UNIVERSITY OF TECHNOLOGY SYDNEY

The Architecture of a Quantum Programming Environment

by

Shusen Liu

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Supervisor: Prof. Runyao Duan

Co-supervisor: Prof. Mingsheng Ying

at the

Centre for Quantum Software and Information

Faculty of Engineering and Information Technology

University of Technology Sydney

October 2018

Certificate of Original Authorship

I, *Shusen Liu* declare that this thesis, is submitted in fulfilment of the requirements for the award of *Doctor of Philosophy Degree*, in *Software Engineering at the Faculty of Engineering and Information Technology* at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise reference or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This thesis is the result of a research candidature conducted jointly with Sun Yat-sen University as part of a collaborative doctoral degree.

This document has not been submitted for qualifications at any other academic institution.

Signed:

Production Note:
Signature removed prior to publication.

Date:

October 2018

The Architecture of a Quantum Programming Environment

by

Shusen Liu

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Abstract

This thesis presents the architecture of quantum programming environment, called QSI, along with its related modules and several quantum experiments. The environment is based on one specific quantum language, namely quantum **while**-language. Some partial experimental results are also presented within QSI.

The first part relates to the architecture, the designing and the implementation of quantum programming environment which provides a new, powerful and flexible environment for developing and implementing quantum programs. First, we study the possible structure of the programming environment which supports a measurement-based case statement and a measurement-based **while**-loop. These two program constructs are extremely convenient for describing large-scale quantum algorithms, such as quantum random walk-based algorithms. We also define a new assembly language called f-QASM (Quantum Assembly Language with feedback) as an interactive command set. The assembly language is compatible with other low-level instruction sets and can be used to directly drive quantum hardware. Moreover, the simulation of syntax of quantum program and the behaviours within the architecture on the classical computer are discussed. Finally, we consider the work-flow which contains the decomposition of unitary matrix to achieve the goal that executing on Noisy Intermediate-Scale Quantum Computer.

The second part concerns the modules based on quantum programming environment: termination analysis module, detective separable unitary module and quantum control module. Along with the architecture, we bring an essential module - termination analysis

module for the loop structure. It can analyze sub-bodies of quantum program and suggest the critical termination information. In addition, we improve the Jordan decomposition step in the original algorithm which consumes extended period for analyzing. This improvement also makes the module more robust on executing. A fast permutation algorithm module clarifies the re-ordering algorithm in case of qubits system. It regenerates the program (unitary operator) which is not in pre-ordered sequence. In the detective separable unitary module, we prove sufficient conditions for separable unitary and its approximate scenario. The result shows there does not exist a universal algorithm for potential parallel executing quantum programs without communications (classical or quantum communications). However, in approximate, there exists a scheme for parallel computing without the help of communication. In this part, two examples for parallel computing are given. Last, in quantum control module, an algorithm is suggested towards automatically generating quantum circuits for quantum case-statement. We believe these analysis modules can help the compiler to optimize the implementation of quantum algorithms.

The third part is devoted to quantum experiment. First, we focus several experiments which can be operated directly by QSI : Qloop, BB84 protocol and Grover search algorithm. After that, with the help of IBM's QISKit, two impressive experiments: distinguishing unitary gates and Bell states are given on real quantum computer. Finally, we combine QSI with Microsoft's LIQU$i|\rangle$ to implement quantum case-statement. These experiments significantly show the quantum power and the scalable framework of the quantum programming environment in practice.

Acknowledgements

The research of this dissertation could not have been performed if not for the assistance, patience, and support of many individuals. First, I would like to express my sincere gratitude to my supervisor Prof. Runyao Duan for supervising my research. His insightful and rigorous instruction has motivated me to constantly purse studies in the quantum area. His encouragement to not give up when the enormous scope of QSI become too overwhelming was my last resource for achieving this goal. It sincerely touched me that he even contributed an amount some of his own time to debugging and the details of the designing. Also, I wish to thank Prof. Mingsheng Ying for introducing me to the topic and for his support on the way. His knowledge and understanding broadened my views on quantum programming and guided me in the right direction.

I would additionally extend my deepest gratitude to Mr Yang He for sharing his wealth of experience in classical programming with me and saving me with some programming techniques. Without the wisdom and skills he has accumulated over the last two decades, our programming environment would not have been developed so quickly.

I also want to show my thanks to Prof. Kan He. The routine discussions during his visit to Sydney illuminated my thinking on fundamental knowledge and made the 'impossible' task of designing the compiler possible.

I also express my thanks to Jemima Moore, who spends much time on proofreading this thesis after final examination. She examines the entire thesis carefully and thoroughly and gives many useful suggestions.

Further, I wish to thank my parents for supporting me as a higher degree research student. They provided me with all their love and care and asked for nothing in return.

Finally, and in particular, I wish to extend my thanks to my love, Ms Huanhua Zhang. She has spent two years with me and fully devoted herself to supporting my research.

Contents

Declaration of Authorship iii

Abstract v

Acknowledgements vii

List of Figures xiii

List of Tables xv

1 Introduction 1
- 1.1 Thesis . . . 1
- 1.2 Background . . . 2
- 1.3 Aims, Objectives and Significance . . . 3
  - 1.3.1 Aims . . . 3
  - 1.3.2 Objectives . . . 4
  - 1.3.3 Significance . . . 5
- 1.4 Research Methods . . . 7
- 1.5 Literature Review . . . 8
  - 1.5.1 Programming Languages and Platforms . . . 8
  - 1.5.2 Quantum Circuits Synthesis and Decompostion . . . 9
  - 1.5.3 Termination Analysis . . . 10
  - 1.5.4 Unitary Discrimination . . . 11
  - 1.5.5 Quantum Control . . . 13
- 1.6 Preliminaries . . . 14
  - 1.6.1 Quantum **while**-language . . . 14
  - 1.6.2 QuGCL . . . 15
- 1.7 Overview . . . 18

2 Quantum Programming Environment 21
- 2.1 Introduction . . . 21
- 2.2 The Structure of QSI . . . 22
  - 2.2.1 Basic Features of QSI . . . 22
  - 2.2.2 Main Components of QSI . . . 24
  - 2.2.3 Implementation of QSI . . . 25

2.3 The Quantum Compiler . . . 27
2.3.1 f-QASM . . . 27
2.3.2 Basic Definitions in f-QASM . . . 28
2.3.3 f-QASM Examples . . . 29
2.3.4 Decomposition of a General Unitary Transformation . . . 32
2.4 The Quantum Simulator . . . 33
2.4.1 Quantum Types . . . 33
2.4.2 Simulating of Quantum Behaviors . . . 35
2.4.3 Simulating Operational Semantics in the Quantum **while**-language 36

3 QSI Modules 41
3.1 Separation of Multipartite Quantum Gates . . . 41
3.1.1 Introduction for Separability of Multipartite Quantum Gates . . . 41
3.1.2 Theoretical Analysis of Quantum Gates Separation . . . 42
3.1.3 Approximate Separation of Multipartite Gates . . . 52
3.1.4 Approximate Separation in a 2-qubit $\frac{1}{2}$-spin System . . . 55
3.1.5 Conclusion and Discussion . . . 58
3.2 Fast Permutation and its Application in Compiler Module . . . 59
3.2.1 Introduction for Permutation . . . 59
3.2.2 General Algorithm for Permutation . . . 60
3.2.3 A Fast Permutation Algorithm Based on Fixed and Ordered Basis . 60
3.3 Termination Analysis Module . . . 62
3.3.1 Introduction for Termination Module . . . 62
3.3.2 Definitions and Theorems . . . 62
3.3.3 Algorithms for Termination Analysis . . . 67
3.3.4 Termination Examples- Qloop . . . 69
3.3.5 Conclusion and Discussion . . . 70
3.4 Quantum Control Module . . . 71
3.4.1 Introduction . . . 71
3.4.2 QMUX Preliminaries . . . 72
3.4.3 Solving Equations . . . 74

4 Experiments 77
4.1 Experiments with QSI . . . 77
4.1.1 Configuration . . . 77
4.1.2 Experiments . . . 79
4.1.3 Serious Coding . . . 96
4.2 Experiments with QISKit and QSI . . . 107
4.2.1 Implementing the Perfect Discrimination of Unitary Operators on IBMQ Cloud Quantum Computer . . . 107
4.2.2 Distinguishing Bell States with Local Measurement on IBMQ . . . 115
4.3 Experiments with QISKit and LIQU$i|\rangle$ . . . 119
4.3.1 Typical Cases . . . 119
4.3.2 Extended 2-qubit QMUX . . . 121

5 Summary 123
5.1 Summary of Contributions . . . 123
5.2 Future work . . . 124

Appendices **127**

Bibliography **127**

List of Figures

2.1 Framework of QSI . . . 23
2.2 QSI The procedure for simulating in the quantum simulation engine . . . 26
2.3 QSI Quantum types layer . . . 33

3.1 Flowchart of a Qloop . . . 69
3.2 Speed of termination analysis algorithms on two programs of different terminating types. . . . 71
3.3 Decomposition of QMUX . . . 73
3.4 QMUX gate rotation decomposition . . . 73
3.5 Decomposition of arbitrary controlled unitary gate . . . 74

4.1 .NET desktop development . . . 78
4.2 MATLAB support assembly . . . 78
4.3 Examples in Project "UnitTest" . . . 79
4.4 CNOT experiment . . . 79
4.5 Termination . . . 81
4.6 Almost-surely termination . . . 81
4.7 Simple BB84 example . . . 82
4.8 Simple BB84 protocol . . . 83
4.9 Multi-clients BB84 console . . . 86
4.10 BB84 with a quantum channel, success . . . 87
4.11 BB84 with a quantum channel, failure . . . 87
4.12 BB84 with statistics and quantum channel . . . 89
4.13 Statistics of success communication via BB84 with channels . . . 90
4.14 Quantum teleporation with f-QASM . . . 92
4.15 Default network structure . . . 92
4.16 Entry to the user code . . . 97
4.17 Teleportation with loops . . . 103
4.18 Compiled file structure . . . 106
4.19 Parallel and sequential discrimination schemes . . . 109
4.20 The quantum circuit (U_p) generates $|\Psi\rangle$ from $|0\rangle \otimes |0\rangle$. . . . 111
4.21 The quantum circuit (U_m) distinguishes $U^{\otimes 2}|\Psi\rangle$ and $|\Psi\rangle$. . . . 111
4.22 Statistical results in the parallel discrimination experiments. . . . 113
4.23 Statistical results in the sequential discrimination experiments. . . . 113
4.24 The discrimination success probability distributions . . . 114
4.25 Circuits for generating Bell states . . . 117
4.26 Measurement results with two copies of a Bell state . . . 118
4.27 One copy experiment and computational measurement. . . . 118

4.28 Controlled-unitary gate decomposition 120
4.29 Demultiplexing multiplexed rotation gates 121

List of Tables

4.1 Quantum variable types and corresponding initial methods 99
4.2 Quantum variable initialisation matrix form 99
4.3 Ideal measurement results on two copies of a Bell state 117
4.4 Measurement results of controlled-Hadamard gate 120
4.5 Measurement results of controlled-identity gate 120
4.6 Measurement result of controlled-Bit flip gate 121

List of Publications

[1] Shusen Liu; Yinan Li; Runyao Duan. Distinguishing unitary gates on the ibm quantum processor (accepted as regular paper). *SCIENCE CHINA Information Sciences*, 2018.

[43] Shusen Liu, Xin Wang, Li Zhou, Ji Guan, Yinan Li, Yang He, Runyao Duan, and Mingsheng Ying. Qsi: a quantum programming environment. *arXiv preprint arXiv:1710.09500*, 2017.

[3] Shusen Liu, Yang He, and Cai Zhang. Towards automatically construct quantum circuits for quantum programs with quantum control. In *2017 IEEE 85th Vehicular Technology Conference (VTC Spring)*, pages 1–5, June 2017. doi: 10.1109/VTC-Spring.2017.8108699.

Chapter 1

Introduction

1.1 Thesis

While loops play a major role in the kernel of many classical programming languages, their power in the quantum world is still being developed. In 2011, Ying and Feng [1] proposed the quantum while-language. This was the first high-level programming language for quantum computers to include verification properties [2]. Moreover, in the original article, the authors proved its correctness and completeness with respect to quantum Hoare logic [1], making it the only quantum programming language to date with proven completeness.

In this thesis, we present a modern programming environment called QSI [1] to support the quantum while-language. The platform is the culmination of previous theoretical work and includes a suite of modules to assist programmers fulfill several functional requirements, such as termination, running-time estimation, simulation, and compilation. A state-of-the-art software architecture behind the programming environment and, with this platform, users can write quantum algorithms and produce corresponding quantum circuits. Moreover, it can connect to real quantum computers and execute programs in the cloud.

[1] Named after research center `http://www.qsi.uts.edu.au`

This thesis discusses the architecture that connects quantum programming with classical topics. Our final vision for the platform is inspired by the concept of 'quantum supremacy'.

1.2 Background

It is well known that quantum computers can solve certain categories of problems much more efficiently than classical computers, for example, Shor's factoring algorithm [3], Grover's search algorithm [4] and more recently Harrow, Hassidim and Lloyd's algorithm for systems of linear equations [5]. In recent years, governments and industries around the globe have been racing to build quantum computers. As was the case in the history of classical computing, once quantum computers are commercialized, programmers will certainly need a modern platform that can express and implement quantum algorithms without considering the trivialities of their circuits. Such a platform will be even more helpful for quantum programming than in classical computing because physically implementing quantum algorithms in a quantum system is somewhat counterintuitive. Using a programming platform could help programmers understand some of these features, which may help to (partially) avoid some common errors.

However, as the development of quantum hardware advances, a variety of barriers to expanding software and application development still remain. For example, Veldhorst et al. [6, 7] from the University of New South Wales used microwave electron-spin resonance (ESR) performed on a global Hadamard gate to demarcate the transitions between tick and tock intervals. Yet applying the same rules to a local Hadamard gate has proven far more elusive.

In 2016, IBM announced their 5-qubit universal quantum computer [8] (*ibmqx2* and *ibmqx4*) and in 2017 it announced a more advanced 16-qubit universal quantum computer [9] (*ibmqx5*). However, both come with several strict constraints. For instance, the number of gates cannot exceed 40, and only a small set of qubits can operate as controlled qubits. Hence, without a proper quantum programming environment, software coding has become a genuinely difficult task for both researchers and developers.

1.3 Aims, Objectives and Significance

The stakeholders are the programmer in quantum community and the researcher who develops and invents quantum protocols and quantum algorithms. Through the platform, the programmer can write the codes with a high-level programming language without concerning the details of corresponding quantum circuits. Researchers can develop quantum protocols (algorithms) using the compiler and the modules of the platform to improve efficiency. The programming environment is an expendable and convenient way to simulate quantum features on classical computers. Also, the programmer can verify properties of their new algorithms such as termination and average running time which are often very necessary for generating basic quantum circuits. Finally, with the flexible compiler, the programmers could send code segments written in high-level language and be translated to an accepted machine-related language for a real quantum computers.

1.3.1 Aims

The aims of this dissertation are as follows:

1. To explicitly describe a flexible framework for a quantum programming environment that supports quantum **while**-language.

   The QSI quantum programming environment fully explores the structures and components for quantum programming. Further, it and its modules are accessible to the public. The specific modules address termination analysis, average running time, and compilation that, in principle, are effective for quantum programming. These toolkits offer programmers deeper insights into quantum behaviours than the mere statistical results of an algorithm can provide.

2. To prove the criteria for unitary decomposition.

   Current theory holds that spontaneously decomposing nontrivial algorithms into parallel or concurrent algorithms is impossible. However, under an approximate scenario, there is a decomposable unitary arbitrarily closing to the given one. This rule opens the way for a possible approach to fault-tolerant programming with parallel quantum computing.

3. To prove some equivalent conditions for checking termination and average running time.

   Ying et al. [10] illustrate several impressive criteria for checking the termination of quantum programs written in the pure quantum **while**-language. They also provide a criterion for calculating the average running time of quantum programs. However these contributions cannot be used directly in calculations because existing conditions need to be satisfied. And, currently, there is no way for a computer to ensure whether they have these properties. Hence, we have explored equivalent conditions for checking termination and average running time that can be calculated by a computer in finite time.

4. To make a hierarchical structure for driving and connecting quantum computer.

   Svore et al. [11] proposed a theoretical command type for quantum programs called QASM. However, the command type is flat and basic and is only able to denote quantum registers and their corresponding operations. When analysing termination and average running-times, we need a hierarchical structure that includes more information in the quantum **while**-language descriptions to create more well-defined information for physics implementation and relative analysis. Moreover, some calculation tasks, such as recursion with super-operators, are highly computationally-intensive. We present some robust structures with high extensibility that are insensitive to scale and can be implemented in a .NET framework.

1.3.2 Objectives

To achieve these aims, the following objectives must be met.

1. Develop and prove more efficient conditions for termination and average running-time than those raised in the article [12]. Implement these conditions on the QSI platform as modules and ensure those modules can evaluate up-to-date quantum algorithms.

   All the proposed computation steps and rules need to be drafted into protocols or algorithms that can be executed on classical computers in finite time. Also, these

protocols or algorithms must have low cross-coupling so they can be easily extended to both parallel and distributed processing models.

2. Prove and explore unitary decomposition conditions in an approximate scenario.

   Unitary decomposition is an accessible pre-condition for trivial concurrent or parallel quantum computation. The goal is to establish that a set can be decomposed in a simple way and develop an algorithm to search for an approximate unitary matrix.

3. Build a platform embedded in C# language to support quantum **while**-language.

1.3.3 Significance

Throughout history, it generally takes a long time for people to realise high-level solutions in conceptual way rather than in detail. The same is true for quantum programs. To formulate solid quantum algorithms, people need a high-level quantum language and a corresponding platform to verify the properties of their quantum programs.

However, most modern quantum programming tools are too technical for quantum researchers to use. For example, LIQU$i|\rangle$, which stems from the F# language, mixes classical and quantum features. Yet, while F# is a great functional language with object-oriented features, LIQU$i|\rangle$ is difficult to use. Programmers must have a background in quantum computing to develop quantum code, plus knowledge of classical computer knowledge to understand its object-oriented features, and a mathematics background to use the functional language features. This is a daunting challenge for any quantum researcher. When considering a non-flat program structure (i.e., without loops or case statements), the user is likely to find the behaviours in LIQU$i|\rangle$ difficult to understand. The most common loop structure used in F# is recursion, but using recursion raises many unsolved theoretical problems in quantum programming. To avoid these issues, LIQU$i|\rangle$ splits classical and quantum programming into two separate parts. Another analogous example is the software provided with IBMQ. IBMQ offers a research-friendly cloud version with a website as its front-end, and the programming process is intuitive. Users only need to drag pre-defined gates to the stave and hit the “run” button to harvest the experimental results. Hence, the users (researchers) do not need much background or to know the details of quantum programming, but this convenience of operations is the result of a trade-off with flexibility.

Programs are limited to operating on a quantum circuit of no more than 40 gates, and these constraints limit a quantum researcher's creativity. As a result, in last few months, IBM has announced that it plans to release a development environment called QISKit that embeds all IBMQ's online features along with other advanced programming features, such as "IF" and "Swap". Even though quantum researchers will still find the platform complicated to learn, its concision and convenience represent a marked improvement to LIQU$i|\rangle$.

There are some potential requirements for developing a quantum programs using the present environments.

1. Researchers need to have experience in several backgrounds. For most, this creates a barrier to exploring quantum programming.

