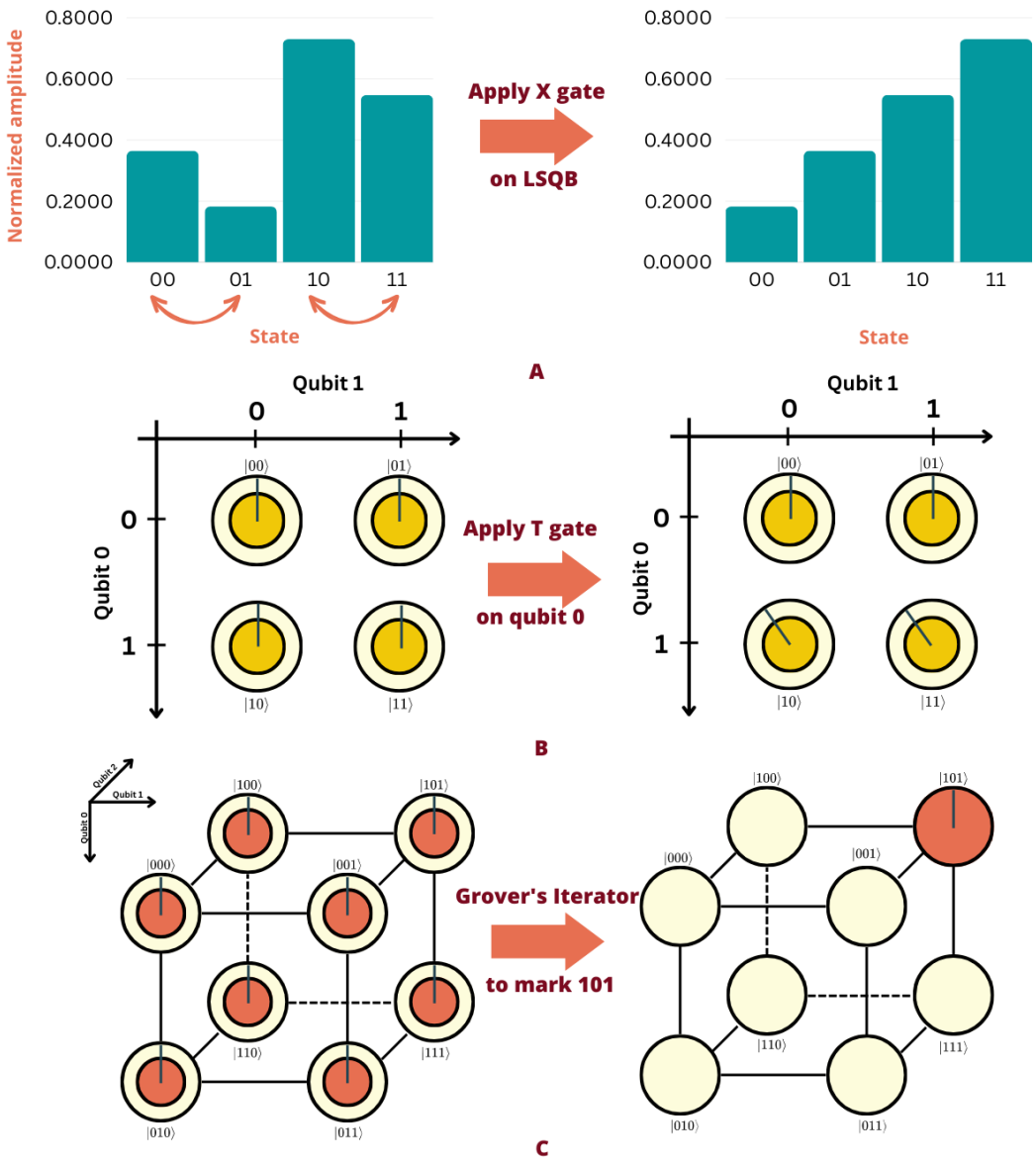


Figure 4.4: Examples of AP and AR block. A. (left) A random amplitude distribution of a 2-qubit state (right) The amplitudes are permuted after applying the NOT gate to the LSQB. B. (left) A two-qubit superposition. (right) The phase is modulated after applying a T gate to the LSQB. C. (left) A uniform superposition of 3 qubits (right) The probability amplitude after applying Grover's iterator to mark the correct answer $|101\rangle$.

ing the phases of quantum states without changing their amplitudes. The primary function of these circuits is to introduce phase shifts in qubits. An example of such circuits is the Quantum Phase Estimation (QPE) algorithm. This algorithm showcases the modulation of quantum states' phases to derive information. Mathematically, for set of states $\alpha_j|j\rangle$, a PM block can be defined as:

$$\sum_j \alpha_j|j\rangle \rightarrow \sum_j \alpha_j e^{i\pi f(j)}|j\rangle \tag{4.3}$$

Where $f(j)$ is a function that calculates the phase shift of a state, $f(j) \in R$. The unitary of a PM block will be a diagonal matrix D with $D_{jj} = e^{i\pi f(j)}$.

- **Amplitude-Redistribution (AR) Blocks** Unlike the Amplitude-Permutation Circuits, these circuits redistribute the amplitudes across various quantum states, thereby harnessing the full potential of quantum superposition and entanglement. An example of an AR block is the Quantum Fourier Transform (QFT). An AR block contains gates that alter interference patterns and create or destroy superposition. AR blocks can be represented as:

$$\sum_j \alpha_j|j\rangle \rightarrow \sum_j \alpha'_j|j\rangle \tag{4.4}$$

Where $\alpha'_j = \sum_k U_{j,k}\alpha_k$, here, $U_{j,k}$ are the elements of the unitary matrix applied to the qubits. An example of a straightforward AR block is only the Hadamard gate.

Though Grover's oracle is an AP block, Grover's iterator (the oracle and the diffusion operator) is an AR block. Figure 4.4-B shows the Q-sphere before and after applying Grover's iterator to a 3-qubit circuit.

Based on the different characteristics of those types, the debugging process will differ significantly. It will even vary within each type. The categorization depends on the size of the circuit as well as the simplicity of creating test vectors to check the functionality of the circuit accurately.

4.5 Testing The Different Circuit Blocks

As discussed in Section 4.4, we divided quantum circuits into three types with entirely different properties, making the process of testing and debugging them significantly different. Moreover, testing and debugging the same type will vary depending on its size and the type of instructions it contains.

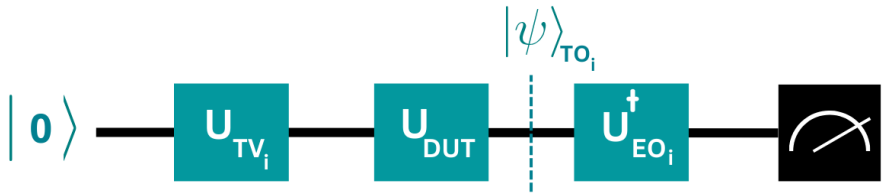


Figure 4.5: One approach to testing quantum circuits, where U_{DUT} is the Device Under Test (full circuit/slice). U_{TVi} is the circuit corresponding to a test vector i. U_{EOi} is the simplified circuit corresponding to the expected behavior for test vector i. If $|\psi\rangle_{TOi} = |\psi\rangle_{EOi}$ the output will be $|0\rangle$.

AP blocks behave like classical programs [185, 187]; hence, we can use classical approaches when testing them. Therefore, the challenge when testing them would primarily be the difficulty of generating test cases that can provide full coverage. For example, for an adder, we can test a few simple inputs using single amplitudes (no superposition), including overflow cases, and reason by induction for the rest. Unfortunately, that approach cannot be extended to AR and PM blocks because they contain quantum properties that are difficult to address using traditional testing and debugging techniques.

Although creating test vectors for AR and PM blocks is not as simple as doing so for the AP blocks, creating test vectors for simple AR and PM blocks should be slightly more straightforward than the complex ones. For example, creating test vectors for a 4-qubit block that contains only 4 Hadamard gates is more straightforward than creating test vectors for a circuit of 20 qubits and 60 different gates.

The size of the block and the types of instructions in it are not the only challenges we face when testing and debugging AR and PM blocks.

Before we discuss the different approaches for each type, we need to discuss an approach to testing PM blocks. To do that, we must define a few important terms:

- U_{DUT}: Device Under Test, which refers to the slice/circuit we are testing/debugging.
- U_{TVi}: Is the circuit corresponding to applying test vector i to U_{DUT}. The results of this is $|\psi\rangle_{TOi}$, where $|\psi\rangle_{TOi} = U_{DUT}U_{TVi}|0\rangle$.
- U_{EOi}: The expected behavior of U_{DUT} for TVi which is $|\psi\rangle_{EOi}$. Because this is specific to a single test vector, it will be substantially simpler than U_{DUT}.

Figure 4.5 shows an approach to testing any quantum circuit (regardless of its type). Essentially, we want to answer the question: *Does* $|\psi\rangle_{TOi} = |\psi\rangle_{EOi}$? If the slice we are testing is correct, these two states are equal; if not, we can conclude that something is

incorrect in the test circuit and proceed with the debugging process. The debugging process will then differ based on the circuit block we target.

To make it easier for the programmer, Cirquo has two testing functions for the two different types of programs. To test AP circuits, the programmer can use the `pClassTester` function with a classical representation of the qubits states. On the other hand, the function `fQuantTester` allows the developer to use state vectors as input/output pairs to test the program for PM and AR circuits.

To see the difference, let us first address AP programs and how they can be debugged and tested using Cirquo. Before we start explaining the difference between testing the two types of circuits, it is important to explain the construction of the test vectors. The general format of a test vector is as shown in Listing 4.4. When constructing tests for AP circuits, the *input* and `expected_out` values would be a list of classical representations of the qubits, while they would be the state vector of the qubits when testing the AR circuits.

```
test_cases = [
    {
        "name": "test 1",
        "input": [],
        "expected_output": []
    },
    .
    .
    .
    {
        "name": "test n",
        "input": [],
        "expected_output": []
    }
]
```

Listing 4.4: The format of test vectors.

4.5.1 Generating Test Cases

One key element missing in all quantum software today is testing, particularly creating test cases. Though some tools offer unit testing for quantum programs, they leave it to

the programmer to create test cases.

Cirquo offers functions to generate basic tests for some commonly used sub-routines in quantum algorithms to ease the load off the programmer. The test generation functions Cirquo offers for the subroutines:

- Quantum Full Adder
- The Diffusion Operator
- The W state
- The GHZ state
- The Dicke State
- Quantum Fourier Transform

For example, if the programmer needs to test their implementation of the W state, they can use the specific function provided by Cirquo to generate tests for the W state based on the number of input qubits. The function to generate tests for any of the provided subroutines works as follows: Given the number of input qubits, the function generates 6 basic inputs corresponding to the different bases $|0\rangle, |1\rangle, |+\rangle, |\rangle, |i\rangle, and |-i\rangle$. Returning to the example of generating tests for the W state, assume that we need to generate tests for a 2-qubit W state. We can use the function `generate_w_state_test_cases(2)` to generate the tests in a format we can pass to the testing function to validate the correctness of a given W state implementation as shown in Listing 4.5.

```
{
'name': 'test 0', 'input': [(1+0j), 0j, 0j, 0j], 'expected_output':
    [0j, (0.707+0j), (0.707+0j), 0j]
}
{
'name': 'test 1', 'input': [0j, 0j, 0j, (1+0j)], 'expected_output':
    [(-2.586e-17+0j), 0j, 0j, (1+0j)]
}
{
'name': 'test +', 'input': [(0.5+0j), (0.5+0j), (0.5+0j), (0.5+0j)],
    'expected_output': [(0.5+0j), (0.707+0j), (-9.872e-17+0j), (0.5+0j
    )]
}
```

```
{
'name': 'test -', 'input': [(0.5+0j), (-0.5+0j), (-0.5+0j), (0.5+0j)
    ], 'expected_output': [(-0.499+0j), (1.0751e-16+0j), (0.707+0j),
    (0.5+0j)]
}
{
'name': 'test i', 'input': [(0.5+0j), 0.5j, 0.5j, (-0.5+0j)], '
    expected_output': [(1.2930e-17+0.5j), (0.353+0.353j), (0.353-0.353
    j), (-0.5-1.293e-17j)]
}
{
'name': 'test -i', 'input': [(0.5+0j), -0.5j, -0.5j, (-0.5+0j)], '
    expected_output': [(1.293e-17-0.5j), (0.3535-0.353j), (0.353+0.353
    j), (-0.5+1.293e-17j)]}
]
```

Listing 4.5: Test vectors for the 4-qubit W state.

This list of tests can then be used with the fQuantTester to run tests on any implementation of the W state to validate its functionality. The W state only needs one parameter to be defined: the number of input qubits. However, some subroutines, namely the Dicke state, require two parameters to be defined, the number of qubits and the Hamming distance between them. To simplify the process for the programmer, each of the given subroutines by Cirquo has its own test generation function. To generate the test vectors for a 3-qubit Dicke state with a Hamming distance of 2, we use the function `generate_dicke_state_test_cases(3,2)`, more details about that will be discussed in Section 5.3.

Similarly, we can use the functions `generate_qft_test_cases` and `generate_qpe_test_cases` to generate test cases for QFT and QPE, respectively. The current implementation of Cirquo generates test cases as state vectors.

4.5.2 Testing and Debugging Amplitude Permutation (AP) Blocks

One example of an AP circuit is the quantum full adder. The full adder is a 4-qubit system, where the inputs are $|A\rangle$, $|B\rangle$, $|C_{in}\rangle$, and $|0\rangle$. $|A\rangle$ and $|B\rangle$ are the qubits we wish to add and $|C_{in}\rangle$ is the carry-in. The output of the full adder is $|A\rangle$, $|B\rangle$, $|S\rangle$, and $|C_{out}\rangle$, here $|S\rangle$ is the sum of qubits $|A\rangle$ and $|B\rangle$ Listing 4.6.

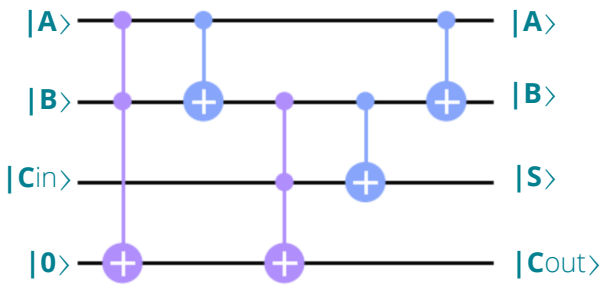


Figure 4.6: The circuit constructing a full adder.

```
def Quant_full_adder(qc, in_qbits,zero_qubit):
    qc.ccx(in_qbits[0],in_qbits[1],zero_qubit)
    qc.cx(in_qbits[0],in_qbits[1])
    qc.ccx(in_qbits[1],in_qbits[2],zero_qubit)
    qc.cx(in_qbits[1],in_qbits[2])
    qc.cx(in_qbits[0],in_qbits[1])
    return qc
```

Listing 4.6: A simple Python and Qiskit implementation of the full adder.

Let's look at this circuit 4.6 from a high level. Without considering the decomposition of the control not and Toffoli gates, we can say that this circuit is AP, using the `catCircuit` function in Cirquo will also point that this circuit is AP. Since this is an AP circuit, we can test it using a classical approach. Cirquo includes two testing functions, one to test AP circuits and one for AR ones. In this case, we will use the AP circuit tester `pClassTester`. The tester function takes on a circuit and a list of test vectors. When testing AP circuits, we will create the test vectors as a list of the desired values of the qubits. For example, to test the full adder, we can pass the test vectors in Listing 4.7.

```
test_cases = [
    {
        "name": "test 1",
        "input": [1,1,1,0],
        "expected_output": [1,1,1,1]
    },
    {
        "name": "test 2",
```

```
        "input": [0,0,0,0],
        "expected_output": [0,0,0,0]
    },
    {
        "name": "test 3",
        "input": [1,0,1,0],
        "expected_output": [1,0,0,1]
    },
    {
        "name": "test 4",
        "input": [0,1,0,0],
        "expected_output": [0,1,1,0]
    },
    {
        "name": "test 5",
        "input": [1,1,0,0],
        "expected_output": [1,1,0,1]
    }
]
```

Listing 4.7: Test vectors for the full adder.

Running the program for these test vectors results in a PASS status for all of them, as seen in Listing 4.8.

```
Testing test 1:
Result:  PASS
Input:  [1, 1, 1, 0]
Output:  [1, 1, 1, 1]
Expected Output:  [1, 1, 1, 1]

Testing test 2:
Result:  PASS
Input:  [0, 0, 0, 0]
Output:  [0, 0, 0, 0]
```

```
Expected Output:  [0, 0, 0, 0]

Testing test 3:
Result:  PASS
Input:  [1, 0, 1, 0]
Output:  [1, 0, 0, 1]
Expected Output:  [1, 0, 0, 1]

Testing test 4:
Result:  PASS
Input:  [0, 1, 0, 0]
Output:  [0, 1, 1, 0]
Expected Output:  [0, 1, 1, 0]

Testing test 5:
Result:  PASS
Input:  [1, 1, 0, 0]
Output:  [1, 1, 0, 1]
Expected Output:  [1, 1, 0, 1]
```

Listing 4.8: Test results when running the test vectors of the correct full adder program.

Now, let us introduce a simple bug to this program. We will add an extra Toffoli gate Listing 4.9, then walk through the testing and see how we can locate that bug if we don't know what it is.

```
def Quant_full_adder(qc, in_qbits,zero_qubit):
    qc.ccx(in_qbits[0],in_qbits[1],zero_qubit)
    qc.cx(in_qbits[0],in_qbits[1])
    qc.ccx(in_qbits[1],in_qbits[2],zero_qubit)
    qc.cx(in_qbits[1],in_qbits[2])
    qc.cx(in_qbits[0],in_qbits[1])
    qc.ccx(in_qbits[0],in_qbits[1],zero_qubit)
```

```
    return qc
```

Listing 4.9: A simple Python and Qiskit implementation of the full adder with an extra CCX gate.

The first step is running the same test for the program containing the bug. This will lead to similar results but different ones (Listing 4.10). When we examine the results, we see that the mismatch only occurs when both qubits $|A\rangle$ and $|B\rangle$ are 1.

```
Testing test 1:
Result:  FAIL
Input:  [1, 1, 1, 0]
Output:  [1, 1, 1, 0]
Expected Output:  [1, 1, 1, 1]

Testing test 2:
Result:  PASS
Input:  [0, 0, 0, 0]
Output:  [0, 0, 0, 0]
Expected Output:  [0, 0, 0, 0]

Testing test 3:
Result:  PASS
Input:  [1, 0, 1, 0]
Output:  [1, 0, 0, 1]
Expected Output:  [1, 0, 0, 1]

Testing test 4:
Result:  PASS
Input:  [0, 1, 0, 0]
Output:  [0, 1, 1, 0]
Expected Output:  [0, 1, 1, 0]

Testing test 5:
Result:  FAIL
```

```
Input:  [1, 1, 0, 0]
Output:  [1, 1, 0, 0]
Expected Output:  [1, 1, 0, 1]
```

Listing 4.10: Test results when running the test vectors of the full adder program containing the bug.

This hints that whatever is causing the error has something to do with the first two qubits. If we used the `gateLoc` function, we know that we have 3 Toffoli gates in the circuit instead of two and that the extra one is applied to the first two qubits. Using similar approaches can help to lead programmers to the source of the bug. In this case, since the error only occurs when the first two qubits are 1, we can assume that they may be the control of some gate, which is when the error occurs. We can know the multi-qubit gates applied to these two qubits using the `gateLoc(qc,' cx', qubits=['q[0]','q[1]'])` function, which will lead us to two Toffoli gates, and then we can remove each of them and test the circuit again. Doing so leads us to determine that the last Toffoli gate is the source of the error.

4.5.3 Testing and Debugging Phase Modulation (PM) Blocks

Testing and debugging AP blocks were relatively straightforward; however, moving on to the PM and AR blocks gets more challenging. This subsection will consider a strategy for debugging a PM block. The most straightforward PM block would be the QPE algorithm.

Before we dive into an example of a QPE with a bug, let us first discuss an approach to testing PM blocks. The strategy we propose here is to use the swap test [188].

The swap test is a fundamental quantum computing procedure used to determine the similarity between two quantum states. It is beneficial for measuring the inner product of two states, which can then be used to calculate their fidelity or similarity.

Consider two quantum states $|\psi\rangle$ and $|\phi\rangle$ that we want to compare. The swap test involves an ancillary qubit (the control qubit) initialized in the state $|0\rangle$ and the two states $|\psi\rangle$ and $|\phi\rangle$. The process of the swap test involves the following steps:

1. Apply a Hadamard gate to the control qubit, putting it into the superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.

2. Perform a CSWAP gate using the control qubit. The CSWAP gate swaps $|\psi\rangle$ and $|\phi\rangle$ only if the control qubit is in the state $|1\rangle$.

3. Apply another Hadamard gate to the control qubit.

4. Measure the control qubit. The probability of measuring $|0\rangle$ is given by $P(0) = \frac{1}{2} + \frac{1}{2}|\langle\psi|\phi\rangle|^2$.

The outcome of the measurement gives us information about the similarity of the two states. If the states are identical, the probability of measuring $|0\rangle$ in the control qubit will be 1. If they are orthogonal, the probability will be $\frac{1}{2}$.

In the swap test circuit, the first qubit is the control qubit, and the other two qubits are the states $|\psi\rangle$ and $|\phi\rangle$ being compared. The CSWAP operation is central to comparing the two states.

In this thesis, we use the swap test to determine if a phase difference exists between the two states. To do that let us consider $|\psi\rangle = |0\rangle + e^{i\theta_1}|1\rangle$ and $|\phi\rangle = |0\rangle + e^{i\theta_2}|1\rangle$. We can then recalculate the probability of measuring 1 or 0 as a function of θ_1 and θ_2 as follows:

The inner product of these two states is:

$$\langle\psi|\phi\rangle = \frac{1}{2}\left(1 + e^{i(\theta_2-\theta_1)}\right). \tag{4.5}$$

We need to find the magnitude squared of this inner product. Since $e^{i(\theta_2-\theta_1)}$ can be expressed as $\cos(\theta_2 - \theta_1) + i\sin(\theta_2 - \theta_1)$, and $\Delta\theta = \theta_2 - \theta_1$ we get:

$$\left|\frac{1}{2}\left(1 + e^{i(\Delta\theta)}\right)\right|^2 = \frac{1}{4}\left([1 + \cos(\Delta\theta)]^2 + [\sin(\Delta\theta)]^2\right). \tag{4.6}$$

The probability $P(0)$ is given by:

$$P(0) = \frac{1}{2} + \frac{1}{2}\left|\langle\psi|\phi\rangle\right|^2. \tag{4.7}$$

Substituting the magnitude squared into this, we get:

$$P(0) = \frac{1}{2} + \frac{1}{8}\left([1 + \cos(\Delta\theta)]^2 + [\sin(\Delta\theta)]^2\right). \tag{4.8}$$

$$P(0) = \frac{3}{4} + \frac{1}{4}\cos(\Delta\theta). \tag{4.9}$$

Since $P(1) = 1 - P(0)$, we substitute our expression for $P(0)$ and simplify:

$$P(1) = \frac{1}{4} - \frac{1}{4}\cos(\Delta\theta). \tag{4.10}$$

To practically implement the swap test in qiskit (Listing 4.11), we can calculate the inner product using the number of shots using the equation $s = 1 - \frac{2}{N}B$. s is the inner product, N is the total number of shots, and B is the number of times 1 was measured.

```
q = QuantumRegister(3, 'q')
c = ClassicalRegister(1, 'c')
circuit = QuantumCircuit(q, c)

circuit.h(q[0])
circuit.cswap(q[0], q[1], q[2])
circuit.h(q[0])

circuit.measure(q[0], c[0])

simulator = Aer.get_backend('qasm_simulator')
nShots = 8192
job = execute(circuit, simulator, shots=nShots)
counts = job.result().get_counts()

if '1' in counts:
    b = counts['1']
else:
    b = 0
s = 1 - (2 / nShots) * b
```

Listing 4.11: An Implementation of a simple swap test using Qiskit.

Since $P(1) = 1 - P(0)$ and $\frac{b}{nShots} = P(1)$, we have:

$$\frac{b}{nShots} = \frac{1}{4} - \frac{1}{4}\cos(\Delta\theta). \tag{4.11}$$

Since we are targeting both the NISQ and FT quantum computers, the number of shots is an essential factor to consider. A higher number of shots is a resource-expensive choice. However, we can estimate the optimal number of shots based on our device. We can calculate a confidence interval with the desired confidence level by collecting data from different devices.

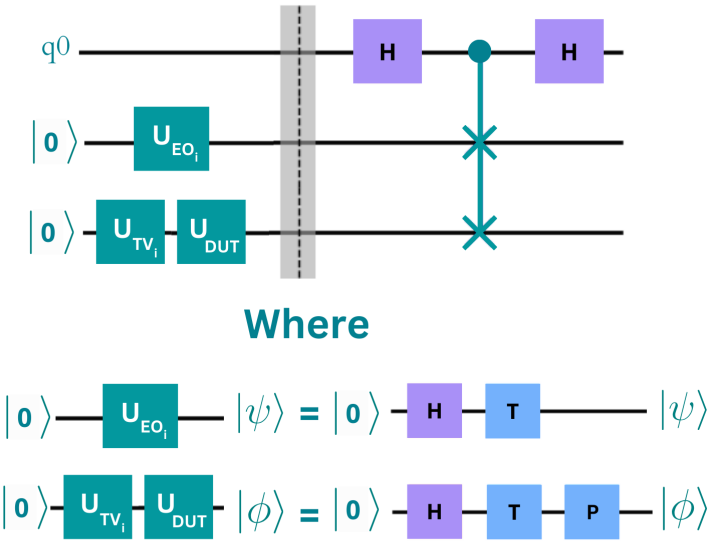


Figure 4.7: Using the swap test to detect the phase difference in a circuit containing a phase error p.

Let us assume we have the circuit shown in Figure 4.7 to see this better. In this circuit, we set our two states $|\psi\rangle$ and $|\phi\rangle$ of qubits 2 and 3 to the $|+\rangle$, and then we apply a T gate to both of them. To introduce an error, we added a phase gate to qubit 3.

Considering we do not know the value of the phase the P gate applies, we can execute the circuit and use the results to estimate that phase. In this case, if we use the code in Listing 4.11, we will end up with $s = 0.76$. We can use the equation $|\langle\psi|\phi\rangle|^2 = \frac{1}{2} + \frac{1}{2}\cos(\Delta\theta)$ to calculate the value of $\Delta\theta$.

$$0.76 = \frac{1}{2} - \frac{1}{2}\cos(\Delta\theta). \tag{4.12}$$

$$\frac{1}{2} = \cos(\Delta\theta). \tag{4.13}$$

Following this, we get $\Delta\theta = \frac{\pi}{3}$. This tells us that the phase gate in that circuit is applying a $\frac{\pi}{3}$, which we can then reverse to remove the error.

Generally, we can express $\delta\theta$ as a function of the squared inner product as:

$$\Delta\theta = cos^{-1}\left(2|\langle\psi|\phi\rangle|^2 - 1\right). \tag{4.14}$$

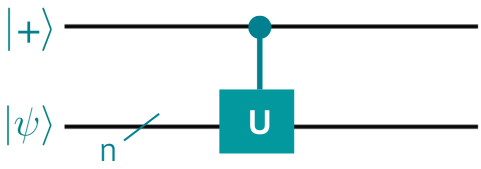


Figure 4.8: A general case of the swap test

Hence, if we know the value of the squared inner product, we can deduct the difference in phase between the two input states.

It is important to note that the Swap test is a special case of the circuit in Figure 4.8. If $|\psi\rangle$ is an Eigenstate of U, the entanglement will not be affected.

Detecting S and T Gate Errors

As discussed in Chapter 4.2, one of the most common errors in quantum circuits is missing or having an extra gate. Since we focus on the gates that alter the phase in this section, how can we use the swap test to detect errors due to additional or missing S to T gates?