2. The languages supported in these programming environment require advanced programming skills to isolate classical and quantum structures. Additionally, the loop and case-statements in classical programming have a tendency to create issues in quantum coding.

3. The details of the algorithms create sticking points for the programmer. Algorithms must be decomposed from general gates into the specific gates defined by the platform. For example, LIQU$i|\rangle$ and IBMQ provide different pre-defined gates, and there are few tools to help free them this trap.

4. There are no toolkits to ensure the correctness of the program with respect to Hoare logic, or even the quantum states. Users must assume that the quantum state is always correct as theory dictates. However, basing decisions on theoretical assumptions will not necessarily prove prudent as the complexity of gates and measurements increases.

5. The quantum programming platforms available today do not allow loops, so repeats must be implemented manually. Loops and the number of allowed gates provided are, arguably, the two most pressing bottlenecks for advancing quantum computing. Hence, when using loops, optimisation is a critical issue as they may quickly consume the provided gates as well as introducing extra quantum noise.

1.4 Research Methods

1. To meet the first aim, we built the QSI programming environment from the ground up – from establishing the principles of its design to implementing the code. Prior to construction, we conducted a code review using LIQU$i|\rangle$ and QISKit to confirm the design was suitable for modern user requirements and the specifications met current industry standards. We monitored IBMQ's quantum programming community forum to keep track of industry developments. We earnestly considered the platform's potential users to ensure the platform would be acceptable to most researchers. As a final step, we executed several experiments on the platform, including BB84, quantum teleportation, and Grover's search, to verify the coding and outputs.

2. To meet the second aim, we took explored the criteria for unitary decomposition and its approximate scenarios. We gradually expanded our proofs beginning with the most trivial two unitary decomposition and induction up to multipartite qubits. Through this process, we proved it is possible to find a decomposable unitary that is close to the target unitary and constructed a corresponding algorithm.

3. To meet the third aim, we researched the literature in the classical computation field for strategies to convert theoretical conditions into computation conditions. After fully studying the original proofs and kernel components, we abandoned the Jordan decomposition as an approach and developed some new propositions that significantly accelerate the process of termination analysis.

4. To meet the fourth aim, we examined the most up-to-date physical quantum computers and their related structures. We catalogued and analysed the advantages and disadvantages of the different approaches to manipulating quantum objects in each. Additionally, for compatibility purposes, we developed an expandable low-level language called f-QASM (Quantum Assembly Language with feedback) to adapt to variety of quantum back-ends.

1.5 Literature Review

1.5.1 Programming Languages and Platforms

Several quantum programming platforms have been developed over the last two decades. The first quantum programming language, QCL, was proposed by Ömer in 1998 [13, 14] and implemented in C++. A very similar quantum programming language, Q language, was defined by Bettelli et al. [15] in 2003, which was implemented as a C++ library. In 2000, qGCL was introduced by Sanders and Zuliani [16] as a quantum extension of GCL (Dijkstra's Guarded Command Language) and pGCL (a probabilistic extension of GCL). Over the last few years, some more scalable and robust quantum programming platforms have emerged. A scalable functional programming language Quipper for quantum computing was proposed by Green et al. [17] in 2013. This was implemented using Haskell as its host language. LIQU$i|\rangle$ was developed in 2014 by Wecker and Svore from QuArc (Microsoft Research Quantum Architecture and Computation Team) [18] as a modern toolset and is embedded in another functional programming language F#. In the same year, the quantum programming language Scafford was defined by JavadiAbhari et al. [19]. Its compilation system, ScaffCC, was presented in the article [20]. Smelyanskiy et al. [21] at Intel built a parallel quantum computing simulator qHiPSTER that can simulate up to 40 qubits on a supercomputer with very high performance.

In a recent new milestone, a new programming language and simulator called Q# was announced by QuArc at the end of 2017. Q# is designed specifically for full-stack quantum computing and supports 32 qubits on a personal computer and up to 40 qubits on Azure (Microsoft Cloud). Almost concurrently, the IBMQ team from IBM published their quantum cloud computer and related Python SDK and Python API tools [22]. It enables general researchers to access the most advanced and cloud-based quantum hardware. As the second renaissance in quantum computation, IBMQ has attracted interested both public and private enterprises.

1.5.2 Quantum Circuits Synthesis and Decompostion

The literature on quantum circuit synthesis can be traced back to Barenco's paper [23], which shows that a set of gates that only consists of one-bit quantum gates and two-bit-exclusive or gates is universal. They also investigated the number of above gates required to implement other gates, such as generalised Toffoli gates. As an extension, they show the exact number of elementary gates required to build up a variety of n-bit Toffoli gates. An impressive outcome of this paper is that they combine Boolean algebra with quantum circuits as an efficient way to simulate an n-bit controlled unitary network.

Another very fundamental but significant contribution came from Dawson and Nielsen [24] in 2005. They showed that a dense set can approximately approach an arbitrary qubit gate. More precisely, if $\mathcal{G}$ is a subset of $SU(2)$ and G is dense in $SU(2)$ there is always an element in $\langle\mathcal{G}\rangle(\langle\mathcal{G}\rangle$ indicates a generating set of a group), such that $\mathcal{G}_l$ is an ϵ-net for $SU(2)$, where $l = O(\log^c(1/\epsilon))$. This algorithm opened up the possibility of universal quantum computing. That is, currently, only a limited number of quantum gates can be produced but some are special gates with high fidelity, such as a Hadamard gate and a controlled-Z gate [7]. This raises the question, can all unitary gates be produced approximately? If so, then what is the method to generate them efficiently?

However, beyond the constraints of quantum hardware, there are still several barriers to develop practical applications for quantum computers. One of the most serious issues is the number of physical qubits that physical machines provide. For example, IBMQ makes quantum computing available to programmers through the cloud, but with far fewer qubits than a practical quantum algorithm requires. Today, quantum hardware is in its infancy. But as the number of available qubits gradually increases, many scholars are beginning to wonder whether the various quantum hardware platforms could be united to work as a single entity and, as a result, bring about a bloom of growth in the number of qubits.

In considering this possibility, one approach that has been touted is concurrent quantum programming. Although quantum-specific environments have only focussed on sequential structures to date, a few researchers have exploited the possibility of parallel or concurrent quantum programming on a general programming platform in different respects. Vizzotto and Costa [25] applied mutually exclusive access to global variables in a concurrent Haskell

programming schema to the case of concurrent quantum programming. Yu and Ying [26] carefully studied termination in concurrent programs. And the papers [27–30] provide mathematical tools for process algebra to describe interactions, communication, and synchronisation.

1.5.3 Termination Analysis

Termination analysis is a crucial task in classical programming language and is a necessary part of modern programming environment. As the quality of software surges, more tools are required for debugging programs. The earliest tool, called 'Syntox', was presented by Bourdoncle in the article [31]. Syntox hinges on abstract debugging in Pascal using the scalar variables in programs. In 2006, Cook, Podelski, and Rybalchenko [32], a team of Microsoft, developed an automatic termination tool that has capacity for large program fragments. After a decade, Brockschmidt et al. [33], also from Microsoft, proposed a new open-source tool called T2. T2 supports automatic temporal-logic proving techniques and can handle a general class of user-provided liveness and safety properties.

Termination with a unitary transformation in a quantum programming was first studied by Ying and Feng [10]. These authors used a Jordan decomposition to develop an analysis algorithm. In 2013, they extended the method to termination with superoperator transformations [12]. The termination of non-deterministic and concurrent quantum programs was studied by Li et al. [34] and Yu and Ying [26] as a problem of quantum Markov systems. Recently, Li and Ying [35] examined termination using the realisability and synthesis of linear ranking super-martingales (LRSMs) and reduced it into a semi-definite programming (SDP) problem.

As quantum programming is more counter-intuitive than probabilistic programming because it includes measurement, superposition, and entanglement, termination in quantum programs has received considerable attention. In any practical quantum system with classical loop structure, termination information is always useful in case of the compiler with explanation type works on infinite steps when executing quantum programs. In the book [2], Ying formally introduced a quantum language with a loop structure, i.e., the **while**-language. The operational semantics, denotational semantics, and related characterisations were systematically researched. Thus, the quantum **while**-language has become

a powerful language for describing some complicated quantum algorithms. Additionally, it is also very simple to be used and has made quantum programming easier and more accessible as problems with termination become a reality.

We discuss termination analysis in Section 3.3, beginning with a review of the definitions and theorems of quantum termination. Then we present two algorithms that have been specifically designed and implemented for analysing termination. One is extended directly into computable conditions from the propositions in the article [12]. The other is an improved algorithm that does not include a Jordan decomposition. The latter algorithm produces highly reliable results at significantly faster speeds. The section concludes with a Qloop termination example to compare the efficiency of each algorithm.

1.5.4 Unitary Discrimination

Perfect discrimination of quantum operations is a fundamental task in quantum mechanics. Several theoretical schemes for discrimination with nonorthogonal unitary operations have been developed with the aim reaching more precise results. The original scheme in [36, 37] efficiently solves the problem using entanglement. Moreover, Duan et al. [38] proved that sequential schemes not only require the same number of runs but are also optimal in any scheme that seeks to discriminate between the unitary operators U and V. Given these theoretical exploits, several pioneering experiments were developed to test the schemes. Liu and Hong [39] conducted an experiment using the sequential scheme with a Ti:Sapphire mode-locked laser. They achieved successful probabilities for U (99.5%) and V (99.6%) on two fixed cases. Zhang et al. [40] also used this laser with the sequential protocol returning success probabilities of over 98%. Laing, Rudolph, and O'Brien [41] performed unitary QPD (Quantum process discrimination) with and without entanglement. The results show a certainty of around 99% without entanglement and greater than 97% certainly with entanglement-assisted unitary QPD.

Recently, the IBMQ quantum computer in the cloud, which is built on superconductors, has opened up a new avenue for performing discrimination experiments. The platform comes with a powerful software toolkit embedded in Python for implementing comprehensive experiments. Called QISKit, the software is a full-stack package that supports a universal operation set for different back-end quantum computer topologies. The toolkit

includes flexible mathematic functions and device commands that can execute dedicated quantum experiments on their cloud hardware.

Essentially, QISKit provides a built-in universal gate basis “CNOT+$U(\theta,\phi,\lambda)$” [42] where

$$\text{CNOT} := \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

and

$$U(\theta,\phi,\lambda) := R_z(\phi)R_y(\theta)R_z(\lambda) = \begin{bmatrix} e^{-i(\phi+\lambda)/2}\cos(\theta/2) & -e^{-i(\phi-\lambda)/2}\sin(\theta/2) \\ e^{i(\phi-\lambda)/2}\sin(\theta/2) & e^{i(\phi+\lambda)/2}\cos(\theta/2) \end{bmatrix}.$$

Indeed, $U(\theta,\phi,\lambda)$ can be used to construct all the elements of $SU(2)$. Thus, QISKit offers flexible universal quantum hardware in the sense that local coding (on the client) performs on quantum circuits. With the help of a quantum compiler, the details of the superconductor quantum computer do not need to be understood.

To perform a discrimination experiment, we need the ability to implement arbitrary unitary operations. The calculation process requires unitary matrices other than rotations, which are a some of the fundamental components in QISKit. One module in QSI developed by our team [43], an arbitrary unitary matrix into real parameters using in-fix notation for R_z and R_y. This approach works for implementing $\theta \in [0,4\pi)$, $\phi \in [0,4\pi)$ and $\lambda \in [0,4\pi)$ [42].

In Section 4.2.1, we demonstrate through experiments that two nonorthogonal single-qubit unitary operations can be discriminated using either a parallel scheme or a sequential scheme on the *ibmqx*4 quantum computer. Both schemes return acceptable probabilities in excess of 85% in all rounds. Comparing the results from each scheme shows that, in most cases, the results from the sequential scheme are more accurate due to entanglement under the effect of environmental noise.

1.5.5 Quantum Control

In classical programming languages, Dijkstra [44] proposed a guarded comments GCL (Guarded Command Language) as a building block for alternative and repetitive constructs. Analogously, in quantum programming languages, one stream of research considers the unitary transformation and measurements process, and the other direction considers quantum control flow and superposition. In the first stream, there are two alternatives: Sanders and Zuliani's quantum programming language (qGCL) and Selinger's QPL. The beginnings of the second stream date back to Altenkirch and Grattage's QML [45] followed by Ying, Yu, and Feng [46] who extracted a novel quantum language QuGCL (Quantum Guarded Command Language) to support the superposition paradigm.

Ying et al. [46] derived a family of algebraic laws of QuGCL which can be utilized in program verification, transformations and compilation. Based on their previous work [10, 12] on quantum program with classical control, they extended a quantum statement $[\![\mathbf{QIF}]\!]$ exploiting quantum control. The $[\![\mathbf{QIF}]\!]$ clause became the key idea to use quantum superposition of computing.

However, when attempting to implement Ying et al.'s work [46] in a collection of software modules based on LIQU$i|\rangle$, we found that quantum choice using superposition is an indirect approach. However, in reflecting on the quantum circuit models that are most commonly used to represent a quantum algorithm, we found that Shende [47] showed an analog of Shannon decomposition with Boolean functions and a quantum multiplexor (QMUX) as far back as 2006, and realised this could be used to generalise several known constructions. That is, QMUX can be used to implement the quantum choice $[\![\mathbf{QIF}]\!]$ in QuGCL.

Hence, in this thesis, we propose a construction based on QMUX to perform $[\![\mathbf{QIF}]\!]$ for quantum programs with quantum control. Further, to construct the QMUX, we have focussed on a way to implement an arbitrary unitary transformation using basic CNOT and quantum rotation gates. Some preliminaries concerning QuGCL and QMUX are reviewed followed by the steps required to solve the unitary matrix equations. Some experimental results of controlled unitary using LIQU$i|\rangle$ are also included. Last, we present a short review of the process for constructing a multi-qubit QMUX. To some extent, the

conclusion indicates that the circuits for quantum programs with quantum control can be constructed with the help of QMUX construction.

1.6 Preliminaries

1.6.1 Quantum while-language

For convenience, a brief review of the quantum **while**-language follows. The quantum **while**-language is a pure quantum language without classical variables. It assumes only a set of quantum variables denoted by the symbols $q_0, q_1, q_2, ...$. However, in practice, almost all existing quantum algorithms involve elements of both classical and quantum computation. Therefore, QSI has been designed such that the quantum **while**-language can be embedded into C#, which brings a significant level of convenience to program design. Some explanations of the quantum program constructs follow; For more detailed descriptions and examples, see [1] and Chapter 3 of the book [2]. The quantum **while**-language is generated using the following simple syntax:

$$\begin{aligned} \mathbf{S} ::= \mathbf{skip} \mid q := |0\rangle \mid \bar{q} = U[\bar{q}] \mid S_1; S_2 \mid & \mathbf{if}\ (\square m \cdot M[\bar{q}] = m \rightarrow S_m)\ \mathbf{fi} \\ \mid & \mathbf{while}\ M[\bar{q}] = 1\ \mathbf{do}\, \mathbf{S}\, \mathbf{od}. \end{aligned}$$

Skip As in the classical **while**-language, the statement **skip** does nothing and terminates immediately.

Initialization The initialisation statement "$q := |0\rangle$" sets the quantum variable q to the basis state $|0\rangle$.

Unitary transformation The statement "$\bar{q} := U[\bar{q}]$" means that a unitary transformation (quantum gate) U is performed on quantum register $\bar{q}$ leaving the other variables unchanged.

Sequential composition As in a classical programming language, in the composition $S_1; S_2$, program S_1 is executed first. Once S_1 terminates, S_2 is executed.

Case statement In the case statement **if** $(\square m \cdot M[\bar{q}] = m \rightarrow S_m)$ **fi**, M is a quantum measurement with m representing its possible outcomes. To execute this statement, M is first performed on the quantum register $\bar{q}$ and a measurement outcome m is

obtained with a certain probability. Then, the subprogram S_m is selected according to the outcome m and executed. The difference between a classical case statement and a quantum case statement is that the state of the quantum program variable $\bar{q}$ is changed after performing the measurement.

while-Loop In the loop **while** $M[\bar{q}] = 1$ **do S od**, M is a "yes-no" measurement with only two possible outcomes: 0 and 1. During execution, M is performed on the quantum register $\bar{q}$ to check the loop guard. If the outcome is 0, the program terminates. If the outcome is 1 the program executes the loop body S and continues. Note that here the state of the program variable $\bar{q}$ is also changed after measuring M.

1.6.2 QuGCL

QuGCL was first formally defined in the book [2] and can be thought of as the quantum counterpart of Dijkstra's GCL (Guarded Command Language).

QuGCL is generated by the syntax

$$\begin{aligned}\mathbf{S} ::= \mathbf{skip} | q := |0\rangle\, | \bar{q} = U[\bar{q}] | S_1; S_2 \\ | \mathbf{if} (\Box m \cdot M[\bar{q}] = m \to S_m) \mathbf{fi} \\ | \mathbf{while} M[\bar{q}] = 1 \mathbf{do\, S\, od} \\ | \mathbf{qif} [\bar{q}] (\Box i \cdot |i\rangle \to S_i) \mathbf{fiq}\,.\end{aligned}$$

The execution of a quantum program can be properly described in terms of the transition between configurations.

Definition 1.1. A quantum configuration is a pair $\langle S, \rho \rangle$, where:

- S is a quantum program or the empty program E;
- $\rho \in D(\mathcal{H}_{all})$ is a partial density operator in $\mathcal{H}_{all}$ and it is used to indicate the (global) state of quantum variables.

The operational semantics of QuGCL follow.

SK $\dfrac{}{\langle \mathbf{skip}, \rho \rangle \to \langle \mathbf{E}, \rho \rangle}$.

As with the **skip** clause in the classical while-language, the statement **skip** does nothing and terminates immediately.

IN $\frac{}{\langle q{:=}|0\rangle,\rho\rangle\rightarrow\langle \mathbf{E},\rho_0^q\rangle}$, where

$$\rho_0^q = \begin{cases} |0\rangle_q \langle 0|\,\rho\,|0\rangle_q \langle 0| \quad + \quad |0\rangle_q \langle 1|\,\rho\,|1\rangle_q \langle 0| \\ \qquad\qquad\qquad\qquad \text{if}\, type(q) = Boolean, \\ \\ \sum_{n=-\infty}^{\infty} |0\rangle_q \langle n| \quad \rho \quad |n\rangle_q \langle 0| \\ \qquad\qquad\qquad\qquad \text{if}\, type(q) = Integer. \end{cases}$$

The initialisation statement "$q := |0\rangle$" sets the quantum variable q to the basis state $|0\rangle$. In fact, initialisation projects an arbitrary state to $span\{|0\rangle\}$. The initialization statement "$q := |0\rangle$" sets quantum variable q to the basis state $|0\rangle$. In fact, the initialization is projecting an arbitrary state to $span\{|0\rangle\}$.

UT $\frac{}{\langle \bar{q}{:=}U[\bar{q}],\rho\rangle\rightarrow\langle \mathbf{E},U\rho U^{\dagger}\rangle}$.

The statement "$\bar{q} := U[\bar{q}]$" means that a unitary gate is performed on quantum register $\bar{q}$, leaving other variables unchanged. Because a closed quantum system can be used by a unitary evolution to describe the process, unitary evolution is designed as a fundamental component of the platform.

SC $\frac{\langle S_1,\rho\rangle\rightarrow\langle S_1',\rho\rangle}{\langle S_1;S_2,\rho\rangle\rightarrow\langle S_1';S_2,\rho\rangle}$.

The S_1 program runs first. After S_1 finishes, S_2 runs. The quantum-**while** language is not designed as a concurrent language so sequential composition is spontaneous.