Before we use the swap test to detect errors due to an S or T gate, let us take a step back and quickly revise the effect of those two gates.

- S Gate: Applies a phase of $\pi/2$ (quarter-turn). Mathematically, $S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$.
- T Gate: Applies a phase of $\pi/4$ (eighth-turn). Mathematically, $T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$.

We can use the swap test, as we did earlier in the section, to detect an error due to an S or T gate.

For an S Gate

The S gate introduces a phase shift of $\frac{\pi}{2}$:

$$\Delta\theta = \frac{\pi}{2}. \tag{4.15}$$

Substituting into the equation:

$$\frac{\pi}{2} = \cos^{-1}\left(2|\langle\psi|\phi\rangle|^2 - 1\right). \tag{4.16}$$

Since $\cos\left(\frac{\pi}{2}\right) = 0$:

$$0 = 2|\langle\psi|\phi\rangle|^2 - 1. \tag{4.17}$$

$$|\langle\psi|\phi\rangle| = \frac{1}{\sqrt{2}}. \tag{4.18}$$

For a T Gate
The T gate introduces a phase shift of $\frac{\pi}{4}$:

$$\Delta\theta = \frac{\pi}{4}. \tag{4.19}$$

Substituting into the equation:

$$\frac{\pi}{4} = \cos^{-1}\left(2|\langle\psi|\phi\rangle|^2 - 1\right). \tag{4.20}$$

Since $\cos\left(\frac{\pi}{4}\right) = \frac{1}{\sqrt{2}}$:

$$\frac{1}{\sqrt{2}} = 2|\langle\psi|\phi\rangle|^2 - 1. \tag{4.21}$$

$$|\langle\psi|\phi\rangle| = \sqrt{\frac{1}{2} + \frac{1}{2\sqrt{2}}}. \tag{4.22}$$

So, if we can calculate the squared inner product of input states, we can deduct if the circuit contains an extra T or S gate.

Effect of Applying Phase Gates to the Circuit

Another common mistake programmers make when implementing their circuits is applying a gate to the wrong qubit. In this section, we are concerned with gates that alter the phase. A phase gate $P(\theta)$ introduces a phase shift θ and is represented by the matrix:

$$P(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$$

For multi-qubit systems, applying a phase gate to a specific qubit affects the unitary matrix of the system. We will explore the effect of applying phase gates, such as the Z, S, and T gates, to different qubits in a multi-qubit system.

First, let us revise the unitaries of some phase gates:

- The Z gate introduces a phase shift of π:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

- The S gate introduces a phase shift of $\frac{\pi}{2}$:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

- The T gate introduces a phase shift of $\frac{\pi}{4}$:

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$$

For an n-qubit system, the matrix resulting from applying a phase gate $P(\theta)$ to qubit q is constructed using the tensor product with identity matrices. The resulting unitary U is:

$$U = I^{\otimes(q-1)} \otimes P(\theta) \otimes I^{\otimes(n-q)}, \tag{4.23}$$

where I is the identity matrix. This ensures that the phase gate only affects the specified qubit while leaving the others unchanged.

Consider a 2-qubit system. Applying a Z gate to qubit 0 (the least significant qubit) results in the following matrix:

$$U = Z \otimes I = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

Here, the unitary U shows that the states $|10\rangle$ and $|11\rangle$ acquire a phase shift of -1, while $|00\rangle$ and $|01\rangle$ remain unchanged.

But what if we have a 3-qubit system? For a 3-qubit system, applying an S gate to qubit 2 (the most significant qubit) results in:

$$U = I \otimes I \otimes S = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$$

This results in:

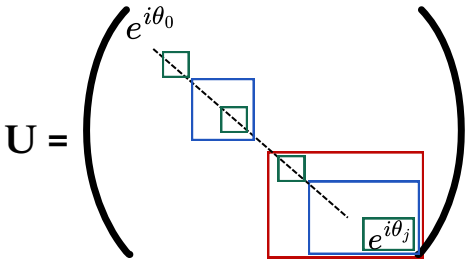


Figure 4.9: The unitary for a three-qubit phase slice. The colored boxes delineate terms affected by errors in the high-order (red), middle (blue), and low-order (green) qubits.

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & i & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & i & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & i & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & i \end{pmatrix}$$

In this matrix, the states $|1xx\rangle$ (where xx can be 00, 01, 10, or 11) acquire a phase shift of i.

From the above, we can reach a generalization of the effect of applying phase gates to different qubits, and we can see the change in the unitary in Figure 4.9.

When applying a phase gate to a specific qubit in an n-qubit system, the unitary matrix is modified such that the phase shift is applied to the diagonal elements where the corresponding qubit is in state $|1\rangle$. This can be generalized as follows:

$$U_{ii} = \begin{cases} e^{i\theta} & \text{if the } q\text{-th bit of } i \text{ is } 1 \\ 1 & \text{if the } q\text{-th bit of } i \text{ is } 0 \end{cases}$$

One approach to finding out if the bug in such circuits is caused by applying a phase gate to the wrong qubit is to examine the matrix resulting from applying that phase gate.

To detect errors in applying phase gates to a specific qubit q in an n-qubit system, we use the following steps:

1. Identify the target qubit q.
2. Prepare superposition states sensitive to phase changes on the target qubit.
3. Apply the phase gate to the intended qubit q.

4. Measure the resulting state to detect phase discrepancies.

We can create test cases to detect a particular case, which is applying the gate to the most significant qubit instead of the least significant qubit creating superposition states helps in detecting phase shifts more clearly. For an n-qubit system, use the Hadamard gate H to prepare superposition states. These states should isolate the effect of the phase gate on the target qubit.

Consider a 3-qubit system where we intend to apply a phase gate to qubit 0 (the least significant bit, LSB). The basis states are:

$$
\begin{aligned}
|\psi_1\rangle &= \frac{1}{\sqrt{2}}(|000\rangle + |100\rangle) \\
|\psi_2\rangle &= \frac{1}{\sqrt{2}}(|001\rangle + |101\rangle) \\
|\psi_3\rangle &= \frac{1}{\sqrt{2}}(|010\rangle + |110\rangle) \\
|\psi_4\rangle &= \frac{1}{\sqrt{2}}(|011\rangle + |111\rangle)
\end{aligned}
$$

We then apply the Z gate to qubit 0, measure the resulting states, and compare them to the expected outcomes to detect any phase discrepancies.

- **Correct Application to Qubit 0**:

$$
\begin{aligned}
|\psi_1\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |100\rangle) &\rightarrow \frac{1}{\sqrt{2}}(|000\rangle - |100\rangle) \\
|\psi_2\rangle = \frac{1}{\sqrt{2}}(|001\rangle + |101\rangle) &\rightarrow \frac{1}{\sqrt{2}}(|001\rangle - |101\rangle) \\
|\psi_3\rangle = \frac{1}{\sqrt{2}}(|010\rangle + |110\rangle) &\rightarrow \frac{1}{\sqrt{2}}(|010\rangle - |110\rangle) \\
|\psi_4\rangle = \frac{1}{\sqrt{2}}(|011\rangle + |111\rangle) &\rightarrow \frac{1}{\sqrt{2}}(|011\rangle - |111\rangle)
\end{aligned}
$$

- **Incorrect Application to Qubit 1** (The middle qubit in the register):

$$|\psi_1\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |100\rangle) \rightarrow \frac{1}{\sqrt{2}}(|000\rangle + |100\rangle)$$
$$|\psi_2\rangle = \frac{1}{\sqrt{2}}(|001\rangle + |101\rangle) \rightarrow \frac{1}{\sqrt{2}}(|001\rangle + |101\rangle)$$
$$|\psi_3\rangle = \frac{1}{\sqrt{2}}(|010\rangle + |110\rangle) \rightarrow \frac{1}{\sqrt{2}}(|010\rangle + |110\rangle)$$
$$|\psi_4\rangle = \frac{1}{\sqrt{2}}(|011\rangle + |111\rangle) \rightarrow \frac{1}{\sqrt{2}}(|011\rangle - |111\rangle)$$

Extension to n-Qubit Systems

The same principle can be applied to systems with more qubits. For an n-qubit system, follow these steps:

1. Use Hadamard gates to create superpositions on the target qubit.
2. Apply the phase gate to the qubit of interest.
3. Measure the resulting states and compare them to the expected outcomes to detect any phase shifts.

For an n-qubit system, prepare superposition states such as:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle_q \otimes |X\rangle_{n-1} + |1\rangle_q \otimes |X\rangle_{n-1})$$

where $|X\rangle_{n-1}$ represents the superposition of the remaining $n-1$ qubits in various basis states. By measuring the resultant phase shifts and comparing them to the expected values, you can determine whether the phase gate was applied correctly.

Using superposition states to detect errors in applying phase gates is simple and scalable to larger systems while leveraging the sensitivity of superposition states to phase changes, providing a robust method for error detection in quantum circuits.

4.5.4 Testing and Debugging Amplitude Redistribution (AR) Blocks

Testing and debugging AP circuits was relatively straightforward; the challenge arises when we deal with AR circuits. Let us consider the implementation of the W state proposed in [159] to walk through an example of debugging an AR circuit. A general W

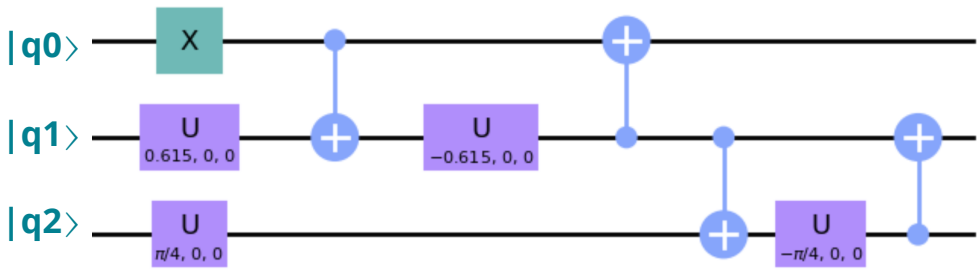


Figure 4.10: The circuit constructing a 3-qubit W state.

state Python and Qiskit function can be seen in Listing 4.12, which, if wanted to create a 3-qubit W state, will result in the circuit in Figure 4.10.

```
def cg (qcir,cQbit,tQbit,theta):
    theta_dash = math.asin(math.cos(math.radians(theta/2)))
    qcir.u(theta_dash,0,0,tQbit)
    qcir.cx(cQbit,tQbit)
    qcir.u(-theta_dash,0,0,tQbit)
    return qcir


def wn (qcir,qbits):
    for i in range(len(qbits)):
        if i == 0:
            qcir.x(qbits[0])
        else:
            p = 1/(len(qbits)-(i-1))
            theta = math.degrees(math.acos(math.sqrt(p)))
            theta = 2* theta
            qcir = cg(qcir,qbits[i-1],qbits[i],theta)
            qcir.cx(qbits[i],qbits[i-1])
```

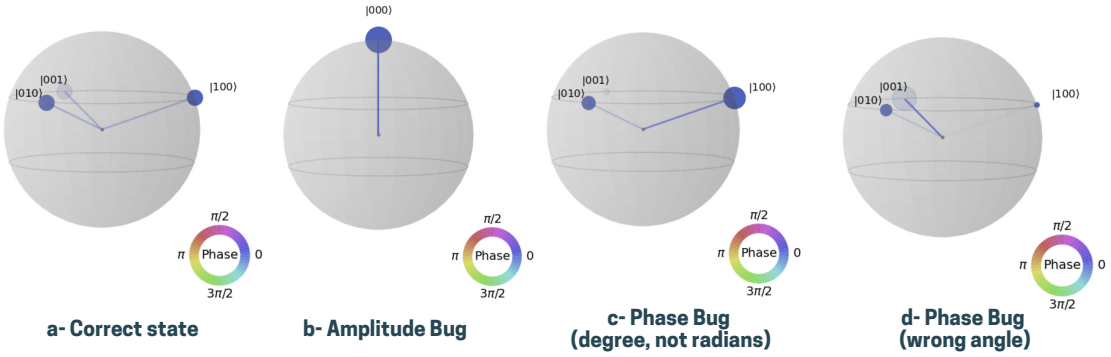


Figure 4.11: The different Q-sphere outputs for the 3-qubit W state.

```
19     return qcir
```

Listing 4.12: An Implementation of the W state as described in [159].

When considering AR programs, different types of bugs can occur that we do not encounter in classical programming. From that perspective, we can categorize the bugs introduced to quantum programs into two types: ones that only affect the amplitude of the qubits and those that affect the phase (in terms of amplitude).

To explain that better, let us attempt injecting two types of bugs into the W state implementation in Listing 4.12. To do that, let us first consider the correct output of that circuit. In the case of three qubits, the output of the W state should follow the equation 4.24.

$$|W_n\rangle = \frac{1}{\sqrt{n}}(|100...0\rangle + ... + |01...0\rangle + |00...01\rangle) \tag{4.24}$$

Hence, if $n = 3$, the result of the program should be $0.58|001\rangle + 0.58|010\rangle + 0.58|100\rangle$. Though only considering the state vector of the results can provide us with some information, having a visual representation of the state of the qubits can provide better information on the phase about the phase. That can be done by displaying the Bloch sphere of the state of the three qubits 4.11-a.

```
1  test_cases = [
2     {
3          "name": "W State Test 1",
4          "input": [1, 0, 0, 0, 0, 0, 0, 0],
5          "expected_output": [0, 1/np.sqrt(3), 1/np.sqrt(3), 0, 1/np.
       sqrt(3), 0, 0, 0]
6      },
```

```
    {
        "name": "W State Test 2",
        "input": [0, 0, 0, 0, 0, 0, 0, 1],
        "expected_output": [0, 0, 0, 0.71, 0, -0.71, 0, 0]
     }
]
```

Listing 4.13: Test vectors for the three-qubit W state.

We can run two of the sample tests offered by Cirquo for this program Listing 4.13 to ensure that it works as expected by using the *fQuantTester* function Listing 4.14.

```

W State Test 1:
Result:  PASS
Input:  1.00|000>
Output:  0.58|001> + 0.58|010> + 0.58|100>
Expected Output:  0.58|001> + 0.58|010> + 0.58|100>


W State Test 2:
Result:  PASS
Input:  1.00|111>
Output:  0.71|011> + -0.71|101>
Expected Output:  0.71|011> + -0.71|101>
```

Listing 4.14: Test results when running the test vectors of the correct three-qubit W state program.

We can start injecting some bugs, starting with amplitude "classical" bugs. "Classical" bugs, in that sense, are like adding or missing a gate that performs a classical operation. In our example, one way to do so would be to remove the NOT gate at the beginning of the circuit.

By removing the NOT gate at the beginning of the circuit, the program's results will be $1.00|000\rangle$, and the Q-Sphere representation of the different qubits can be seen in Figure 4.11-b. As shown in the figure, removing this led to the wrong amplitude (the probability of measuring the state– the size of the point of the vector) in the results. If we run the tests on this circuit, both tests will fail.

To locate the source of the error here, we can use the process of elimination with the functions offered by Cirquo. To start, we can use the *gateLoc* function to count the different gates in the circuit. This would only be helpful if the programmer knows how the correct circuit is supposed to look. However, if we consider the two tests $|000\rangle$ and $|111\rangle$, we see when the input is $|000\rangle$, the output is the same as the input, as in the gates in the circuit was not activated.

We can start by assuming that the source of the error is an initialization problem. Hence, we can run the circuit with three different inputs with each qubit activated, making the states $|001\rangle$, $|010\rangle$, and $|001\rangle$. Then, use the fQuantAnalyzer function to run these tests, Which leads to the results in Listing 4.15. From there, we can see that when the first qubit is in state $|1\rangle$, the results are correct; hence, if we add a NOT gate on that qubit at the beginning of the circuit and rerun the original tests, we will see that they both pass. From that, we can conclude that the bug was due to a missing NOT gate on qubit 0.

```
Testing W State Test 1:
Name: W State Test 1
Input: 1.00|100>
Output: 0.58|100> + 0.58|010> + 0.58|001>


Testing W State Test 2:
Name: W State Test 2
Input: 1.00|010>
Output: 0.71|110> + 0.71|101>


Testing W State Test 3:
Name: W State Test 3
Input: 1.00|001>
Output: 1.00|011>
```

Listing 4.15: Test results when running the test vectors of the three-qubit W state program with amplitude bug to locate the bug.

Though some gates can be considered "classical" on a higher level, their implementation may not be. For example, we can consider the CNOT or Toffoli gates as classical

gates (as we did in Subsection 4.5.2); however, the decomposition of those gates may often—contain phase gates. So, if we are testing an implementation of these gates, we must ensure that the overall program does not introduce phase to the qubits at the end of the circuit.

Often, when a bug is introduced through a quantum gate (phase or Hadamard), it affects both the final amplitude redistribution of the state of the qubits and their phase. Returning to our example of the three-qubit W state, in line 2 of Listing 4.10, we can introduce a phase error in two different ways. We can enter the phase for *theta_dash* in degrees instead of radians or by passing the wrong angle to the cos function.

```
W State Test 1:
Result:  FAIL
Input:  1.00|000>
Output:  0.89|001> + 0.42|010> + 0.18|100>
Expected Output:  0.58|001> + 0.58|010> + 0.58|100>

W State Test 2:
Result:  FAIL
Input:  1.00|111>
Output:  0.38|011> + -0.92|101>
Expected Output:  0.71|011> + -0.71|101>
```

Listing 4.16: Test results when running the test vectors of the three-qubit W state program with phase error in degrees instead of radians corresponding to Figure 4.11-c.

If we use degrees instead of radians, we get $-0.24|001\rangle + 0.51|010\rangle + 0.83|100\rangle$, and the final states of the qubits are shown in Figure 4.11-c, whereas if we pass the wrong angle—$\theta/4$ instead of $\theta/2$—we get $0.89|001\rangle + 0.42|010\rangle + 0.18|100\rangle$, and the final state of the qubits would look like Figure 4.11-d.

When we consider the case where the angle of the function was passed using degrees instead of radians and use the testing function with the same test as before, it leads to both tests failing Listing 4.16. If we examine the Q-Sphere, we see that state $|001\rangle$ has a different color than the other two states, which means it has a different phase. From this, we can assume that the error is phase-related. In a Q-Sphere, each line represents an answer state. The color of the line represents the state's phase, and the dot on the sphere's surface represents the probability amplitude of measuring the state. The implementation of the W state used here has two angles: *theta* and *theta_dash*. Since *theta_dash* depends

on *theta*, we can start by analyzing the lines of code containing *theta*. Doing so leads us to notice the missing conversion to radians in line 2 Listing 4.12.

4.6 Overview of Cirquo

The tool discussed in this thesis is built using Python on top of the Qiskit module. In Qiskit, any quantum circuit is built using an object class QuantumCircuit. Any QuantumCircuit object can contain QuantumRegisters, ClassicalRegisters, different quantum gates, and measurement operations. The Qiskit QuantumCircuit object contains many proprieties and characteristics. In order to build our tool, we extended this class to include a few new commands to include breakbarriers to cut the circuit and gate tracking option. In addition to these, we added new functionalities to perform horizontal and vertical slicing. We can divide the functionality the debugging tool adds to Qiskit into two categories, methods added to the QuantumCircuit class and the debugger's core functionality.

Methods Added to The QuantumCircuit Class

Since all quantum circuits that can be built using Qiskit use the QuantumCircuit object, we decided to extend that class to include the debugging-needed methods instead of creating a whole new type. That was accomplished by adding two methods:

1. `breakbarrier()`: a new object type based on Qiskit's barrier class that is used to pinpoint where the tool is going to cut the circuit when using the Vertical tool function (`VSlicer`).

2. `gateInfo()`: a method that, when the debugging mode is enabled, is used to store information about all gates added to the circuit. The information is the gate type, the number of occurrences, and where in the code this gate was added to the circuit.

The Core Functionality of Cirquo

Cirquo is built using Python on top of the Qiskit package[1]. In Qiskit, any quantum circuit is built using the object class QuantumCircuit. Any QuantumCircuit object can contain QuantumRegisters, ClassicalRegisters, different quantum gates, and measurement operations. The Qiskit QuantumCircuit object contains many properties and characteristics.

[1]Cirquo is built on 'qiskit-terra': '0.25.2', 'qiskit': '0.44.2', 'qiskit-aer': '0.12.0', 'qiskit-ignis': '0.7.1', 'qiskit-ibmq-provider': '0.20.2', 'qiskit-nature': '0.6.2'

In order to build our tool, we extended this class to include a few new commands to include breakbarriers to cut the circuit and gate tracking options.

The methods added to the QuantumCircuit class and the new functions offered by Cirquo are:

1. `breakbarrier()`: a new object type based on Qiskit's barrier class that is used to pinpoint where the tool is going to cut the circuit when using the Vertical tool function (`VSlicer`).

2. `gateInfo()`: a method that, when the debugging mode is enabled, is used to store information about all gates added to the circuit. The information is the gate type, the number of occurrences, and where in the code this gate was added to the circuit.

3. `startDebug()`: This function enables the debugging mode by extending the QuantumCircuit Class to include both *breakbarrier* and *gateInfo* methods.

4. `endDebug()`: This function disables the debugging mode.

5. `VSlicer()`: This function takes a QuantumCircuit object that contains breakbarriers and then divides the circuit based on the location of those *breakbarrier* and returns the original circuit as well as a list of subcircuits corresponding to the circuit dividing based on the *breakbarrier* locations.

6. `HSlicer()`: This function removes unused qubits or QuantumRegisters from a subcircuit after using the vertical slicer.

7. `gateLoc()`: This function takes a circuit or a subcircuit, a gate, and a qubit or a list of qubits (optional), and displays how many times and where in the code this gate was added to the circuit.

8. `catCircuit()`: This function takes a circuit and, based on its size, unitary, and the containing gates, categorizes it into either Amplitude Permutation, Phase Modulation, or Amplitude Redistribution.

9. `pClassAnalyzer()`: This function is used to run tests on Amplitude Permutation circuits to observe the circuit's behavior.

10. `fQuantAnalyzer()`: This function is used to run tests on Phase Modulation and Amplitude Redistribution circuits to observe the circuit's behavior.

11. `pClassTester()`: This function is used to test Amplitude Permutation blocks.

12. `fQuantTester()`: A function to run tests on Phase Modulation and Amplitude Redistribution blocks.

13. `applySwapTest()`: A function to apply the swap test and calculates $\delta\theta$ for Phase Modulation blocks.

The API implementation of Cirquo can be found in Appendix C. Once we obtain the smaller circuits from the slicer—or already have the circuit—and know its categorization, we need to start testing it and observe its behavior. Cirquo offers two testing functions, `pClassTester` and `fQuantTester`, to run tests on Amplitude Permutation, Phase Modulation, and Amplitude Redistribution circuits, respectively.

The `pClassTester` allows the programmer to pass the test cases as a list of classical values of the qubits—that is, 0 or 1—that input is converted to a state vector before running the circuit. That is to make testing the classical circuits more intuitive than how we test classical software. For example, if a programmer needs to test a quantum full-adder circuit and wants to test the circuit results if the inputs were 1, 1 and 1, they can pass the `pClassTester` a list [1,1,1].

The `fQuantTester` takes a list of state vectors and their corresponding outputs and runs the circuit accordingly. Testing Phase Modulation and Amplitude Redistribution circuits can be challenging; hence, programmers often don't know how to test these circuits. Cirquo also contains implementation and sample test vectors for common building blocks in quantum algorithms.

If a programmer wants to test an Amplitude Permutation circuit, they can describe the inputs and corresponding outputs as state vectors. For example, if we were to test a simple Bell pair generator, we would need to format the input and output tests as state vectors. So, in the case of a 2-qubit Bell pair generator circuit consisting of a Hadamard gate and a CNOT gate, if the initial state of the qubits is $|0\rangle$ and $|0\rangle$, then the results should be $0.71|00\rangle + 0.71|11\rangle$. To prepare that for the tester, the input state will be $[1, 0, 0, 0]$ because the state vector of 2 qubits consists of 2^2 cases. In addition to the test samples, programmers can also use the pre-defined functions of these building blocks to assist them in constructing their circuits.

A summary of the methods and functions added to Qiskit to allow the tool to function in Table 4.3.

Understanding how the circuit behaves is the key to making the testing and debugging process more efficient. Therefore, Cirquo offers some functions that provide information to make testing the circuit a better experience for the programmer.

When an error occurs while executing a circuit on actual hardware, it is often due to the hardware (machine-related) or the quantum logic (algorithm and implementation). Machine-related errors are low-level hardware-specific faults such as gate errors, readout errors, thermal relaxation errors, measurements errors, etc. [189, 190, 191]. Fixing these errors requires implementing quantum error correction and some efficient error minimizing technique [192] (or getting very lucky on a given run). However, only quantum logic errors can occur if the circuit is simulated.

Quantum logic errors are more difficult to resolve in quantum circuits than in classical programs, partly because most quantum algorithms select an outcome from a probability

Table 4.3: Cirquo API Overview.

Instruction	Type	Parameters	Description
breakbarrier	QuantumCircuit attribute	None	Used on a QuantumCircuit object to mark where the circuit needs to be sliced.
gateInfo	QuantumCircuit attribute	None	Keep track of all gates added to the circuit when debugging mode is enabled.
VSlicer	Function	QuantumCircuit	Divides the QuantumCircuit into a list of sub-circuits vertically cut based on the breakbarriers added to the QuantumCircuit object.
HSlicer	Function	QuantumCircuit	Removes unused regiters from the QuantumCircuit or slices.
startDebug	Function	None	Starts the debugging mode by adding brakbarrier and gatInfo to the Qiskit QuantumCircuit Class.
endDebug	Function	None	Ends the debugging mode by enabling the original Qiskit QuantumCircuit Class.
gateLoc	Function	QuantumCircuit, Gate, Qubits	Returns the number of times and line of code the given gate is added to the QuantumCircuit.
catCircuit	Function	QuantumCircuit	Categorizes the circuit.
pClassAnalyzer	Function	QuantumCircuit, Tests	Run different inputs Amplitude Permutation circuits.
fQuantAnalyzer	Function	QuantumCircuit, Tests	Run different inputs on Phase Modulation and Amplitude Redistribution circuits.
pClassTester	Function	QuantumCircuit, Tests	Run and validate tests on pseudo-classical circuits.
fQuantTester	Function	QuantumCircuit, Tests	Run and validate tests on Amplitude Permutation circuits.

distribution at the end of a run and partly because of their use of superposition, entanglement, and interference. Thus, isolating the circuit section containing the error requires careful reasoning and narrowing down operations to make the error as reproducible and visible as possible.

One aspect of this is the need for the programmer to work both forward and backward through the toolchain, examining the circuit at the gate level and the higher-level functions that generated the circuit. Cirquo allows the programmer to track where in the code each gate was added to the circuit.

Chapter 5

Usage Examples

In the previous chapter, we proposed the ideas and strategies for testing and debugging quantum circuits implemented in Cirquo. Building upon that foundation, this chapter will delve into practical walkthrough examples, demonstrating how these methodologies can be applied to some well-known quantum algorithms. We will illustrate how the proposed tools—the circuit slicer, categorizer, and various testing strategies—can be effectively utilized to enhance the development process of quantum software.

This chapter provides a step-by-step guide on dissecting and understanding the complexities of quantum circuits through detailed examples showing how the circuit slicer can break down large quantum circuits into smaller, more manageable units, facilitating easier analysis and debugging. We will also demonstrate how the circuit categorizer helps identify critical components of the circuit that significantly affect the overall computation.

5.1 Example 1: Grover's Algorithm for The Triangle Finding Problem

To better understand how we can Cirquo can help developers understand their circuits and locate errors in them, let us consider an example of an implementation of Grover's algorithm applied to the triangle finding problem, where a graph is given. We try to find a 3-node complete graph within the larger graph (*a triangle*). This problem has been addressed both classically [141], [142] and quantumly [143],[144]. Although there are various ways this problem can be solved quantumly, using Grover is one of the most straightforward approaches, as presented in Chapter [**?**]

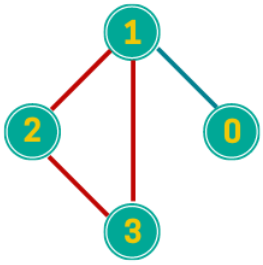


Figure 5.1: A 4-node graph with a triangle between nodes 1,2 and 3.