IF $\frac{}{\langle \mathbf{if}\,(\square m\cdot M[\bar{q}]{=}m\rightarrow S_m)\,\mathbf{fi},\rho\rangle\rightarrow\langle S_m,M_m\rho M_m^{\dagger}\rangle}$, for each possible outcome m of meaurement $M = \{M_m\}$.

The case statement can be rewritten as

$$
\begin{aligned}
&\textbf{If}(\Box m \cdot M[\bar{q}] = m \to S_m)\textbf{fi})\\
\equiv &\textbf{if}\ M[\bar{q}] = m_1 \to S_{m_1}\\
&\Box \qquad m_2 \to S_{m_2} \qquad .\\
&\ldots\\
&\Box \qquad m_n \to S_{m_n}
\end{aligned}
$$

The first step in executing of this case statement is to perform a measurement M on the quantum variable $\bar{q}$ and check the output result index. A corresponding subprogram S_m is then chosen based on the output result index. The difference between a classical case statement and a quantum case statement is that the variable in a quantum case statement must be changed to the state in measurement output index once a measurement is performed.

Loop (L0) $\overline{\langle \textbf{while}(M[\bar{q}]=1)\textbf{do}\, S\, \textbf{od}, \rho\rangle \to \langle E, M_0 \rho M_0^{\dagger}\rangle}$,

(L1) $\overline{\langle \textbf{while}(M[\bar{q}]=1)\textbf{do}\, S\, \textbf{od}, \rho\rangle \to \langle S_m, M_m \rho M_m^{\dagger}\rangle}$.

Quantum loops are a generalisation of classical loops. The guard is a Boolean equality, and M is a set of measurements. If the measurement set has two members, it either degenerates into a "yes-no" measurement or the user can define the guard behaviours by denoting the compared number. At the very beginning of the program, users can redefine the general measurement set as "yes-no" measurement. If the guard is true, the program will enter into the loop body. Otherwise, the program terminates immediately.

QIF $[\![S]\!](|i\rangle\,|\varphi\rangle) = |i\rangle\,([\![S_i]\!]\,|\varphi\rangle)$

Let $S_1, S_2, \ldots, S_n$ be a collection of general quantum programs whose state spaces are the same Hilbert space $\mathcal{H}$ and the external quantum system called the "coin" system which can either be a single system or a composite system. Assume that the state space of system $\bar{q}$ is an n-dimensional Hilbert space $\mathcal{H}_q$ and $\{|i\rangle\}_{i=1}^{n}$ with an orthonormal basis. As such, a quantum case statement S can be defined by

combining programs $S_1, S_2, \dots, S_n$ along the basis $\{|i\rangle\}$:

$$\begin{array}{rl} S \equiv \textbf{qif}[\bar{q}]: & |1\rangle \rightarrow S_1 \\ \square & |2\rangle \rightarrow S_2 \\ & \dots \\ \square & |n\rangle \rightarrow S_n \\ \textbf{fiq} & \end{array}.$$

When considering the program body as collection of a unitary operators, the definition can be converted to $U(|i\rangle |\varphi\rangle) = |i\rangle U_i |\varphi\rangle$ for any $\varphi \in \mathcal{H}$ and for any $1 \leq i \leq n$. The operator U is called the guarded composition of $U_i(1 \leq i \leq n)$ along the basis $\{|i\rangle\}$ and written as $U \equiv \bigoplus_{i=1}^{n} U_i$. Actually, this can be explained as a QMUX which was introduced by Shende [47].

1.7 Overview

The main purpose of this research is to study the architecture of a quantum programming environment and its related modules. Moreover, the related algorithms for implementing quantum programming environment need to be carefully studied.

In Chapter 2, we describe the QSI's architecture and construct the connections between the semantics and the matrix computation. In addition, we introduce the compiler as the key component of programming environment and explain how it has been designed to achieve the primary goal of generating code that can operate on quantum hardware. The quantum simulator, designed to verify quantum code on a classical computer, is also presented.

In Chapter 3, each of the three modules are discussed in detail. First, we explore the separation of multipartite quantum gates. We find that most unitary operators cannot simply be decomposed into tensor product formals, which means that most algorithms cannot be paralleled naturally. Here, we present an approach to arbitrary unitary using the tensor products of the unitary evolutions in approximate conditions. Second, we propose a fast permutation method to adjust a given unitary in a target system that can

adapt to the memory limitations of a computer. Unlike the permutation matrix method, our experiments show this method is able to support high-dimensional matrix adjustments. Third, we discuss termination analysis and the two algorithms designed to check for correct termination in quantum programs. One algorithm is a direct extension of the propositions in [12] into computable conditions. The other is an improved algorithm that relies on the Jordan decomposition to achieve highly reliable results and significantly improved speed. Lastly, we introduce the quantum control module as a part of the QuGCL language. We explicitly describe the quantum control statement $[\![\mathbf{QIF}]\!]$ can be achieved by QMUX and further show how to construct the syntax of a given clause to implement QuGCL on a general quantum computer's circuits.

Chapter 4 focuses how QSI can work with other platforms. Many features of QSI are demonstrated through a series of experiments. These include: the most fundamental gate – the CNOT gate; the most famous communication protocol, BB84 – and its channel cases; quantum Google PageRank; and even Grover's search. We also illustrate self-coding for programmers and its related APIs (Application Program Interfaces). The power of QSI is illustrated in the way it connects to real quantum computers using the QISKit. Two interesting and fundamental experiments with the IBMQ quantum computer in the cloud are provided, namely unitary discrimination and distinguishing Bell states with local measurements. The results showcase IBMQ as a new medium for performing quantum algorithms and the useful programming environment QSI offers as a supplement. Methods of connecting with LIQU$i|\rangle$, an older quantum platform, are also explained. These methods allow users to access to quantum simulations with the functional language F#. The method to construct QMUX are conducted through the links between these two platforms.

In Chapter 5, we summarise the contributions of this and discuss future work.

These results are based on the following papers:

- Shusen Liu; Yinan Li; Runyao Duan, "Distinguishing Unitary Gates on the IBM Quantum Processor". in SCIENCE CHINA Information Sciences (accepted as regular paper).

- Shusen Liu; Xin Wang; Li Zhou; Ji Guan; Yinan Li; Yang He; Ruanyao Duan and Mingsheng Ying, 2017. "$Q|SI\rangle$: a quantum programming environment". arXiv preprint arXiv:1710.09500.
- Shusen Liu; Li Zhou; Ji Guan; Yang He; Runyao Duan; Mingsheng Ying, (2017-05-09). "Q$|SI\rangle$: A Quantum Programming Language". SCIENTIA Sinica Information. 47 (10): 1300.
- Shusen Liu; Yang He and Cai Zhang. "Towards Automatically Construct Quantum Circuits for Quantum Programs with Quantum Control", IEEE 85th Vehicular Technology Conference (VTC Spring), 2017, 1-5.

Chapter 2

Quantum Programming Environment

2.1 Introduction

This research presents a powerful and flexible new quantum programming environment called QSI [1], named after my research center – the Centre for Quantum Software and Information. The *core* of QSI is *a quantum programming language* and *its compiler*. This language is a quantum extension of the **while**-language. It was first defined in [1] along with a careful study of its operational and denotational semantics (see also [2], Chapter 3). The language includes a measurement-based case statement and a measurement-based **while**-loop. These two program constructs are extremely convenient for describing large-scale quantum algorithms, e.g., quantum random walk-based algorithms.

For operations with quantum hardware, we have defined a new assembly language called f-QASM (Quantum Assembly Language with feedback) as an interactive command set. f-QASM is an extension of the instruction set QASM (Quantum Assembly Language) introduced in [48]. A feedback instruction has been added that allows measurement-based case and loop statements to be implemented efficiently. A compiler then transforms the quantum **while**-program into a sequence of f-QASM instructions and further generates a

[1] `http://www.qcompiler.com`

corresponding quantum circuit equivalent to the program (i.e., a sequence of executable quantum gates). QSI also has a module for optimising quantum circuits as well as a module to simulate its quantum programs developed in the environment on a classical computer.

2.2 The Structure of QSI

This section provides an introduction to the basic structure of QSI , leaving the details to be described in subsequent sections. QSI is designed to offer a unified general-purpose programming environment to support the quantum **while**-language. It includes a compiler for quantum **while**-programs, a quantum computation simulator, and a module for the analysis and verification of quantum programs. We have implemented QSI as a deeply embedded domain-specific platform for quantum programming using the host language C#.

QSI 's framework is shown in Figure 2.1.

2.2.1 Basic Features of QSI

A description of the main features of QSI follows:

Language support QSI is the first platform to support the quantum **while**-language. Specifically, it allows users to program with measurement-based case statements and **while**-loops. The two program constructs provide more efficient and clearer descriptions of some quantum algorithms, such as quantum walks and Grover's search algorithm.

Quantum type enriched Compared to other simulators and analysis tools, QSI supports quantum types beyond pure qubit states, such as density operators, mixed states, and so on. These types have unified operations and can be used in different scenarios. This feature provides highly flexible usability and facilitates the programming process.

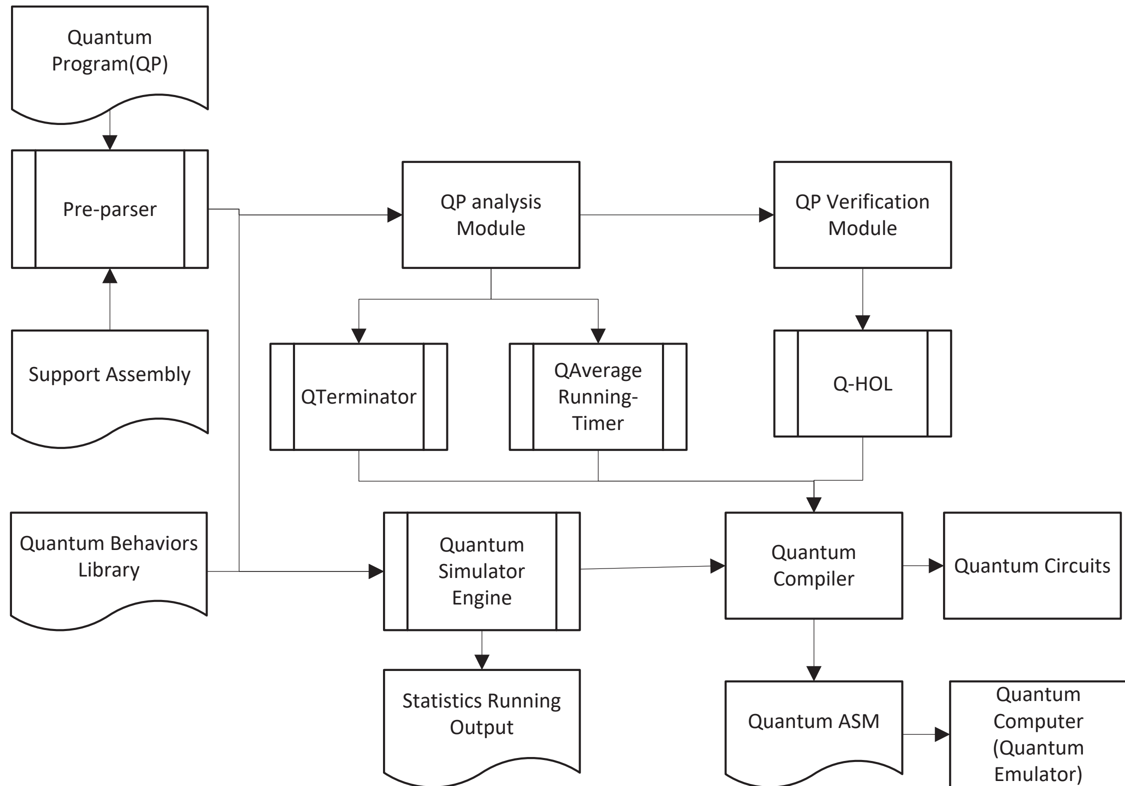


FIGURE 2.1: Framework of QSI . Rectangle modules indicate the main process stream in QSI with respect to the execution components. The double-edged rectangles indicate auxiliary modules. The ripple rectangles are the input and output data structures or files. The QP Verification Module with Q-HOL will link to the theorem prover proposed by Liu et al. [49] but is currently still in development.

Dual mode QSI has two executable modes. "Running-time execution" mode simulates quantum behaviors in one-shot experiments. "Static execution" mode is mainly designed for quantum compilation, analysis, and verification.

f-QASM instruction set Defined as an extension of Quantum Assembly Language (QASM) [48], f-QASM is essentially a quantum circuit description language that can be adapted for a variety of purposes. In this language, each line has only one command. f-QASM's 'goto' structure contains more information than the original QASM [48] or space efficient QASM-HL [20]. f-QASM can also be used for further optimization and analysis.

Quantum circuits generation Similar to modern digital circuits in classical computing, quantum circuits provide a low-level representation of quantum algorithms [48]. Our

compiler can produce a quantum circuit from a program written in the high-level quantum **while**-language.

Arbitrary unitary operator implementation The QSI platform includes Solovay-Kitaev algorithm [24] together with two-level matrix decomposition algorithm [50] and a quantum multiplexor (QMUX) algorithm [47]. Therefore, an arbitrary unitary operator could be transferred into a quantum circuit consisting of quantum gates from a small pre-defined set of basic gates once these are available from quantum chip manufactures.

Gate-by-gate description Similar to other quantum simulators, QSI has a gate-by-gate description feature. Some basic quantum gates are inherently provided in our platform. Users can use them to build their desired quantum circuits gate-by-gate. We have also provided a decomposition function to generate arbitrary 2-dimensional controlled-unitary gates for emulation feasibility.

2.2.2 Main Components of QSI

The QSI platform consists of four main parts.

Quantum Simulation Engine This component includes some support assemblies, a quantum mechanics library and a quantum simulator engine. The support assemblies provide support for the quantum types and quantum language. More specifically, they provide a series of quantum objects, and reentrant encapsulated functions to play the role of the quantum program constructs **if** and **while**. The quantum mechanics library provides the behaviors for quantum objects such as unitary transformation and measurement including the result and post-state. The quantum simulator engine is designed as an execution engine. It accepts quantum objects and their rules from the quantum mechanics library and converts them into probability programming that can be executed on a classical computer.

Quantum Program (QP) Analysis Module This module currently comprises two submodules for the static analysis mode: the "QTerminator" and the "QAverage Running-Timer". The former provides the terminating information, and the latter evaluates

the running time of the given program. Their output is sent to the quantum compiler at the next stage for further use.

QP Verification Module This module is a tool for verifying the correctness of quantum programs and is still under development. It is based on quantum Hoare logic, which was introduced by Ying in [1]. One possibility for its future advancement is to link QSI to the quantum theorem prover developed by Liu et al [49].

Quantum Compiler The compiler consists of a series of tools to map a high-level source program that represents a quantum algorithm into the relevant language of a quantum device [48], e.g., f-QASM and further into a quantum circuit. Our goal is to be able to implement any source code without considering the details of the devices, it will ultimately run on, i.e., to automatically construct a quantum circuit based on the source code. In future, a tool to optimise the quantum circuit will be added.

2.2.3 Implementation of QSI

One of the basic problems during implementation is how to use probabilistic and classical algorithms to simulate quantum behaviors. To support quantum operations, QSI has been enriched with data structures from a quantum simulation engine. Figure 2.2 shows the procedure for simulating a quantum engine. Three types of languages are supported: the pure quantum **while**-language, the classical **while**-language and a mixed language. The engine starts a support flow path when it detects the existence of a quantum section of the program. Then, the engine checks the quantum type for each variable and operator and executes the corresponding support assembly, which is explained as a constrained object on a classical computer. As mentioned, one of the main features of QSI are the quantum while-language support assemblies, with aid programming in the quantum while-language. All of the behaviours that consider semantics are explained by probabilistic algorithms. The outputs are extended into C# languages and can be executed directly on a .NET Framework or can be explained in f-QASM and OpenQASM by the compiler.

The quantum simulation engine involves numerous matrix computations and operations. In QSI , Math.net is used for matrix computation. Math.NET is an open-source initiative to build and maintain toolkits that cover fundamental mathematics. It targets both the

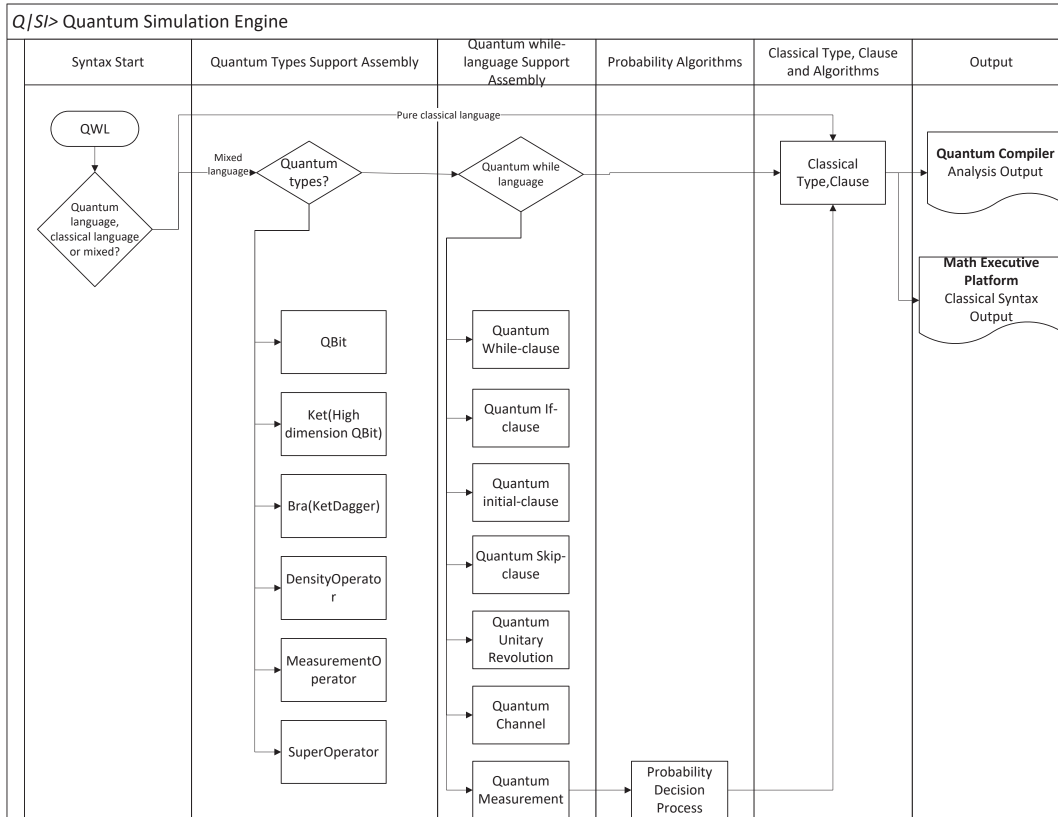


FIGURE 2.2: QSI The procedure for simulating in the quantum simulation engine. Three types of languages are supported: the pure quantum while-language, the classical while-language, and a mixed language. The engine starts a support flow path when it detects the existence of a quantum section of the program. Then, the engine checks the quantum type for each variable and operator and executes the corresponding support assembly, which is explained as a constrained object on a classical computer. As mentioned, one of the main features of QSI are the quantum while-language support assemblies, with aid programming in the quantum while-language. All of the behaviours that consider semantics are explained by probabilistic algorithms. The outputs are extended into C# languages and can be executed directly on a .NET Framework or can be explained in f-QASM and OpenQASM by the compiler.

everyday and the advanced needs of .NET developers[2]. It includes numerical computing, computer algebra, signal processing and geometry. Math.net is also able to accelerate matrix calculations when the simulation includes a MIC (Many Integrated Core Architecture) device.