For this example problem, consider the 4-node graph in Figure 5.1. As mentioned earlier, we need three algorithmic steps to implement Grover's algorithm: stage preparation, the oracle, and the diffusion operator. The oracle and diffusion operator need to be repeated a total number of `opt_iter` cyclically. This optimal number of times depends on two factors, the size of the search space N and the number of answers for our problem m (how many triangles in the graph, in this case, one) and is calculated using $opt\_iter = \left\lfloor \frac{\pi}{4}\sqrt{\frac{N}{m}} \right\rfloor$ [138].

In our example, we will limit the search space using a W state preparation followed by NOT gates. Hence, the search space will be states containing three 1s ($|1110\rangle, |0111\rangle, ..., |1011\rangle$. Using that formula and in case of $m = 1$, `opt_iter` will be 1.

To implement this circuit, we need 6 ancillary qubits and 3 qubits flag that will be set to $|111\rangle$ only if a triangle is found in the graph. We can write this using Python and Qiskit as shown in Listing 5.1.

```
from qiskit, import QuantumRegister, ClassicalRegister,
    QuantumCircuit
import numpy as np
import math as m
def grover():
    n_nodes = 4
    N = 2**n_nodes  # Hilbert space size
    #Defone needed qubits
    nodes_qubits = QuantumRegister(n_nodes, name='nodes')
    ancilla = QuantumRegister(6, name = 'anc')
```

```
    flag = QuantumRegister(3, name='check_qubits')
    class_bits = ClassicalRegister(n_nodes, name='class_reg')
    tri_flag = ClassicalRegister(3, name='tri_flag')
    qc = QuantumCircuit(nodes_qubits, ancilla, flag, class_bits,
tri_flag)
    # Initialize quantum flag qubits in |-> state
    qc.x(flag[2])
    qc.h(flag[2])
    # Initializing i/p qubits in superposition
    qc.h(nodes_qubits)
    # Calculate optimal iteration count
    iterations = round(m.pi/4.sqrt(N))
    #in case of debugging, we will make iteration = 1
    for i in np.arange(iterations):
        oracle(n_nodes, qc, nodes_qubits, ancilla, flag)
        diffusion(qc, nodes_qubits, ancilla)
        qc.breakbarrier() #for debugging only, must be used after
startDebug() is called
    return qc
```

Listing 5.1: Python and Qiskit code implementing Grover's algorithm for the triangle finding problem.

First, we will run the algorithm using 5000 shots. What we expect to get is `'|0111>': 5000`, however, we got `'|1111>': 1285, '|0110>': 1239, '|0101>': 1256, '|0011>': 1220` indicating that there is a bug in the algorithm.

Now, we need to debug the circuit.

Step 1: Slicing the circuit

To do that, we will first need to access debugger mode by calling the `startDebug` function to be able to use the **breakbarrier** and `Vslicer` function with the mini mode.

Then, we will slice the circuit using the `VSlicer` function based on the algorithmic steps shown in Figure 5.2, resulting in three subcircuits.

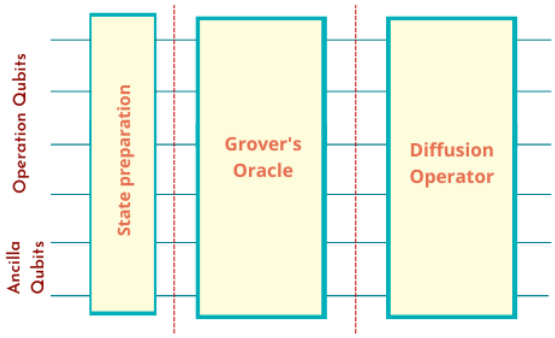


Figure 5.2: Slicing the circuit according to the algorithmic steps

Step 2: Categorize the slices

Having the subcircuit, we can pass them through the `catCircuit` function to categorize them according to their behavior. The categorize will return the following categorizations:

- The state prep » "Amplitude Redistribution".
- The oracle » "Amplitude Permutation".
- The diffusion » "Amplitude Redistribution".

Knowing the categories of the different slices will assess us in the testing and debugging process.

Step 3: Testing the different Slices

The goal is to test each slice and ensure they perform as expected.

Let us start with the state prep, which is the W state. The W state is an AR circuit, so we will use the `fQuantTester` function to test it and the `generate_w_state_test_cases(4)` to generate test cases to use, which will print the results in Listing 5.2.

```
Testing test 0:
Result:  PASS
Input:  1.00|0000>
Output:  0.50|1000> + 0.50|0100> + 0.50|0010> + 0.50|0001>
Expected Output:  0.50|1000> + 0.50|0100> + 0.50|0010> + 0.50|0001>

```

```
Testing test 1:
Result:  PASS
Input:  1.00|1111>
Output:  -0.41|1010> + 0.41|1001> + 0.82|1111>
Expected Output:  -0.41|1010> + 0.41|1001> + 0.82|1111>

Testing test +:
Result:  PASS
Input:  0.25|0000> + 0.25|1000> + .... + 0.25|0111> + 0.25|1111>
Output:  0.25|0000> + 0.34|1000> + .... + 0.13|0111> + 0.35|1111>
Expected Output:  0.25|0000> + 0.34|1000> + .... + 0.13|0111> +
    0.35|1111>

Testing test -:
Result:  PASS
Input:  0.25|0000> + -0.25|1000> + .... + -0.25|0111> + 0.25|1111>
Output:  -0.25|0000> + -0.09|1000> + .... + 0.08|0111> + 0.06|1111>
Expected Output:  -0.25|0000> + -0.09|1000> + .... + 0.09|1011> +
    0.08|0111> + 0.06|1111>

Testing test i:
Result:  PASS
Input:  0.25|0000> + 0.00+0.25j|1000> + .... + 0.00-0.25j|0111> +
    0.25|1111>
Output:  -0.00+0.25j|0000> + 0.12+0.22j|1000> + .... + -0.10+0.23j
    |0111> + 0.20-0.14j|1111>
Expected Output:  -0.00+0.25j|0000> + 0.12+0.22j|1000> + .... +
    -0.10+0.23j|0111> + 0.20-0.14j|1111>

Testing test -i:
Result:  PASS
Input:  0.25|0000> + 0.00-0.25j|1000> + .... + 0.00+0.25j|0111> +
    0.25|1111>
```

```
Output:  -0.00-0.25j|0000> + 0.12-0.22j|1000> + .... + -0.10-0.23j
    |0111> + 0.20+0.14j|1111>
Expected Output:  -0.00-0.25j|0000> + 0.12-0.22j|1000> + .... +
    -0.10-0.23j|0111> + 0.20+0.14j|1111>
```

Listing 5.2: Test results for the W state.

The passing tests tell us that the bug is not likely in the state preparation, so we move on to test the oracle.

The oracle circuit is packaged in a function that aims to mark the correct answer ($|0111\rangle$). Based on the categorization, the oracle is an AP circuit, so we can use the `pClassTester` function to test it. To do that, however, we need to create some test cases. The oracle's job is to count the edges for each state based on the input graph. If the state represents a triangle in the graph, flag qubits will be 1. The set of tests we will use is in Listing 5.3.

```
    test_cases_ora = [
    {
        "name": "test 1",
        "input": [0,1,1,1,0,0,0,0,0],
        "expected_output": [1,1,1,0,0,0,0,0,1]
    },
    {
        "name": "test 2",
        "input": [1,1,1,1,0,0,0,0,0],
        "expected_output": [1,1,1,1,0,0,0,0,0]
    },
    {
        "name": "test 3",
        "input": [1,1,0,0,0,0,0,0,0],
        "expected_output": [1,1,0,0,0,0,0,0,0]
    }
]
```

Listing 5.3: Tests for the oracle.

We can then pass those tests to the `pClassTester` function along with our oracle, which leads to the results in Listing 5.10 showing that the tests pass.

```
Testing test 1:
Result:  PASS
Input:  [0, 1, 1, 1, 0, 0, 0, 0, 0]
Output:  [1, 1, 1, 0, 0, 0, 0, 0, 1]
Expected Output:  [1, 1, 1, 0, 0, 0, 0, 0, 1]

Testing test 2:
Result:  PASS
Input:  [1, 1, 1, 1, 0, 0, 0, 0, 0]
Output:  [1, 1, 1, 1, 0, 0, 0, 0, 0]
Expected Output:  [1, 1, 1, 1, 0, 0, 0, 0, 0]

Testing test 3:
Result:  PASS
Input:  [1, 1, 0, 0, 0, 0, 0, 0, 0]
Output:  [1, 1, 0, 0, 0, 0, 0, 0, 0]
Expected Output:  [1, 1, 0, 0, 0, 0, 0, 0, 0]
```

Listing 5.4: The results of testing the oracle.

The last part of the circuit we need to test is the diffusion operator. Since we have already tested the other parts of the circuit and made sure that they function correctly, we can confidently say that the error occurred with the diffusion operator. The diffusion operator is added to the circuit to execute a rotation about the average to increase the probability of measuring the correct answer or the answer marked by the oracle. Generally, if we used the Hadamard gates as state preparation, the diffusion operator should look as shown in Figure 5.3-A. For our example, the diffusion operator code is shown in Listing 5.5

```
def grover_diff(qc, nodes_qubits,edge_anc,ancilla,stat_prep,
    inv_stat_prep):
    qc.append(inv_stat_prep,qargs=nodes_qubits)
    qc.x(nodes_qubits)
    #=====================================================
        #3 control qubits Z gate
    create_3cz_circuit(qc,len(nodes_qubits)-1,nodes_qubits[::-1],
```

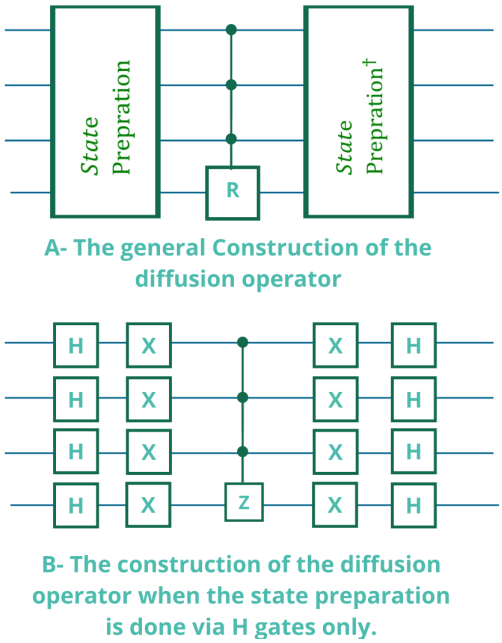


Figure 5.3: The general construction of the diffusion operator.

```
    ancilla)
7   #=====================================================
8   qc.x(nodes_qubits)
9   qc.append(stat_prep,qargs=nodes_qubits)
```

Listing 5.5: The function constructing the diffusion operator (that results in an error due to an extra NOT gate in line 7).

We need to look at the definition of the diffusion operator. Mathematically, the diffusion operator D is defined as D = $state\_prep$ R $state\_prep^{\dagger}$, where R is a zero reflection or a zero-phase shift. This phase shift can be calculated by $R = 2|0\rangle^{\otimes n}\langle 0|^{\otimes n} - I_n$, where I_n is the identity matrix on n qubits [41, 193, 194]. When we examine this equation and try to test the function in Listing 5.5 correctly, we implement it.

Since we already tested the state prep, we can use the `VSlicer` to slice the diffusion even further so we can focus on the implementation of the $C^{\otimes 3}Z$ gate (Listing 5.6).

```
1   def create_3cz_circuit(qc):
2
3   c3z_circuit = QuantumCircuit(4, name='C3Z')
4
```

```
    # Add the gate sequence for the C3Z gate
    c3z_circuit.h(3)  # Hadamard gate to transform Z to X basis
    c3z_circuit.mct([0, 1, 2], 3)  # Multi-controlled Toffoli gate
    c3z_circuit.h(3)  # Hadamard gate back to original basis
    c3z_circuit.z(3)
    # Convert the circuit to a gate
    c3z_gate = c3z_circuit.to_gate()

    # Append the C3Z gate to the provided quantum circuit
    qc.append(c3z_gate, [0, 1, 2, 3])

    return qc
```

Listing 5.6: The implementation of the $C^{\otimes 3}Z$ gate.

The diffusion part is an AR circuit; however, passing the CZ circuit to the categorizer will show that it is a PM circuit. First, we need to create some tests for the $C^{\otimes 3}Z$ circuit. The circuit should only apply a phase of π to the target qubit when all control qubits are 1. Accordingly, we can see the tests for this in Listing 5.7.

```
    test_cases_cnz = [
    {
        "name": "test 1",
        "input": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                  0, 0, 0, 0, 1],
        "expected_output": [0, 0, 0, 0, 0, 0, 0, 0, 0,
                            0, 0, 0, 0, 0, 0, -1]
    },
    {
        "name": "test 2",
        "input": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                  0, 0, 1/np.sqrt(2), 1/np.sqrt(2)],
        "expected_output": [0, 0, 0, 0, 0, 0, 0, 0, 0,
                            0, 0, 0, 0, 0, 1/np.sqrt(2), -1/np.sqrt
    (2)]
    },
```

```
    {
        "name": "test 3",
        "input": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                  0, 0, 0, 0, 1/np.sqrt(2), -1/np.sqrt(2)],
        "expected_output": [0, 0, 0, 0, 0, 0, 0, 0,
                            0, 0, 0, 0, 0, 0, 1/np.sqrt(2), 1/np.sqrt
    (2)]
    }
]
```

Listing 5.7: The tests for the control Z function.

All tests fail when we run those tests on implementing the $C^{\otimes 3}Z$, as shown in Listing 5.8.

```
Test Results:

Testing test 1:
Result:  FAIL
Input:  1.00|1111>
Output:  -1.00|1110>
Expected Output:  -1.00|1111>

Testing test 2:
Result:  FAIL
Input:  0.71|0111> + 0.71|1111>
Output:  0.71|0110> + -0.71|1110>
Expected Output:  0.71|0111> + -0.71|1111>

Testing test 3:
Result:  FAIL
Input:  0.71|0111> + -0.71|1111>
Output:  0.71|0110> + 0.71|1110>
Expected Output:  0.71|0111> + 0.71|1111>
```

Listing 5.8: The results of testing the $C^{\otimes 3}Z$ circuit.

Now that we know that the source of the bug is in the $C^{\otimes 3}Z$ implementation, we can use the Swap test to determine the phase difference between the expected results and the results we obtained. The test circuit we use is shown in Figure 5.4, corresponding to the code in Listing 5.9.

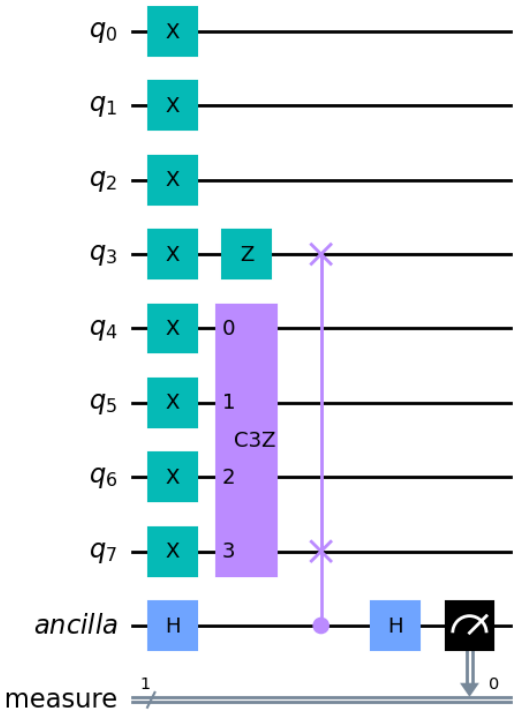


Figure 5.4: Using the swap test on the $C^{\otimes 3}Z$ circuit

The Swap test tells us that the phase difference is π, which means an extra Z gate or a P gate with phase π applied to the target qubit in implementing the $C^{\otimes 3}Z$ gate. To figure that out, we use the `gateLoc` function, which leads us to an extra Z gate in the `create_3cz_circuit`. Removing that Z gate and rerunning the algorithm will produce the correct results.

```
#Prepare test circuit
qc1 = QuantumCircuit(4)
#initialize([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1])
qc1.x([0,1,2,3])
qc1.z(3)
#prepare the C3Z circuit
```

```
qc2 = QuantumCircuit(4)
qc2.x([0,1,2,3])
qc2 = create_3cz_circuit(qc2, 3, [0,1,2,3], None)
qc2.draw(output='mpl')
# Combine the two circuits and add a swap test
combined_circuit = QuantumCircuit(8)  # New circuit to hold both
    state circuits
combined_circuit.compose(qc1, qubits=[0, 1, 2, 3], inplace=True)
combined_circuit.compose(qc2, qubits=[4, 5, 6, 7], inplace=True)
combined_circuit = apply_swap_test1(combined_circuit, [(3,7)])
combined_circuit.draw(output='mpl')
```

Listing 5.9: Using the Swap test to test the $C^{\otimes 3}Z$ circuit.

5.2 Example 2: The $C^{\otimes n}Z$ Gate

In quantum computing, a multi-control Z gate ($C^{\otimes n}Z$) applies a Z gate to a target qubit if all control qubits are in the $|1\rangle$ state. The gate can be constructed using multiple Toffoli gates and ancilla qubits. The decomposition typically involves creating an ancilla qubit representing the AND of the control qubits and then applying a controlled-Z operation.

Mathematically, a $C^{\otimes n}Z$ gate with n control qubits and one target qubit can be represented as:

$$C^{\otimes n}Z = (I - 2|1\rangle\langle 1|)^{\otimes(n+1)}$$

where I is the identity matrix, and the outer product $|1\rangle\langle 1|$ ensures that the Z gate is applied only when all control qubits are in the state $|1\rangle$.

In Qiskit, a multi-control Z gate can be implemented by constructing a custom gate using a series of Toffoli (controlled-controlled-NOT gates) and single-qubit gates. The following code demonstrates how to implement a $C^{\otimes n}Z$ gate (a 3-control Z gate) using Qiskit:

```
from qiskit import QuantumCircuit, Aer, execute
from qiskit.circuit.library import MCXGate

# Define a function to add a multi-control Z gate to a QuantumCircuit
def multi_control_z(circuit, controls, target):
```

```
    # Number of control qubits
    n = len(controls)

    # Create an ancilla qubit
    ancilla = circuit.qregs[0].size
    circuit.add_register(QuantumRegister(1, 'ancilla'))

    # Apply multi-controlled X (MCX) gate
    mcx = MCXGate(n)
    circuit.append(mcx, controls + [ancilla])

    # Apply Z gate to target
    circuit.cz(ancilla, target)

    # Uncompute the ancilla
    circuit.append(mcx, controls + [ancilla])

# Initialize the quantum circuit
num_controls = 3
qc = QuantumCircuit(num_controls + 1)

# Define control and target qubits
control_qubits = [i for i in range(num_controls)]
target_qubit = num_controls

# Add the multi-control Z gate to the circuit
multi_control_z(qc, control_qubits, target_qubit)

# Draw the circuit
print(qc.draw())

# Execute the circuit on a statevector simulator
backend = Aer.get_backend('statevector_simulator')
```

```
result = execute(qc, backend).result()
statevector = result.get_statevector()

# Print the statevector
print(statevector)
```

Listing 5.10: An implementation of a general $C^{\otimes n}Z$ gate.

In this implementation, the `multi_control_z` function creates an ancilla qubit, uses an MCX (multi-control X) gate to compute the AND of the control qubits onto the ancilla, applies a CZ gate between the ancilla and the target qubit, and then uncomputes the ancilla. This approach leverages Qiskit's built-in `MCXGate` to simplify the construction of the multi-control logic. The quantum circuit is then executed on a statevector simulator to verify its correctness.

Adding an extra gate is one of the interesting bugs that can occur when implementing quantum programs. However, the effect of such a bug depends highly on the location where the extra gate was added.

Taking a deeper look into implementing the $C^{\otimes 3}Z$ introduced in 5.1. We used the categorize, tester, and swap test to locate an extra Z gate before the second Hadamard gate (in Listing 5.11).

```
    def create_3cz_circuit(qc):

    c3z_circuit = QuantumCircuit(4, name='C3Z')

    # Add the gate sequence for the C3Z gate
    c3z_circuit.h(3)  # Hadamard gate to transform Z to X basis
    c3z_circuit.mct([0, 1, 2], 3)  # Multi-controlled Toffoli gate
    c3z_circuit.h(3)  # Hadamard gate back to the original basis
    # Convert the circuit to a gate
    c3z_gate = c3z_circuit.to_gate()

    # Append the C3Z gate to the provided quantum circuit
    qc.append(c3z_gate, [0, 1, 2, 3])

    return qc
```

Listing 5.11: The implementation of the $C^{\otimes 3}Z$ gate.

However, for this example, we want to show that this extra gate makes a big difference based on its location. In fact, where we add the extra gate will completely change the circuit's category.

Let us try adding this extra gate in two locations and see how the circuit behaves:

- Add the extra Z gate before the second H gate.
- Add the extra Z gate after the second H gate.

Now, we know that a CZ gate, regardless of the number of control qubits, is a phase gate, so if we pass it to the `catCircuit` function, we should get "Phase Modulation."

But, if the extra gate is added before the H gate, the results of the categorizer will be "Amplitude Redistribution." In Section 4.5, we discussed that we would follow different strategies to test the various types of the circuit. If we had gotten the correct categorization, we could use the swap test to figure out the phase difference.

Though we can use the swap test regardless of the categorization, it will only offer helpful information for PM circuits.

Luckily, we can locate the extra gate in both cases using the `gateLoc` function, which may not always be the case as the circuit size and complexity increase. However, getting an AR categorization instead of a PM one is a helpful hint for the debugging process.

5.3 Example 3: Entangled Symmetric States

A symmetric state in quantum computing often refers to a quantum state that remains invariant under some transformations or permutations. In other words, exchanging the parts or elements of the state does not alter it. Such states are used in various quantum algorithms and protocols, such as quantum teleportation, superdense coding, and quantum error correction.

The study of symmetric states is more comprehensive than systems of only two qubits. Studying the properties of symmetric states in larger systems is an essential area of research in quantum information theory. There are various examples of such symmetric states; for instance, the GHZ (Greenberger–Horne–Zeilinger) state is a type of multi-qubit symmetric state that plays a crucial role in many quantum protocols.

Consider a two-qubit system. The system's state is symmetric if swapping the two qubits doesn't change the state. For instance, Bell states:

$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$,

is a symmetric state. If you swap the two qubits, the state remains the same. The same logic applies to the case of the state:

$|\psi\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$

Generally speaking, a state is symmetric if it is unchanged under any permutation of the qubits. For this set of examples, we will consider 3 symmetric states: GHZ, W,

and Dicke. An example of testing an implementation of the W state was covered in Chapter 4.5.

5.3.1 The GHZ State

The GHZ (Greenberger–Horne–Zeilinger) state is a specific kind of entangled quantum state that involves three or more qubits. It was named after the scientists Daniel Greenberger, Michael Horne, and Anton Zeilinger, who first studied it.

The GHZ state for three qubits is often written as:
$|GHZ\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$

In this state, if you measure one of the qubits, you'll also collapse the state of the other two. For instance, if you measure the first qubit and find it in state $|0\rangle$, you know with certainty that if you measure the other two qubits, you'll also find them in state $|0\rangle$. Similarly, if you measure the first qubit and find it in state $|1\rangle$, you'll find the other two in state $|1\rangle$. This property holds no matter how far apart the qubits are, demonstrating the phenomenon of quantum entanglement.

A circuit representation of a 3-qubit GHZ state can be seen in Figure 5.5. This circuit is built following the code in Listing 5.12 and the output state will be 5.13.

```
def ghz(qcir,qbits):
    if len(qbits) == 1:
        qcir.h(qbits)
    else:
        for i in range(len(qbits)):
            if i == 0:
                qcir.h(qbits[i])
            else:
                qcir.cx(qbits[i-1],qbits[i])
    return qcir

n = 3
nodes_qubits = QuantumRegister(n, name='q')
qc = QuantumCircuit(nodes_qubits)
qc = ghz(qc, nodes_qubits)
result = execute(qc, S_simulator).result()
output_state = result.get_statevector()
```

```
18 state_to_ket(output_state)
```

Listing 5.12: Python and Qiskit code implementing of the GHZ state.

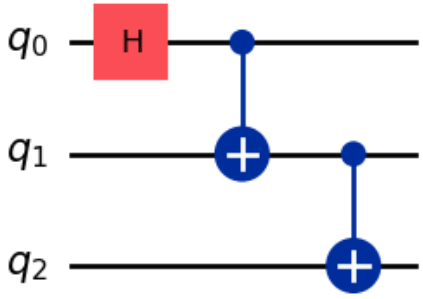


Figure 5.5: A three-qubit circuit implementation of a 3 qubit GHZ state.

```
'0.71|000> + 0.71|111>'
```

Listing 5.13: Python output for the 3-qubit GHZ state.

We can use the `generate_ghz_state_test_cases` function to generate a set of test cases which we can then use as an input to `fQuantTester` to examine the behaviour of another implementation of the GHZ state and locate bugs. We must pass the number of qubits to the `generate_ghz_state_test_cases` function and use the result dictionary. For the purposes of our example, we will generate test cases for a 3-qubit system (Listing 5.14)

```
1 {'name': 'test 0', 'input': [(1+0j), 0j, 0j, 0j, 0j, 0j, 0j, 0j], '
    expected_output': [(0.707+0j), 0j, 0j, 0j, 0j, 0j, 0j, (0.707+0j)
    ]}
2
3 {'name': 'test 1', 'input': [0j, 0j, 0j, 0j, 0j, 0j, 0j, (1+0j)], '
    expected_output': [(0.707+0j), 0j, 0j, 0j, 0j, 0j, 0j, (0.707+0j)
    ]}
4
5 {'name': 'test +', 'input': [(0.353+0j), (0.353+0j), (0.353+0j),
    (0.353+0j), (0.353+0j), (0.353+0j), (0.353+0j), (0.353+0j)], '
    expected_output': [(0.707+0j), 0j, 0j, 0j, 0j, 0j, 0j, (0.707+0j)
    ]}
```

```

{'name': 'test -', 'input': [(0.353+0j), (-0.353+0j), (-0.353+0j),
    (0.353+0j), (-0.353+0j), (0.353+0j), (0.353+0j), (-0.353+0j)], '
    expected_output': [(0.707+0j), 0j, 0j, 0j, 0j, 0j, 0j, (0.707+0j)
    ]}

{'name': 'test i', 'input': [(0.353+0j), 0.353j, 0.353j, (-0.353+0j),
     0.353j, (-0.353+0j), (-0.353+0j), -0.353j], 'expected_output':
    [(0.707+0j), 0j, 0j, 0j, 0j, 0j, 0j, (0.707+0j)]}

{'name': 'test -i', 'input': [(0.353+0j), -0.353j, -0.353j, (-0.353+0
    j), -0.353j, (-0.353+0j), (-0.353+0j), 0.353j], 'expected_output':
     [(0.707+0j), 0j, 0j, 0j, 0j, 0j, 0j, (0.707+0j)]}
```

Listing 5.14: Test cases for a 3-qubit GHZ state.

As an example of that, assume we have a buggy implementation of the GHZ state shown in Listing 5.15.

```
n = 3
nodes_qubits = QuantumRegister(n, name='q')
qc_t = QuantumCircuit(nodes_qubits)
qc_t.h(0)
qc_t.cx(1,2)
qc_t.cx(0,1)
qc_t.draw('mpl')
result_t = execute(qc_t, S_simulator).result()
output_state_t = result_t.get_statevector()
state_to_ket(output_state_t)
```

Listing 5.15: An implementing of the GHZ state with a bug.