In static analysis mode, Roslyn is our chosen tool for auxiliary code analysis. Roslyn is a set of open-source compilers and code analysis APIs for C# and Basic languages. Since

[2] `https://www.mathdotnet.com`

our platform is embedded in the .NET Framework for the C# language, Roslyn is used as a parser to produce an abstract syntax tree (AST) for further analysis.

2.3 The Quantum Compiler

The compiler works as a connection between different devices and data structures and serves several different functions. It produces f-QASM code, which can be used to emulate a real or virtual quantum processor. It provides quantum circuits for quantum chip design. It also optimizes quantum circuits.

The QSI compiler is heavily dependent on other modules. It collects data structures from the quantum simulation engine and splits the program into several parts, e.g. the variables, the quantum gates, the measurements, the entry and exit points for each clause along with their positions. It constructs an AST (Abstract Syntax Tree) from the program, then reconstructs the program as a sequence of f-QASM instructions for further use. Based on f-QASM, the compiler provides a method for decomposing the unitary operators. It can decompose an arbitrary unitary operator $U(n)$ into a sequence of basic quantum gates from a pre-defined set $\{U_1, U_2, \ldots, U_m\}$ where $U_1, U_2, \ldots, U_m \in U(2)$ (qubit gate). This corresponds to a scenario in quantum device development: people need universal computation in spite of only the few of gates, that manufacturers can currently produce. Further, the quantum **while**-language delivers the power of loops, but it also increases the complexity of compilation. A quantum program with a loop structure is much harder to trace than the one without loops. The QP Analysis module provides static analysis tools including a “QTerminator” for termination checking and a “QAverage Running-Timer” for calculating the expected running time. In addition, the QP Verification module, still in development, is being designed to verify quantum programs. Once complete, programmers will be able to insert to debug program behaviors.

2.3.1 f-QASM

QASM (Quantum Assembly Language) is widely used in modern quantum simulators. It was first introduced in [48] and is defined as a technology-independent reduced-instruction-set computing assembly language extended by a set of quantum instructions based on the

quantum circuit model. The article [51] carefully characterizes its theoretical properties. In 2014, A.JavadiAbhari et al. [20] defined a space-consuming flat description and denser hierarchical description of QASM, called QASM-HL. Recently, Smith et al. [52] proposed a hybrid QASM for classical-quantum algorithms and applied it in Quil. Quil is the front-end of Forest which is a tool for quantum programming and simulation that works in the cloud.

We propose a specific QASM format, called f-QASM (Quantum Assembly Language with feedback). The most significant motivation behind our variation is to translate the inherent logic of quantum program written in a high-level programming language into a simple command set, so there is only one command in every line or period. However, a further motivation is to solve an issue raised on the IBM QASM 2.0 list and provide conditional operators for feedback on measurement outcomes.

2.3.2 Basic Definitions in f-QASM

Let us first define the registers:

- Define $\{r_1, r_2, \ldots\}$ as a finite set of classical registers.
- Define $\{q_1, q_2, \ldots\}$ as a finite set of quantum registers.
- Define $\{fr_1, fr_2, \ldots\}$ as a finite set of flag registers. These are a special kind of classical register often used to illustrate partial results for a code segment. In most cases, the flag registers can not be operated directly by any users code.

There are also two kinds of basic operations:

- Define the command "$op(q)$" as $q \coloneqq op(q)$, where op is a unitary operator and q is a quantum register.
- Define the command "$\{op\}(q)$" as $r \coloneqq \{op\}(q)$, where $\{op\}$ is a set of measurement operators, q is a quantum register, and r is a classical register.

After defining registers and operations, we can define some assembly functions:

- Define "$INIT(q)$" as $q := |0\rangle\langle 0|$, where q is a quantum register. The value of q is assigned into $\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$.

- Define "$OP\{q, num\}$", where q is a quantum register, $num \in \mathbb{N}$ and OP is an operator, in another functional form of $q := op(q)$. num can be 0 or other value, i.e., 0 means the unitary operator belongs to a pre-defined set of basic quantum gates which can be prepared by either the manufacturer or the user. Otherwise, num can only be ignored or used after being decomposed into basic gates, or be ignored.

- Define "$MOV(r_1, r_2)$", r_1 and r_2 as the classical registers. This function assigns the value of the register r_2 to the register r_1 and empties r_2.

- Define "$CMP(r_1, r_2)$" as $fr_1 = \delta(r_1, r_2)$ or as $fr_1 = (r_1 == r_2)$, where r_1, r_2 are two classical registers, δ is the function comparing whether r_1 is equal to r_2: if r_1 is equal to r_2 then $fr_1 = 1$; otherwise $fr_1 = 0$.

- Define "$JMP\ l_0$" as the current command which goes to the line indexed by l_0.

- Define "$JE\ l_0$" as indexing the value of fr_1 and jumping. If fr_1 is equal to 1 then the compiler executes $JMP\ l_0$, otherwise it does nothing.

2.3.3 f-QASM Examples

Some simple examples to help readers understand f-QASM follow,

2.3.3.1 Initialisation

$\mathbf{q} := |\mathbf{0}\rangle$ means the program initialises the quantum register q in the state $|0\rangle$. In f-QASM, initializing two quantum registers $Q1$ and $Q2$ in the state $|0\rangle$ would be written as

```
INIT(Q1);
INIT(Q2);
```

2.3.3.2 Unitary transformation

$\bar{\mathbf{q}} = \mathbf{U}[\bar{\mathbf{q}}]$ means the program performs a unitary transformation on the register q. The compiler checks whether the unitary matrix is a basic gate or not. An example program segment of unitary transformation follows:

```
hGate(q1);
```

Here we support *hGate* is a Hadamard gate performed on single qubit, i.e., $hGate = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$. To transform this into an f-QASM instruction, it is written as

```
hGate(q1, 0);
```

2.3.3.3 Case statement

The following program segment is written as a case statement in quantum **while**-language:

```
QIf(m(q1)
() =>
{
xGate(q1);
},
() =>
{
hGate(q1);
}
);
zGate(q1);
```

where *hGate* is a Hadamard gate performed on single qubit, *xGate* is a bit-flip gate performing on single qubit $xGate = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, and *zGate* is a phase-flip gate $zGate = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$. Here we assume that all the gates can be provided. M is a user-defined measurement. The compiler interprets this segment as the following f-QASM instructions:

```
MOV(r,{M}(q1));
CMP(r,0);
```

```
JE L1;
CMP(r,1);
JE L2;
L1:
xGate(q1,0);
JMP L3;
L2:
hGate(q1,0)
JMP L3;
L3:
zGate(q1,0);
```

2.3.3.4 Loop

A loop construct is provided using $QWhile(M(q))$, where $QWhile$ is a keyword, M is a measurement and q is a quantum register. An example program segment with quantum **while**-loop follows:

```
QWhile(m(q1),
() =>
{
xGate(q1);
}
);
hGate(q1);
```

Both *hGate* and *xGate* are basic gates as above. These can be transformed into f-QASM as follows:

```
L1:
MOV(r,{M}(q1));
CMP(r,0);
JE L2;
XGate(q1,0);
```

```
JMP L1;
L2:
hGate(q1,0);
```

2.3.4 Decomposition of a General Unitary Transformation

Given a set of basic gates $\{U_1, U_2, \ldots, U_n\}$. If any unitary operator can be approximated to an arbitrary accuracy with a sequence of gates from this set, then the set is said to be universal [50].

In the compiler, there are two kinds of built-in decomposition algorithms. One is the QR method given in [23, 50], which consists of the following steps:

1. An arbitrary unitary operator is decomposed exactly into (the composition of) a sequence of unitary operators that act non-trivially only on a subspace spanned by two computational basis states;
2. Each unitary operator, which only acts non-trivially on a subspace spanned by two computational basis states are further expressed using single qubit gates ($U(2)$) and the CNOT gate;
3. Each single qubit gate can be decomposed into a sequence of gates from a given small set of basic (single qubit) gates using the Solovay-Kitaev theorem [24].

The other is the QSD method presented in [47], consisting of the following steps:

1. An arbitrary operator is decomposed into three multiplexed rotations and four generic $U(2^{d-1})$ operators, where d is the number of qubits;
2. Repeatedly execute step 1 until $U(4)$ is generated;
3. The $U(4)$ operator is decomposed into $U(2)$ operators with two extra CNOT gates;
4. Each single qubit gate in $U(2)$ is decomposed into gates from a given small set of basic (single qubit) gates using the Solovay-Kitaev theorem [24].

2.4 The Quantum Simulator

2.4.1 Quantum Types

Data types can be extended from classical computing to quantum computing. For example, quantum generalisations of Boolean and integer variables were introduced in [1]. The state space of a quantum Boolean variable is a 2-dimensional Hilbert space **Boolean** = $\mathcal{H}_2$, and the state space of a quantum integer variable is an infinite-dimensional Hilbert space **integer** = $\mathcal{H}_\infty$. In QSI , every kind of quantum variable has its own initialization method and operation. Currently, QSI contains only finite-dimensional quantum variables, but infinite-dimensional variables will be added in the future. The quantum types used in QSI are presented in Figure 2.3.

The entire set of quantum types are defined as subclasses of one virtual base class called $QuantumTypes\langle T\rangle$. This virtual base class has only been introduced to indicate that all of the derived subclasses are quantum objects. From the virtual base class $QuantumTypes\langle T\rangle$, two extended virtual base classes inherit: $Vector\langle T\rangle$, which represents a class of quantum variables which share some vector rules, and $Matrix\langle T\rangle$, which represents a class of quantum operators that share some operator rules.

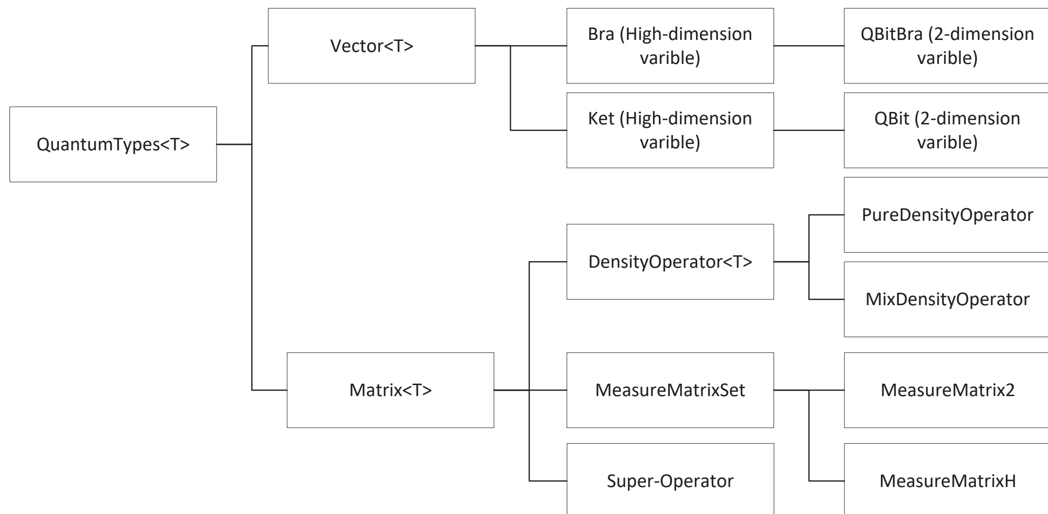


FIGURE 2.3: QSI Quantum types layer

Quantum variables come in two basic types: *Ket* is used to denote a quantum variable of arbitrary dimension, and type *Bra* is the conjugate transpose of *Ket*. Two specialised (sub)types *QBit* and *QBitBra* are provided for 2-dimensional quantum variables. Note that both are compatible when we consider the Boolean type as a subtype of an integer. Also, these types must follow a few rules, as explained in the next sections.

Normalised states If a qubit is written as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the result when measured on a computational basis is either 0 with a probability of $|\alpha|^2$ or 1 with a probability of $|\beta|^2$. Since these probabilities must sum to 1, it obeys $|\alpha|^2 + |\beta|^2 = 1$. Thus, the length of a vector should be normalised to 1 during initialization and computation. For convenience, QSI provides a function *QBit.NormlizeSelf*() to keep the norm of the variable types *QBit* and *Ket*.

Hidden states It is well-known that information about a *QBit* or a *Ket* cannot be extracted unless the state is measured. However, as indicated by Nielsen and Chuang in [50] , "Nature evolves a closed quantum system of qubits, not performing any 'measurements', she apparently does keep track of all the continuous variables describing the state, like α and β ". In QSI , we use the following trick to simulate quantum computing: a quantum state is a black box- each part in the box cooperates with others, but an external viewer knows nothing. Functions and other object methods including unitary transformation or a quantum channel know the quantum state exactly, but a viewer knows nothing about this hidden information unless it is measured. Thus, we classify the state of information storage as a "*Protect*" class, which means that information about a quantum state cannot be accessed easily.

The matrix form is widely used in (the semantics of) the quantum **while**-language. There are three categorised of matrix: *DensityOperator*$\langle T\rangle$, *MeasureMatrixSet* and *SuperOperator*. *DensityOperator*$\langle T\rangle$ is a virtual basic class with two sub-classes: *PureDensityOperator* and *MixDensityOperator*. In fact, the difference between *PureDensityOperator* and *MixDensityOperator* is that only *MixDensityOperator* accepts an ensemble, namely a set of probabilities and their corresponding states, which can be expressed by a *PureDensityOperator*$\langle T\rangle$ or a *Vector*$\langle T\rangle$. The object quantum variable ρ of *DensityOperator*$\langle T\rangle$ satisfies the following two conditions: (1) ρ has trace 1; and (2) ρ is a positive operator. Every

operation on an object triggers a verification of these conditions to ensure the object is a real density operator. *MeasureMatrixSet* is a measurement containing an array of matrix $M = \{M_0, M_1, \ldots, M_n\}$ satisfying a completeness condition $\sum_i M_i^\dagger M_i = I$. Defining a quantum measurement in this way is very flexible. There are two built-in measurements – a plus-minus basis $\{|+\rangle, |-\rangle\}$ and a computation basis $\{|0\rangle, |1\rangle\}$, so that users can easily use their desired measurement. A *SuperOperator* can be used to simulate an open quantum system. It uses an array of Kraus operators $\mathcal{E} = \{E_0, E_1, \ldots, E_n\}$ satisfying $\sum_i E_i^\dagger E_i \leq I$ as a representation.

2.4.2 Simulating of Quantum Behaviors

The basis of simulating quantum computation is to simulate the quantum behaviors defined by the four basic postulates of quantum mechanics [50]:

- **Postulate 1**: Associated with any isolated physical system is a complex vector space with an inner product (Hilbert space) known as the state space of the system. The system is completely described by its state vector, which is a unit vector in the system's state space.

  In QSI , a function in Math.net called

  double ConjugateDotProduct(Vector⟨T⟩ other)

  is used to support the inner product.

- **Postulate 2**: The evolution of a closed quantum system is described by a unitary transformation. That is, the state $|\psi\rangle$ of the system at time t_1 is related to the state $|\psi'\rangle$ of the system at time t_2 by a unitary operator U which depends only on the time t_1 and t_2. $|\psi'\rangle = U|\psi\rangle$.

  To simulate this feature in QSI , we have added the function $UnitaryTrans$ to some of our quantum types in a closed quantum system, such as *QBit*, *Ket* and *DensityOperator⟨T⟩*. In addition, the static global function *SuperMatrixTrans* is provided to describe the dynamics of an open quantum system as a super-operator $\mathcal{E}$.

- **Postulate 3**: Quantum measurements are described by a collection $\{M_m\}$ of measurement operators. These are operators that act on the state space of the system being measured. The index m refers to the measurement outcomes that may occur in the experiment. If the state of the quantum system is $|\psi\rangle$ before the measurement, then the probability that the result m occurs is given by $p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle$ and the state of the system after the measurement is $\frac{M_m|\psi\rangle}{\langle\psi|M_m^\dagger M_m|\psi\rangle}$.

  In QSI , quantum measurements are simulated with a modified Monte Carlo method, i.e., simulating measurement with a pseudo-random number sampling.

- **Postulate 4**: The state space of a composite physical system is the tensor product of the state spaces of the component physical systems. Moreover, if we have systems numbered 1 through n, and system number i is prepared in the state $|\psi_i\rangle$, then the joint state of the total system is $|\psi_1\rangle \otimes |\psi_2\rangle \otimes \ldots \otimes |\psi_n\rangle$.

  The function *void KroneckerProduct (Matrix⟨T⟩ other, Matrix⟨T⟩ result)* is used in the tensor product method, which is embedded in Math.net.

2.4.3 Simulating Operational Semantics in the Quantum while-language

Simulating the computation of a program written in the quantum **while**-language is based on simulating the operational semantics of the language. To clearly delineate coding in mixed classic-quantum programs in the quantum simualation engine, a quantum **if**-clause is denoted as **cif** and a quantum **while**-clause is denoted as **cwhile**. To simulate these two functions, the related function methods are encapsulated in the Quantum Mechanics Library.

The execution of a quantum program can be conveniently described in terms of transitions between configurations.

Definition 2.1. A quantum configuration is a pair $\langle S, \rho\rangle$, where:

- S is a quantum program or the empty program E (termination);
- ρ is a partial density operator used to indicate the (global) state of quantum variables.

With the preparations in the previous subsections, we are able to simulate the transition rules that define the operational semantics of the quantum while-language:

Skip

$$\frac{}{\langle \mathbf{skip}, \rho\rangle \to \langle \mathbf{E}, \rho\rangle}.$$

The statement **skip** does nothing and terminates immediately. Both I-identity operation and the null clause satisfy this procedure requirement for simulation in QSI .

Initialisation

$$\frac{}{\langle q := |0\rangle, \rho\rangle \to \langle \mathbf{E}, \rho_0^q\rangle},$$

where

$$\rho_0^q = \begin{cases} |0\rangle_q \langle 0| \rho |0\rangle_q \langle 0| \quad + \quad |0\rangle_q \langle 1| \rho |1\rangle_q \langle 0| \text{ if } type(q) = Boolean, \\ \\ \sum_{n=-\infty}^{\infty} |0\rangle_q \langle n| \quad \rho \quad |n\rangle_q \langle 0| \text{ if } type(q) = Integer. \end{cases}$$

The initialization statement "$q := |0\rangle$" sets the quantum variable q to the basis state $|0\rangle$.

In QSI , initialization has two forms. When the variable q is a *QBit*, it is explained as $[\![q := |0\rangle]\!](\rho) = |0\rangle\langle 0|\rho|0\rangle\langle 0| + |0\rangle\langle 1|\rho|1\rangle\langle 0|$; otherwise, it is explained as $[\![q := |0\rangle]\!](\rho) = \sum_{n=0}^{d} |0\rangle\langle n|\rho|n\rangle\langle 0|$, where d is the dimension of the quantum variable q. Moreover, a more flexible initialization method is provided with the help of unitary transformation.

Unitary evolution

$$\frac{}{\langle \bar{q} := U[\bar{q}], \rho\rangle \to \langle \mathbf{E}, U\rho U^\dagger\rangle}.$$

The statement "$\bar{q} := U[\bar{q}]$" means that the unitary gate U is performed on the quantum register $\bar{q}$ leaving other variables unchanged.

A corresponding method called *QuantumTypes⟨T⟩ .UnitaryTrans(Matrix⟨T⟩ other)* has been designed for *QBit*, *Ket*, *DensityOperator⟨T⟩* objects to perform this function. This function accepts a unitary operator and performs the operator on the variable with null returns. We have also provided a global function called

UnitaryGlobalTrans(QuantumType⟨T⟩, Matrix⟨T⟩)

that perform an arbitrary unitary matrix on quantum variables.

The quantum **while**-languages do not include any assignment claim for a pure state because a unitary operator U exists for any pure state $|\psi\rangle$ satisfies $|\psi\rangle = U|0\rangle$. Therefore, any pure state can be produced from a combination of an initialisation clause and a unitary transformation clause. However, for convenience, QSI provides a flexible state claim to initialise a *QBit*, or a *Ket* using a vector, and to initialise a *DensityOperator⟨T⟩* using a positive matrix.

Sequential composition

$$\frac{\langle S_1, \rho\rangle \to \langle S_1', \rho\rangle}{\langle S_1; S_2, \rho\rangle \to \langle S_1'; S_2, \rho'\rangle}.$$

The current version of the quantum **while**-language is not designed for concurrent programming. Thus sequential composition is spontaneous.

Case statement

$$\frac{}{\langle \mathbf{if}(\square m \cdot M[\bar{q}] = m \to S_m)\mathbf{fi}, \rho\rangle \to \langle S_m, M_m \rho M_m^\dagger\rangle},$$

for each possible outcome m of measurement $M = \{M_m\}$.

The first step in executing of the case statement is performing a measurement M on the quantum variable $\bar{q}$ and observing the output result index. The corresponding subprogram S_m is then chosen according to the index.

Case statements in QSI use an encapsulated function with the prototype

$$\mathbf{cif}(QuantumTypes\langle T\rangle, MeasureMatrixSet, Func\langle T\rangle, Func\langle T\rangle \ldots)$$

.

By default, the $Func\langle T\rangle$ sequence is a subprogram corresponding to a measurement output index, i.e., the nth $Func\langle T\rangle$ corresponds to the nth measurement output index. We have also considered cases where the user has not provided a subprogram corresponding to every measurement output index. In these situations, QSI 's strategy is to automatically skip that clause if the outcome index exceeds the value of $Func\langle T\rangle$. In fact, in this case, nothing needs to be done to the variables except for a measurement.

Another difference between a classical and a quantum case statement is that quantum case statement variables must be modified into the state corresponding the measurement output index after performing a measurement. We call the function to return the measurement result and go to the correct subprogram

$$int\, Measu2ResultIndex(MeasureMatrixSet)$$

, which then inherently calls the $void\, StateChange(int)$ and changes the variable $\bar{q}$ to the corresponding state after the measurement.

Loop statement

$$\text{(L0)} \quad \frac{}{\langle \mathbf{while}(M[\bar{q}] = 1)\mathbf{do}\, S\, \mathbf{od}, \rho\rangle \to \langle \mathbf{E}, M_0 \rho M_0^\dagger\rangle},$$

$$\text{(L1)} \quad \frac{}{\langle \mathbf{while}(M[\bar{q}] = 1)\mathbf{do}\, S\, \mathbf{od}, \rho\rangle \to \langle S; \mathbf{while}(M[\bar{q}] = 1)\mathbf{do}\, S\, \mathbf{od}, M_1 \rho M_1^\dagger\rangle}.$$

To implement this loop statement in QSI , we use an encapsulated function with the prototype

$$\mathbf{cwhile}(QuantumTypes\langle T\rangle, MeasureMatrixSet, int, Func\langle T\rangle)$$

This function accepts quantum types, a measurement, and an integer. Then, it compares the measurement result with the given integer in the guard. If the guard has a value of '1', it enters into the loop body, otherwise it terminates. In addition,

the state will have been changed after being measured in the guard. The function

$$int\, Measu2ResultIndex(MeasureMatrixSet)$$

is called to return the guard index and go to the correct subprogram, then it inherently calls the $void\, StateChange(int)$ as per the case statement.

Chapter 3

QSI Modules

3.1 Separation of Multipartite Quantum Gates

3.1.1 Introduction for Separability of Multipartite Quantum Gates

Along with the motivation to increase the number of accessible qubits in quantum hardware, one approach is concurrent quantum programming. Although the recent quantum-specific environments have only focussed on sequential structures, some researchers have exploited the possibility of parallel or concurrent quantum programming in different ways. For example, Vizzotto and Costa [25] applied mutually exclusive access to global variables for concurrent programming in Haskell to the case of concurrent quantum programming. Yu and Ying [26]carefully studied the termination of concurrent programs. The papers [27–30] provide mathematical tools for process algebras that describe interactions, communication, and synchronisation.

The majority of this chapter is devoted to the separability of unitary gates, which plays an important role in most programming languages as an access point to concurrent quantum programming. When implementing parallel programs, the very first obstacle is to separate multipartite quantum gates into the tensor products of local gates. If separation is possible, a potential parallel execution will result naturally. Here, we provide the sufficient and necessary conditions for the separability of multipartite gates. Unsurprisingly, multipartite

quantum gates seldom exist that can be separated simply. However, we can confirm there is always a separable gate close to a non-separable gate in certain approximate conditions.

Moreover, we show an approximate separable example in a 2-qubit system.

3.1.2 Theoretical Analysis of Quantum Gates Separation

In this analysis, let $\mathcal{H}_k$ be a separable complex Hilbert space of finite or infinite dimension and let $\otimes_{k=1}^n \mathcal{H}_k$ the tensor product of $\mathcal{H}_k$s. Denote $\mathcal{B}(\otimes_{k=1}^n \mathcal{H}_k)$,$\mathcal{U}(\otimes_{k=1}^n \mathcal{H}_k)$ and $\mathcal{B}_s(\otimes_{k=1}^n \mathcal{H}_k)$ as the set of all bounded linear operators, unitary operators and self-adjoint operators on the underlying space $\otimes_{k=1}^n \mathcal{H}_k$.

In finite-dimensional cases, a correspondence exists from the unitary group to the space of self-adjoint operators through the formula $U = \exp[it\mathbf{H}]$. However, in infinite dimensional systems not all unitary gates can be expressed using the above formula. For this purposes of this example, when a multipartite system has finite dimensions, let U be an arbitrary unitary operator on a complex Hilbert space and let $\mathbb{T}$ be the unit circumference $\{\exp[it] : t \in [0, 2\pi]\}$. As far as we know, the spectrum of U $\sigma(U) \subseteq \mathbb{T}$. We write $\sigma(U) = \{\exp[it] : t \in \Omega \subseteq [0, 2\pi]$. It follows that U has the spectral integral

$$U = \int_{t\in\Omega} \exp[it] dE_t,$$

where $\{E_t\}_{t\in\Omega}$ is the set of spectral projections. Taking $\mathbf{H} = \int_{t\in\Omega} t dE_t$, we have $\mathbf{H}$ as a self-adjoint operator and we have $U = \exp i\mathbf{H}$. However, if $\mathbf{H}$ is a self-adjoint operator with a spectral set of Δ, then $\mathbf{H} = \int_{t\in\Delta} t dE_t$. Let

$$U = \exp[i\mathbf{H}] = \int_{t\in\Omega} \exp[it] dE_t,$$

then U is a unitary operator.

Let U be a multipartite gate on the composite system $\otimes_{k=1}^n \mathcal{H}_k$. We say that U is separable (local or decomposable) if there exist quantum gates U_k on $\mathcal{H}_k$ such that

$$U = \otimes_{k=1}^n U_k \,. \tag{3.1}$$

Next, we estabilish the separability problem for multipartite gates as follows.

The Separability Problem: Consider the multipartite system $\otimes_{k=1}^{n}\mathcal{H}_k$. If $U = \exp[i\mathbf{H}]$ with $\mathbf{H} = \sum_{i=1}^{N_H} A_i^{(1)} \otimes A_i^{(2)} \otimes ... \otimes A_i^{(n)}$ for a multipartite unitary gate U, do any unitary operators U_k on $\mathcal{H}_k$ exist such that $U = \otimes_{k=1}^{n} U_k$? Further, how does the structure of each U_k depend on the exponents of $A_k^{(j)}$, $i = 1, 2, ..., n$?

Remark 3.1. Note that when the dimension of $\otimes_{k=1}^{n}\mathcal{H}_k$ is finite, every unitary gate U has the form $U = \exp[i\mathbf{H}]$ with $\mathbf{H} = \sum_{i=1}^{N_H} A_i^{(1)} \otimes A_i^{(2)} \otimes ... \otimes A_i^{(n)}$ and $N_H < \infty$. Generally speaking, in the decomposition of $\mathbf{H} = \sum_{i=1}^{N_H} A_i^{(1)} \otimes A_i^{(2)} \otimes ... \otimes A_i^{(n)}$ with $N_H < \infty$, many selections of the operator set $\{A_i^{(j)}\}_{i,j}$ (even $A_i^{(j)}$ exist that may not be self-adjoint. Hence, we call the self-adjoint decomposition of $\mathbf{H}$ if each $A_i^{(j)}$ is self-adjoint). For an arbitrary (self-adjoint or non-self-adjoint) decomposition of $\mathbf{H} = \sum_{i=1}^{N_H} B_i^{(1)} \otimes B_i^{(2)} \otimes ... \otimes B_i^{(n)}$, there exists a self-adjoint decomposition $\mathbf{H} = \sum_{i=1}^{N_H} A_i^{(1)} \otimes A_i^{(2)} \otimes ... \otimes A_i^{(n)}$ such that ([53])

$$\operatorname{span}\{B_1^{(j)}, B_2^{(j)}, ..., B_n^{(j)}\} = \operatorname{span}\{A_1^{(j)}, A_2^{(j)}, ..., A_n^{(j)}\}\,.$$

In the following, we always assume that $\mathbf{H}$ takes its self-adjoint decomposition.

To answer the separability question, we begin the discussion with a simple case of $\mathbf{H}$: the length $N_H = 1$, i.e., $\mathbf{H} = A_1 \otimes A_2 \otimes ... \otimes A_n$. Let us first deal with a case where $n = 2$.

Theorem 3.2. *Let $\mathcal{H}_1 \otimes \mathcal{H}_2$ be an arbitrary dimensional bipartite system. For a quantum gate $U = \exp[i\mathbf{H}] \in \mathcal{U}(\mathcal{H}_1 \otimes \mathcal{H}_2)$ with $\mathbf{H} = A \otimes B$, the following statements are equivalent:*

- *(I) there exist unitary operators C, D such that $U = C \otimes D$;*
- *(II) one of A, B belongs to $\mathbb{R}I$, and there exist real scalars α, β such that either $C = \exp[i(tA + \alpha I)], D = I$ if $B = tI$, or $D = \exp[i(sB + \beta I)], C = I$ if $A = sI$.*

Before the proof of Theorem 3.2, let us recall the following lemmas about separating vectors in von Neumman algebras. Let $\mathcal{A}$ be a C*-algebra on a Hilbert space $\mathcal{H}$. We call the vector $|x_0\rangle$ is a separating vector of $\mathcal{A}$ if $T(|x\rangle) = 0 \Rightarrow T = 0$ for all $T \in \mathcal{A}$.

Lemma 3.3. *[54] Every Abel C*-algebra has separating vectors.*

Proof of Theorem 3.2 (II)⇒ (I) is obvious. Next we show (I) ⇒ (II).

For any unit vector $|x\rangle, |x'\rangle$ in the first system and $|y\rangle, |y'\rangle$ in the second system. It is obvious that

$$\begin{aligned} U|xy\rangle\langle x'y'| &= \exp[iA\otimes B]|xy\rangle\langle x'y'| \\ &= |xy\rangle\langle x'y'| + iA\otimes B|xy\rangle\langle x'y'| - \frac{A^2\otimes B^2|xy\rangle\langle x'y'|}{2!} - ... \\ &+ i^k\frac{A^k\otimes B^k|xy\rangle\langle x'y'|}{k!} + ... \end{aligned} \tag{3.2}$$

We also have the equation

$$U|xy\rangle\langle x'y'| = C\otimes D|xy\rangle\langle x'y'|. \tag{3.3}$$

Connecting Eq. 3.2 and 3.3 and taking a partial trace of the second (first) system respectively, we have

$$\begin{aligned} \langle y|D|y'\rangle C|x\rangle\langle x'| &= \langle y|y'\rangle|x\rangle\langle x'| + \langle y|B|y'\rangle A|x\rangle\langle x'| \\ &- \langle y|B^2|y'\rangle\frac{A^2}{2!}|x\rangle\langle x'| - ... + i^k\langle y|B^k|y'\rangle\frac{A^k}{k!}|x\rangle\langle x'| + ... \end{aligned}$$

and

$$\begin{aligned} \langle x|C|x'\rangle D|y\rangle\langle y'| &= \langle x|x'\rangle|y\rangle\langle y'| + \langle x|A|x'\rangle B|y\rangle\langle y'| \\ &- \langle x|A^2|x'\rangle\frac{B^2}{2!}|y\rangle\langle y'| - ... + i^k\langle x|A^k|x'\rangle\frac{B^k}{k!}|y\rangle\langle y'| + ... \end{aligned}$$

It follows from the arbitrariness of $|x'\rangle$ and $|y'\rangle$ that

$$\begin{aligned} \langle y|D|y'\rangle C|x\rangle &= \langle y|y'\rangle I|x\rangle + \langle y|B|y'\rangle A|x\rangle \\ &- \langle y|B^2|y'\rangle\frac{A^2}{2!}|x\rangle - ... + i^k\langle y|B^k|y'\rangle\frac{A^k}{k!}|x\rangle + ... \end{aligned} \tag{3.4}$$

and

$$\langle x|C|x'\rangle D|y\rangle = \langle x|x'\rangle I|y\rangle + \langle x|A|x'\rangle B|y\rangle$$
$$- \langle x|A^2|x'\rangle \frac{B^2}{2!}|y\rangle - ... + i^k \langle x|A^k|x'\rangle \frac{B^k}{k!}|y\rangle + ... \quad (3.5)$$

Next, we divide the three cases to complete the proof.

Case 1. $B = tI$. Taking $y' = y$ in Eq. 3.4, we have

$$\langle y|D|y'\rangle C|x\rangle = I|x\rangle + A|x\rangle - t^2 \frac{A^2}{2!}|x\rangle - ... + i^k t^k \frac{A^k}{k!}|x\rangle + ... = \exp[itA]|x\rangle .$$

Note that C and $\exp[itA]$ are unitary, so $C = \exp[i\alpha]\exp[itA] = \exp[i(tA+\alpha I)]$. It follows that $U = \exp[i(tA + \alpha I)] \otimes I$.

Case 2. $A = sI$. Similar to Case 1, we have $D = \exp[i\beta]\exp[isB] = \exp[i(sB + \beta I)]$ for some B. It follows that $U = \exp[i(sB + \beta I)] \otimes I$.

Case 3. $A, B \notin \mathbb{R}I$. In this case, a contradiction is induced, so that Case 3 may not happen. Dividing the following two subcases, have

Subcase 3.1. Either dim $\mathcal{H}_1 \otimes \mathcal{H}_2 < \infty$, or dim $\mathcal{H}_1 \otimes \mathcal{H}_2 = \infty$ but both A and B have two distinct eigenvalues. It follows that there exist two different real numbers t_1, t_2 such that $A|x_1\rangle = t_1|x_1\rangle$ and $A|x_2\rangle = t_2|x_2\rangle$, and $s_1 \neq s_2$ such that $B|y_1\rangle = s_1|y_1\rangle$ and $B|y_2\rangle = s_2|y_2\rangle$. Taking $|x\rangle = |x'\rangle = |x_1\rangle$ and $|x\rangle = |x'\rangle = |x_2\rangle$ in Eq. 3.5 respectively, and $|y\rangle = |y'\rangle = |y_1\rangle$ and $|y\rangle = |y'\rangle = |y_2\rangle$ in Eq. 3.4 respectively, we have that

$$\langle x_1|C|x_1\rangle D = \exp[t_1 B], \langle x_2|C|x_2\rangle D = \exp[t_2 B] ,$$

$$\langle y_1|D|y_1\rangle C = \exp[s_1 A], \langle y_2|D|y_2\rangle C = \exp[s_2 A] .$$

It follows that

$$\langle x_1|C|x_1\rangle = \frac{\exp[s_1 t_1]}{\langle y_1|D|y_1\rangle}, \langle x_2|C|x_2\rangle = \frac{\exp[s_1 t_2]}{\langle y_1|D|y_1\rangle} .$$

So

$$\frac{\langle y_1|D|y_1\rangle \exp[t_1 B]}{\exp[s_1 t_1]} = D = \frac{\langle y_1|D|y_1\rangle \exp[t_2 B]}{\exp[s_1 t_2]}.$$

Taking the inner product for $|y_2\rangle$ on both sides of the above equation, we have

$$\frac{\exp[t_1 s_2]}{\exp[t_1 s_1]} = \frac{\exp[t_2 s_2]}{\exp[t_2 s_1]}.$$

It follows that $\exp[t_1 s_2 - t_1 s_1] = \exp[t_2 s_2 - t_2 s_1]$, i.e., $t_1 = t_2$. This is a contradiction.

Subcase 3.2. When Subcase 3.1 does not happen, since $A, B \notin \mathbb{R}I$, then A or B has the continuous spectrum set. Assume that A has the continuous spectrum set, similarly we can deal with the case when B has the continuous spectrum set. Let $\mathcal{A} = \text{cl span}\{I, A, A^2, ..., A^n, ...\}$ be an Abelian C*-algebra. By Lemma 3.3 , it has a separating vector $|x_0\rangle$. Replacing $|x\rangle$ with $|x_0\rangle$ and taking vectors the $|y\rangle, |y'\rangle$ satisfies $\langle y|D|y'\rangle = 0$. Then,

$$\begin{aligned} 0 &= \langle y|D|y'\rangle C|x_0\rangle \\ &= \langle y|y'\rangle I|x_0\rangle + \langle y|B|y'\rangle A|x_0\rangle - \langle y|B^2|y'\rangle \frac{A^2}{2!}|x_0\rangle - ... + i^k \langle y|B^k|y'\rangle \frac{A^k}{k!}|x_0\rangle + ... \\ &= (\sum_k \lambda_k A^k)|x_0\rangle \end{aligned} \tag{3.6}$$

where $\lambda_k = \frac{i^k \langle y|B^k|y'\rangle}{k!}$. It follows from the definition of separating vectors that $\sum_k \lambda_k A^k = 0$. Next we first show that each $\lambda_k = 0$. For any fixed $|y\rangle, |y'\rangle$, note that the function $f(z) = \sum_k \lambda_k z^k$ is analytic. Since $f(A) = 0$, the spectrum set of $f(A)$ $\sigma(f(A))$ contains the unique element 0, i.e.,

$$\{0\} = \sigma(f(A)) = \{f(\lambda)|\lambda \in \sigma(A)\}.$$

Note that A has the continuous spectrum set. So the analytic function $f(z)$ is zero. Then each $\lambda_k = 0$. It follows that

$$\langle y|B^k|y'\rangle = 0, \ \ k = 0, 1, 2, ..., n, ...$$

for any vectors $|y\rangle, |y'\rangle$ satisfying $\langle y|D|y'\rangle = 0$. When $k = 0$, we have $\langle y|D|y'\rangle = 0 \Rightarrow \langle y|y'\rangle = 0$ for any vector $|y\rangle, |y'\rangle$. Thus $D \in \mathbb{R}I$. When $k = 1$, we have that $\langle y|D|y'\rangle = 0 \Rightarrow \langle y|B|y'\rangle = 0$

for any vector $|y\rangle, |y'\rangle$. Then D, B is linearly dependent. So $B \in \mathbb{R}I$, which is a contradiction to Case 3.