Running the tests we generated on this implementation will lead to all of them failing as the output this time is not the expected one (Listing 5.16). We can see a visual representation of the correct results vs. the buggy ones in Figure 5.6.

Creating special test cases is difficult since the GHZ is an Amplitude Redistribution circuit, creating special test cases is difficult. So, the best way to approach this issue is to consider the steps of creating a 3-qubit GHZ state.

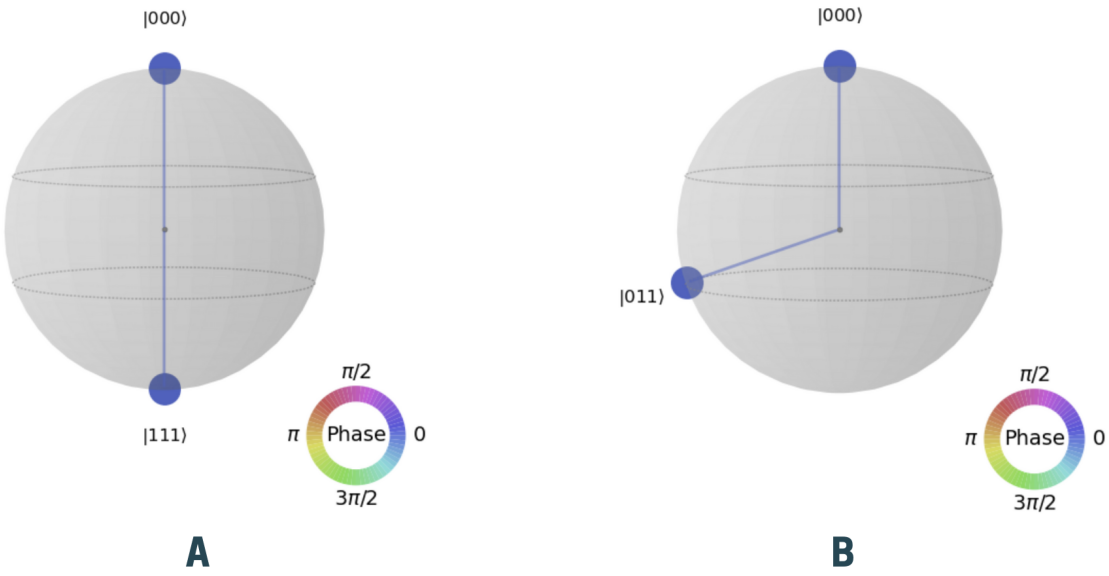


Figure 5.6: The Q-sphere for the 3-qubit GHZ state. A- the correct answer, B- the buggy results.

To create a 3-qubit GHZ, we need to:

- After the Hadamard on the first qubit, the state should be $\frac{1}{\sqrt{2}}(|000\rangle + |100\rangle)$.
- After the first CNOT, with the first qubit as the control and the second qubit as the target, the state should be $\frac{1}{\sqrt{2}}(|000\rangle + |110\rangle)$.
- After the second CNOT, with the second qubit as the control and the third qubit as the target, the state should be $\frac{1}{\sqrt{2}}(|000\rangle + |111\rangle)$.

Now, we can use the `gateLoc` functions to follow these steps until we find the error.

Which will lead us to know that the circuit has the correct number and types of gates. So, the error can either be an order problem or applying the gates to the wrong qubits. Since we only have two gates, we can switch the order of the gates and run the tests again and see if they pass.

In this case, switching the order fixes the bug, but if it did not, then we can deduct that the error is due to applying the gates to the wrong qubits (using the wring control/target combination).

```
0.71|000> + 0.71|011>
```

Listing 5.16: The output of a faulty 3-qubit GHZ state.

5.3.2 The Dicke State

A Dicke state defined for an N-qubit system is defined by the number of qubits and the hamming distance. Specifically, a Dicke state $|D(N,k)\rangle$ is a state with N qubits with a k qubits hamming distance. They are in the state $|1\rangle$, and the rest are in state $|0\rangle$.

For instance, the Dicke state $|D(3,1)\rangle$ with 3 qubits and 1 qubit in state $|1\rangle$ corresponds to the three-qubit W state:

$|D(3,1)> = \frac{1}{\sqrt{3}}(|001\rangle + |010\rangle + |100\rangle)$

However, the Dicke state $|D(3,2)\rangle$ with 3 qubits and 2 qubits in state $|1\rangle$ would be:

$|D(3,2)> = \frac{1}{\sqrt{3}}(|011\rangle + |101\rangle + |110\rangle)$

And, the Dicke state $|D(4,2)\rangle$ with 4 qubits and 2 qubits in state $|1\rangle$ would be:

$|D(4,2)\rangle = \frac{1}{\sqrt{6}}(|0011\rangle + |0101\rangle + |0110\rangle + |1001\rangle + |1010\rangle + |1100\rangle)$

The Dicke state $|D(N,k)\rangle$ is a superposition of all possible permutations of k qubits with equal amplitudes for each term. We can write a general implementation for the Dicke state as seen in Listing 5.17. Using that code, a $|D(3,2)\rangle$ Dicke circuit would look like Figure 5.7, and the output state will be 5.18.

```
#Implementation of dicke state as described in arXiv:1904.07358v1
def scs(qc,qubits,n,l):
    #the qubits will be inputed from buttom to top
    w = math.sqrt(l/n)
    #print(math.sqrt(l/n))
    if len(qubits) == 2:
        qc.cx(qubits[1],qubits[0])
        theta = 2*(math.acos(w))
        qc.cry(theta,qubits[0],qubits[1])
        qc.cx(qubits[1],qubits[0])
    elif len(qubits) == 3:
        qc.cx(qubits[2],qubits[0])
        theta = 2*(math.acos(w))
        qc.mcry(theta, [qubits[0],qubits[1]],qubits[2], None)
        qc.cx(qubits[2],qubits[0])
    else:
        raise CircuitError("Unvalid number of qubits")
    #qc.barrier()
    return qc

```

```
def dicke(qc,qubits,n,k):
    qc.x(qubits[n-k:])
    d = {}
    l = 1
    while l<=k:
        if n == k:
            k -= 1
        else:
            t = []
            for i in range(1,k+1):
                if i == 1:
                    t.append([n-1,n-i-1])
                else:
                    t.append([n-1,n-i,n-i-1])
            d[str(n)+str(k)] = t #[::-1]
            n -= 1
    d["32"] = [[2,1],[2,1,0]]
    d["21"] = [[1,0]]
    for key in d.keys():
        item = d[key]
        l = 1
        for sub in item:
            fn_in = []
            for i in sub:
                fn_in = [qubits[i] for i in sub]
            temp = key[0] if len(key)== 2 else key[:-1]
            qc = scs(qc,fn_in,int(temp),l)
            l += 1
    return qc

n = 3
k = 2
nodes = QuantumRegister(n, name='q')
```

```
54 qc = QuantumCircuit(nodes)
55 qc = dicke(qc,nodes,n,k)
56 result = execute(qc, S_simulator).result()
57 output_state = result.get_statevector()
58 state_to_ket(output_state)
```

Listing 5.17: Python and Qiskit code implementing the Dicke State.

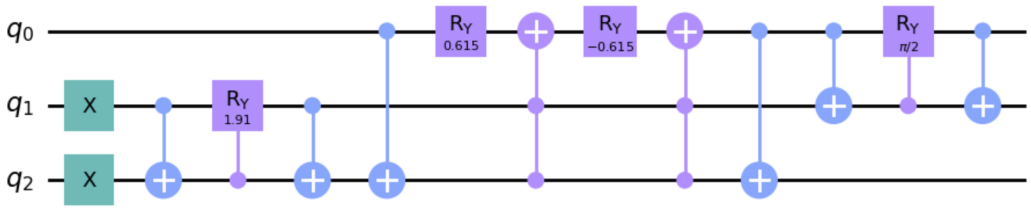


Figure 5.7: A three-qubit circuit implementation of Dicke states.

```
'0.58|011> + 0.58|101> + 0.58|110>'
```

Listing 5.18: Python output of a 3-qubit Dicke state.

Like the GHZ state, we can use the `generate_dicke_state_test_cases` to generate test cases for different implementations. However, unlike the GHZ state, we must pass two values to the `generate_dicke_state_test_cases` function, the number of qubits n and the hamming distance k. For example, for a $|D(3,2)\rangle$, the function returns the test cases shown in Listing 5.19.

```
1 {'name': 'test 0', 'input': [(1+0j), 0j, 0j, 0j, 0j, 0j, 0j, 0j], '
    expected_output': [0j, 0j, 0j, (0.577+0j), 0j, (0.577+0j),
    (0.577+0j), 0j]}
2 {'name': 'test 1', 'input': [0j, 0j, 0j, 0j, 0j, 0j, 0j, (1+0j)], '
    expected_output': [0j, (0.707+0j), (-0.707+0j), 0j, (-0+0j), 0j, 0
    j, 0j]}
3 {'name': 'test +', 'input': [(0.353+0j), (0.353+0j), (0.353+0j),
    (0.353+0j), (0.353+0j), (0.353+0j), (0.353+0j), (0.353+0j)], '
    expected_output': [(0.354+0j), (0.598+0j), (0.098+0j), (0.606+0j),
     (-0.085+0j), (-0.057+0j), (0.063+0j), (0.354+0j)]}
```

```
{'name': 'test -', 'input': [(0.353+0j), (-0.353+0j), (-0.353+0j),
    (0.353+0j), (-0.353+0j), (0.353+0j), (0.353+0j), (-0.353+0j)], '
    expected_output': [(0.354+0j), (-0.598+0j), (-0.098+0j), (0.606+0j
    ), (0.085+0j), (-0.057+0j), (0.063+0j), (-0.354+0j)]}
{'name': 'test i', 'input': [(0.353+0j), 0.353j, 0.353j, (-0.353+0j),
     0.353j, (-0.353+0j), (-0.353+0j), -0.353j], 'expected_output':
    [(-0.354+0j), 0.098j, 0.598j, (-0.198+0j), -0.085j, (0.465+0j),
    (0.346+0j), 0.354j]}
{'name': 'test -i', 'input': [(0.353+0j), -0.353j, -0.353j, (-0.353+0
    j), -0.353j, (-0.353+0j), (-0.353+0j), 0.353j], 'expected_output':
     [(-0.354+0j), -0.098j, -0.598j, (-0.198+0j), 0.085j, (0.465+0j),
    (0.346+0j), -0.354j]}
```

Listing 5.19: Test cases for a 3-qubit Dicke state.

One common error in circuits containing phase gates is entering the rotation angle in the wrong measurement unit (degrees instead of radians).

```
def scs(qc,qubits,n,l):
    #the qubits will be input from buttom to the top
    if len(qubits) == 2:
        qc.cx(qubits[1],qubits[0])
        theta = math.degrees(2*(math.acos(math.sqrt(l/n))))
        qc.cry(theta,qubits[0],qubits[1])
        qc.cx(qubits[1],qubits[0])
    elif len(qubits) == 3:
        qc.cx(qubits[2],qubits[0])
        theta = math.degrees(2*(math.acos(math.sqrt(l/n))))
        qc.mcry(theta, [qubits[0],qubits[1]],qubits[2], None)
        qc.cx(qubits[2],qubits[0])
    else:
        raise CircuitError("Invalid number of qubits")
    return qc

def dicke(qc,qubits,n,k):
```

```
    qc.x(qubits[n-k:])
    d = {}
    l = 1
    while l<=k:
        if n == k:
            k -= 1
        else:
            t = []
            for i in range(1,k+1):
                if i == 1:
                    t.append([n-1,n-i-1])
                else:
                    t.append([n-1,n-i,n-i-1])
            d[str(n)+str(k)] = t #[::-1]
            n -= 1
    d["32"] = [[2,1],[2,1,0]]
    d["21"] = [[1,0]]
    for key in d.keys():
        item = d[key]
        l = 1
        for sub in item:
            fn_in = []
            for i in sub:
                fn_in = [qubits[i] for i in sub]
            temp = key[0] if len(key)== 2 else key[:-1]
            qc = scs(qc,fn_in,int(temp),l)
            l += 1
    return qc
```

Listing 5.20: Dicke implementation with a bug (wrong measurement units for the angles).

Consider the code block in Listing 5.20, where the angle is entered in degrees, leading to somewhat wrong results (Listing 5.21).

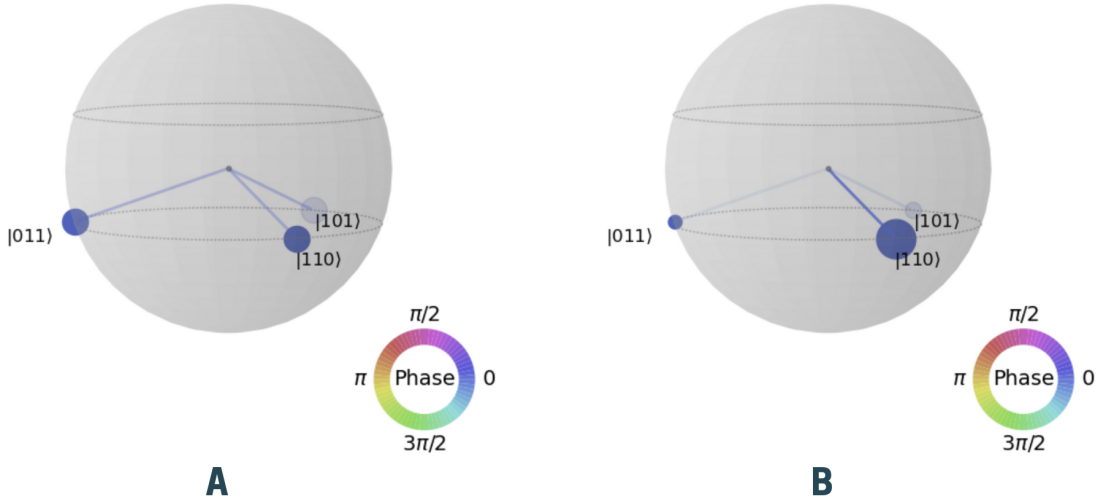


Figure 5.8: The Q-sphere for the 3-qubit Dicke state. A- the correct answer, B- the buggy results.

```
-0.65|011> + -0.65|101> + -0.40|110>
```

Listing 5.21: The output of the faulty Dicke state.

If we look closer at these results and compare them to the correct ones, we can see that the states themselves are accurate. However, their distribution is not. To better understand the effect of the bug, Figure 5.8 shows the Q-sphere of the right vs the buggy versions of the code.

The Q-sphere shows that the phases for both the correct and wrong answers are the same. Based on the discussion of the different categories of circuits in Section 4.4, the Dicke state is an Amplitude Redistribution circuit (the controlled-rotation gates cause entanglement). Since the states are all negative in the faulty results (or not as expected), we can check the lines of code that alter the phase (e.g., rotation gates).

In the implementation of the Dicke state we are using, the angle `theta` is used as an argument for two multi-control Y rotation gates. Since using the wrong measuring units is a common mistake, ensuring that `theta` is in radians is a logical first step. In our example, this should solve the problem. However, if the angle is in radians, but an error still occurs, we need to ensure the value of the angle is correct by going back to the implementation details and cross-checking the values.

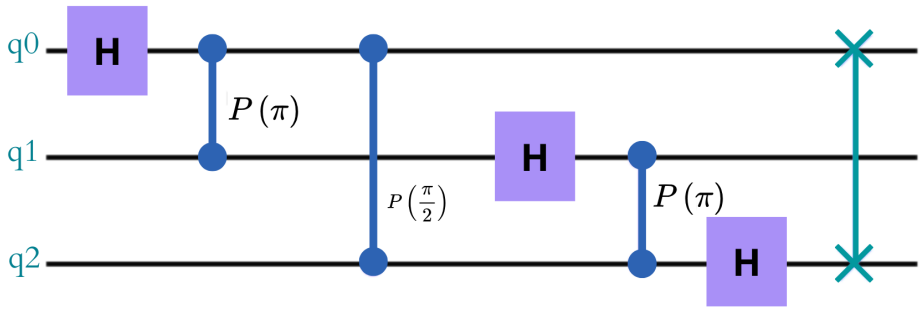


Figure 5.9: A 3-qubit QFT circuit.

5.4 Example 4: Quantum Fourier Transform

The QFT is a fundamental operation in quantum computing, pivotal for algorithms like Shor's algorithm [40] for factoring and quantum phase estimation. Ensuring its correct implementation is vital for the success of these quantum algorithms. The QFT on an n-qubit state is defined as

$$QFT|j\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{2\pi i j k/2^n} |k\rangle. \tag{5.1}$$

A general implementation using Qiskit can be seen in Listing 5.22. The code here offers an important clue to debugging AR circuits: the code is parameterized in n, the number of qubits in the QFT. Thus, the programmer can test and debug their code using small values of n and develop confidence in the behavior of larger instances. We can use simulators and Cirquo's functions to examine the code and test propositions about the circuit.

```
def qft(circuit, n):
    """Applies QFT on the first n qubits in circuit"""
    for j in range(n):
        circuit.h(j)
        for k in range(j+1, n):
            circuit.cp(2 * pi / (2**(k - j)), k, j)
    for j in range(n//2):
```

```
8         circuit.swap(j, n-j-1)
```

Listing 5.22: An Implementation of the QFT using Qiskit.

One approach to creating efficient test vectors is to focus on the properties of the QFT, such as linear shift-invariance and parallelism.

So we can focus on these properties to create test cases as well as some corner cases to test the algorithm's behavior, for example:

- **The simplest states**: Ensure the fundamental functionality of the QFT (all zeroes and all ones). These are basic test cases because it is fairly simple to know the results of the QFT on them.
- **Edge Cases**: Test the handling of single amplitudes set to 1, which can reveal issues in the phase rotations by preparing $|k\rangle$ for $k \neq 0$.
- **Superposition States**: Verify the linearity and ability to handle superpositions critical for quantum algorithms.
- **Equal Superposition**: Check the symmetry and uniformity essential for understanding periodic structures.
- **Phase States**: Ensure the correct handling of complex phases.

All Zeroes State

The simplest input state is $|00\ldots0\rangle$. The QFT of this state should result in a uniform superposition of all basis states:

$$\text{QFT}|000\ldots0\rangle = \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} |y\rangle. \tag{5.2}$$

All Ones State

The input state $|111\ldots1\rangle$ referring to an n-qubit state where all qubits are in the $|1\rangle$ state should transform to a state with alternating phases:

$$\text{QFT}|111\ldots1\rangle = \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{2\pi i \cdot y/2^n} |y\rangle. \tag{5.3}$$

Single Qubit States

For single qubit states such as $|100\ldots0\rangle$, $|010\ldots0\rangle$, and $|000\ldots1\rangle$, the QFT should apply phase rotations corresponding to the position of the single qubit set to 1:

$$\text{QFT}|2^k\rangle = \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{2\pi i \cdot 2^k \cdot y/2^n} |y\rangle. \tag{5.4}$$

Superposition States

States such as $|\psi_1\rangle = \frac{1}{\sqrt{2}}(|000\rangle + |010\rangle)$ and $|\psi_2\rangle = \frac{1}{\sqrt{2}}(|001\rangle + |111\rangle)$ test the linearity and interference patterns of the QFT:

$$\text{QFT}(|\psi\rangle) = \frac{1}{\sqrt{2}}(\text{QFT}(|000\rangle) + \text{QFT}(|010\rangle)). \tag{5.5}$$

Equal Superposition State

The equal superposition state $|\phi\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle$ should transform to a state where the QFT reveals the periodicity:

$$\text{QFT}(|\phi\rangle) = |0\rangle. \tag{5.6}$$

Phase States

States with phase factors such as $|\psi\rangle = \frac{1}{\sqrt{2}}(|000\rangle + i|001\rangle)$ test the handling of complex coefficients and phase shifts:

$$\text{QFT}(|\psi\rangle) = \frac{1}{\sqrt{2}}(\text{QFT}(|000\rangle) + i\text{QFT}(|001\rangle)). \tag{5.7}$$

In this example, we will focus on the linear shift-invariant aspect of the QFT. This result of applying the QFT on a periodic state shows the periodicity and phase relations encoded by the original state transformed into a new superposition reflecting these frequency domain properties. A periodic n-qubit state ($|\psi(n, r, l)\rangle$) is a state that shows a periodic behavior and can be defined by its period r and its shift l. We can express that as

$$|\psi(n, r, l)\rangle = \sum_{k=0}^{2^n-1} c_k |k\rangle \tag{5.8}$$

where c_k are the coefficients of the computational basis states $|k\rangle$, and they reflect the

periodicity and shift of the state. Specifically, c_k is non-zero for states $|k\rangle$ that satisfy $(k - l) \mod r = 0$. For example, consider $|\psi(3, 2, 1)\rangle$

$$|\psi(3, 2, 1)\rangle = \frac{1}{\sqrt{8}} \left(|001\rangle + |011\rangle + |101\rangle + |111\rangle\right). \tag{5.9}$$

We can now apply the QFT to state $|\psi(3, 2, 1)\rangle$ and observe how it preserves the period while transforming it to the frequency domain. For a 3-qubit system ($n = 3$), the QFT equation simplifies to

$$QFT|j\rangle = \frac{1}{\sqrt{8}} \sum_{k=0}^{7} e^{2\pi i jk/8} |k\rangle. \tag{5.10}$$

We can see the circuit for a 3-qubit QFT in Figure 5.9. Applying the QFT to $|\psi(2, 1)\rangle$ gives

$$\begin{aligned} |\psi_{QFT}\rangle = \frac{1}{\sqrt{8}} \Bigg(& \sum_{k=0}^{7} e^{2\pi i \cdot 1 \cdot k/8} |k\rangle + \sum_{k=0}^{7} e^{2\pi i \cdot 3 \cdot k/8} |k\rangle \\ & + \sum_{k=0}^{7} e^{2\pi i \cdot 5 \cdot k/8} |k\rangle + \sum_{k=0}^{7} e^{2\pi i \cdot 7 \cdot k/8} |k\rangle \Bigg). \end{aligned} \tag{5.11}$$

After simplifying this equation, we get

$$|\psi_{QFT}\rangle = \frac{1}{\sqrt{2}} (|0\rangle - |2\rangle + |4\rangle - |6\rangle). \tag{5.12}$$

The calculation demonstrates how the QFT reveals the periodicity and phase relationships encoded in the original state $|\psi(n, r, l)\rangle$, transforming it into a new superposition that reflects these properties in the frequency domain. That effect can be seen in Figure 5.9-A.

Consider the QFT for $n = 1024$; full simulation of this circuit is well beyond classical capabilities. For a fault-tolerant machine, we expect to use it frequently. In Shor's algorithm, the QFT is far less expensive than the modular exponentiation portion of the circuit so that the execution cost will be reasonable. As a first test, the programmer can run the complete QFT circuit on a chosen test vector or set of test vectors using the approach of Figure 4.5. If this test returns an error, the next step is to reduce the scale of the tests to something that can be simulated.

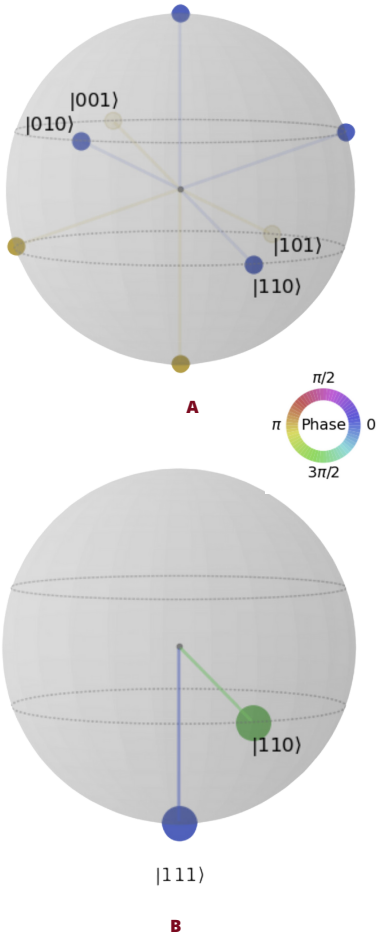


Figure 5.10: (A) The Q-sphere results for the periodic state $|\psi(3,2,1)\rangle$. (B) The Q-sphere results of the 3-qubit QFT where a Hadamard gate on qubit 0 is missing.

Assume that the user forgot to apply a Hadamard gate to qubit 0 at the beginning of the algorithm's implementation. (This kind of error can be expected to be common due to off-by-one programming mistakes in loops, as in Listing 5.22.) Simulating the circuit for $|\psi(3,2,1)\rangle$ will lead to the results shown in Figure 5.10-B. We can see that the output does not match our expected results, but we do not know what might be causing the error.

Because the most common error when implementing quantum algorithms is often a missing/extra gate, we can get the gates count using Qiskit's `count_ops()`. However, this will only return the total count, including the measurement, which can be helpful to use with Cirquo's `gateLoc()` to get the specific qubit and line of code where a gate was added.

We know that an n-qubit QFT includes n Hadamard gates, $\frac{n(n-1)}{2}$ controlled phase rotations are required, and $\lfloor \frac{n}{2} \rfloor$ SWAP gates.

These two functions allow us to discover that qubit 0 has no Hadamard gates applied; adding one (or correcting the loop conditions) and executing the circuit again fixes the error. Repeating the smaller tests up to the simulation limit and re-testing the entire circuit on the FT computer increases our confidence in the overall circuit.

In the prior subsections, we worked with single amplitudes in test vectors. Because AR blocks modify the set of amplitudes, a single-amplitude input will be a multiple-amplitude output. With many amplitudes, testing results will be stochastic, and reconstructing the amplitude distribution will be difficult.

An alternative would be to engineer an input test vector to generate a single amplitude on output. We could do this, for example, by taking a desired output, directly calculating the inverse of the AR block, and using that output as the initial input test vector. Unfortunately, in many cases, the direct creation of that input vector will be as complicated as the entire AR block. However, the approaches shown in Figures 4.5 and 4.7 can be used, for example, to confirm that a new implementation of the QFT produces the same output as a standard library implementation. Working entirely from scratch will be difficult, but comparing versions with new optimizations or features can be more straightforward using these techniques once bootstrapped.

Chapter 6

Discussions and Related Work

Advancements in quantum software tools for debugging are critical to support the increasingly complex quantum computing applications. Essential improvements include the development of quantum-aware debugging tools that can operate within the constraints of quantum mechanics such as observing or measuring a quantum state can cause its wave function to collapse, potentially altering the state and erasing essential quantum properties such as superposition and entanglement. However, as we get closer to fault-tolerant quantum computers, we will need tools and systems to test and debug quantum programs. We also need a version of these tools for the current generation of quantum computers so that programmers and researchers can design and develop programs for future generations.

Developing a mature debugging and testing process for quantum computers will require many years of trials and research. Still, we hope this work will be one of the first steps toward developing a mentality for thinking about quantum programs. Since one of the challenges of debugging and testing quantum programs is the nature of quantum algorithms and how they can be quite counterintuitive, providing the programmer with tools to help them better understand the program's behavior and offer some testing assistance can make a big difference.

An example of a quantum testing and debugging effort is the debugging and testing aspect of Q#. Q# is a quantum development environment developed by Microsoft with a syntax designed to look and feel similar to C#. Q# offers various ways to test NISQ-era quantum programs. It offers unit testing functionality over its quantum circuits object. Unit testing is a commonly used testing and debugging technique in classical computing that is very helpful if the program's results are already known or can be easily calculated. However, if the circuit behavior is challenging to understand, writing test cases for it

will be challenging. Most quantum programs tend to be counterintuitive because of how quantum computers operate; hence, more than unit testing functionality is needed to be more helpful to developers, especially if they are new to quantum software development.

Moreover, some work focused on debugging quantum circuits at run-time [195], using statistical methods [196], or quantum cloning [197]. Some work has also been done on trying to establish a quantum programming development cycle, similar to the classical software cycle, to suggest different techniques that can be used in each step [105] [66] and the testing and debugging tactics for quantum algorithms [66]. In addition to some work on the progress of formal verification methods in the quantum arena [198].