This completes the proof. □

When the bipartite system has finite dimensions, the following corollary applies.

Corollary 3.4. *Assume that dim* $(\mathcal{H}_1 \otimes \mathcal{H}_2) < \infty$. *For a bipartite quantum gate* $U = \exp[i\mathbf{H}] \in \mathcal{U}(\mathcal{H}_1 \otimes \mathcal{H}_2)$ *with* $\mathbf{H} = A \otimes B$, U *is separable if and only if one of* A, B *belongs to* $\mathbb{R}I$, *and there exist real scalars* α, β *such that either* $C = \exp[i(tA + \alpha I)], D = I$ *if* $B = tI$, *or* $D = \exp[i(sB + \beta I)], C = I$ *if* $A = sI$.

Next, we extend the result for Theorem 3.2 to a multipartite case. Before introducing the claim, some notations are required. Let A_i be self-adjoint operators on $\mathcal{H}_i$, $i = 1, 2, ..., n$ such that $\mathbf{H} = A_1 \otimes A_2 \otimes ... \otimes A_n$. If there exists at most one element in the set $\{A_1, A_2, ..., A_n\}$ which does not belong to the set $\mathbb{R}I$, we define a scalar

$$\delta(A_j) = \begin{cases} \prod_{k \neq j} \lambda_k, \ (A_j \notin \mathbb{R}I) \\ 0, \ (A_j \in \mathbb{R}I) \end{cases} \tag{3.7}$$

where $A_k = \lambda_k I$ when $A_k \in \mathbb{R}I$.

Based on Theorem 3.2, we reach the following conclusion in the multipartite case.

Theorem 3.5. *Let* $\otimes_{i=1}^n \mathcal{H}_i$ *be the arbitrary dimensional multipartite system. For a multipartite quantum gate* $U = \exp[i\mathbf{H}] \in \mathcal{U}(\otimes_{i=1}^n \mathcal{H}_i)$ *with* $\mathbf{H} = A_1 \otimes A_2 \otimes ... \otimes A_n$, *the following statements are equivalent:*

(I) there exist unitary operators $C_i \in \mathcal{U}(\mathcal{H}_i)$ $(i = 1, 2, ..., n)$ *such that* $U = \otimes_{i=1}^n C_i$;

(II) there exist at most one element in $\{A_i\}_{i=1}^n$ *not belong to* $\mathbb{R}I$, *and there is a number of unit modulus number* λ *such that*

$$U = \lambda \otimes_{j=1}^n \exp[i\delta(A_j) A_j]. \tag{3.8}$$

Proof. (II) $\Rightarrow$ (I) is obvious. To prove (I) $\Rightarrow$ (II), we use induction on n.

According to Theorem 3.2, (I) $\Rightarrow$ (II) holds true for $n = 2$. Assume that the implication is true for $n = k$. Now let $n = k + 1$. We have

$$\exp[iA_1 \otimes A_2 \otimes ... \otimes A_{k+1}] = \exp[i\mathbf{H}] = C_1 \otimes C_2 \otimes ... \otimes C_k \otimes C_{k+1} = T \otimes C_{k+1}\,.$$

It follows from Theorem 3.2 that either $A_{k+1} \in \mathbb{R}I$ or $A_1 \otimes A_2 \otimes ... \otimes A_k \in \mathbb{R}I$. If $A_1 \otimes A_2 \otimes ... \otimes A_k \in \mathbb{R}I$, then each A_i belongs to $\mathbb{R}I$. According to the induction assumption, (II) holds true. If $A_{k+1} \in \mathbb{R}I$, assume that $A_{k+1} = wI$, then

$$\exp[i\mathbf{H}] = \exp[iwA_1 \otimes A_2 \otimes ... \otimes A_k] \otimes I = C_1 \otimes C_2 \otimes ... \otimes C_k \otimes I.$$

It follows from the induction assumption that (II) holds true. Eq. 3.8 is obtained by repeating uses of (II) in Theorem 3.2. This completes the proof. □

Similarly, in the finite dimensional case, we have the following corollary.

Corollary 3.6. *Assume that $dim(\otimes_{i=1}^n \mathcal{H}_i) < \infty$. For a multipartite quantum gate $U = \exp[i\mathbf{H}] \in \mathcal{U}(\otimes_{i=1}^n \mathcal{H}_i)$ with $\mathbf{H} = A_1 \otimes A_2 \otimes ... \otimes A_n$, U is separable if and only if there exist at most one element in $\{A_i\}_{i=1}^n$ that does not belong to $\mathbb{R}I$, and there is a unit-model number λ such that*

$$U = \lambda \otimes_{j=1}^n \exp[\delta(A_j)A_j].$$

Next we turn to a general case of $\mathbf{H}$: $1 < N_H < \infty$.

Before the main results, an important tools needs to be introduced. The Zassenhaus formula [55] states that

$$\exp[A + B] = \exp[A]\exp[B]\mathcal{P}_z(A, B)\,, \tag{3.9}$$

where $\mathcal{P}_z(A, B) = \Pi_{i=2}^{\infty} \exp[C_i(A, B)]$ and each a term $C_i(A, B)$ is a homogeneous Lie polynomial in the variables A, B. That is, $C_i(A, B)$ is a linear combination (with rational coefficients) of the commutators in the form $[V_1...[V_2, ..., [V_{m-1}, V_m]...]]$ with $V_i \in \{A, B\}$ ([55, 56]). Especially, $C_2(A, B) = -\frac{1}{2}[A, B]$ and $C_3(A, B) = \frac{1}{3}[B, [A, B]] + \frac{1}{6}[A, [A, B]]$. As shown, if $\Pi_{i=2}^{\infty} \exp[C_i(A, B)]$ is a multiple of the identity, $\exp[A]\exp[B] = \lambda \exp[A + B]$. However, when $AB = BA$, $\Pi_{i=2}^{\infty} \exp[C_i(A, B)] \in \mathbb{C}I$, what happens in a multi-variable

case? We have

$$\begin{aligned}\exp[\sum_{i=1}^{N} A_i] &= \prod_{i=1}^{N} \exp[A_i]\mathcal{P}_z(A_{N-1}, A_N)\cdot\mathcal{P}_z(A_{N-2}, A_{N-1}+A_N)...\mathcal{P}_z(A_1, \sum_{j=2}^{N} A_j)\\ &= \prod_{i=1}^{N}\exp[A_i]\prod_{k=1}^{N}\mathcal{P}_z(A_k, \sum_{j=k+1}^{N} A_j).\end{aligned} \tag{3.10}$$

Assume that a multipartite quantum gate $U = \exp[-it\mathbf{H}]$ with $\mathbf{H} = \sum_{i=1}^{N_H} T_i = \sum_{i=1}^{N_H} A_i^{(1)} \otimes A_i^{(2)} \otimes ... \otimes A_i^{(n)}$. If there exists at most one element in each set $\{A_i^{(1)}, A_i^{(2)}, ..., A_i^{(n)}\}$ that does not belong to the set $\mathbb{R}I$, we define the function:

$$\delta(A_k^{(i)}) = \begin{cases}\prod_{k\neq i}\lambda_j^{(k)}, & (A_j^{(i)} \notin \mathbb{R}I)\\ 0, & (A_j^{(i)} \in \mathbb{R}I)\end{cases} \tag{3.11}$$

where $A_j^{(k)} = \lambda_j^{(k)} I$ if $A_j^{(k)} \in \mathbb{R}I$.

Theorem 3.7. *For a multipartite quantum gate $U \in \mathcal{U}(\otimes_{k=1}^{n}\mathcal{H}_k)$, if $U = \exp[-it\mathbf{H}]$ with $\mathbf{H} = \sum_{i=1}^{N_H} T_i = \sum_{i=1}^{N_H} A_i^{(1)} \otimes A_i^{(2)} \otimes ... \otimes A_i^{(n)}$ and each $A_i^{(j)}$ has two eigenvalues at least for any i, j, the product of homogeneous Lie polynomials in Eq. 3.10 $\prod_{k=1}^{N}\mathcal{P}_z(T_k, \sum_{j=k+1}^{N} T_j) \in \mathbb{C}I$ and there exists at most one element in each set $\{A_i^{(1)}, A_i^{(2)}, ..., A_i^{(n)}\}$ that does not belong to the set $\mathbb{R}I$, then*

$$U = U^{(1)} \otimes U^{(2)} \otimes ... \otimes U^{(n)}, \tag{3.12}$$

where $U^{(i)}$ is the local quantum gate on H_i,

$$U^{(i)} = \prod_{k=1}^{N_H} \exp[it\delta(A_k^{(i)})A_k^{(i)}],$$

where $\delta_k^{(i)}$ is defined by Eq. 3.11.

Remark 3.8. In Theorem 3.7, we provided a sufficient condition for the separability of a multipartite gate. However, this condition is not easy to check since the product of homogeneous Lie polynomials in Eq. 3.10 $\prod_{k=1}^{N}\mathcal{P}_z(T_k, \sum_{j=k+1}^{N} T_j)$ is complex and difficult to calculate. We observe that if $[T_k, T_l] \in \mathbb{C}I$ for any pair k, l, then $\prod_{k=1}^{N}\mathcal{P}_z(T_k, \sum_{j=k+1}^{N} T_j) \in \mathbb{C}I$. To make this easier to check, if $[T_k, T_l] \in \mathbb{C}I$ and there exists at most one element in $\{A_i^{(1)}, A_i^{(2)}, ..., A_i^{(n)}\}$ that does not belong to the set $\mathbb{R}I$, then U has the tensor product

decomposition in Eq 3.12. Impressively, in the finite-dimensional case, if $[T_k, T_l] \in \mathbb{C}I$, $[T_k, T_l]$ has to be zero. Moreover, when the system is of finite-dimension, the condition "each $U_i^{(j)}$ has two eigenvalues at least for any i, j" can be omitted.

Proof of Theorem 3.7 Let $U \in \mathcal{U}(\otimes_{k=1}^{n} \mathcal{H}_k)$, $U = \exp[-it\mathbf{H}]$, and $\mathbf{H} = \sum_{i=1}^{N_H} T_i = \sum_{i=1}^{N_H} A_i^{(1)} \otimes A_i^{(2)} \otimes ... \otimes A_i^{(n)}$.

Let us first observe that for any real number r, $\exp[rT] = (\exp[T])^r$. Further, $\exp[rT \otimes S] = \exp[rT] \otimes \exp[rS]$ if $\exp[T \otimes S] = \exp[T] \otimes \exp[S]$. Indeed, for an arbitrary positive integer N, it follows from Baker's formula that $\exp[NT] = (\exp[T])^N$. In addition, $\exp[T] = \exp[\frac{T}{M} \cdot M]$, so $\exp[\frac{T}{M}] = (\exp[T])^{\frac{1}{M}}$. In summary, for any rational number a, $\exp[aT] = (\exp[T])^a$, so does any real number.

According to the assumption and the definition of $\delta_j^{(i)}$, when writing

$$\prod_{k=1}^{N} \mathcal{P}_z(T_k, \sum_{j=k+1}^{N} T_j) = \lambda I \,,$$

it follows from Theorem 3.5 that

$$\begin{aligned} U &= \exp[it\mathbf{H}] \\ &= \exp[it(\sum_{i=1}^{N_H} T_i)] \\ &= \prod_{i=1}^{N_H} \exp[itT_i] \prod_{k=1}^{N} \mathcal{P}_z(T_k, \sum_{j=k+1}^{N} T_j) \\ &= \lambda \prod_{i=1}^{N_H} \exp[itT_i] \\ &= \lambda \prod_{i=1}^{N_H} \exp[itA_i^{(1)} \otimes A_i^{(2)} \otimes ... \otimes A_i^{(n)}] \\ &= \lambda \prod_{k=1}^{N_H} \exp[it\delta_k^{(1)} A_k^{(1)}] \otimes \prod_{k=1}^{N_H} \exp[it\delta_k^{(2)} A_k^{(2)}] \otimes ... \otimes \prod_{k=1}^{N_H} \exp[it\delta_k^{(n)} A_k^{(n)}] \,. \end{aligned}$$

Absorbing the unit mode scalar λ, let $U^{(i)} = \prod_{k=1}^{N_H} \exp[it\delta_k^{(i)} A_k^{(i)}]$. This completes the proof. $\square$

Some examples of separable gates follow.

Here, we assume that the Planck constant equals to 1 and denote σ_X, σ_Y and σ_Z as the Pauli matrices $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$.

Example 3.1. *In the 2-qubit composite spin-$\frac{1}{2}$ system, the total spin operator $\mathbf{S}^2$ is defined by $\mathbf{S}^2 = S_X^2 + S_Y^2 + S_Z^2$, where*

$$S_X = \sigma_X \otimes I + I \otimes \sigma_X \,,$$

$$S_Y = \sigma_Y \otimes I + I \otimes \sigma_Y \,,$$

$$S_Z = \sigma_Z \otimes I + I \otimes \sigma_Z \,.$$

The three operators S_X, S_Y, S_Z assign the X, Y, Z components of the spin to the composite system respectively.

Considering the Hamiltonian $H_X = I \otimes \sigma_X + \sigma_X \otimes I$ and since H_X satisfies the separation conditions in Theorem 3.5, we can decompose the X-spin quantum gate by applying the Eq. 3.12 as follows:

$$\begin{aligned} U_X &= \exp[-itH_X] \\ &= \exp[-itI \otimes \sigma_X - it\sigma_X \otimes I] \\ &= \exp[i\delta_1^{(1)} I] \exp[i\delta_2^{(1)} \sigma_X] \otimes \exp[i\delta_1^{(2)} \sigma_X] \exp[i\delta_2^{(2)} I] \,. \end{aligned}$$

It follows from the definition of $\delta_j^{(i)}$ that $\delta_1^{(1)} = 0$, $\delta_2^{(1)} = -t$, $\delta_1^{(2)} = -t$ and $\delta_2^{(2)} = 0$. Thus, we have

$$U_X = \exp[-itH_X] = \exp[-it\sigma_X] \otimes \exp[-it\sigma_X] \,.$$

Similarly, we have

$$U_Y = \exp[-itH_Y] = \exp[-it\sigma_Y] \otimes \exp[-it\sigma_Y] \,.$$

$$U_Z = \exp[-itH_Z] = \exp[-it\sigma_Z] \otimes \exp[-it\sigma_Z] \,.$$

Generally, we have

$$U_{A,B} = \exp[-itH_{A,B}] = \exp[-it\sigma_A] \otimes \exp[-it\sigma_B] \,,$$

where $H_{A,B} = A \otimes I + I \otimes B$.

Further, let us consider the so-called special 7-parameter Hamiltonian introduced in [57], where

$$\mathbf{H} = \sum_{i=1}^{4} (a_i \sigma_i \otimes I + I \otimes b_i \sigma_i),$$

with $a_0 + b_0 = \mathrm{tr}(\mathbf{H})$ (i.e., with seven not eight parameters). We rewrite $\mathbf{H} = (\sum_{i=1}^{4} a_i \sigma_i) \otimes I + I \otimes (\sum_{i=1}^{4} b_i \sigma_i)$.

$$U = \exp[-it\mathbf{H}] = \exp[-it(\sum_{i=1}^{4} a_i \sigma_i)] \otimes \exp[-it(\sum_{i=1}^{4} b_i \sigma_i)].$$

Example 3.2. *In the bipartite continuous variable system $\mathcal{H}_A \otimes \mathcal{H}_B$, let $(\hat{x}, \hat{p})$ be the pair of the position and momentum operators on the system $\mathcal{H}_A$. For any real numbers r_1, r_2, s_1, s_2, and $\mathbf{H} = (r_1 \hat{x} + s_1 \hat{p}) \otimes I + (r_2 \hat{x} + s_2 \hat{p}) \otimes I$. Since*

$$[r_1 \hat{x} + s_1 \hat{p}, r_2 \hat{x} + s_2 \hat{p}] = -i\hbar(r_1 s_2 + r_2 s_1) I,$$

So $\mathbf{H}$ satisfies the separation conditions of Theorem 3.5 (see Remark 3.8). It follows that

$$\begin{aligned}
U &= \exp[i\mathbf{H}] \\
&= \exp[-i\frac{\hbar(r_1 s_2 + r_2 s_1)}{2}] \exp[i(r_1 \hat{x} + s_1 \hat{p})] \otimes \exp[i(r_2 \hat{x} + s_2 \hat{p})] \\
&= \exp[-i\frac{\hbar(r_1 s_2 + r_2 s_1)}{2}] \exp[-i\frac{\hbar r_1 s_1}{2}] \exp[ir_1 \hat{x}] \exp[is_1 \hat{p})] \\
&\quad \otimes \exp[-i\frac{\hbar r_2 s_2}{2}] \exp[ir_2 \hat{x}] \exp[is_2 \hat{p})] \\
&= \exp[-i\frac{\hbar(r_1 s_2 + r_2 s_1 + r_1 s_1 + r_2 s_2)}{2}] \exp[ir_1 \hat{x}] \exp[is_1 \hat{p})] \otimes \exp[ir_2 \hat{x}] \exp[is_2 \hat{p})].
\end{aligned}$$

3.1.3 Approximate Separation of Multipartite Gates

In this section, we turn to the approximate separation problem with multipartite gates.

ϵ-approximate separation question *Given a positive scalar ϵ and a multipartite quantum gate $U \in \mathcal{U}(\otimes_{k=1}^{n}\mathcal{H}_k)$, whether or not there are local gates $U_i \in \mathcal{U}(\mathcal{H}_i)$ such that*

$$d(U, \otimes_{i=1}^{n} U_i) < \epsilon, \tag{3.13}$$

where $d(\cdot,\cdot)$ is a distance of two operators. We consider U to be ϵ-approximate separable if Eq. 3.13 holds true. Further, the way to search these local gates U_i.

Remark 3.9. Note that the set of tensor products of local unitary gates $\mathcal{U}_l = \{\otimes_{i=1}^{n} U_i | U_i \in \mathcal{U}(H_i)\}$ is closed, it follows that $d(U, \mathcal{U}_l) = \epsilon_0 > 0$ when U is not separable. So Eq. 3.13 holds true only if ϵ is greater then ϵ_0. So ϵ can not be chosen freely.

In the following theorem, we provide an upper bound for the distance between an arbitrary unitary operator and a separable one.

Theorem 3.10. *Let $U = \exp[it\mathbf{H}] \in \mathcal{U}(\otimes_{k=1}^{n}\mathcal{H}_k)$ be a multipartite quantum gate with $\mathbf{H}$ and $U_i = \exp[it\mathbf{H}_i] \in \mathcal{U}(\mathcal{H}_i)$ with $\mathbf{H}_i \in \mathcal{B}_s(\mathcal{H}_i)$. Then*

(I)

$$\|U - \otimes_{i=1}^{n} U_i\| \leq M \|\mathbf{H} - \sum_i \hat{\mathbf{H}}_i\|, \tag{3.14}$$

where $\hat{\mathbf{H}}_i = I_1 \otimes I_2 \otimes \ldots I_{i-1} \otimes \mathbf{H}_i \otimes I_{i+1} \otimes \ldots I_n$, I_j is the identity on $\mathcal{H}_j$, $M = |t| \| \exp[-it\sum_{i=1}^{n} \hat{\mathbf{H}}_i] \| \| \exp[-it\mathbf{H}] \|$ and $\| \cdot \|$ denotes an arbitrary norm of operators.