Our proposed tool, Cirquo, provides programmers with some functionalities that they can use to validate the performance of their circuits. Still, another valuable assistance that Cirquo offers is 8 extendable building blocks (adders, the diffusion operator, the W state, the Dicke state, the GHZ state, the QFT, the QPE, and the discrete quantum walk), as well as testing vectors for each of them that the programmer can use if they wish to implement these blocks themselves.

The goal behind Cirquo is to combine, extend, redefine, and bridge three common classical debugging techniques. Firstly is the Binary-like search [199, 200] often referred to as "binary search debugging" or "bisecting." The main idea behind binary-like search debugging is to divide the program into smaller sections, test them individually, and eliminate bug-free sections. Many version control systems, like **Git**, have built-in tools to automate this process. For instance, Git offers the *git bisect* command, which automates the binary search debugging process, making it easier to find the commit that introduced a regression or bug. Cirquo incorporates this by offering the slicer, the testers, and the sample building block implementations and test vectors. Another debugging approach Cirquo utilizes is hypothesis testing [201, 202]. Hypothesis testing, in general, is a statistical approach to analyzing and understanding the behavior of some given data. In the context of software engineering and development, it refers to the process of applying different tests to a program (or a slice), observing its behavior, and then making decisions accordingly. Cirquo utilizes hypothesis testing by offering two executing functions for the Amplitude Permutation, Phase Modulation, and Amplitude Redistribution programs (versions of the testing functions that only run the circuit for different inputs), allowing the user to observe the circuit's behavior for different inputs.

6.1 Verifying Quantum States

Quantum state verification is pivotal in quantum computing and quantum information theory. It involves methods to ascertain the exact state of a quantum system, often represented by qubits. This process is intricate due to the fundamental principles of quantum mechanics. This thesis considers methods like the swap test and quantum tomography and examines their mathematical foundations, limitations, and challenges.

Swap Test and Fidelity Check

The swap test determines the similarity or orthogonality of two quantum states, $|\psi\rangle$ and $|\phi\rangle$. The test utilizes an ancillary qubit, initially in the state $|0\rangle$, and a series of controlled-swap (Fredkin) gates. The ancillary qubit's final state depends on the similarity of $|\psi\rangle$ and $|\phi\rangle$. Mathematically, the probability of finding the ancillary qubit in the $|0\rangle$ state after the swap test is given by:

$$P(0) = \frac{1}{2} + \frac{1}{2}|\langle\psi|\phi\rangle|^2 \tag{6.1}$$

The swap test's limitation is its binary nature; it quantifies similarity but doesn't provide specific state details. It is also probabilistic, requiring several iterations for accurate results.

An important note here is the relation between the swap test and fidelity checks. We can estimate the fidelity F between an experimental state $|\psi\rangle$ and a theoretical state $|\phi\rangle$. Fidelity is mathematically defined as:

$$F(|\psi\rangle, |\phi\rangle) = |\langle\psi|\phi\rangle|^2 \tag{6.2}$$

This method is less demanding than full tomography but offers limited insights into the state's specifics.

Technically, the swap test is an efficient and practical implementation of a fidelity check [203] that we can execute on NISQ devices and FT ones in the future.

Tomography

Tomography is a set of methods used to reconstruct quantum states or processes.

State tomography [204, 205] involves reconstructing the quantum state itself. For a state $|\psi\rangle$, state tomography aims to estimate the density matrix $\rho = |\psi\rangle\langle\psi|$. The process involves performing measurements in different bases and statistically reconstructing ρ from the outcomes.

Process tomography [206, 207, 208], on the other hand, reconstructs the quantum operation (or channel) applied to a state. For an operation $\mathcal{E}$, process tomography estimates the transformation that $\mathcal{E}$ imparts on input states.

Though tomography provides comprehensive information about the system, it is very resource-intensive, requiring exponential measurements for larger systems.

It is an expensive process, so selective process tomography became a focus for many researchers. Though still expensive, it is significantly less resource-consuming than the original process tomography algorithm.

While QPT provides comprehensive information about the quantum process, its exponential scaling limits its use to small systems. SQPT, with its more manageable scaling, becomes a practical choice for analyzing specific aspects of larger quantum processes, offering a balance between resource requirements and possible information to be obtained.

Randomized Benchmarking

Randomized benchmarking can be used to assess the quality of quantum gates and processes [209]. It is robust against specific errors but does not verify the quantum state directly. Instead, it provides an average error rate over a set of quantum operations.

Each method faces some limitations that include:

- Swap Test provides limited information about state specifics.
- Quantum Tomography has big scalability issues and high computational demands.
- Checking the fidelity provides limited insights into the quantum state.
- Randomized benchmarking would not directly provide state information and can be challenging for people without a background in quantum information.

Furthermore, all methods must contend with quantum decoherence and noise, which can lead to inaccurate results.

6.2 Entanglement and Debugging Quantum Circuits

One of the main limitations and challenges we will face is addressing entanglement, not just the work in this thesis, but also debugging quantum programs in general. This problem will become more prominent when we attempt to slice the circuit. For example, consider a simple 2-qubit circuit shown in Figure 6.1.

Let us examine this circuit (the circuit for the Swap test) closely and see how it behaves for different values of $q0$. To do that, we will consider 3 cases, $q0 = |0\rangle$, $|+\rangle$,

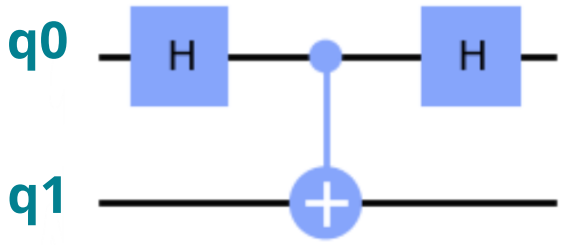


Figure 6.1: A simple circuit showing the challenges of debugging quantum circuits.

and an arbitrary state $|\psi\rangle$. This circuit consists of three steps: $(H \otimes I) \cdot CNOT \cdot (H \otimes I)$, we will go step by step for each value of $q0$ and check the intermediate state of the qubits after each operation (to keep thing simple, the initial value of $q1$ will be set to 1).

Starting with $q0 = 0$:

$$\begin{aligned}
\text{Initial State:} \quad & |00\rangle \\
\text{After First Operation:}(H \otimes I): \quad & \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) \\
\text{After CNOT:} \quad & \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \\
\text{After Second Operation:}(H \otimes I): \quad & \frac{1}{2}[(|0\rangle + |1\rangle) \otimes |0\rangle + (|0\rangle - |1\rangle) \otimes |1\rangle] \\
& = \frac{1}{2}(|00\rangle + |10\rangle + |01\rangle - |11\rangle)
\end{aligned}$$

Similarly, assume we start with $q0 = |+\rangle$, the formals for the intermediate states will be:

$$\begin{aligned}
\text{Initial State:} \quad & |+\rangle \otimes |0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) \\
\text{After First Operation } (H \otimes I): \quad & \frac{1}{2}(|0\rangle + |1\rangle) \otimes |0\rangle + \frac{1}{2}(|0\rangle - |1\rangle) \otimes |0\rangle = |00\rangle \\
\text{After CNOT:} \quad & |00\rangle \\
\text{After Second Operation } (H \otimes I): \quad & \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)
\end{aligned}$$

Now, what if we need to know the state of $q0$? If $q0$ is $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where α and β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$. In that case, the intermediate

states will be:

$$
\begin{aligned}
\text{Initial State:} \quad & (\alpha|0\rangle + \beta|1\rangle) \otimes |0\rangle = \alpha|00\rangle + \beta|10\rangle \\
\text{After First Operation } (H \otimes I): \quad & \frac{\alpha}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle + \frac{\beta}{\sqrt{2}}(|0\rangle - |1\rangle) \otimes |0\rangle \\
& = \frac{1}{\sqrt{2}}(\alpha|00\rangle + \alpha|10\rangle + \beta|00\rangle - \beta|10\rangle) \\
\text{After CNOT:} \quad & \frac{1}{\sqrt{2}}(\alpha|00\rangle + \alpha|11\rangle + \beta|00\rangle - \beta|11\rangle) \\
\text{After Second Operation } (H \otimes I): \quad & \frac{1}{2}[\alpha(|0\rangle + |1\rangle) \otimes |0\rangle + \alpha(|0\rangle - |1\rangle) \otimes |1\rangle \\
& \quad + \beta(|0\rangle + |1\rangle) \otimes |0\rangle - \beta(|0\rangle - |1\rangle) \otimes |1\rangle] \\
& = \frac{1}{2}[\alpha|00\rangle + \alpha|10\rangle + \alpha|01\rangle - \alpha|11\rangle \\
& \quad + \beta|00\rangle + \beta|10\rangle - \beta|01\rangle + \beta|11\rangle]
\end{aligned}
$$

Assume we only want to consider qubit 0; one may guess that since it has two Hadamard gates applied to it if we start with $|0\rangle$ then at the end if we measure it, we should get 0 ($|0\rangle \otimes H \otimes H = |0\rangle \otimes I = |1\rangle$). However, that is not always true. It highly depends on the value of qubit 1 and how it entangles with qubit 0 because of the CNOT gate. In cases where the result is not a pure state but a mixed one, qubit 0 will not be 0 after measurement, making creating test cases for that relatively simple slice even more challenging.

This is the main bottleneck in debugging quantum circuits. Though we can create test cases for the pure states, achieving full coverage would be a serious problem.

6.3 Limitations of the Proposed Debugging and Testing Techniques

Debugging quantum circuits is essentially performing process tomography at a high level. We expect the system's state to evolve over time, and the goal of our testing is to determine whether the system is following these expectations.

In Section 4.5, we propose different strategies to use when testing and debugging the various types of blocks, such as using the swap test for PM blocks or targeting the algorithm's properties to create test vectors for AR circuits. Though these approaches are theoretically valid to test and debug these circuits, they still have limitations.

Creating useful test vectors

Testing AR blocks poses unique challenges. The probabilistic outcomes of quantum states require statistical methods for validation, diverging from the deterministic testing used in classical computing. Entanglement further complicates testing, as the state of one qubit is interdependent with others, obscuring individual analysis. One way to approach this is by creating test vectors that prove the properties of the block under test, similar to what we demonstrated in Section 4.5. Using basic test vectors (applying the block for different bases) can provide the user with information to assist them in debugging.

The increase in needed resources

The swap test circuit looks fairly simple, but its complexity is directly tied to the size of the states being compared. For two quantum states, each consisting of n qubits, the circuit complexity $C_{\text{swap test}} = n$, where $C_{\text{swap test}}$ denotes the number of controlled-SWAP gates needed.

Each CSWAP gate has a significant cost in terms of quantum resources. Depending on the architecture of the quantum processor, implementing a CSWAP gate typically involves multiple elementary quantum gates, such as CNOT and single-qubit gates.

Therefore, the resource cost and potential error rates increase with the number of CSWAP gates used. As n grows, the swap test circuit becomes more complex and more challenging to execute accurately due to increased gate operations and the associated error rates.

A similar argument can be made for implementing and executing process tomography (QPT) [206, 207].

QPT aims to reconstruct the complete quantum process, represented by a superoperator acting on a density matrix. The complexity of QPT scales exponentially with the number of qubits n in the system [210]. The number of experimental configurations required for full reconstruction grows [208] is

$$C_{\text{QPT}} = 4^n \tag{6.3}$$

In contrast, Selective Process Tomography (SQPT) [211, 212] focuses on reconstructing only specific elements of the process matrix. The number of experimental configurations in SQPT depends on the number of elements being selectively measured. This selective approach reduces the scaling, often to a polynomial relation with the number of qubits, depending on the targeted elements. Thus, for a subset of elements k [213], the scaling is

$$C_{\text{SQPT}} \approx \text{poly}(n, k) \tag{6.4}$$

The difference in scaling between QPT and SQPT has significant implications for

their applicability in large quantum systems.

Estimating the optimal number of shots needed

Estimating the required number of shots in quantum circuit simulations depends on the type of circuit we are testing/executing. We only need one shot for the AP blocks to get the results. However, that is different when examining PM and AR blocks. These blocks need more shots to have a higher accuracy. The swap test, for example, involves considerations about statistical accuracy, computational resources, and the specifics of the quantum states being compared. Considering that the measurement process in quantum computing follows a binomial distribution [214] where each shot results in either a success or failure. Hence, the relation between the number of shots and the standard deviation (σ) of the measurement outcome can be described by

$$\sigma = \sqrt{\frac{p(1-p)}{N}} \tag{6.5}$$

p is the probability of measuring a particular outcome, and N is the number of shots. While there is no simple formula to calculate the optimal number directly, we can determine a suitable number of shots based on the requirements for statistical accuracy. We can use the σ with a specific accuracy value Z to calculate the confidence interval (CI) of the number of shots needed using the formula

$$CI = p \pm Z \times \sigma \tag{6.6}$$

Distinguishing between hardware and software errors

A critical aspect of quantum program testing is acknowledging the differences in debugging on simulators, NISQ machines, and fault-tolerant quantum computers. Limitations in program size and hardware-related errors in NISQ machines pose significant challenges. While these issues might be resolved with fault-tolerant quantum computers, classical computers will still play a vital role in debugging and testing quantum programs.

Developing a deep understanding of circuit behavior, identifying sources of errors, and creating effective test cases are essential for building the quantum intuition necessary to advance the field toward the fault-tolerant era.

Chapter 7

Conclusion

7.1 Summary

As quantum computers evolve towards fault tolerance, the demand for tools assisting developers in crafting and refining their quantum programs will surge. While there has been significant focus on the software side of quantum computing, with the development of quantum-intuitive programming languages, there is still a need to provide developers with robust tools for testing and debugging.

Validating the behavior of quantum circuits remains the most significant challenge quantum technology faces today. If a debugging tool is not developed, this challenge will become more difficult as the circuit size increases and the ability of current hardware improves.

It is crucial to pinpoint the specific implementation level to debug quantum programs effectively. Quantum programs can be implemented on different levels: the high-level algorithmic design, the intermediate circuit layout, and the foundational level of electrical pulses interacting with the hardware. These implementation levels have challenges, but overall, quantum programs are difficult to test and debug regardless of the implementation level.

This thesis introduces a comprehensive suite of software tools (Cirquo) and insights for developers to test and understand their quantum programs. This suite encompasses circuit slicing, categorization, gate tracking, and testing functionalities. Additionally, it provides foundational building blocks for testing and debugging quantum algorithms.

We also present an analysis of frequently encountered bugs in quantum software collected from discussions on platforms like StackOverflow, StackExchange, and GitHub (similar to bug clustering [215, 216] classically). This categorization can serve as a valuable source for developers in the bug identification step of debugging.

To summarize this thesis:

- Explores commonly occurring bugs reported by programmers on platforms like GitHub, StackOverflow, and StackExchange.
- Proposes the construction of a simple circuit slicer to reduce the size of the circuit being examined by the programmer.
- Explains a categorization of quantum circuits into three types based on their effects on the qubits' states.
- Discusses the implementation and test vectors of commonly used subroutines in quantum algorithms.
- Proposes strategies for creating efficient test vectors and debugging different types of circuits.
- Covers a debugging helper function to track and locate specific gates within a quantum circuit.

Quantum circuit testing and debugging is a research field with numerous unresolved questions and challenges. This research contributes to shaping the future of quantum program testing and debugging while offering valuable insights into the field's current state by introducing a circuit slicer, categorizer, and targeted testing methods. It represents a significant leap towards overcoming the limitations of quantum circuits' increasing complexity and size. These tools are designed to be hardware-independent, offering a versatile solution across various quantum computing platforms.

For instance, consider the application of the circuit slicer tool within a quantum algorithm used for molecular simulation—a field where quantum computing promises substantial breakthroughs. Typically, such simulations involve complex and large-scale quantum circuits that challenge even the most advanced classical simulation techniques. Using the circuit slicer, developers can decompose these large circuits into smaller, more manageable blocks. This decomposition allows for isolated testing and debugging, which is far less resource-intensive and provides more precise insights into the functionality and potential issues within specific circuit sections. Similarly, the circuit categorizer can be used to pinpoint how a particular circuit block influence the overall performance and outcomes of the simulation, enabling more precise optimizations and adjustments. These examples illustrate the utility of the proposed tools and highlight their potential to significantly expedite the development and deployment of quantum computing solutions.

7.2 Future Directions

This work focuses on helping the programmer better understand the circuit behavior by allowing them to look into smaller parts of it, learn the functionality of that part, and use the provided test functions to examine different input/output combinations.

The work in this thesis is a solid first step not only in the field of quantum software but also in the field of quantum education. Some possible future expansions of this work include:

1. **Development of Specialized Quantum Software Testing Frameworks:** Including the creation of broader testing frameworks that cover a specific subset of quantum algorithms by focusing on how these algorithms manipulate the state of the qubits to reach an answer.

2. **Integration of Classical and Quantum Software Development Tools:** There is a potential to develop integrated development environments (IDEs) that merge classical and quantum software tools, simplifying the development of hybrid algorithms.

3. **An In-depth Investigation on Calculating the Optimal Number of Shots:** We propose a simple approach to find the confidence interval for the optimal number of shots. However, developing an algorithm to calculate the optimal number of shots for a given algorithm will significantly value testing and debugging quantum circuits.

4. **Distinguish Between Hardware and Software Bugs:** One of the challenges that quantum software still faces, particularly in the NISQ era, is distinguishing between errors caused by the hardware and the software.

If we desire to debug a quantum computer, we will need to be more specific about the level of implementation we are targeting. In quantum computing, there are different levels of implementation, starting with the higher level, which is the algorithmic implementation. The circuit and lower levels consist of electrical pulses executed on actual hardware. Regardless of our target level, scaling is another challenge since we use classical computers to debug quantum circuits. Even if we attempt to use NISQ devices, the scalability challenge remains. However, we need tools and strategies when we reach fault-tolerant quantum computers.

Although attempting to bridge ideas and methods from the classical side to the quantum side can be beneficial, these approaches must be edited to adapt to the nature of quantum algorithms and add new methods and techniques that help us better predict and understand the behavior of quantum programs. Cirquo is meant to assist the programmer in locating bugs due to the implementation of the algorithm. It does not help if the bug is due to the hardware or if there is any way to distinguish between the software-caused or hardware-caused bugs.

7.3 Influence of this work

This thesis's contributions mark an advancement in the field of quantum software, addressing some of the key challenges faced by developers and researchers. By introducing the comprehensive suite of tools within the Cirquo package, this work provides a robust framework for testing, debugging, and optimizing quantum programs. These tools, including circuit slicing, categorization, gate tracking, and targeted testing functionalities, offer a solution for managing the complexity of quantum circuits as the field advances toward fault-tolerant quantum computing.

I believe that this thesis will be impactful for multiple reasons, one of which is its focus on the practical challenges of quantum software development. By analyzing and categorizing frequently encountered bugs from platforms like GitHub, StackOverflow, and StackExchange, this work offers a resource for developers, aiding in the bug identification process and providing insights into common issues. This empirical approach ensures that the tools developed are grounded in the real-world experiences of quantum programmers, making them highly relevant and effective.

Furthermore, this thesis sets a precedent for the future of quantum software development by emphasizing the importance of hardware-independent solutions. The tools and methodologies proposed are designed to be versatile across various quantum computing platforms, ensuring broad applicability and fostering interoperability in the rapidly evolving quantum landscape. This forward-thinking approach addresses current challenges and anticipates future developments, positioning this work as a foundational reference for ongoing and future research in quantum software testing and debugging. This thesis contributes to accelerating the adoption and advancement of quantum computing technologies by improving the reliability and efficiency of quantum program development.

Appendix A

List of Papers and Presentations

A.1 First Author Papers and Presentations

A.1.1 Peer-Reviewed Journals

1. Sara Ayman Metwalli and Rodney Van Meter, "Testing and Debugging Quantum Circuits," in IEEE Transactions on Quantum Engineering, vol. 5, pp. 1-15, 2024, Art no. 2500415, doi: 10.1109/TQE.2024.3374879.

2. Sara Ayman Metwalli, François Le Gall and Rodney Van Meter "Finding Small and Large kClique Instances on a Quantum Computer," in IEEE Transactions on Quantum Engineering, vol. 1, pp. 1-11, 2020, Art no. 3102911, doi: 10.1109/TQE.2020.3045692.

3. Sara Ayman Metwalli and Yuko Hara-Azumi. 2019. SSA-AC: Static Significance Analysis for Approximate Computing. ACM Trans. Des. Autom. Electron. Syst. 24, 3, Article 34 (May 2019), 17 pages. https://doi.org/10.1145/3314575.

A.1.2 International Conferences

Oral Presentation

1. Sara Ayman Metwalli and Rodney Van Meter, "A Tool For Debugging Quantum Circuits," 2022 IEEE International Conference on Quantum Computing and Engineering (QCE), Broomfield, CO, USA, 2022, pp. 624-634, doi: 10.1109/QCE53715.2022.00085.

2. Sara Ayman Metwalli, Jeffrey Cross, and Hideki Mori, “Going Deep with edX Insights and TokyoTechX First MOOC." *Open edX 2016*. Stanford University, USA.

Poster Presentations

1. Sara Ayman Metwalli and Rodney Van Meter, "A Categorization of Bugs in Quantum Programs," 2023 IEEE International Conference on Quantum Computing and Engineering (QCE), Bellevue, WA, USA, 2023, pp. 346-347, doi: 10.1109/QCE57702.2023.10275.

2. Sara Ayman Metwalli, François Le Gall and Rodney Van Meter, "Finding Small and Large kClique Instances on a Quantum Computer," 2020 IEEE International Conference on Quantum Computing and Engineering (QCE), Virtual.

3. Sara Ayman Metwalli and Yuko Hara-Azumi, “SEA-AC: Symbolic Execution-based Analysis towards Approximate Computing." *IEEE MICRO51*, 2018, Fukuoka, Japan.

A.1.3 Other Presentations

1. Kenneth Heitritter and Sara Ayman Metwalli “qBraid: The Quantum Ecosystem All in One Place.” *IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2023, Washington, USA.

2. Sara Ayman Metwalli, “Quantum Search with Python and Qiskit.” *Women Who Code DataPy*, 2019, Atlanta, USA.

3. Sara Ayman Metwalli, “Programming Quantum Teleportation.” *Women Who Code CONNECT Asia*, 2019, Singapore.

4. Sara Ayman Metwalli and Todd Tilma, “Building an OpenStax-based MATLAB Grader Homework System for a Physics Class." *Mathworks Asia Summit*, 2018, Tokyo, Japan.

A.2 Non-First Author Papers and Presentations

1. Ryosuke Satoh, Michal Hajdusek, Naphan Benchasattabuse, Shota Nagayama, Kentaro Teramoto, Takaaki Matsuo, Sara Ayman Metwalli, Poramet Pathumsoot, Takahiko Satoh, Shigeya Suzuki, and Rodney Van Meter, "QuISP: a Quantum

Internet Simulation Package", *IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2022, Colorado, USA.

A.2.1 Teaching

1. Probability (Keio University) / 2019F.

A.2.2 Certificates and Awards

Awards

1. Grace Hopper Celebration Awardee 2021.
2. Qiskit Camp Asia 1st place 2019.
3. Co-Valedictorian for Master's Degree, TokyoTech, 2018.

Certificates

1. Qiskit Advocate IBM, 2021.
2. IBM Certified Associate Developer 2021.
3. Certificate of Quantum Excellence, IBM 2020.

Appendix B

Quantum Bugs

In this appendix we display a table of the 123 bugs collected and discussed in Chapter 4.2.

Bug source	Comment	Bug Category	Runtime error?	Library/ PL Used	Date Created
https://github.com/Qiskit/qiskit-terra/issues/356	QASM misinterprates the circuit when running it	Classical Data Post-processing	Yes	Qiskit	Mar 21, 2018
https://github.com/Qiskit/qiskit-terra/issues/537	error matrix is not unitary	Quantum Circuit	Yes	Qiskit	Jun 5, 2018
https://github.com/Qiskit/qiskit-terra/issues/596	User attempts to run an empty circuit	Job Handling	Yes	Qiskit	Jun 26, 2018
https://stackoverflow.com/questions/51387285/ getting-error-released-qubits-are-not-in-zero-state-in-q-quantum-dev-kit	qubit needs to be reset at the end of operations?	Quantum Circuit	Yes	Q#	Jul 17, 2018
https://stackoverflow.com/questions/51600882/q-aggregate-exception-error	Execution delay error, must include a wait function	Job Handling	Yes	Q#	Jul 30, 2018
https://stackoverflow.com/questions/51618651/getting-error-a-set-statement-must-have-a-variable-or-array-element-target-whe	Version problem	Job Handling	Yes	Q#	Jul 31, 2018
https://quantumcomputing.stackexchange.com/questions/3929/incorrectly-calculating-probability-amplitudes-for-3-qbit-circuit?rq=1	LSQB first? Misreading the measurment results	Classical Data Post-processing	No	Qiskit	Aug 8, 2018
https://github.com/Qiskit/qiskit-terra/issues/783	error matrix is not unitary	Quantum Circuit	Yes	Qiskit	Aug 16, 2018
https://quantumcomputing.stackexchange.com/questions/4142/ how-does-one-obtain-amplitude-information-in-q	User didn't know what function to use	Classical Data Post-processing	No	Q#	Sep 3, 2018
https://quantumcomputing.stackexchange.com/questions/4260/how-to-create-a-condition-on-only-one-classical-bit-when-we-have-a-total-of-2-cl	Wrong implementation (the user should use two classical register instead of one)	Quantum Circuit	Yes	Qiskit	Sep 20, 2018
https://quantumcomputing.stackexchange.com/questions/31957/ qiskit-error-circuit-circuit-91-contains-invalid-instructions	Transpiling needed	Job Handling	Yes	Qiskit	Sep 24, 2018
https://quantumcomputing.stackexchange.com/questions/4283/how-can-i-resolve-the-error-qiskit-error-the-same-provider-has-already-been-reg	Token resistered more than once	Job Handling	Yes	Qiskit	Sep 24, 2018
https://stackoverflow.com/questions/52701261/qiskit-nameerror-name-q0-is-not-defined	global vs local variable and wrong naming	Classical Semantics	Yes	Qiskit	Oct 8, 2018
https://github.com/Qiskit/qiskit-terra/issues/1446	User using function from_qasm_str wrong	Classical Semantics	Yes	Qiskit	Dec 7, 2018
https://quantumcomputing.stackexchange.com/questions/5557/wait-gate-throws-an-error-notimplementederror-no-decomposition-rules-defin	Deprecation	Job Handling	Yes	Qiskit	Feb 25, 2019
https://github.com/Qiskit/qiskit-terra/issues/1985	Transpile error	Classical Data Post-processing	Yes	Qiskit	Mar 19, 2019
https://github.com/Qiskit/qiskit-aer/issues/135	Measure used wrong	Quantum Circuit	No	Qiskit	Apr 9, 2019
https://quantumcomputing.stackexchange.com/questions/5959/ grovers-algorithm-returns-skewed-probability-distribution	Using the reset function in the wrong location in the circuit	Quantum Circuit	No	Qiskit	Apr 23, 2019
https://quantumcomputing.stackexchange.com/questions/6692/how-do-i-get-out-2-measurements-from-the-same-execution-on-qiskit	wrong use of the APi	Classical Data Post-processing	Yes	Qiskit	Jul 5, 2019
https://quantumcomputing.stackexchange.com/questions/6697/ creating-and-running-parallel-circuits-in-qiskit	Deprecation	Job Handling	Yes	Qiskit	Jul 6, 2019
https://quantumcomputing.stackexchange.com/questions/ 6755/controlled-initialize-instruction?rq=1	DAG error	Quantum Circuit	Yes	Qiskit	Jul 12, 2019
https://quantumcomputing.stackexchange.com/questions/ 7129/how-to-obtain-qubits-amplitude-in-qiskit	measure instead of using statevector	Classical Data Post-processing	No	Qiskit	Aug 30, 2019
https://quantumcomputing.stackexchange.com/questions/8547/ cirq-ry-complains-that-it-needs-one-argument	Wrong syntax of using gates	Classical Semantics	Yes	Cirq	Oct 22, 2019
https://quantumcomputing.stackexchange.com/questions/ 8694/is-there-a-mistake-in-the-vqe-ansatz-in-cirqs-tutorial	wrong order of operations	Quantum Circuit	No	Cirq	Nov 6, 2019
https://quantumcomputing.stackexchange.com/questions/9224/ how-to-plot-histogram-or-bloch-sphere-for-multiple-circuits	Misuse of the function draw in Qiskit	Classical Semantics	No	Qiskit	Dec 15, 2019
https://quantumcomputing.stackexchange.com/questions/ 9209/how-to-use-parallel-executions-of-circuits	Wrong function used	Classical Semantics	No	Qiskit	Dec 15, 2019
https://quantumcomputing.stackexchange.com/questions/ 9246/quantum-phase-estimation-implementation	Wrong algo implementation	Quantum Circuit	No	Qiskit	Dec 18, 2019
https://github.com/Qiskit/qiskit-terra/issues/3799	Function usage not clearly documented	Classical Semantics	Yes	Qiskit	Feb 5, 2020
https://quantumcomputing.stackexchange.com/questions/9828/generate-the-state-frac-0-rangle-1-rangle-sqrt2-with-qiskit-proble?rq=1	Z gate behaviour	Classical Data Post-processing	No	Qiskit	Feb 11, 2020
https://quantumcomputing.stackexchange.com/questions/ 9871/achieve-a-control-gate-with-2-hadamard-coins	Wrong gate decomposition	Quantum Circuit	No	Qiskit	Feb 15, 2020

Table B.1: Bugs Table 1

Bug source	Comment	Bug Category	Does it produce a runtime error?	API Used	Date Created
https://quantumcomputing.stackexchange.com/questions/9943/how-to-make-circuit-for-randomly-selected-gate	Wrong use of the APi	Quantum Circuit	Yes	Qiskit	Feb 22, 2020
https://github.com/Qiskit/qiskit-aer/issues/664	User misused the measurment fucntion	Quantum Circuit	No	Qiskit	Mar 19, 2020
https://stackoverflow.com/questions/60918011/implement-quantum-teleportation-in-qiskit	Wrong measurment location	Quantum Circuit	No	Qiskit	Mar 29, 2020
https://quantumcomputing.stackexchange.com/questions/11492/vqe-for-beginners-using-tutorial-and-cirq	plotting the wrong values	Classical Data Post-processing	No	Cirq	Apr 11, 2020
https://github.com/Qiskit/qiskit-terra/issues/4144	use decompose instead of transpile	Classical Semantics	No	Qiskit	Apr 13, 2020
https://stackoverflow.com/questions/62661255/2-entangled-qubit-gives-all-states-with-25	User lack of knowldge of how circuits work	Quantum Circuit	No	Qiskit	Jun 30, 2020
https://quantumcomputing.stackexchange.com/questions/12770/microsoft-q-namespace-declarations-can-only-occur-at-a-global-scope-error	Name conflict	Classical Semantics	Yes	Q#	Jul 3, 2020
https://quantumcomputing.stackexchange.com/questions/12838/visualizing-custom-gates-in-cirq	Misuse of the function CircuitDiagramInfo	Classical Semantics	Yes	Cirq	Jul 8, 2020
https://stackoverflow.com/questions/63283443/my-qiskit-codes-output-differ-from-the-lecturer-ryan-o-donnell	Qiskit bit ordering flip issue (misread by the user)	Classical Data Post-processing	No	Qiskit	Aug 6, 2020
https://stackoverflow.com/questions/63663248/in-quantum-teleportation-should-the-x-gate-be-before-z-gate-possible-error-on	wrong gates order	Quantum Circuit	No	Qiskit	Aug 31, 2020
https://quantumcomputing.stackexchange.com/questions/13615/qiskit-plot-state-qsphere-phase-error	Phase in Qsphere is wrong due to Qiskit interal commands	Job Handling	No	Qiskit	Sep 6, 2020
https://quantumcomputing.stackexchange.com/questions/13822/grover-algorithm-in-q	Wrong iteration number	Quantum Circuit	No	Q#	Sep 19, 2020
https://github.com/Qiskit/qiskit-terra/issues/5098	function not working as expected	Job Handling	Yes	Qiskit	Sep 21, 2020
https://quantumcomputing.stackexchange.com/questions/14147/odd-behavior-with-qiskit-pauli-operators?rq=1	Phase introduced due to using Qiskit's control() method	Job Handling	No	Qiskit	Oct 14, 2020
https://github.com/Qiskit/qiskit-terra/issues/5249	User used the wrong command	Classical Semantics	No	Qiskit	Oct 16, 2020
https://quantumcomputing.stackexchange.com/questions/14286/multiple-bipartite-entangled-state-in-cirq	Misunderstanding of the append method functionality	Classical Data Post-processing	Yes	Cirq	Oct 21, 2020
https://stackoverflow.com/questions/64707625/visualizing-circuits-in-qiskit-with-matplotlib	Using the wrong import syntax	Classical Semantics	Yes	Qiskit	Nov 6, 2020
https://github.com/quantumlib/Cirq/issues/3549	adding multiple gates using the wrong function	Quantum Circuit	Yes	Cirq	Dec 1, 2020
https://quantumcomputing.stackexchange.com/questions/14945/why-is-the-measurement-result-always-1-expected-to-find-uniformly-random-measu	Measuring the wrong thing	Quantum Circuit	No	Cirq	Dec 3, 2020
https://quantumcomputing.stackexchange.com/questions/15275/how-can-i-fix-this-parametrization-error-in-qiskit	pramatized circuit error	Job Handling	Yes	Qiskit	Dec 26, 2020
https://github.com/Qiskit/qiskit-terra/issues/5580	Wrong implementation of the CRZ gate	Quantum Circuit	No	Qiskit	Dec 30, 2020
https://stackoverflow.com/questions/65733183/qbit-collapse-before-quantum-teleportation	Missing gate and wrong measurement	Quantum Circuit	No	Qiskit	Jan 15, 2021
https://quantumcomputing.stackexchange.com/questions/15649/cirq-result-of-rotating-qubit-measurements-never-come-0-1-or-1-0-always-c	User switched reading qubits	Quantum Circuit	No	Cirq	Jan 23, 2021
https://quantumcomputing.stackexchange.com/questions/15711/cirq-n-qubit-ghz-state	Uncomplete implementation of the circuit	Quantum Circuit	Yes	Cirq	Jan 27, 2021
https://quantumcomputing.stackexchange.com/questions/15925/q-sphere-representation-of-bell-states	Missing gate	Quantum Circuit	No	Qiskit	Feb 8, 2021

Table B.2: Bugs Table 2

Bug source	Comment	Bug Category	Does it produce a runtime error?	API Used	Date Created
https://quantumcomputing.stackexchange.com/questions/15966/was-the-quantum-circuit-attribute-iden-renamed	Deprecation	Job Handling	Yes	Qiskit	Feb 10, 2021
https://stackoverflow.com/questions/66198976/valueerror-numpy-ndarray-size-changed-on-importing-qiskit	Version problem	Job Handling	Yes	Qiskit	Feb 14, 2021
https://stackoverflow.com/questions/66197227/return-two-numbers-in-q-sharp-q-quantum-development-kit	wrong return type syntax	Classical Semantics	Yes	Q#	Feb 14, 2021
https://quantumcomputing.stackexchange.com/questions/16636/setting-the-initial-state-of-density-matrix-simulation-in-cirq	Function doesn't realize shape	Job Handling	Yes	Cirq	Mar 22, 2021
https://github.com/quantumlib/Cirq/issues/3954	angles in radians not degrees	Quantum Circuit	No	Cirq	Mar 23, 2021
https://github.com/Qiskit/qiskit-aer/issues/1192	model larger than machine can handle	Classical Data Post-processing	Yes	Qiskit	Mar 24, 2021
https://github.com/Qiskit/qiskit-terra/issues/6255	Statevector and QASM mismatch results	Classical Data Post-processing	No	Qiskit	Apr 19, 2021
https://quantumcomputing.stackexchange.com/questions/17259/qiskit-qft-matrix-does-not-match-with-dft-matrix?rq=1	Misunderstanding of output	Classical Data Post-processing	No	Qiskit	Apr 26, 2021
https://stackoverflow.com/questions/67362847/attributeerror-quantumcircuit-object-has-no-attribute-config	Missing function before execution	Quantum Circuit	Yes	Qiskit	May 3, 2021
https://stackoverflow.com/questions/67411006/python-calling-q-file-on-ionq-qpu-results-in-error-about-a-system-text-json-ve	Version problem	Job Handling	No	Q#	May 6, 2021
https://quantumcomputing.stackexchange.com/questions/17651/setting-initial-state-in-qiskit-unitary-simulator	Using unsupported function for the simulator	Job Handling	Yes	Qiskit	May 26, 2021
https://github.com/Qiskit/qiskit-terra/issues/6540	Cant convert to controled because of nested circuits	Classical Data Post-processing	Yes	Qiskit	Jun 8, 2021
https://github.com/Qiskit/qiskit-terra/issues/6571	Missing function attribute	Classical Semantics	No	Qiskit	Jun 14, 2021
https://quantumcomputing.stackexchange.com/questions/18448/how-to-perform-a-plot-histogram-for-a-circuit	Misunderstanding of the circuit	Classical Data Post-processing	No	Qiskit	Jul 16, 2021
https://stackoverflow.com/questions/68475347/why-i-get-different-phases-with-jupyter-and-quantum-composer	Difference in the global phase between code and simulator	Quantum Circuit	No	Qiskit	Jul 21, 2021
https://quantumcomputing.stackexchange.com/questions/20662/cirq-qaoa-tasks-returns-iteration-over-a-0-d-array-error	Error in reading JSON in Cirq	Classical Data Post-processing	Yes	Cirq	Jul 29, 2021
https://github.com/Qiskit/qiskit-terra/issues/6892	Missing function attribute	Classical Semantics	No	Qiskit	Aug 10, 2021
https://quantumcomputing.stackexchange.com/questions/20894/saving-statevector-on-more-than-one-location-in-a-quantum-circuit-in-qiskit	Not enough knowldge of API	Classical Data Post-processing	Yes	Qiskit	Aug 20, 2021
https://github.com/Qiskit/qiskit-terra/issues/7032	must decompose first	Classical Data Post-processing	Yes	Qiskit	Sep 16, 2021
https://stackoverflow.com/questions/69245836/getting-figure-x-when-drawing-quantum-circuit-with-qiskit-mpl-output-mo	Using the draw function wrong	Classical Semantics	No	Qiskit	Sep 19, 2021
https://stackoverflow.com/questions/69598995/qiskits-draw-only-shows-the-circuits-name-and-not-the-architecture	Not using decompose	Classical Data Post-processing	No	Qiskit	Oct 16, 2021
https://stackoverflow.com/questions/70095529/atribute-error-when-running-the-qiskit-vqe-alogrithum	Version problem	Job Handling	Yes	Qiskit	Nov 24, 2021
https://stackoverflow.com/questions/70093492/qisikit-statevector-of-quantum-teleportation-circuit	LSQB first? Misreading the measurment results	Classical Data Post-processing	No	Qiskit	Nov 24, 2021
https://quantumcomputing.stackexchange.com/questions/23421/quantum-circuit-error-in-qiskit	measurement error (using the measurement in the wrong location)	Quantum Circuit	Yes	Qiskit	Dec 26, 2021
https://quantumcomputing.stackexchange.com/questions/23471/error-cannot-unroll-the-circuit-to-the-given-basis	Wrong function input	Classical Semantics	Yes	Qiskit	Dec 29, 2021

Table B.3: Bugs Table 3

Bug source	Comment	Bug Category	Does it produce a runtime error?	API Used	Date Created
https://stackoverflow.com/questions/70754687/plot-histogramresult-get-counts-error-in-qiskit	Missing measurment function	Quantum Circuit	Yes	Qiskit	Jan 18, 2022
https://quantumcomputing.stackexchange.com/questions/24202/wrong-expected-value-from-cirq-paulistring	user used the wrong function	Classical Semantics	No	Cirq	Feb 21, 2022
https://quantumcomputing.stackexchange.com/questions/24369/what-is-wrong-with-my-circuit-for-the-fourth-root-of-x	Gates flipped	Quantum Circuit	No	Qiskit	Mar 4,2022
https://stackoverflow.com/questions/71456850/dqcperror-the-problem-is-not-dqcp-while-implementing-a-qsvm-in-ibm-quantum-exp	Version problem	Job Handling	Yes	Qiskit	Mar 13, 2022
https://stackoverflow.com/questions/71546461/result-and-iqarray-could-not-be-found	missing function	Classical Semantics	Yes	Q#	Mar 20, 2022
https://stackoverflow.com/questions/71768097/qiskit-qaoa-compile-error-evolvedop-object-has-no-attribute-broadcast-argumen	Version problem	Job Handling	Yes	Qiskit	Apr 6, 2022
https://stackoverflow.com/questions/71818251/valueerror-setting-an-array-element-with-a-sequence-for-scikit-learn	Type error when using quantum kernal	Classical Data Post-processing	Yes	Qiskit	Apr 10, 2022
https://quantumcomputing.stackexchange.com/questions/26212/qiskit-library-gaussian-does-not-accept-parametric-expression?rq=1	Qiskit have both guassian and Guassian functions	Job Handling	Yes	Qiskit	Apr 29, 2022
https://stackoverflow.com/questions/72215342/qiskit-matplotlib-drawings-have-incorrect-shapes	the transpiler missing attributes	Job handling	No	Qiskit	May 12, 2022
https://stackoverflow.com/questions/72322760/import-modules-for-qiskit	Typo	Classical Semantics	Yes	Qiskit	May 20, 2022
https://stackoverflow.com/questions/72417370/qiskit-no-counts-for-experiment	Missing measurment function	Quantum Circuit	No	Qiskit	May 28, 2022
https://stackoverflow.com/questions/72413224/i-am-getting-no-memory-for-the-experiment-error-although-i-dont-see-any-mistak	Wrong attributes	Classical Semantics	Yes	Qiskit	May 28, 2022
https://stackoverflow.com/questions/72646187/qiskit-twolocal-does-not-print-quantum-circuit	User didn't use the unroll to display full circuit	Quantum Circuit	No	Qiskit	Jun 16, 2022
https://stackoverflow.com/questions/72785037/aws-braket-python-sdk-user-is-not-authorized-to-perform-on-quantum-device	Authorizarion error	Job handling	Yes	Braket	Jun 28, 2022
https://stackoverflow.com/questions/72875257/cannot-import-qiskit-modulenotfounderror-no-module-named-qiskit-accelerate	Compatibility error	Job handling	Yes	Qiskit	Jul 5, 2022
https://stackoverflow.com/questions/73239349/why-am-i-getting-nameerror-name-qiskit-is-not-defined-on-my-program-when-i	wrong command when calling excute function	Classical Semantics	Yes	Qiskit	Aug 4, 2022
https://quantumcomputing.stackexchange.com/questions/27782/circuit-gives-unexpected-phase-factor-cirq	global phase issue	Classical Data Post-processing	No	Cirq	Aug 15, 2022
https://stackoverflow.com/questions/73477635/bundled-qc-circuit-on-qiskt-index-error	Missing function input	Classical Semantics	Yes	Qiskit	Aug 24, 2022
https://github.com/qiskit-community/qiskit-nature/issues/816	user misunderstanding of algo	Quantum Circuit	No	Qiskit	Sep 4, 2022
https://stackoverflow.com/questions/73646675/error-while-running-circuit-drawoutput-mpl	missing library	Classical Semantics	Yes	Qiskit	Sep 8, 2022
https://stackoverflow.com/questions/73963180/this-worked-last-weekqiskit-library-issue	import issue/ environment related error	Classical Semantics	Yes	Qiskit	Oct 5, 2022
https://stackoverflow.com/questions/74239149/tensorcirucit-c-draw-nameerror	import issue/ environment related error	Classical Semantics	Yes	Qiskit	Oct 28, 2022
https://quantumcomputing.stackexchange.com/questions/28972/number-of-qubits-does-not-match-the-number-of-qubits-of-the-observable?rq=1	The user used the SparsePauliOp wrong	Quantum Circuit	Yes	Qiskit	Nov 12, 2022
https://quantumcomputing.stackexchange.com/questions/29038/qiskit-error-circuit-contains-invalid-instructions	Gate not supported by simulator	Job Handling	Yes	Qiskit	Nov 18, 2022
https://quantumcomputing.stackexchange.com/questions/29303/conversion-error-from-a-qiskit-circuit-to-a-qasm-string-and-back	Error due to qiskit update	Job Handling	Yes	Qiskit	Dec 8, 2022

Table B.4: Bugs Table 4

Bug source	Comment	Bug Category	Does it produce a runtime error?	API Used	Date Created
https://stackoverflow.com/questions/74830919/syntax-for-creating-immutable-array-in-q	wrong input	Quantum Circuit	Yes	Q#	Dec 17, 2022
https://stackoverflow.com/questions/75162746/ket-draw-not-showing-00-qiskit	function input missing	Classical Semantics	No	Qiskit	Jan 18, 2023
https://quantumcomputing.stackexchange.com/questions/29831/grover-ccz-gate-on-ibm-quantum-lab-unexpected-outputs	Wrong order of qubits in output	Classical Data Post-processing	No	Qiskit	Jan 24, 2023
https://stackoverflow.com/questions/75237445/keyerror-0-problem-is-there-a-way-to-fix	return type chaged from list to dict in qiskit update	Classical Data Post-processing	Yes	Qiskit	Jan 25, 2023
https://stackoverflow.com/questions/75424858/issue-in-qiskit-vqc-with-amplitude-encoding-for-state-preparation	User mistake in input	Quantum Circuit	Yes	Qiskit	Feb 12, 2023
https://stackoverflow.com/questions/75553239/attributeerror-cant-set-attribute-in-qiskit-grover-algorithm	attribute error, user trying to assign value to read only property	Classical Semantics	Yes	Qiskit	Feb 24, 2023
https://stackoverflow.com/questions/75688554/recursionerror-maximum-recursion-depth-exceeded-when-training-with-quantumker	runtime error	Job Handling	Yes	Qiskit	Mar 9, 2023
https://stackoverflow.com/questions/76030148/quantum-fourier-transformation-qiskit-implementation	method not found due to qiksit updates	Job Handling	Yes	Qiskit	Apr 16, 2023
https://stackoverflow.com/questions/76084475/incorrect-results-in-ground-state-evaluation-via-qiskit-nature-when-using-qiskit	Different results for estimators due to shots number	Classical Data Post-processing	No	Qiskit	Apr 23, 2023
https://stackoverflow.com/questions/76144355/combining-quantum-circuits-in-qiskit	Deprecation error, can't append two circuits using +	Job handling	Yes	Qiskit	May 1, 2023
https://stackoverflow.com/questions/76152273/qiskit-no-counts-for-experiment-0	missing measurment function	Quantum Circuit	Yes	Qiskit	May 2, 2023
https://quantumcomputing.stackexchange.com/questions/32422/resolving-aws-braket-validationexception-an-error-occurred-validationexcepti	error due to cirq implementation	Job Handling	Yes	Cirq	May 3, 2023
https://stackoverflow.com/questions/76229341/issues-encountered-in-using-scikit-quant-optimizers-in-qiskit-vqe-workflows-aer	anstaz init error	Quantum Circuit	Yes	Qiskit	May 11, 2023
https://quantumcomputing.stackexchange.com/questions/32581/grovers-algorithm-for-max-cut-problem-implementation-in-cirq	grover's oracle implementation issue	Quantum Circuit	No	Cirq	May 15, 2023
https://quantumcomputing.stackexchange.com/questions/32698/the-amount-of-clbit-arguments-0-does-not-match-the-instruction-expectation-16	API uses wrong function	Job Handling	Yes	Qiskit	May 24, 2023
https://stackoverflow.com/questions/76392444/import-error-when-running-shors-algorithm-on-qiskit	Deprecation error	Job Handling	Yes	Qiskit	Jun 2, 2023
https://stackoverflow.com/questions/76415391/why-am-i-getting-this-qiskit-error-code-qr-not-defined	Typo	Classical Semantics	Yes	Qiskit	Jun 6, 2023
https://github.com/qiskit-community/qiskit-nature/issues/1197	wrong function return	Classical Data Post-processing	No	Qiskit	Jun 6, 2023
https://github.com/qiskit-community/qiskit-nature/issues/816	user misunderstanding of algo	Quantum Circuit	No	Qiskit	Sep 4, 2022
https://stackoverflow.com/questions/73646675/error-while-running-circuit-drawoutput-mpl	missing library	Classical Semantics	Yes	Qiskit	Sep 8, 2022
https://stackoverflow.com/questions/73963180/this-worked-last-weekqiskit-library-issue	import issue/ environment related error	Classical Semantics	Yes	Qiskit	Oct 5, 2022
https://stackoverflow.com/questions/74239149/tensorcirucit-c-draw-nameerror	import issue/ environment related error	Classical Semantics	Yes	Qiskit	Oct 28, 2022
https://quantumcomputing.stackexchange.com/questions/28972/number-of-qubits-does-not-match-the-number-of-qubits-of-the-observable?rq=1	The user used the SparsePauliOp wrong	Quantum Circuit	Yes	Qiskit	Nov 12, 2022
https://quantumcomputing.stackexchange.com/questions/29038/qiskit-error-circuit-contains-invalid-instructions	Gate not supported by simulator	Job Handling	Yes	Qiskit	Nov 18, 2022
https://quantumcomputing.stackexchange.com/questions/29303/conversion-error-from-a-qiskit-circuit-to-a-qasm-string-and-back	Error due to qiskit update	Job Handling	Yes	Qiskit	Dec 8, 2022

Table B.5: Bugs Table 5

Appendix C

Suite Code and Structure

C.0.1 The Slicer

```
# This file includes the slicer functions
import re
from qiskit import QuantumCircuit, QuantumRegister
from .helpers import get_barrier_locs, slice_list_with_indeces,
    remove_barrier

def Vslicer(inputCir, mode="mini"):
    gates_list = inputCir.data
    barrier_locs = get_barrier_locs(gates_list)
    slicedCir = slice_list_with_indeces(gates_list, list(barrier_locs
    .values()))
    cleanCir = remove_barrier(slicedCir)
    qubitsInCir = str(inputCir.qubits)
    pattern = r"QuantumRegister\(\d+,\s*'\w+'\)"
    result = re.findall(pattern, qubitsInCir)
    result = sorted(set(result), key=lambda x: result.index(x))
    qunRegs = {}
    for i in list(result):
```

```
        info = i[16:-1].split(', ')
        qunRegs[info[1][1:-1]] = info[0]
    Dcir = QuantumCircuit()
    for key in qunRegs.keys():
        Dcir.add_register(QuantumRegister(int(qunRegs[key]), name=key
    ))
    cirList = []
    for cir in cleanCir:
        count = 1
        cirName = "sub circuit" + str(count)
        temp = Dcir.copy(name=cirName)
        for gate in cir:
            if gate[0].name == 'measure':
                pass
            else:
                temp.append(gate[0], qargs=gate[1])
        cirList.append(temp)
        count += 1
    if mode == "mini":
        return cirList
    elif mode == "accom":
        cirListAcc = [cirList[0]]
        for i in range(1, len(cirList)):
            count = 2
            cirName = "sub circuit" + str(count)
            temp = cirListAcc[i-1].copy(name=cirName)
            temp.extend(cirList[i])
            cirListAcc.append(temp)
            count += 1
        return cirListAcc
    else:
        return "Invalid mode"

```

```
def Hslicer(cirSlice, wanted_regs):
    d = cirSlice.data
    pattern = r"QuantumRegister\(\d+,\s*'\w+'\)"
    result = re.findall(pattern, str(d))
    result = sorted(set(result), key=lambda x: result.index(x))
    qunRegs = {}
    for i in list(result):
        info = i[16:-1].split(', ')
        reg_name = info[1][1:-1]
        if reg_name in wanted_regs:
            qunRegs[reg_name] = info[0]
    sub_circuit = QuantumCircuit()
    for reg_name in qunRegs.keys():
        temp_reg = QuantumRegister(int(qunRegs[reg_name]), name=
    reg_name)
        sub_circuit.add_register(temp_reg)
    for gate in cirSlice:
        if any(str(qubit.register.name) in wanted_regs for qubit in
    gate[1]):
            qargs = [qubit for qubit in gate[1] if qubit.register.
    name in wanted_regs]
            sub_circuit.append(gate[0], qargs=qargs)
    return sub_circuit
```

Listing C.1: The Slicer Functions

C.0.2 The Categorizer

```
def is_permutation_matrix(matrix):
    """Check if the matrix is a permutation matrix."""
    size = matrix.shape[0]
    for i in range(size):
```

```
        if not (np.sum(matrix[i,:] == 1) == 1 and np.sum(matrix[:,i] == 1) == 1):
            return False
    return True

def is_diagonal_matrix(matrix, tol=1e-10):
    """Check if the matrix is diagonal, considering a tolerance for small imaginary parts."""
    return np.allclose(matrix, np.diag(np.diagonal(matrix)), atol=tol)

def catCircuit(circuit):
    """Categorize the quantum circuit based on its unitary."""
    # Simulate the circuit to get the unitary matrix
    backend = Aer.get_backend('unitary_simulator')
    t_circ = transpile(circuit, backend)
    result = backend.run(t_circ).result()
    unitary = result.get_unitary(t_circ)

    # Convert the Operator to a numpy array
    unitary_array = np.asarray(unitary)

    # Print the unitary matrix nicely
    np.set_printoptions(precision=3, suppress=True)
    print("Unitary matrix:")
    print(np.array2string(unitary_array, separator=', '))

    # Analyze the unitary matrix
    if is_permutation_matrix(unitary_array):
        return "Amplitude Permutation (AP)"
    elif is_diagonal_matrix(unitary_array):
        return "Phase Modulation (PM)"
    else:
```

```
35         return "Amplitude Redistribution (AR)"
```

Listing C.2: The Slicer Functions

C.0.3 Gate Tracker

```
class QuantumCircuitWrapper(_original_QuantumCircuit):
_original_QuantumCircuit = QuantumCircuit

class QuantumCircuitWrapper(_original_QuantumCircuit):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.gateInfo = dict()

    def __getattr__(self, name):
        attr = super().__getattr__(name)
        if callable(attr):
            def method_with_traceback(*args, **kwargs):
                if name not in self.gateInfo:
                    self.gateInfo[name] = [traceback.extract_stack()
    [-2]]
                else:
                    self.gateInfo[name].append(traceback.
    extract_stack()[-2])
                return attr(*args, **kwargs)
            return method_with_traceback
        else:
            return attr

    def breakbarrier(self, *qargs, **kwargs):
        if 'breakbarrier' not in self.gateInfo:
            self.gateInfo['breakbarrier'] = [traceback.extract_stack
    ()[-2]]
```

```
25          else:
26              self.gateInfo['breakbarrier'].append(traceback.
    extract_stack()[-2])
27          return super().barrier(*qargs, **kwargs)
28
29  def startDebug():
30      global QuantumCircuit
31      QuantumCircuit = QuantumCircuitWrapper
32
33  def endDebug():
34      global QuantumCircuit
35      QuantumCircuit = _original_QuantumCircuit
36
37
38  def gateLoc(circuit, gate_name, qubits=None):
39      all_occurrences = circuit.gateInfo.get(gate_name, [])
40      occurrence_idx = 0
41      filtered_occurrences = []
42
43      for gate, *applied_qubits in circuit.data:
44          if gate.name == gate_name:
45              if qubits:
46                  qinv = [f"{q.register.name}[{q.index}]" for q in
    applied_qubits[0]]
47                  if lists_have_same_items(qubits, qinv):
48                      filtered_occurrences.append(all_occurrences[
    occurrence_idx])
49              else:
50                  filtered_occurrences.append(all_occurrences[
    occurrence_idx])
51              occurrence_idx += 1
52
53      print(f"Gate '{gate_name}' has {len(filtered_occurrences)}
```

```
    occurrences.")
54      for idx, trace in enumerate(filtered_occurrences, 1):
55          print(f"{idx}. Called at file {trace.filename} \nIn function
    {trace.name} line {trace.lineno} {(trace.line)}", end="\n
    ============================\n")
```