(II) If the norm is chosen as the operator norm, then

$$\|U - \otimes_{i=1}^{n} U_i\|_o \leq |t| \|\mathbf{H} - \sum_i \hat{\mathbf{H}}_i\|_o. \tag{3.15}$$

Remark 3.11. The norm $\| \cdot \|$ in Eq. 3.14 can be selected freely. For example in Eq. 3.14, when we choose the operator norm $\| \cdot \|_o$ defined by $\|A\|_o = \sup_x \frac{\|Ax\|}{\|x\|}$, then $M = |t|$, since $\| \exp[-it\sum_{i=1}^{n} \hat{\mathbf{H}}_i] \|_o = 1 = \| \exp[-it\mathbf{H}] \|_o$. So Eq. 3.14 can be simplified as Eq. 3.15. In the finite-dimensional case, the norm $\| \cdot \|$ can be selected as arbitrary matrix norm, including the trace norm and Hilbert-Schmidt norm.

Before the proof of Theorem 3.10, we need the following lemmas. The first lemma is obvious from Theorem 3.5.

Lemma 3.12. *For self-adjoint operators $A_i s$, $\otimes_{i=1}^n \exp[-itA_i] = \exp[\sum_{i=1}^n(-it\hat{A}_i)]$, where $\hat{A}_i = I_1 \otimes I_2 \otimes \dots I_{i-1} \otimes A_i \otimes I_{i+1} \otimes \dots I_n$.*

Lemma 3.13. *([58])* $\exp[A+B] - \exp[A] = \int_0^1 \exp[(1-t)A]B\exp[t(A+B)]\mathrm{d}t.$

Proof of Theorem 3.2 According to the assumption, it follows from Lemma 3.12 and Lemma 3.13 that

$$\begin{aligned}
\|U - \otimes_{i=1}^n U_i\| &= \|\exp[-it\mathbf{H}] - \otimes_{i=1}^n \exp[-it\mathbf{H}_i]\| \\
&= \|\exp[-it\mathbf{H}] - \exp[\sum_{i=1}^n(-it\hat{\mathbf{H}}_i)]\| \\
&= \|\int_0^1 dx \exp[(1-x)(-it\hat{\mathbf{H}}_i)](-it\mathbf{H} - \sum_{i=1}^n(-it\hat{\mathbf{H}}_i))\exp[x(-it\mathbf{H})]\| \\
&\le \|\exp[-it\hat{\mathbf{H}}_i]\|\| - it\mathbf{H} - \sum_{i=1}^n(-it\hat{\mathbf{H}}_i)\|\|\exp[-it\mathbf{H}]\| \\
&= |t|\|\exp[-it\sum_{i=1}^n\hat{\mathbf{H}}_i]\|\|\mathbf{H} - \sum_{i=1}^n\hat{\mathbf{H}}_i\|\|\exp[-it\mathbf{H}]\|.
\end{aligned}$$

Let $M = |t|\|\exp[-it\sum_{i=1}^n\hat{\mathbf{H}}_i]\|\|\exp[-it\mathbf{H}]\|$, we complete the proof. □

The next task is to design an operational program for approximate separation of a multipartite quantum gate.

To arrive at the approximate separation for a given approximate bound ϵ and a multipartite gate $U = \exp[it\mathbf{H}] \in \mathcal{U}(\otimes_{k=1}^n\mathcal{H}_k)$, we find self-adjoint operators $\hat{\mathbf{H}}_i$ such that in Eq. 3.14,

$$\|\mathbf{H} - \sum_i\hat{\mathbf{H}}_i\| < \frac{\epsilon}{M}, \tag{3.16}$$

so Eq. 3.13 holds true.

We take the real variable set as $\{a_{k_i}^{(i)}\}_{k_i=1}^{N_i}$ with $N_i = \dim(\mathcal{H}_i) \le +\infty$, $i = 1,2,...,n$, where each $a_{k_i}^{(i)}$ is unknown. Let

$$\Lambda_i = \quad \mathrm{diag}\{a_{k_i}^{(i)}\}_{k_i=1}^{N_i}.$$

Note that $\hat{\mathbf{H}}_i$ in Theorem 3.10 will belong to the following local unitary orbit:

$$\mathcal{U}^{local} = \{I \otimes (U_i \text{diag}(\Lambda_i) U_i^*) \otimes I : \forall\ U_i \in \mathcal{U}(\mathcal{H}_i)\}.$$

So $\sum_i \hat{\mathbf{H}}_i$ will be in

$$\mathcal{U}^{local}(\sum_i \hat{\mathbf{H}}_i) = \{\sum_i I \otimes (U_i \text{diag}(\Lambda_i) U_i^*) \otimes I : \forall\ U_i \in \mathcal{U}(\mathcal{H}_i)\}.$$

Thus, Eq. 3.16 is equivalent to

$$d(\mathbf{H}, \mathcal{U}^{local}(\sum_i \hat{\mathbf{H}}_i)) < \frac{\epsilon}{M}. \tag{3.17}$$

In summary, the following sequential approach is designed to arrive at approximate separation.

For a given multipartite gate $U = \exp[it\mathbf{H}] \in \mathcal{U}(\otimes_{k=1}^n \mathcal{H}_k)$ and the bound ϵ:

1. Assume the real variable set $\{a_{k_i}^{(i)}\}_{k_i=1}^{N_i}$, and select the available norm for the distance is $d(\mathbf{H}, \mathcal{U}^{local}(\sum_i \hat{\mathbf{H}}_i))$;
2. Design a program to calculate the minimal value:
$$\inf d(\mathbf{H}, \mathcal{U}^{local}(\sum_i \hat{\mathbf{H}}_i));$$
3. When $m < \epsilon/M$, estabilish the corresponding the parameters of Λ_i and the local unitary operators U_is. Therefore, we obtain each $\mathbf{H}_i$, and the local gates $U_i = \exp[-it\mathbf{H}_i]$.

3.1.4 Approximate Separation in a 2-qubit $\frac{1}{2}$-spin System

Example 3.3. *In a 2-qubit $\frac{1}{2}$-spin system, an arbitrary quantum gate $U = \exp[it\mathbf{H}]$ with a 4×4 Hermitian matrix $\mathbf{H}$. Let $d(A, B) = \|A - B\|_o$. Assume a 2×2 real diagonal matrix $\Lambda_1 = \text{diag}(r_1, r_2)$ and $\Lambda_2 = \text{diag}(s_1, s_2)$. To calculate the distance, use*

$$d(\mathbf{H}, \mathcal{U}^{local}(\Lambda_1 \otimes I_2 + I_1 \otimes \Lambda_2)) = \sup_{U_1, U_2} \|\mathbf{H} - U_1 \Lambda_1 U_1^* \otimes I_2 + I_1 \otimes U_2 \Lambda_2 U_2^*\|_o, \tag{3.18}$$

where U_1, U_2 run through all 2×2 unitary matrices.

$$\|\mathbf{H} - (U_1\Lambda_1U_1^* \otimes I_2 + I_1 \otimes U_2\Lambda_2U_2^*)\|_o = \lambda_{\max}(\mathbf{H} - (U_1\Lambda_1U_1^* \otimes I_2 + I_1 \otimes U_2\Lambda_2U_2^*)) \tag{3.19}$$

where $\lambda_{\max}(A)$ denotes the maximal eigenvalue of $\sqrt{AA^*}$.

Taking a simple example. Let $\epsilon_0 > 0$ and $U = \exp[-it\mathbf{H}_X]$ with $\mathbf{H}_X = \sigma_X \otimes \sigma_X$, where $\sigma_X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$. From Theorem 3.2, U can not be separated directly. In the following discussion, we take an technique for Eq 3.18.

$$\begin{aligned} &\|\mathbf{H}_X - (U_1\Lambda_1U_1^* \otimes I_2 + I_1 \otimes U_2\Lambda_2U_2^*)\|_o \\ \le\ &\|\frac{1}{2}\mathbf{H}_X - U_1\Lambda_1U_1^* \otimes I_2\|_o + \|\frac{1}{2}\mathbf{H}_X - I_1 \otimes U_2\Lambda_2U_2^*\|_o\,. \end{aligned} \tag{3.20}$$

The following explanations dealing with each of the four cases separately. To simplified the cases, we have chosen U_1, U_2 as the real orthogonal matrices with the parameters θ_1, θ_2.

Case 1. U_1, U_2 are all symplectic. An element T in the local unitary orbit $\mathcal{U}^{local}(\Lambda_1 \otimes I_2)$ have the following form:

$$\begin{pmatrix} r_1\cos^2\theta_1 + r_2\sin^2\theta_1 & 0 & (r_2 - r_1)\cos\theta_1\sin\theta_1 & 0 \\ 0 & r_1\cos^2\theta_1 + r_2\sin^2\theta_1 & 0 & (r_2 - r_1)\cos\theta_1\sin\theta_1 \\ (r_2 - r_1)\cos\theta_1\sin\theta_1 & 0 & r_1\sin^2\theta_1 + r_2\cos^2\theta_1 & 0 \\ 0 & (r_2 - r_1)\cos\theta_1\sin\theta_1 & 0 & r_1\sin^2\theta_1 + r_2\cos^2\theta_1 \end{pmatrix}$$

So

$$\|\frac{1}{2}\mathbf{H}_X - U_1\Lambda_1U_1^* \otimes I_2\|_o = \frac{1}{2}(r_1 + r_2 \pm \sqrt{(r_1 + r_2)^2\cos^2 2\theta_1 + ((r_2 - r_1)\sin 2\theta_1 \pm 1)^2}\,.$$

Similarly, an element in the local unitary orbit $\mathcal{U}^{local}(I_1 \otimes \Lambda_2)$ has the following form:

$$\begin{pmatrix} s_1\cos^2\theta_2 + s_2\sin^2\theta_2 & (s_2 - s_1)\cos\theta_2\sin\theta_2 & 0 & 0 \\ (s_2 - s_1)\cos\theta_2\sin\theta_2 & s_1\sin^2\theta_2 + s_2\cos^2\theta_2 & 0 & 0 \\ 0 & 0 & s_1\cos^2\theta_2 + s_2\sin^2\theta_2 & (s_2 - s_1)\cos\theta_2\sin\theta_2 \\ 0 & 0 & (s_2 - s_1)\cos\theta_2\sin\theta_2 & s_1\sin^2\theta_2 + s_2\cos^2\theta_2 \end{pmatrix}.$$

So

$$\|\frac{1}{2}\mathbf{H}_X - U_1\Lambda_1 U_1^* \otimes I_2\|_o = \frac{1}{2}(s_1 + s_2 \pm \sqrt{(s_1+s_2)^2\cos^2 2\theta_2 + ((s_2-s_1)\sin 2\theta_2 \pm 1)^2}.$$

It follows that

$$\begin{aligned}
&\|\mathbf{H}_X - U_1\Lambda_1 U_1^* \otimes I_2 + I_1 \otimes U_2\Lambda_2 U_2^*\|_o \\
\le &\frac{1}{2}(r_1 + r_2 \pm \sqrt{(r_1+r_2)^2\cos^2 2\theta_1 + ((r_2-r_1)\sin 2\theta_1 \pm 1)^2} \\
&+ \frac{1}{2}(s_1 + s_2 \pm \sqrt{(s_1+s_2)^2\cos^2 2\theta_2 + ((s_2-s_1)\sin 2\theta_2 \pm 1)^2} \\
=&B_1(r_1, r_2, s_1, s_2, \theta_1, \theta_2).
\end{aligned}$$

Case 2. U_1 is symplectic but U_2 is not. Similar to the ideal to deal solution in Case 1, we have

$$\begin{aligned}
&\|\mathbf{H}_X - U_1\Lambda_1 U_1^* \otimes I_2 + I_1 \otimes U_2\Lambda_2 U_2^*\|_o \\
\le &\frac{1}{2}(r_1 + r_2 \pm \sqrt{(r_1+r_2)^2\cos^2 2\theta_1 + ((r_2-r_1)\sin 2\theta_1 \pm 1)^2} \\
&+ \frac{1}{2}(s_1 - s_2\cos 2\theta_2 \pm \sqrt{(s_1+s_2)^2\cos^2 2\theta_2 + ((s_2-s_1)\sin 2\theta_2 \pm 1)^2} \\
= &B_2(r_1, r_2, s_1, s_2, \theta_1, \theta_2).
\end{aligned}$$

Case 3. U_2 is symplectic but U_1 is not. Similar to the ideal solution in Case 1, we have

$$\begin{aligned}
&\|\mathbf{H}_X - U_1\Lambda_1 U_1^* \otimes I_2 + I_1 \otimes U_2\Lambda_2 U_2^*\|_o \\
\le &\frac{1}{2}(r_1 - r_2\cos 2\theta_1 \pm \sqrt{(r_1+r_2)^2\cos^2 2\theta_1 + ((r_2-r_1)\sin 2\theta_1 \pm 1)^2} \\
&+ \frac{1}{2}(s_1 + s_2 \pm \sqrt{(s_1+s_2)^2\cos^2 2\theta_2 + ((s_2-s_1)\sin 2\theta_2 \pm 1)^2} \\
= &B_3(r_1, r_2, s_1, s_2, \theta_1, \theta_2).
\end{aligned}$$

Case 4. U_1, U_2 are not symplectic. Similar to the ideal solution in Case 1, we have

$$\begin{aligned}
&\|\mathbf{H}_X - U_1\Lambda_1 U_1^* \otimes I_2 + I_1 \otimes U_2\Lambda_2 U_2^*\|_o \\
\leq\ & \frac{1}{2}(r_1 + r_2 \pm \sqrt{(r_1+r_2)^2\cos^2 2\theta_1 + ((r_2-r_1)\sin 2\theta_1 \pm 1)^2} \\
&+ \frac{1}{2}(s_1 + s_2 \pm \sqrt{(s_1+s_2)^2\cos^2 2\theta_2 + ((s_2-s_1)\sin 2\theta_2 \pm 1)^2} \\
=\ & B_4(r_1, r_2, s_1, s_2, \theta_1, \theta_2).
\end{aligned}$$

Thus, we have the following conclusion.

Proposition 3.14. *Let $\epsilon_0 > 0$ and $U = \exp[it\sigma_X \otimes \sigma_X]$ on $\mathcal{H}_A \otimes \mathcal{H}_B$. Then there exists a U_A, U_B on $\mathcal{H}_A, \mathcal{H}_B$ such that $\|U - U_A \otimes U_B\|_o < \epsilon_0$ if*

$$\max_j B_j(r_1, r_2, s_1, s_2, \theta_1, \theta_2) < \frac{\epsilon_0}{|t|}.$$

3.1.5 Conclusion and Discussion

In Section 3.1.2, we established a number of evaluation criteria for the separability of multipartite gates. These criteria demonstrate that all A except one $A \in \{A_i\}_{i=1}^n$ should belong to $\mathbb{R}I$ to be able to separate multipartite gate U where $\mathbf{H} = A_1 \otimes A_2 \otimes \ldots \otimes A_n$ and $U = \exp[i\mathbf{H}]$. As shown in Theorem 3.5, a random chosen multipartite gate cannot fundamentally satisfy this separability condition. We devoted Section 3.1.3 to the existence of the infimum of the gap between U and a local gate U_i and illustrated the search algorithm approaching to the given unitary using local gates. Moreover, as shown in the examples, a 2-qubit composite spin-$\frac{1}{2}$ system is a very practical tool for checking the criteria.

This work directly suggests that there are very few quantum computational tasks (quantum circuits) that can be automatically parallelised. Concurrent quantum programming and parallel quantum programming still needs to be researched for a greater understanding of quantum-specific features concerning the separability of quantum states, local operations, classical communication and even quantum networks.

3.2 Fast Permutation and its Application in Compiler Module

3.2.1 Introduction for Permutation

Permutation is a necessary aspect of designing a practical quantum compiler and quantum simulator, especially in multi-qubit scenarios. Permutations greatly influence the execution of the system in terms of both usability and reliability. In this section, we explore permutation and propose two permutation algorithms. The first is trivial, relying only on a permutation matrix that is completely described through permutations of its basis. Further, the matrix is used to transfer the original state matrix. The second is a special case of the first algorithm, where each term is treated as a combination of bases. Once all the relationships between the original basis and the new basis have been identified, the algorithm prevents a large permutation matrix from being stored.

As we have already introduced the QSI framework and its assumption, we will not repeat them here. The assumptions for the permutation module are the same. The following example demonstrates one permutation scenario with the QSI platform with multi-qubit operations.

Consider three subsystems, $q1, q2, q3 \in \mathcal{H}_2^1 \otimes \mathcal{H}_2^2 \otimes \mathcal{H}_2^3$, where $\mathcal{H}_2$ is a 2-dimensional Hilbert space as defined in Section 3.1.1, a code segment like:

```
U (q1,q3);
```

The code means $U \in \mathcal{H}_2^1 \otimes \mathcal{H}_2^3$. Both the simulator and compiler usually require U' which is the corresponding impact of U on the whole system. A trivial algorithm is that $U \otimes I$ for the whole system and a permutation is only performed on the last step to result in 'correct' order. For example, for the code segment $U\,(q1, q3)$, we have $U' = U^{13} \otimes I_2^2$ where $I_2^2 \in \mathcal{H}_2^2$. Then $U'^{123} = P^{132->123}(U'^{132})$ where $P^{132->123}$ is a permutation matrix.

In most case, we need to convert the unordered system into an ordered system. Let the ordered system is $\mathcal{H}^1, \mathcal{H}^2, \mathcal{H}^3 \ldots$ and denote $P^{132->123}$ to P^{132} by default.

3.2.2 General Algorithm for Permutation

A trivial algorithm for finding a permutation matrix P finds the basis transformation. The basic idea is that all transformations can be written as a matrix when the base of the transformation are fixed. The transformation of the matrix can be completely described by the transformation of basis. Thus, the description of the basis transformation could substitute the matrix as P.

The permutation matrix can be constructed using the following steps,

1. Define the set of standard orthogonal bases $S = \{|v_1\rangle, |v_2\rangle, \ldots, |v_n\rangle\}$.

2. Describe the transformation of the relationship between the basis of the target system and the defined basis. For example, for $P^{312} \in \mathcal{H}^1 \otimes \mathcal{H}^2 \otimes \mathcal{H}^3$, the transformation can be describe as $|v_1\rangle \to |v_3\rangle$, $|v_2\rangle \to |v_1\rangle$, $|v_3\rangle \to |v_2\rangle$.

3. Finally, the permutation matrix can be written as $P^{312} = |v_3\rangle\langle v_1| + |v_1\rangle\langle v_2| + |v_2\rangle\langle v_3|$.

A few of conclusions can be made from these steps:

1. In general, the permutation matrix is a sparse matrix. There is always a position taken by 1 in every column or row when standard bases are used. That means storing matrix is a low efficiency use of the computer's memory. Also, the permutation matrix is sometimes *enormous*, i.e., with m systems, of n dimensions, the permutation would be a $n^m \times n^m$ sparse matrix.

2. The permutation matrix is an elementary column transformation if we choose a standard orthogonal basis. This provides insight that the target matrix U could be adjusted directly without P.

3. The permutation process needs to be completely described as the basis whether or not changes are taken into consideration. In any large dimensional system, this would lead to an effortless step.

3.2.3 A Fast Permutation Algorithm Based on Fixed and Ordered Basis

Writing any matrix that represents a transformation implies the bases are fixed and ordered.

A fast permutation algorithm can be simply based on the basis transformation rather than generating the permutation matrix itself, thus saving space for the sparse matrix.