Listing C.3: The Slicer Functions

C.0.4 Testing

Generating inputs

```
1  def generate_input_states(num_qubits):
2      # Define the input states
3      states = {
4          "0": [1, 0],
5          "1": [0, 1],
6          "+": [1/np.sqrt(2), 1/np.sqrt(2)],
7          "-": [1/np.sqrt(2), -1/np.sqrt(2)],
8          "i": [1/np.sqrt(2), 1j/np.sqrt(2)],
9          "-i": [1/np.sqrt(2), -1j/np.sqrt(2)]
10     }
11
12     input_states = {}
13     for name, state in states.items():
14         qc = QuantumCircuit(num_qubits)
15         for qubit in range(num_qubits):
16             qc.initialize(state, qubit)
17         input_states[name] = Statevector.from_instruction(qc).data.
    tolist()
18
19     return input_states
```

Listing C.4: The Slicer Functions

Generating test cases

```
def generate_w_state_test_cases(num_qubits):
    input_states = generate_input_states(num_qubits)
    test_cases = []
    for name, in_state in input_states.items():
        #print(in_state)
        w_reg = QuantumRegister(num_qubits)
        w_circuit = QuantumCircuit(w_reg)
        w_circuit.initialize(in_state, list(range(num_qubits)))
        w_circuit = wn(w_circuit,w_reg)
        w_state = Statevector.from_instruction(w_circuit).data.tolist
    ()
        #print(w_state)
        test_cases.append({"name": f"test {name}", "input": in_state,
     "expected_output": w_state})

    #test_cases = [{"name": f"test {name}", "input": state, "
    expected_output": w_state} for name, state in input_states.items()
    ]

    return test_cases

def generate_ghz_state_test_cases(num_qubits):
    # Generate GHZ state
    ghz_circuit = QuantumCircuit(num_qubits)
    ghz_circuit.h(0)
    for qubit in range(num_qubits - 1):
        ghz_circuit.cx(qubit, qubit + 1)
    ghz_state = Statevector.from_instruction(ghz_circuit).data.tolist
    ()

    input_states = generate_input_states(num_qubits)
```

```
    test_cases = [{"name": f"test {name}", "input": state, "
    expected_output": ghz_state} for name, state in input_states.items
    ()]

    return test_cases

def generate_dicke_state_test_cases(num_qubits,k):
    input_states = generate_input_states(num_qubits)
    test_cases = []
    for name, in_state in input_states.items():
        dicke_reg = QuantumRegister(num_qubits)
        dicke_circuit = QuantumCircuit(dicke_reg)
        dicke_circuit.initialize(in_state, list(range(num_qubits)))
        dicke_circuit = dicke(dicke_circuit,dicke_reg,num_qubits,k)
        dicke_state = Statevector.from_instruction(dicke_circuit).
    data.tolist()
        test_cases.append({"name": f"test {name}", "input": in_state,
     "expected_output": dicke_state})


    return test_cases

def generate_qft_test_cases(num_qubits):
    input_states = generate_input_states(num_qubits)
    test_cases = []
    for name, in_state in input_states.items():
        qft_reg = QuantumRegister(num_qubits)
        qft_circuit = QuantumCircuit(qft_reg)
        qft_circuit.initialize(in_state, list(range(num_qubits)))
        qft_circuit = qft(qft_circuit,num_qubits)
        qft_state = Statevector.from_instruction(qft_circuit).data.
    tolist()
        test_cases.append({"name": f"test {name}", "input": in_state,
```

```
    "expected_output": qft_state})

    return test_cases

def generate_full_adder_test_cases():
    """Generate test cases for a full adder with 4 input/output
    qubits."""
    test_cases = []

    # Full adder has 3 inputs: A, B, and Cin, plus one 0 input
    # There are 2^3 = 8 possible input combinations for A, B, and Cin
    inputs = [
        [0, 0, 0, 0],
        [0, 0, 1, 0],
        [0, 1, 0, 0],
        [0, 1, 1, 0],
        [1, 0, 0, 0],
        [1, 0, 1, 0],
        [1, 1, 0, 0],
        [1, 1, 1, 0]
    ]

    for i, input_bits in enumerate(inputs):
        a, b, cin, _ = input_bits
        sum_bit = (a ^ b ^ cin)  # Sum bit calculation using XOR
        cout_bit = (a & b) | (cin & (a ^ b))  # Carry out calculation

        expected_output = [a, b, sum_bit, cout_bit]

        test_case = {
            "name": f"test {i + 1}",
            "input": input_bits,
            "expected_output": expected_output
```

```
        }

        test_cases.append(test_case)

    return test_cases

def create_diffusion_operator_test_cases(num_qubits):
    test_cases = []

    backend = Aer.get_backend('statevector_simulator')

    for i in range(6):
        # Create a quantum circuit with the given number of qubits
        qc = QuantumCircuit(num_qubits)

        # Apply Hadamard gates to create an initial superposition
    state
        qc.h(range(num_qubits))

        # Get the initial statevector (input)
        initial_statevector = execute(qc, backend).result().
    get_statevector()
        initial_statevector = np.asarray(initial_statevector)  #
    Explicitly cast to numpy array

        # Define a simple oracle that marks one of the states (e.g.,
    |0...0> state)
        oracle = QuantumCircuit(num_qubits)
        oracle.x(range(num_qubits))
        oracle.h(num_qubits-1)
        oracle.mcx(list(range(num_qubits-1)), num_qubits-1)  # multi-
    controlled Toffoli
        oracle.h(num_qubits-1)
```

```
        oracle.x(range(num_qubits))

        # Define the Grover operator (diffusion operator)
        grover_op = GroverOperator(oracle)

        # Apply the Grover operator to the circuit
        qc.append(grover_op, range(num_qubits))

        # Get the final statevector (expected output)
        final_statevector = execute(qc, backend).result().
    get_statevector()
        final_statevector = np.asarray(final_statevector)  #
    Explicitly cast to numpy array

        # Create the test case
        test_case = {
            "name": f"Test case {i+1}",
            "input": initial_statevector.tolist(),  # Convert to list
     for JSON serialization
            "expected_output": final_statevector.tolist()  # Convert
    to list for JSON serialization
        }

        # Add the test case to the list
        test_cases.append(test_case)

    return test_cases
```

Listing C.5: The Slicer Functions

Running Inputs

```
def fQuantAnalyzer(circuit, test_cases):
```

```
    backend = Aer.get_backend("statevector_simulator")
    analysis_results = []

    for test_case in test_cases:
        # Normalize the input state
        input_state = test_case["input"]
        norm = np.linalg.norm(input_state)
        input_state = input_state / norm
        state = Statevector(input_state)

        test_circuit = QuantumCircuit(circuit.num_qubits)
        test_circuit.initialize(state.data, range(circuit.num_qubits)
    )

        # Append the circuit
        test_circuit = test_circuit.compose(circuit)

        # Execute the test circuit
        result = execute(test_circuit, backend).result()
        output_state = result.get_statevector()

        # Store the analysis results
        analysis_result = {
            "name": test_case["name"],
            "input": state_to_ket(input_state),
            "output": state_to_ket(output_state)
        }
        analysis_results.append(analysis_result)

    return analysis_results

def display_results(test_results):
    for test_result in test_results:
```

```
        print(f"Testing {test_result['name']}:")

        for key, value in test_result.items():
            # Check if the key is "pass" and modify the value accordingly
            if key == "pass":
                value = "PASS" if value else "FAIL"

            # Capitalize the first letter of the key and print the value
            print(f"{key.capitalize()}: {value}")

        print("\n")
```

Listing C.6: The Slicer Functions

Running Tests

```
from qiskit import Aer, execute, QuantumCircuit
import numpy as np
from qiskit.quantum_info import Statevector
from .helpers import state_to_ket, normalize_global_phase, classical_to_statevector, statevector_to_classical

def fQuantTester(circuit, test_cases):
    backend = Aer.get_backend("statevector_simulator")
    test_results = []

    for test_case in test_cases:
        input_state = test_case["input"]
        norm = np.linalg.norm(input_state)
        input_state = input_state / norm
        state = Statevector(input_state)
```

```
15          test_circuit = QuantumCircuit(circuit.num_qubits)
16          test_circuit.initialize(state.data, range(circuit.num_qubits)
    )
17
18          test_circuit = test_circuit.compose(circuit)
19
20          result = execute(test_circuit, backend).result()
21          output_state = result.get_statevector()
22
23          expected_output_state = test_case["expected_output"]
24
25          normalized_output_state = normalize_global_phase(output_state
    )
26          normalized_expected_output_state = normalize_global_phase(
    Statevector(expected_output_state))
27
28          is_equal = np.allclose(normalized_output_state,
    normalized_expected_output_state, atol=1e-6)
29
30          test_result = {
31              "name": test_case["name"],
32              "pass": is_equal,
33              "input": state_to_ket(input_state),
34              "output": state_to_ket(output_state),
35              "expected_output": state_to_ket(expected_output_state)
36          }
37          test_results.append(test_result)
38          print(f"Testing {test_result['name']}:")
39          print("Result: ", "PASS" if test_result["pass"] else "FAIL")
40          print("Input: ", test_result["input"])
41          print("Output: ", test_result["output"])
42          print("Expected Output: ", test_result["expected_output"])
43          print("\n")
```

```
44
45      return test_results
46
47  def pClassicalTester(circuit, test_cases):
48      backend = Aer.get_backend("statevector_simulator")
49      test_results = []
50
51      for test_case in test_cases:
52          print(f"Testing {test_case['name']}:")
53
54          input_state = classical_to_statevector(test_case["input"])
55          expected_output_state = classical_to_statevector(test_case["
    expected_output"])
56
57          test_circuit = QuantumCircuit(circuit.num_qubits)
58
59          for i, bit in enumerate(test_case["input"]):
60              if bit == 1:
61                  test_circuit.x(i)
62
63          test_circuit = test_circuit.compose(circuit)
64
65          result = execute(test_circuit, backend).result()
66          output_state = result.get_statevector()
67
68          output_classical = statevector_to_classical(output_state)
69
70          is_equal = np.array_equal(output_classical[::-1], test_case["
    expected_output"])
71
72          test_result = {
73              "name": test_case["name"],
74              "pass": is_equal,
```

```
            "input": test_case["input"],
            "output": output_classical,
            "expected_output": test_case["expected_output"]
        }
        test_results.append(test_result)

        print("Result: ", "PASS" if is_equal else "FAIL")
        print("Input: ", test_result["input"])
        print("Output: ", test_result["output"][::-1])
        print("Expected Output: ", test_result["expected_output"])
        print("\n")

    return test_results

def applySwapTest(circuit, qubit_pairs, shots=8192):
    from qiskit import QuantumRegister, ClassicalRegister

    ancilla = QuantumRegister(1, 'ancilla')
    c = ClassicalRegister(1, 'measure')
    circuit.add_register(ancilla)
    circuit.add_register(c)

    circuit.h(ancilla[0])

    for reg1_index, reg2_index in qubit_pairs:
        circuit.cswap(ancilla[0], reg1_index, reg2_index)

    circuit.h(ancilla[0])

    circuit.measure(ancilla[0], c[0])

    backend = Aer.get_backend('qasm_simulator')
    result = execute(circuit, backend, shots=shots).result()
```

```
    counts = result.get_counts(circuit)

    if '1' in counts:
        b = counts['1']
    else:
        b = 0

    s = abs(1 - (2 / shots) * b)

    delta_theta = np.arccos(round((2*s-1)))

    delta_theta_degrees = np.degrees(delta_theta)
    delta_theta_pi = delta_theta / np.pi

    print("Delta Theta: {delta_theta_pi}pi rad")

    return circuit
```

Listing C.7: The Slicer Functions

C.0.5 Quantum Subroutines

```

\begin{lstlisting}[language=Python, caption=Diffusion for non-uniform
    state preparation.]
def cnz(qc, num_control, node, anc):
    """
    num_control :  number of control qubit of cnz gate
    node :              node qubit
    anc :                ancillaly qubit
    """
    if num_control>2:
        qc.ccx(node[0], node[1], anc[0])
```

```
        for i in range(num_control-2):
            qc.ccx(node[i+2], anc[i], anc[i+1])
        qc.cz(anc[num_control-2], node[num_control])
        for i in range(num_control-2)[::-1]:
            qc.ccx(node[i+2], anc[i], anc[i+1])
        qc.ccx(node[0], node[1], anc[0])
    if num_control==2:
        qc.h(node[2])
        qc.ccx(node[0], node[1], node[2])
        qc.h(node[2])
    if num_control==1:
        qc.cz(node[0], node[1])

def grover_diff(qc, nodes_qubits,edge_anc,edge_flag,ancilla,stat_prep
    ,inv_stat_prep):
    qc.append(inv_stat_prep,qargs=nodes_qubits)
    qc.x(nodes_qubits)
    #=====================================================
        #control qubits Z gate
    cnz(qc,len(nodes_qubits)-1,nodes_qubits[::-1],ancilla)
    #=====================================================
    qc.x(nodes_qubits)
    qc.append(stat_prep,qargs=nodes_qubits)
```

Listing C.8: General Diffusion Operator

```
def qft_rotations(circuit, n):
    if n == 0: # Exit function if circuit is empty
        return circuit
    n -= 1 # Indexes start from 0
    circuit.h(n) # Apply the H-gate to the most significant qubit
    for qubit in range(n):
        # For each less significant qubit, we need to do a
        # smaller-angled controlled rotation:
```

```
        circuit.cp(pi/2**(n-qubit), qubit, n)

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

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

Listing C.9: QFT

```
def cg (qcir,cQbit,tQbit,theta):
    theta_dash = math.asin(math.cos(math.radians(theta/2)))
    qcir.u(theta_dash,0,0,tQbit)
    qcir.cx(cQbit,tQbit)
    qcir.u(-theta_dash,0,0,tQbit)
    return qcir


def wn (qcir,qbits):
    for i in range(len(qbits)):
        if i == 0:
            qcir.x(qbits[0])
            #qcir.barrier()
        else:
            p = 1/(len(qbits)-(i-1))
            theta = math.degrees(math.acos(math.sqrt(p)))
            theta = 2* theta
            qcir = cg(qcir,qbits[i-1],qbits[i],theta)
            qcir.cx(qbits[i],qbits[i-1])
```

```
            #qcir.barrier()
    return qcir
```

Listing C.10: W State

```
def ghz(qcir,qbits):
    if len(qbits) == 1:
        qcir.h(qbits)
    else:
        for i in range(len(qbits)):
            if i == 0:
                qcir.h(qbits[i])
            else:
                qcir.cx(qbits[i-1],qbits[i])
    return qcir
```

Listing C.11: GHZ State

```
def scs(qc,qubits,n,l):
    #the qubits will be inputed from buttom to top
    #print(math.sqrt(l/n))
    if len(qubits) == 2:
        qc.cx(qubits[1],qubits[0])
        theta = 2*(math.acos(math.sqrt(l/n)))
        qc.cry(theta,qubits[0],qubits[1])
        qc.cx(qubits[1],qubits[0])
    elif len(qubits) == 3:
        qc.cx(qubits[2],qubits[0])
        theta = 2*(math.acos(math.sqrt(l/n)))
        qc.mcry(theta, [qubits[0],qubits[1]],qubits[2], None)
        qc.cx(qubits[2],qubits[0])
    else:
        raise CircuitError("Unvalid number of qubits")
    return qc

```

```
18 def dicke(qc,qubits,n,k):
19     qc.x(qubits[n-k:])
20     d = {}
21     l = 1
22     while l<=k:
23         if n == k:
24             k -= 1
25         else:
26             t = []
27             for i in range(1,k+1):
28                 if i == 1:
29                     t.append([n-1,n-i-1])
30                 else:
31                     t.append([n-1,n-i,n-i-1])
32             d[str(n)+str(k)] = t #[::-1]
33             n -= 1
34     d["32"] = [[2,1],[2,1,0]]
35     d["21"] = [[1,0]]
36 #=================================================================
37     for key in d.keys():
38         item = d[key]
39         l = 1
40         for sub in item:
41             fn_in = []
42             #print("=========================")
43             #print(sub)
44             for i in sub:
45                 fn_in = [qubits[i] for i in sub]
46             #print(int(key[0]),l)
47             #print(fn_in)
48             qc = scs(qc,fn_in,int(key[0]),l)
49             l += 1
```

```
50      return qc
```

Listing C.12: Dicke State

C.0.6 Helper functions

```
1  def state_to_ket(state):
2      ket = ''
3      state = np.asarray(state)
4      n_qubits = int(np.log2(len(state)))
5      for i, amplitude in enumerate(state):
6          if abs(amplitude) > 1e-6:
7              if ket != '':
8                  ket += ' + '
9              amplitude_str = f'{amplitude:.2f}' if amplitude.imag != 0
       else f'{amplitude.real:.2f}'
10             ket += f'{amplitude_str}|{i:0{n_qubits}b}>'
11     return ket
```

Listing C.13: A function to convert statevector to Dirac notation.

```
1  def ket_to_state(ket):
2      # Split the ket string into terms
3      terms = ket.split(' + ')
4
5      # Extract the number of qubits from the first term
6      n_qubits = len(re.search(r'\|(\d+)\>', terms[0]).group(1))
7
8      # Initialize the state vector
9      state = np.zeros(2**n_qubits, dtype=complex)
10
11     for term in terms:
12         # Extract the amplitude and the state from the term
```

```
        amplitude_str, state_str = re.match(r'(-?\d+\.?\d*(?:[eE
    ][-+]?\d+)?j?)\|(\d+)\>', term).groups()

        # Convert the amplitude to a complex number
        amplitude = complex(amplitude_str)

        # Convert the state string to an integer
        state_int = int(state_str, 2)

        # Set the corresponding element of the state vector
        state[state_int] = amplitude

    return state
```

Listing C.14: A function to convert Dirac notation to statevector.

```
def execute_for_bloch(qc):
    simulator = Aer.get_backend('statevector_simulator')
    result = execute(qc, simulator).result()
    statevector = result.get_statevector()
    return statevector

def execute_for_histogram(qc):
    qc_with_meas = qc.copy()
    qc_with_meas.measure_all()
    simulator = Aer.get_backend('qasm_simulator')
    result = execute(qc_with_meas, simulator, shots=1000).result()
    counts = result.get_counts()
    return counts
```

Listing C.15: Run same circuit for different outputs.

Bibliography

[1] John Preskill. Quantum computing in the NISQ era and beyond. *Quantum*, 2:79, 2018.

[2] T.D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, and J.L. O'Brien. Quantum computers. *Nature*, 464:45–53, March 2010.

[3] Rodney Van Meter. A #quantumcomputerarchitecture tweetstorm. `https://doi.org/10.5281/zenodo.3496597`, sep 2019.

[4] Ashley Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2:15023, 2016.

[5] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in science & engineering*, 19(2):41–50, 2017.

[6] R. P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6):467–488, 1982.

[7] Mark Shand and Jean Vuillemin. Fast implementations of RSA cryptography. In *Proceedings of IEEE 11th Symposium on Computer Arithmetic*, pages 252–259. IEEE, 1993.

[8] Derrick H Lehmer and Richard E Powers. On factoring large numbers. *Bulletin of the American Mathematical Society*, 37(10):770–776, 1931.

[9] Adi Shamir. Factoring large numbers with the twinkle device. In *Cryptographic Hardware and Embedded Systems: First InternationalWorkshop, CHES'99 Worcester, MA, USA, August 12–13, 1999 Proceedings 1*, pages 2–12. Springer, 1999.

[10] Adi Shamir and Eran Tromer. Factoring large numbers with the twirl device. In *Annual International Cryptology Conference*, pages 1–26. Springer, 2003.

[11] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, oct 1997.

[12] Joe P Buhler, Hendrik W Lenstra, and Carl Pomerance. Factoring integers with the number field sieve. In *The development of the number field sieve*, pages 50–94. Springer, 1993.

[13] Dan Boneh and Ramarathnam Venkatesan. Breaking RSA may not be equivalent to factoring. In *Advances in Cryptology—EUROCRYPT'98: International Conference on the Theory and Application of Cryptographic Techniques Espoo, Finland, May 31–June 4, 1998 Proceedings 17*, pages 59–71. Springer, 1998.

[14] Aamer Nadeem and M Younus Javed. A performance comparison of data encryption algorithms. In *2005 international Conference on information and communication technologies*, pages 84–89. IEEE, 2005.

[15] Ashley Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2(1), jan 2016. https://doi.org/10.1038/2Fnpjqi.2015.23.

[16] Román Orús, Samuel Mugel, and Enrique Lizaso. Quantum computing for finance: Overview and prospects. *Reviews in Physics*, 4:100028, 2019.

[17] Daniel J Egger, Claudio Gambella, Jakub Marecek, Scott McFaddin, Martin Mevissen, Rudy Raymond, Andrea Simonetto, Stefan Woerner, and Elena Yndurain. Quantum computing for finance: State-of-the-art and future prospects. *IEEE Transactions on Quantum Engineering*, 1:1–24, 2020.

[18] Andreas Bayerstadler, Guillaume Becquin, Julia Binder, Thierry Botter, Hans Ehm, Thomas Ehmer, Marvin Erdmann, Norbert Gaus, Philipp Harbach, Maximilian Hess, et al. Industry quantum computing applications. *EPJ Quantum Technology*, 8(1):25, 2021.

[19] Abhishek Pandey and V Ramesh. Quantum computing for big data analysis. *Indian Journal of Science*, 14(43):98–104, 2015.

[20] Tawseef Ayoub Shaikh and Rashid Ali. Quantum computing in big data analytics: A survey. In *2016 IEEE international conference on computer and information technology (CIT)*, pages 112–115. IEEE, 2016.

[21] Frank Phillipson. Quantum computing in logistics and supply chain management-an overview. *arXiv preprint arXiv:2402.17520*, 2024.

[22] Charles H Bennett, François Bessette, Gilles Brassard, Louis Salvail, and John Smolin. Experimental quantum cryptography. *Journal of cryptology*, 5:3–28, 1992.

[23] Stefano Pirandola, Ulrik L Andersen, Leonardo Banchi, Mario Berta, Darius Bunandar, Roger Colbeck, Dirk Englund, Tobias Gehring, Cosmo Lupo, Carlo Ottaviani, et al. Advances in quantum cryptography. *Advances in optics and photonics*, 12(4):1012–1236, 2020.

[24] Nicolas Gisin, Grégoire Ribordy, Wolfgang Tittel, and Hugo Zbinden. Quantum cryptography. *Reviews of modern physics*, 74(1):145, 2002.

[25] Yudong Cao, Jonathan Romero, Jonathan P Olson, Matthias Degroote, Peter D Johnson, Mária Kieferová, Ian D Kivlichan, Tim Menke, Borja Peropadre, Nicolas PD Sawaya, et al. Quantum chemistry in the age of quantum computing. *Chemical reviews*, 119(19):10856–10915, 2019.

[26] John P Lowe and Kirk Peterson. *Quantum chemistry*. Elsevier, 2011.

[27] Dan-Bo Zhang, Hongxi Xing, Hui Yan, Enke Wang, and Shi-Liang Zhu. Selected topics of quantum computing for nuclear physics. *Chinese Physics B*, 30(2):020306, 2021.

[28] Iulia Buluta and Franco Nori. Quantum simulators. *Science*, 326(5949):108–111, 2009.

[29] Iulia M Georgescu, Sahel Ashhab, and Franco Nori. Quantum simulation. *Reviews of Modern Physics*, 86(1):153, 2014.

[30] Bela Bauer, Sergey Bravyi, Mario Motta, and Garnet Kin-Lic Chan. Quantum algorithms for quantum chemistry and quantum materials science. *Chemical Reviews*, 120(22):12685–12717, 2020.

[31] Hongbin Liu, Guang Hao Low, Damian S Steiger, Thomas Häner, Markus Reiher, and Matthias Troyer. Prospects of quantum computing for molecular sciences. *Materials Theory*, 6(1):11, 2022.

[32] Gautam Kumar, Sahil Yadav, Aniruddha Mukherjee, Vikas Hassija, and Mohsen Guizani. Recent advances in quantum computing for drug discovery and development. *IEEE Access*, 2024.

[33] Lov K. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Phys. Rev. Lett.*, 79:325–328, Jul 1997. `https://link.aps.org/doi/10.1103/PhysRevLett.79.325`.

[34] Mingsheng Ying. Quantum computation, quantum theory and ai. *Artificial Intelligence*, 174(2):162–176, 2010.

[35] Vedran Dunjko and Hans J Briegel. Machine learning & artificial intelligence in the quantum domain: a review of recent progress. *Reports on Progress in Physics*, 81(7):074001, 2018.

[36] NIST National Institute of Standards and Technology. The quantum algorithm zoo, 2021. `https://quantumalgorithmzoo.org/`.

[37] Arjen K Lenstra, Hendrik W Lenstra Jr, Mark S Manasse, and John M Pollard. The number field sieve. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 564–572, 1990.

[38] Michael Case. A beginner's guide to the general number field sieve. *Oregon State University, ECE575 Data Security and Cryptography Project*, 2003.

[39] Daniel J Bernstein and Arjen K Lenstra. A general number field sieve implementation. In *The development of the number field sieve*, pages 103–126. Springer, 2006.

[40] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[41] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[42] Diana Manjarres, Itziar Landa-Torres, Sergio Gil-Lopez, Javier Del Ser, Miren Nekane Bilbao, Sancho Salcedo-Sanz, and Zong Woo Geem. A survey on applications of the harmony search algorithm. *Engineering Applications of Artificial Intelligence*, 26(8):1818–1831, 2013.

[43] Osama Moh'D Alia and Rajeswari Mandava. The variants of the harmony search algorithm: an overview. *Artificial Intelligence Review*, 36:49–68, 2011.

[44] Damian R Musk. A comparison of quantum and traditional fourier transform computations. *Computing in Science & Engineering*, 22(6):103–110, 2020.

[45] Lars S Madsen, Fabian Laudenbach, Mohsen Falamarzi Askarani, Fabien Rortais, Trevor Vincent, Jacob FF Bulmer, Filippo M Miatto, Leonhard Neuhaus, Lukas G Helt, Matthew J Collins, et al. Quantum computational advantage with a programmable photonic processor. *Nature*, 606(7912):75–81, 2022.

[46] Chithralekha Balamurugan, Kalpana Singh, Ganeshvani Ganesan, and Muttukrishnan Rajarajan. Post-quantum and code-based cryptography—some prospective research directions. *Cryptography*, 5(4):38, 2021.

[47] Elija Perrier. Ethical quantum computing: A roadmap. *arXiv preprint arXiv:2102.00759*, 2021.

[48] David P. DiVincenzo. The physical implementation of quantum computation. *Fortschritte der Physik*, 48(9-11):771–783, sep 2000. https://doi.org/10.48550/arXiv.quant-ph/0002077.

[49] Wojciech Hubert Zurek. Decoherence, einselection, and the quantum origins of the classical. *Reviews of modern physics*, 75(3):715, 2003.

[50] Maximilian Schlosshauer. Decoherence, the measurement problem, and interpretations of quantum mechanics. *Reviews of Modern physics*, 76(4):1267, 2005.

[51] Wojciech Hubert Zurek and Juan Pablo Paz. Decoherence, chaos, and the second law. *Physical Review Letters*, 72(16):2508, 1994.

[52] Maximilian A Schlosshauer. *Decoherence: and the quantum-to-classical transition*. Springer Science & Business Media, 2007.

[53] Joschka Roffe. Quantum error correction: an introductory guide. *Contemporary Physics*, 60(3):226–245, 2019.

[54] Raymond Laflamme, Cesar Miquel, Juan Pablo Paz, and Wojciech Hubert Zurek. Perfect quantum error correcting code. *Physical Review Letters*, 77(1):198, 1996.

[55] A Robert Calderbank and Peter W Shor. Good quantum error-correcting codes exist. *Physical Review A*, 54(2):1098, 1996.

[56] Emanuel Knill and Raymond Laflamme. Theory of quantum error-correcting codes. *Physical Review A*, 55(2):900, 1997.

[57] Nathalie P de Leon, Kohei M Itoh, Dohun Kim, Karan K Mehta, Tracy E Northup, Hanhee Paik, BS Palmer, Nitin Samarth, Sorawis Sangtawesin, and David W Steuerman. Materials challenges and opportunities for quantum computing hardware. *Science*, 372(6539):eabb2823, 2021.

[58] David Kielpinski, Chris Monroe, and David J Wineland. Architecture for a large-scale ion-trap quantum computer. *Nature*, 417(6890):709–711, 2002.

[59] MF Gonzalez-Zalba, S de Franceschi, E Charbon, T Meunier, M Vinet, and AS Dzurak. Scaling silicon-based quantum computing using CMOS technology. *Nature Electronics*, 4(12):872–884, 2021.

[60] Shuntaro Takeda and Akira Furusawa. Toward large-scale fault-tolerant univeRSAl photonic quantum computing. *APL Photonics*, 4(6):060902, 2019.

[61] R Gilmore, CM Bowden, and LM Narducci. Classical-quantum correspondence for multilevel systems. *Physical Review A*, 12(3):1019, 1975.

[62] Wojciech H Zurek. Decoherence, chaos, quantum-classical correspondence, and the algorithmic arrow of time. *Physica Scripta*, 1998(T76):186, 1998.

[63] Enrique Moguel, Javier Berrocal, José García-Alonso, and Juan Manuel Murillo. A roadmap for quantum software engineering: Applying the lessons learned from the classics. In *Q-SET@ QCE*, pages 5–13, 2020.

[64] Jianjun Zhao. Quantum software engineering: Landscapes and horizons. *arXiv preprint arXiv:2007.07047*, 2020.

[65] Manuel A Serrano, Ricardo Pérez-Castillo, and Mario Piattini. *Quantum software engineering*. Springer Nature, 2022.

[66] Andriy Miranskyy, Lei Zhang, and Javad Doliskani. On testing and debugging quantum software. *arXiv preprint arXiv:2103.09172*, 2021.

[67] Manuel De Stefano, Fabiano Pecorelli, Dario Di Nucci, Fabio Palomba, and Andrea De Lucia. Software engineering for quantum programming: How far are we? *Journal of Systems and Software*, 190:111326, 2022.

[68] Mingsheng Ying, Nengkun Yu, Yuan Feng, and Runyao Duan. Verification of quantum programs. *Science of Computer Programming*, 78(9):1679–1700, 2013.

[69] Marco Lewis, Sadegh Soudjani, and Paolo Zuliani. Formal verification of quantum programs: Theory, tools, and challenges. *ACM Transactions on Quantum Computing*, 5(1):1–35, 2023.

[70] Fabian Bauer-Marquart, Stefan Leue, and Christian Schilling. symqv: automated symbolic verification of quantum programs. In *International Symposium on Formal Methods*, pages 181–198. Springer, 2023.

[71] Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. Coqq: Foundational verification of quantum programs. *Proceedings of the ACM on Programming Languages*, 7(POPL):833–865, 2023.

[72] John Clarke and Frank K. Wilhelm. Superconducting quantum bits. *Nature*, 453(7198):1031–1042, 2008.

[73] Michel H. Devoret and Robert J. Schoelkopf. Superconducting circuits for quantum information: An outlook. *Science*, 339(6124):1169–1174, 2013.

[74] Philip Krantz, Morten Kjaergaard, Fei Yan, Terry P. Orlando, Simon Gustavsson, and William D. Oliver. A quantum engineer's guide to superconducting qubits. *Applied Physics Reviews*, 6(2):021318, 2019.

[75] Rainer Blatt and David Wineland. Entangled states of trapped atomic ions. *Nature*, 453(7198):1008–1015, 2008.

[76] C. Monroe and J. Kim. Scaling the ion trap quantum processor. *Science*, 339(6124):1164–1169, 2013.

[77] Hartmut Häffner, Christian F. Roos, and Rainer Blatt. Quantum computing with trapped ions. *Physics Reports*, 469(4):155–203, 2008.

[78] Sankar Das Sarma, Michael Freedman, and Chetan Nayak. Topologically protected qubits from a possible non-abelian fractional quantum hall state. *Phys. Rev. Lett.*, 94:166802, Apr 2005. https://link.aps.org/doi/10.1103/PhysRevLett.94.166802.

[79] Chetan Nayak, Steven H. Simon, Ady Stern, Michael Freedman, and Sankar Das Sarma. Non-abelian anyons and topological quantum computation. *Reviews of Modern Physics*, 80(3):1083–1159, 2008.

[80] A. Yu. Kitaev. Fault-tolerant quantum computation by anyons. *Annals of Physics*, 303(1):2–30, 2003.

[81] J. L. O'Brien. Optical quantum computing. *Science*, 318(5856):1567–1570, 2007.

[82] J. L. O'Brien, A. Furusawa, and J. Vučković. Photonic quantum technologies. *Nature Photonics*, 3(12):687–695, 2009.

[83] Emanuel Knill, Raymond Laflamme, and Gerard J. Milburn. A scheme for efficient quantum computation with linear optics. *Nature*, 409(6816):46–52, 2001.

[84] Daniel Loss and David P. DiVincenzo. Quantum computation with quantum dots. *Physical Review A*, 57(1):120–126, 1998.

[85] J. M. Elzerman, R. Hanson, L. H. Willems van Beveren, B. Witkamp, L. M. K. Vandersypen, and L. P. Kouwenhoven. Single-shot read-out of an individual electron spin in a quantum dot. *Nature*, 430(6998):431–435, 2004.

[86] R. Hanson, L. P. Kouwenhoven, J. R. Petta, S. Tarucha, and L. M. K. Vandersypen. Spins in few-electron quantum dots. *Reviews of Modern Physics*, 79(4):1217–1265, 2007.

[87] M. D. Lukin and M. I. Imamoğlu. Controlling photons using electromagnetically induced transparency. *Nature*, 413(6853):273–276, 2001.

[88] Anders Sørensen, Mikhail Lukin, Ignacio Cirac, and Peter Zoller. Quantum computing with collective ensembles of multilevel systems. *Nature*, 409(6816):63–66, 2001.

[89] B. E. Kane. A silicon-based nuclear spin quantum computer. *Nature*, 393(6681):133–137, 1998.

[90] Daniel A Lidar and Todd A Brun. *Quantum error correction*. Cambridge university press, 2013.

[91] Simon J Devitt, William J Munro, and Kae Nemoto. Quantum error correction for beginners. *Reports on Progress in Physics*, 76(7):076001, 2013.

[92] Li-Heng Henry Chang, Shea Roccaforte, Rose Xu, and Paul Cadden-Zimansky. Geometric visualizations of single and entangled qubits. *arXiv preprint arXiv:2212.03448*, 2022.

[93] Jonas Bley, Eva Rexigel, Alda Arias, Nikolas Longen, Lars Krupp, Maximilian Kiefer-Emmanouilidis, Paul Lukowicz, Anna Donhauser, Stefan Küchemann, Jochen Kuhn, et al. Visualizing entanglement, measurements and unitary operations in multi-qubit systems. *arXiv preprint arXiv:2305.07596*, 2023.

[94] Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. Going beyond bell's theorem. pages 69–72, 1989.

[95] Daniel M. Greenberger, Michael A. Horne, Abner Shimony, and Anton Zeilinger. Bell's theorem without inequalities. *American Journal of Physics*, 58:1131–1143, 1990.

[96] Hans J. Briegel and Robert Raussendorf. Persistent entanglement in arrays of interacting particles. *Physical Review Letters*, 86:910–913, 2001.

[97] Qiskit documentation, Mar 2020. `https://qiskit.org/documentation`.

[98] Shin Nishio, Yulu Pan, Takahiko Satoh, Hideharu Amano, and Rodney Van Meter. Extracting success from IBM's 20-qubit machines using error-aware compilation. *J. Emerg. Technol. Comput. Syst.*, 16(3), May 2020. `https://doi.org/10.1145/3386162`.

[99] Andrew W Cross, Lev S Bishop, Sarah Sheldon, Paul D Nation, and Jay M Gambetta. Validating quantum computers using randomized model circuits. *Physical Review A*, 100(3):032328, 2019.

[100] Andrew M Steane. Overhead and noise threshold of fault-tolerant quantum error correction. *Physical Review A*, 68(4):042322, 2003.

[101] Dario Gil and William MJ Green. The future of computing: Bits+ neurons+ qubits. *arXiv preprint arXiv:1911.08446*, 2019.

[102] Charles H Baldwin, Karl Mayer, Natalie C Brown, Ciarán Ryan-Anderson, and David Hayes. Re-examining the quantum volume test: Ideal distributions, compiler optimizations, confidence intervals, and scalable resource estimations. *Quantum*, 6:707, 2022.

[103] David C McKay, Ian Hincks, Emily J Pritchett, Malcolm Carroll, Luke CG Govia, and Seth T Merkel. Benchmarking quantum processor performance at scale. *arXiv preprint arXiv:2311.05933*, 2023.

[104] Xuedong Hu, Rogerio de Sousa, and S Das Sarma. Decoherence and dephasing in spin-based solid state quantum computers. In *Foundations Of Quantum Mechanics In The Light Of New Technology: ISQM—Tokyo'01*, pages 3–11. World Scientific, 2002.

[105] Benjamin Weder, Johanna Barzen, Frank Leymann, Marie Salm, and Daniel Vietz. The quantum software lifecycle. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Architectures and Paradigms for Engineering Quantum Software*, pages 2–9, 2020.

[106] Robert S. Sutor. *Dancing with Qubits*. Packt Publishing, 2019.

[107] John Preskill. Lecture notes for ph/cs 219 at caltech. available on the web, 2019.

[108] Bettina Heim, Mathias Soeken, Sarah Marshall, Chris Granade, Martin Roetteler, Alan Geller, Matthias Troyer, and Krysta Svore. Quantum programming languages. *Nature Reviews Physics*, 2(12):709–722, 2020.

[109] Simon Gay. Quantum programming languages: Survey and bibliography. *Bulletin of the European Association for Theoretical Computer Science*, June 2005.

[110] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 286–300, 2020.

[111] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 333–342, 2013.

[112] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, D Bucher, FJ Cabrera-Hernández, J Carballo-Franquis, A Chen, CF Chen, et al. Qiskit: An open-source framework for quantum computing. 16, 2019.

[113] Google AI. Cirq, a python framework for creating, editing, and invoking noisy intermediate scale quantum (NISQ) circuits, 2018.

[114] Rigetti Computing. Pyquil: Quantum programming in python. `https://github.com/rigetti/pyquil`.

[115] University of Melbourne. QUI, a quantum user interface tool, 2018. `https://qui.research.unimelb.edu.au`.

[116] Alan C Santos. The IBM quantum computer and the IBM quantum experience. *arXiv preprint arXiv:1610.06980*, 2016.

[117] Craig Gidney. Algorithmic assertions - craig gidney's computer science blog. `https://algassert.com/`.

[118] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

[119] Aleks Kissinger and John van de Wetering. Pyzx: Large scale automated diagrammatic reasoning. *arXiv preprint arXiv:1904.04735*, 2019.

[120] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t| ket>: a retargetable compiler for NISQ devices. *Quantum Science and Technology*, 6(1):014003, 2020.

[121] Prakash Murali, Norbert Matthias Linke, Margaret Martonosi, Ali Javadi Abhari, Nhung Hong Nguyen, and Cinthia Huerta Alderete. Full-stack, real-system quantum computer studies: Architectural comparisons and design insights. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 527–540. IEEE, 2019.

[122] Michael Booth, Jesse Berwald, Uchenna Chukwu, John Dawson, Raouf Dridi, DeYung Le, Mark Wainger, and Steven P Reinhardt. Qci qbsolv delivers strong classical performance for quantum-ready formulation. *arXiv preprint arXiv:2005.11294*, 2020.

[123] Frederic T Chong, Diana Franklin, and Margaret Martonosi. Programming languages and compiler design for realistic quantum hardware. *Nature*, 549(7671):180–187, 2017.

[124] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Sylvain Collange, and Fernando Magno Quintao Pereira. Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 113–125, New York, NY, USA, 2018. ACM. http://doi.acm.org/10.1145/3168822.

[125] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm. *arXiv preprint arXiv:1411.4028*, 2014.

[126] Tad Hogg and Dmitriy Portnov. Quantum optimization. *Information Sciences*, 128(3):181 – 197, 2000.

[127] Jarrod R McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, 18(2):023023, 2016.

[128] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J Love, Alán Aspuru-Guzik, and Jeremy L O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature communications*, 5, 2014.

[129] C A Trugenberger. Quantum optimization for combinatorial searches. *New Journal of Physics*, 4:26–26, apr 2002.

[130] J. R. Johansson, P. D. Nation, and F. Nori. Qutip: An open-source python framework for the dynamics of open quantum systems. *Computational Physics Communications*, 183:1760, 2012.

[131] K. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler. Q# : Enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL2018)*, 2018. https://doi.org/10.1145/3183895.3183901.

[132] N. Khammassi, I. Ashraf, X. Fu, C.G. Almudever, and K. Bertels. Qx: A high-performance quantum computer simulation platform. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2017*, pages 464–469, 2017.

[133] D-Wave Systems Inc. D-wave ocean documentation. https://docs.ocean.dwavesys.com/en/stable/.

[134] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M Sohaib Alam, Guillermo Alonso-Linaje, B AkashNarayanan, Ali Asadi, et al. Pennylane: Automatic differentiation of hybrid quantum-classical computations. *arXiv preprint arXiv:1811.04968*, 2018.

[135] Stewart Smith. Quantumjavascript. `https://quantumjavascript.app/`.

[136] Constantin Gonzalez. Cloud based qc with amazon braket. *Digitale Welt*, 5:14–17, 2021.

[137] Xin-Chuan Wu, Shavindra P Premaratne, and Kevin Rasch. A comprehensive introduction to the intel quantum sdk. In *Proceedings of the 2023 Introduction on Hybrid Quantum-Classical Programming Using C++ Quantum Extension*, pages 1–2. 2023.

[138] Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp. Tight bounds on quantum searching. *Fortschritte der Physik: Progress of Physics*, 46(4-5):493–505, 1998.

[139] Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[140] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[141] J Longina Castellanos, Susana Gómez, and Valia Guerra. The triangle method for finding the corner of the l-curve. *Applied Numerical Mathematics*, 43(4):359–373, 2002.

[142] Virginia Vassilevska Williams, Joshua R Wang, Ryan Williams, and Huacheng Yu. Finding four-node subgraphs in triangle time. In *Proceedings of the twenty-sixth annual ACM-SIAM symposium on discrete algorithms*, pages 1671–1680. SIAM, 2014.

[143] François Le Gall. Improved quantum algorithm for triangle finding via combinatorial arguments. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 216–225. IEEE, 2014.

[144] Frédéric Magniez, Miklos Santha, and Mario Szegedy. Quantum algorithms for the triangle problem. *SIAM Journal on Computing*, 37(2):413–424, 2007.

[145] Andrew M Childs, Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. Finding cliques by quantum adiabatic evolution. *arXiv preprint quant-ph/0012104*, 2000.

[146] Andrew M Childs and Jason M Eisenberg. Quantum algorithms for subset finding. *arXiv preprint quant-ph/0311038*, 2003.

[147] Gabriel Valiente. *Algorithms on trees and graphs*. Springer Science & Business Media, 2013.

[148] M Pelillo. Encyclopedia of optimization, chapter heuristics for maximum clique and independent set, 2001.

[149] Nicholas Rhodes, Peter Willett, Alain Calvet, James B Dunbar, and Christine Humblet. Clip: similarity searching of 3d databases using clique detection. *Journal of chemical information and computer sciences*, 43(2):443–448, 2003.

[150] Frederick S Kuhl, Gorden M Crippen, and Donald K Friesen. A combinatorial algorithm for calculating ligand binding. *Journal of Computational Chemistry*, 5(1):24–34, 1984.

[151] National Research Council et al. *Mathematical challenges from theoretical/computational chemistry*. National Academies Press, 1995.

[152] William HE Day and David Sankoff. Computational complexity of inferring phylogenies by compatibility. *Systematic Biology*, 35(2):224–229, 1986.

[153] Dongdai Lin, Xiaofeng Wang, and Moti Yung. *Information security and cryptology: 11th International Conference, Inscrypt 2015, Beijing, China, November 1-3, 2015, revised selected papers*. Springer, 2016.

[154] R. H. Dicke. Coherence in spontaneous radiation processes. *Phys. Rev.*, 93:99–110, Jan 1954. https://link.aps.org/doi/10.1103/PhysRev.93.99.

[155] S K Özdemir, J Shimamura, and N Imoto. A necessary and sufficient condition to play games in quantum mechanical settings. *New Journal of Physics*, 9(2):43–43, Feb 2007. http://dx.doi.org/10.1088/1367-2630/9/2/043.

[156] R. Prevedel, G. Cronenberg, M. S. Tame, M. Paternostro, P. Walther, M. S. Kim, and A. Zeilinger. Experimental realization of dicke states of up to six qubits for multiparty quantum networking. *Physical Review Letters*, 103(2), Jul 2009. http://dx.doi.org/10.1103/PhysRevLett.103.020503.

[157] Géza Tóth. Multipartite entanglement and high-precision metrology. *Physical Review A*, 85(2), Feb 2012. http://dx.doi.org/10.1103/PhysRevA.85.022322.

[158] Andreas Bärtschi and Stephan Eidenbenz. Deterministic preparation of dicke states. In *International Symposium on Fundamentals of Computation Theory*, pages 126–139. Springer, 2019.

[159] Diogo Cruz, Romain Fournier, Fabien Gremion, Alix Jeannerot, Kenichi Komagata, Tara Tosic, Jarla Thiesbrummel, Chun Lam Chan, Nicolas Macris, Marc-André Dupertuis, et al. Efficient quantum algorithms for ghz and w states, and implementation on the IBM quantum computer. *Advanced Quantum Technologies*, 2(5-6):1900015, 2019.

[160] Andrew M Childs, Eddie Schoute, and Cem M Unsal. Circuit transformations for quantum architectures. *arXiv preprint arXiv:1902.09102*, 2019.

[161] Pablo Andrés-Martínez and Chris Heunen. Automated distribution of quantum circuits via hypergraph partitioning. *Physical Review A*, 100(3):032308, 2019.

[162] Shin Nishio, Yulu Pan, Takahiko Satoh, Hideharu Amano, and Rodney Van Meter. Extracting Success from IBM's 20-Qubit Machines Using Error-Aware Compilation, 2020. to appear; preprint available as arXiv:1903.10963.

[163] Beatrice Nash, Vlad Gheorghiu, and Michele Mosca. Quantum circuit optimizations for NISQ architectures. *Quantum Science and Technology*, 5(2):025010, mar 2020. https://doi.org/10.1088/2F2058-9565/2Fab79b1.

[164] Swamit S Tannu and Moinuddin K Qureshi. Mitigating measurement errors in quantum computers by exploiting state-dependent bias. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pages 279–290, 2019.

[165] Robin Harper and Steven T Flammia. Fault-tolerant logical gates in the IBM quantum experience. *Physical review letters*, 122(8):080504, 2019.

[166] Elementary operations qiskit 0.10.4 documentation. https://bit.ly/367Dpcw.

[167] Alessandro Orso and Gregg Rothermel. Software testing: a research travelogue (2000–2014). In *Future of Software Engineering Proceedings*, pages 117–132. 2014.

[168] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

[169] Kai Pan, Sunghun Kim, and E James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14:286–315, 2009.

[170] Robert Love. *Linux Kernel Development: Linux Kernel Development _p3*. Pearson Education, 2010.

[171] Tegawendé F Bissyandé, Laurent Réveillère, Julia L Lawall, and Gilles Muller. Diagnosys: automatic generation of a debugging interface to the linux kernel. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 60–69, 2012.

[172] Pengzhan Zhao, Jianjun Zhao, Zhongtao Miao, and Shuhan Lan. Bugs4q: A benchmark of real bugs for quantum programs. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1373–1376. IEEE, 2021.

[173] José Campos and André Souto. Qbugs: A collection of reproducible bugs in quantum algorithms and a supporting infrastructure to enable controlled quantum software testing and debugging experiments. In *2021 IEEE/ACM 2nd International Workshop on Quantum Software Engineering (Q-SE)*, pages 28–32. IEEE, 2021.

[174] Junjie Luo, Pengzhan Zhao, Zhongtao Miao, Shuhan Lan, and Jianjun Zhao. A comprehensive study of bug fixes in quantum programs. *arXiv preprint arXiv:2201.08662*, 2022.

[175] Matteo Paltenghi and Michael Pradel. Bugs in quantum computing platforms: An empirical study. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022. `https://doi.org/10.1145/3527330`.

[176] Matteo Paltenghi and Michael Pradel. Morphq: Metamorphic testing of the qiskit quantum computing platform. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2413–2424, 2023.

[177] Danny Cohen. On holy wars and a plea for peace. *Computer*, 14(10):48–54, 1981.

[178] Michael A Nielsen. Cluster-state quantum computation. *Reports on Mathematical Physics*, 57(1):147–161, 2006.

[179] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[180] A Kotok. DEC debugging tape. *Memo MIT-1 (rev.), MIT (Dec. 1961)*, 1961. `https://www.computerhistory.org/pdp-1/189cc577e7b13aafbb0efab4c547d262/`.

[181] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[182] Butler W Lampson and Kenneth A Pier. A processor for a high-performance personal computer. In *Proceedings of the 7th annual symposium on Computer Architecture*, pages 146–160, 1980.

[183] Wei Tang, Teague Tomesh, Martin Suchara, Jeffrey Larson, and Margaret Martonosi. CutQC: using small quantum computers for large quantum circuit evaluations. In *Proceedings of the 26th ACM International conference on architectural support for programming languages and operating systems*, pages 473–486, 2021.

[184] Andrew Eddins, Mario Motta, Tanvi P Gujarati, Sergey Bravyi, Antonio Mezzacapo, Charles Hadfield, and Sarah Sheldon. Doubling the size of quantum simulators by entanglement forging. *PRX Quantum*, 3(1):010309, 2022.

[185] C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Develop.*, 17:525–532, 1973.

[186] C. H. Bennett. Notes on the history of reversible computation. *IBM J. of Research and Development*, 32(1), 1988. reprinted in IBM J. R.&D. Vol. 44 No. 1/2, Jan./Mar. 2000, pp. 270–277.

[187] Charles H. Bennett. Notes on the history of reversible computation. *IBM J. of Research and Development*, 32(1), 1988. reprinted in IBM J. R.&D. Vol. 44 No. 1/2, Jan./Mar. 2000, pp. 270–277.

[188] Steph Foulds, Viv Kendon, and Tim Spiller. The controlled swap test for determining quantum entanglement. *Quantum Science and Technology*, 6(3):035002, 2021.

[189] Swamit S Tannu and Moinuddin K Qureshi. Mitigating measurement errors in quantum computers by exploiting state-dependent bias. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 279–290, 2019.

[190] Kristan Temme, Sergey Bravyi, and Jay M. Gambetta. Error mitigation for short-depth quantum circuits. *Phys. Rev. Lett.*, 119:180509, Nov 2017. `https://link.aps.org/doi/10.1103/PhysRevLett.119.180509`.

[191] Suguru Endo, Simon C. Benjamin, and Ying Li. Practical quantum error mitigation for near-future applications. *Phys. Rev. X*, 8:031027, Jul 2018. `https://link.aps.org/doi/10.1103/PhysRevX.8.031027`.

[192] Ying Li and Simon C. Benjamin. Efficient variational quantum simulator incorporating active error minimization. *Phys. Rev. X*, 7:021050, Jun 2017. `https://link.aps.org/doi/10.1103/PhysRevX.7.021050`.

[193] Gilles Brassard, Peter Hoyer, Michele Mosca, and Alain Tapp. Quantum amplitude amplification and estimation. *Contemporary Mathematics*, 305:53–74, 2002.

[194] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[195] Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. Projection-based runtime assertions for testing and debugging quantum programs. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. https://doi.org/10.1145/3428218.

[196] Yipeng Huang and Margaret Martonosi. Statistical assertions for validating patterns and finding bugs in quantum programs. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 541–553, 2019.

[197] Nan Jiang, Zichen Wang, and Jian Wang. Debugging quantum programs using probabilistic quantum cloning. *Available at SSRN 4511772*, 2023.

[198] Robert Rand, Jennifer Paykin, and Steve Zdancewic. Qwire practice: Formal verification of quantum circuits in coq. *arXiv preprint arXiv:1803.00699*, 2018.

[199] Shreya Srivastva and Saru Dhir. Debugging approaches on various software processing levels. In *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, volume 2, pages 302–306. IEEE, 2017.

[200] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. Apollo: Automatic detection and diagnosis of performance regressions in database systems. *Proceedings of the VLDB Endowment*, 13(1):57–70, 2019.

[201] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on software engineering*, 32(10):831–848, 2006.

[202] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2):67–92, 2008.

[203] Qisheng Wang, Zhicheng Zhang, Kean Chen, Ji Guan, Wang Fang, Junyi Liu, and Mingsheng Ying. Quantum algorithm for fidelity estimation. *IEEE Transactions on Information Theory*, 69(1):273–282, 2022.

[204] Marcus Cramer, Martin B Plenio, Steven T Flammia, Rolando Somma, David Gross, Stephen D Bartlett, Olivier Landon-Cardinal, David Poulin, and Yi-Kai Liu. Efficient quantum state tomography. *Nature communications*, 1(1):149, 2010.

[205] Matthias Christandl and Renato Renner. Reliable quantum state tomography. *Physical Review Letters*, 109(12):120403, 2012.

[206] Maurice S Beck et al. *Process tomography: principles, techniques and applications*. Butterworth-Heinemann, 2012.

[207] Masoud Mohseni, Ali T Rezakhani, and Daniel A Lidar. Quantum-process tomography: Resource analysis of different strategies. *Physical Review A*, 77(3):032322, 2008.

[208] Giacomo Torlai, Christopher J Wood, Atithi Acharya, Giuseppe Carleo, Juan Carrasquilla, and Leandro Aolita. Quantum process tomography with unsupervised learning and tensor networks. *Nature Communications*, 14(1):2858, 2023.

[209] Emanuel Knill, Dietrich Leibfried, Rolf Reichle, Joe Britton, R Brad Blakestad, John D Jost, Chris Langer, Roee Ozeri, Signe Seidelin, and David J Wineland. Randomized benchmarking of quantum gates. *Physical Review A*, 77(1):012307, 2008.

[210] Leonardo Zambrano, Luciano Pereira, Sebastián Niklitschek, and Aldo Delgado. Estimation of pure quantum states in high dimension at the limit of quantum accuracy through complex optimization and statistical inference. *Scientific Reports*, 10(1):12781, 2020.

[211] Ariel Bendersky, Fernando Pastawski, and Juan Pablo Paz. Selective and efficient quantum process tomography. *Physical Review A*, 80(3):032116, 2009.

[212] Ariel Bendersky, Fernando Pastawski, and Juan Pablo Paz. Selective and efficient estimation of parameters for quantum process tomography. *Physical review letters*, 100(19):190403, 2008.

[213] Akshay Gaikwad, Krishna Shende, Arvind, and Kavita Dorai. Implementing efficient selective quantum process tomography of superconducting quantum gates on IBM quantum experience. *Scientific reports*, 12(1):3688, 2022.

[214] Scott Aaronson. *Quantum computing since Democritus*. Cambridge University Press, 2013.

[215] Alice X Zheng, Michael I Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: simultaneous identification of multiple bugs. In *Proceedings of the 23rd international conference on Machine learning*, pages 1105–1112, 2006.

[216] David Andrzejewski, Anne Mulhern, Ben Liblit, and Xiaojin Zhu. Statistical debugging using latent topic models. In *European conference on machine learning*, pages 6–17. Springer, 2007.