Suppose there is a pure quantum state v in $\mathcal{H}_8$, and thus the implication bases are $\{|1\rangle, |2\rangle, \ldots, |8\rangle\}$, i.e., $v = a_1|1\rangle + a_2|2\rangle + \ldots + a_8|8\rangle$. For convenience, we also suppose the current system in the memory where the unitary gate U works is $\mathcal{H}_2^1 \otimes \mathcal{H}_2^2 \otimes \mathcal{H}_2^3$, the state vector $v_{123} = a_1|000\rangle + a_2|001\rangle + \ldots + a_8|111\rangle$, i.e., $v = \sum_{i,j,k=0}^{i,j,k=1} a_{ijk}|ijk\rangle$.

Assuming the permutation matrix converted P^{312} to $|ijk\rangle \to |kij\rangle$, then $v_{312} = a_1|000\rangle + a_2|100\rangle + a_3|001\rangle + a_4|101\rangle + a_5|010\rangle + a_6|110\rangle + a_7|011\rangle + a_8|111\rangle$. As the matrix basis are often fixed and ordered by standard basis, the state v can be written with $v_{123} = a_1|000\rangle + a_3|001\rangle + a_5|010\rangle + a_7|011\rangle + a_2|100\rangle + a_4|101\rangle + a_6|110\rangle + a_8|111\rangle$, i.e., v can be written as $v = \{a_1, a_3, a_5, a_7, a_2, a_4, a_6, a_8\}$ with the standard basis. The transformation steps follow,

1. Find a set of ordered standard orthogonal bases . Express each item as a vector or a matrix using this basis, i.e., $A = \sum_{ij}\langle i|A|j\rangle |i\rangle\langle j|$.

2. Following to the permutation requirement P, a new basis should be constructed and the relationship $|i\rangle \to |i'\rangle$ will be stored.

3. Adjust the system into the formal order and find the coefficient $\langle i|A|j\rangle$ for each term.

Compared to a general permutation algorithm, we directly adjust the position of elements for each term to avoid using n^{2m} auxiliary space, where m is the number of systems and n is the dimension of each system for storing for permutation matrix. In fact, as previously mentioned, the permutation matrix is a kind of elementary column transformation, where the fast algorithm considers the order of the bases rather than the matrix. In a 2-dimensional (qubit) system ($\dim(\mathcal{H}^1) = \dim(\mathcal{H}^2) = \ldots = \dim(\mathcal{H}^n) = 2$), we may find that the binary expression of the basis is just the system combination, e.g., in 3-qubit systems, $|1\rangle$ might be coded as $|000\rangle$, $|2\rangle$ might be coded as $|001\rangle$ and so on. Every position is enough for one system and thus, the basis transfer would be expressed as a byte transfer. Moreover, in classical computation systems, the binary computation is always faster than other operations.

3.3 Termination Analysis Module

3.3.1 Introduction for Termination Module

Termination analysis is an essential part in programming – especially quantum programming. Measurement, entanglement, and even superposition are the foundations of bizarre behaviour in quantum programs. In any practical system with a loop structure, termination information is always useful that the compiler with explanation type may work on infinite steps in executing quantum programs. Even if termination analysis despiting the input state, it can provide an indication of the general termination scenarios in a quantum program. Hence, in this section, we review the definitions and theorems of quantum termination, followed by the two algorithms we developed for analysing termination. Lastly, we provide an example of termination – Qloop – to display the efficiency of the algorithms.

3.3.2 Definitions and Theorems

A general quantum loop can be shorten to the form,

$$\textbf{while}(M[\bar{q}] \in X)\{\bar{q} = \mathcal{E}(\bar{q});\} \quad ,$$

where $\bar{q}$ is a quantum variable, M is a measurement that satisfies $M_0^\dagger M_0 + M_1^\dagger M_1 = I$, X is the set of measurement results, $\mathcal{E}$ is a complete positive and trace preserving (CPTP) map and a denotation for a **while** sub-program, and $\mathcal{E}(\cdot) = \sum_i E_i(\cdot)E_i^\dagger$.

We denote $\mathcal{G}(\rho) = \sum_i E_i M_1 \rho M_1^\dagger E_i^\dagger$, G in below text is the matrix representation of $\mathcal{G}$. That is $G = \sum_i (E_i M_1) \otimes (E_i M_1)^*$. $|\Phi\rangle$ is the maximally entangled state.

Definition 3.15. (I) We say that a program terminates if the non-termination probability in the nth step

$$p_n^{NT} = tr([\mathcal{E}_1 \circ (\mathcal{E} \circ \mathcal{E}_1)^{n-1}](\rho_0)) = 0$$

for some positive integer n.

(II) We say that the program almost-surely terminates if $\lim_{n\to\infty} p_n^{NT} = 0$, where p_n^{NT} is as in finite termination definition.

Prior to outlining the full results, some elementary results are provided as follows. The first lemma is obvious.

Lemma 3.16. *(I) The program terminates if, and only, if* $G^n(\rho_0 \otimes I)|\Phi\rangle = 0$ *for some integer* $n \geq 0$*;*

(II) The program almost-surely terminates if, and only if, $\lim_{n\to\infty} G^n(\rho_0 \otimes I)|\Phi\rangle = 0$.

This leads to the following important lemma in termination analysis [12].

Lemma 3.17. *(I) The program terminates if, and only if,* $G^n|\Phi\rangle = 0$ *for some integer* $n \geq 0$,

(II) The program almost-surely terminates if $\lim_{n\to\infty} G^n|\Phi\rangle = 0$.

However, it is not easy to check whether a quantum program terminates or almost-surely terminates with the above definition or lemmas. But, by splitting the vectors, these assertions can be converted into a computable proposition that assesses the correct termination type through some operational algorithms (see Section 3).

Recall that G is the matrix representation of the quantum program $\mathcal{G}$. Therefore, the eigenvalues of G will have norms of 1 or less. Denoted as E_λ, the eigensubspace of G's eigenvalue λ gives the space decomposition $H = E_0 \oplus E_{(0,1)} \oplus E_1$, where $E_{(0,1)} = \bigvee_{|\lambda|\in(0,1)} E_\lambda$ and $E_1 = \bigvee_{|\lambda|=1} E_\lambda$.

Theorem 3.18. *Suppose the Jordan decomposition of G is* $G = SJ(G)S^{-1}$ *with an invertible* S *and* $S^{-1}|\phi\rangle = \begin{pmatrix} |u\rangle \\ |v\rangle \\ |w\rangle \end{pmatrix}$ *based on the space decomposition* $H = E_0 \oplus E_{(0,1)} \oplus E_1$*. Then the following statements hold true:*

(I) If $|v\rangle$ *and* $|w\rangle$ *are both* 0*, the program* G *is terminating.*

(II) If $|w\rangle$ *is nonzero, the program* G *is non-terminating.*

(III) The program G *is almost-surely terminating if and only if* $|w\rangle$ *is* 0.

Proof. The proofing technique is similar to that of Proposition 5.1 in the article [12].

Suppose $G = SJ(G)S^{-1}$, where $J(G) = diag(J_{k_1}(\lambda_1), J_{k_2}(\lambda_2), \ldots, J_{k_l}(\lambda_l))$. All of the eigenvalues $0 \leq |\lambda_i| \leq 1$ [59]. Without loss of generality, we can assume $\lambda_1 = \ldots = \lambda_s = 0$,

$0 < |\lambda_{s+1}| \leq \ldots \leq |\lambda_t| < 1$ and $|\lambda_{t+1}| = \ldots = |\lambda_l| = 1$. As $G^n = SJ(G)^n S^{-1}$, $G^n |\Phi\rangle = SJ(G)^n S^{-1} |\Phi\rangle$. Based on the space decomposition $H = E_0 \oplus E_{(0,1)} \oplus E_1$, we write

$$J(G) = \begin{pmatrix} A & 0 & 0 \\ 0 & B & 0 \\ 0 & 0 & C \end{pmatrix} \quad \texttt{and} \quad S^{-1} |\Phi\rangle = \begin{pmatrix} |u\rangle \\ |v\rangle \\ |w\rangle \end{pmatrix},$$

where

$$A = diag(J_{k_1}(\lambda_1), \ldots, J_{k_s}(\lambda_s)), B = diag(J_{k_{s+1}}(\lambda_{s+1}), \ldots, J_{k_t}(\lambda_t)),$$

$$C = diag(J_{k_{t+1}}(\lambda_{t+1}), \ldots, J_{k_l}(\lambda_l)),$$

$|u\rangle$ is an s-dimensional vector, $|v\rangle$ is a $(t-s)$-dimensional vector, $|w\rangle$ is a $(l-t)$-dimensional vector and $S^{-1} |\phi\rangle$ is a combination of $|u\rangle$,$|v\rangle$ and $|w\rangle$. It follows that $J(G)^n S^{-1} |\Phi\rangle$ can be written as

$$J(G)^n S^{-1} |\Phi\rangle = \begin{pmatrix} A^n |u\rangle \\ B^n |v\rangle \\ C^n |w\rangle \end{pmatrix}. \qquad (*)$$

Checking (I) first, note that $\lambda_1, \ldots, \lambda_s = 0$ and $\lambda_{s+1}, \ldots, \lambda_l \neq 0$, $J_{k_{s+1}}(\lambda_{s+1}), \ldots, J_{k_l}(\lambda_l)$ are non-singular in Eq. (*). If $|v\rangle$ and $|w\rangle$ are both 0, then according to Eq. (*),

$$J(G)^n S^{-1} |\Phi\rangle = \begin{pmatrix} A^n |u\rangle \\ 0 \\ 0 \end{pmatrix}.$$

Now with an n greater than the maximal size of the Jordan blocks $J_{k_j}(0)$, $j = 1, 2, ..., s$, $A^n = 0$. It follows that $J(G)^n S^{-1} |\Phi\rangle = 0$. Thus $G^n|\Phi\rangle = 0$ and, therefore, G is terminating and (I) is proved. Moreover, if $|w\rangle \neq 0$, then $J(G)^n S^{-1} |\Phi\rangle \neq 0$ for any n since C is non-singular. So $G^n|\Phi\rangle \neq 0$ for any n, and (II) is proved.

Turning to (III), note that always $\lim_{n\to\infty} A^n = 0$ and $\lim_{n\to\infty} B^n = 0$. Further, $|w\rangle$ is zero if and only if $C^n|w\rangle = 0$ since C is non-singular. It follows that $|w\rangle$ is 0 if, and only if,

$$\lim_{n\to\infty} J(G)^n S^{-1}|\Phi\rangle = \begin{pmatrix} A^n|u\rangle \\ B^n|v\rangle \\ C^n|w\rangle \end{pmatrix} = 0\,,$$

Therefore, (III) holds true. □

The following proposition concerns the output matrix representation of terminating program and almost-surely terminating programs which can be used for nested and concurrent program analysis.

Proposition 3.19. *Suppose that G is defined in Theorem 3.18 and $N_0 = E_0 \otimes E_0^*$.*

(I) If the program G terminates after k-steps, then the matrix representation of the program becomes $G' = \sum_{n=0}^{k-1} N_0 G^n$.

(II) If the program G almost-surely terminates, then the matrix representation of the program becomes $G' = \sum_{n=0}^{\infty} N_0 G^n = N_0(I-G)^{-1}$.

Concluding Proposition 5.1 in the article [12], we can draw another two sufficient conditions.

Proposition 3.20. *(I) If $G^d|\Phi\rangle = 0$, where d is the rank of G, then the program G terminates.*

(II) If $|\Phi\rangle$ is orthogonal to all eigenvectors of $G^\dagger$ corresponding to eigenvalues with module 1, then the program G almost-surely terminates.

Theorem 3.21 is the lower bound of the terminating steps, i.e., for any terminating program, when the executing step $n_{any} \geq n_{\inf}$, the program must terminate. Proposition 3.19 determines the fixed output matrix representation of a program.

Theorem 3.21. *If the program G terminates, then the lower bound of the terminating steps*

$$n_{\inf} = \max_{J_{k_1},\ldots,J_{k_s}} \min\{k'_j, n'_j\}\,,$$

where k'_j is the size of the jth Jordan block J_{k_j} of G with $0 \le j \le s$, n'_j is the number of non-zero elements in the vector block corresponding to the jth Jordan block of G.

Proof. Let $B_1 = J_{k_1}(0), \dots, B_s = J_{k_s}(0)$ and $S^{-1}|\Phi\rangle = |y\rangle = (|y_1\rangle, \dots, |y_s\rangle)^T$. Therefore, G is terminating if and only if

$$B|y\rangle = \begin{pmatrix} B_1 & & \\ & \ddots & \\ & & B_s \end{pmatrix} \begin{pmatrix} |y_1\rangle \\ \vdots \\ |y_s\rangle \end{pmatrix} = \begin{pmatrix} B_1|y_1\rangle \\ \vdots \\ B_s|y_s\rangle \end{pmatrix}.$$

Further, if the program is terminating, then there is an n such that

$$B^n|y\rangle = \begin{pmatrix} B_1^n|y_1\rangle \\ \vdots \\ B_s^n|y_s\rangle \end{pmatrix} = 0. \qquad (\dagger)$$

To complete the proof, we only need to show that for each block under the condition $\min\{k'_j, n'_j\}$, every block must ensure $B_j^{n'_j}|y_j\rangle = 0$. $n_{\text{inf}} = \max\min\{k'_j, n'_j\}$ is enough to ensure all blocks are 0, i.e., $B|y\rangle = 0$.

Now, with Eq.($\dagger$), $B_1^n|y_1\rangle = 0, \dots, B_s^n|y_s\rangle = 0$ for each block. It is clear that after k'_j times with the size of k'_j, $B_j^{k'_j} = 0$ where $0 \le j \le s$ as follows

$$B_j = \begin{pmatrix} 0 & 1 & & & \\ & 0 & 1 & & \\ & & \ddots & & \\ & & & 0 & 1 \\ & & & & 0 \end{pmatrix}, \quad B_j^{k'_j-1} = \begin{pmatrix} 0 & 0 & & 0 & 1 \\ & 0 & 0 & & 0 \\ & & \ddots & & \\ & & & 0 & 0 \\ & & & & 0 \end{pmatrix}, \quad B_j^{k'_j} = \begin{pmatrix} 0 & 0 & & 0 & 0 \\ & 0 & 0 & & 0 \\ & & \ddots & & \\ & & & 0 & 0 \\ & & & & 0 \end{pmatrix}.$$

However, suppose $|y\rangle = (y_1, y_2, \dots, y_j)^T$ for each block $J_{k_j}(0)$. Here we have

$$B_j^{n'_j}|y_j\rangle = B_j^{n'_j-1}B_j|y_j\rangle = B_j^{n'_j-1}\begin{pmatrix} 0 & 1 & & & \\ & 0 & 1 & & \\ & & \ddots & & \\ & & & 0 & 1 \\ & & & & 0 \end{pmatrix}\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_j \end{pmatrix} = B_j^{n'_j-1}\begin{pmatrix} y_2 \\ y_3 \\ \vdots \\ y_j \\ 0 \end{pmatrix}$$
$$= B_j^{n'_j-1}\left|y_j^1\right\rangle = B_j^{n'_j-2}\left|y_j^2\right\rangle = \dots = \left|y_j^n\right\rangle .$$

If the first n'_j number of $|y_j\rangle = (y_1, y_2, \dots, y_j)^T$ is not 0, then the first $n'_j - 1$ number of $\left|y_j^1\right\rangle = (y_2, y_3, \dots, y_j)^T$ is not 0. After n'_j times, $\left|y_j^{n'_j}\right\rangle = 0$.

Thus, we can conclude that for each block under the condition $\min\{k'_j, n'_j\}$, it ensures every block $B_j^{n'_j}|y_j\rangle = 0$. $n_{\text{inf}} = \max\min\{k'_j, n'_j\}$ is enough to ensure all blocks are 0, i.e., $B|y\rangle = 0$. This completes the proof. □

3.3.3 Algorithms for Termination Analysis

In Section 3.3.2, we outlined several useful theorems and propositions for termination analysis. Here, we provide two computational algorithms for termination analysis. The first algorithm considers Jordan decomposition and only has efficient complicity from a theoretical perspective. The second algorithm is more flexible and efficient theoretically and practically.

Algorithm 3.1 (Algo 3.1) re-illustrates the process we used to prove Theorem 3.18. The input of the function *CheckTermination* is the matrix representation of $\mathcal{G}$. As the termination is defined for programs that exclude an input quantum state, our concern is programs with measurement, M_0 or M_1 and loop body $\mathcal{E}$ and $\mathcal{G}$ is enough to check termination. The outputs of Algorithm 3.1 are: *State*, which is byte represents termination, almost-surely termination or non-termination and *Instead*, which is also a matrix representation for substituting the program segment as a new super-operator $\sum_{n=0}^{k-1} N_0 G^n$ where N_0 is the matrix representation of measurement operator M_0. It can be easily inferred that for a k-step termination program the first $k-1$ steps can be treated as G^n, and the last step must be M_0 which shows the after-measurement effect. Lines 10-12 are for cases

Algorithm 3.1 Check Terminating Algorithm with Jordan Decomposition

1: **function** [STATE,INSTEAD]=CHECKTERMINATION(G)
2: $|\phi\rangle \leftarrow MaxEntangledState$ where $d(|\phi\rangle) = d(G)$
3: $[J, S] \leftarrow Jordan(G)$ s.t. $G = SJS^{-1}$
4: $|\phi\rangle' \leftarrow S^{-1} |\phi\rangle$
5: $\begin{pmatrix} |u\rangle \\ |v\rangle \\ |w\rangle \end{pmatrix} \leftarrow |\phi\rangle'$ where $d(|u\rangle) =$ *number of eignvalues with value* 0, $d(|w\rangle) =$ *number of eignvalues with module* 1, $d(|v\rangle) = d(|\phi\rangle') - d(|u\rangle) - d(|w\rangle)$.
6: **if** $|v\rangle = 0$ and $|w\rangle = 0$ **then**
7: k← CalSteps(G)
8: **return** State←Finite Termination
9: **return** Instead←$\sum_{n=0}^{k-1} N_0 G^n$
10: **if** $|w\rangle$=0 **then**
11: **return** State←Almost-surely termination
12: **return** Instead←$\sum_{n=0}^{\infty} N_0 G^n = N_0(I - G)^{-1}$
13: **return** State←Non-termination or Unknown
14: **return** Instead←$NULL$

Algorithm 3.2 Check Terminating Algorithm without Jordan Decomposition

1: **function** [STATE,INSTEAD]=CHECKTERMINATION(G)
2: $|\phi\rangle \leftarrow MaxEntangledState$
3: $[V, D] \leftarrow$ *eig*($G^\dagger$)
4: $D_1 \leftarrow D$ *where their eigenvalues with module* 1
5: **if** $|\phi\rangle \perp D_1$ **then**
6: **return** State←Almost-surely termination
7: **return** Instead←$\sum_{n=0}^{\infty} N_0 G^n = N_0(I - G)^{-1}$
8: **if** $G^d |\phi\rangle = 0$ where d is the dimension of G **then**
9: k← CalSteps(G)
10: **return** State←Finite termination
11: **return** Instead←$\sum_{n=0}^{k-1} N_0 G^n$
12: **return** State←Non-termination
13: **return** Instead←$NULL$

where the program almost-surely terminates. According to Proposition 3.19, the infinite steps effect can be substituted by $N_0(I-G)^{-1}$. The last scenario is the non-termination or an unknown program. In Algorithm 3.1, the most significant barrier for implementation is the Jordan decomposition on Line 3. Even though Jordan decomposition can achieve a time complexity of $\mathcal{O}(n^3)$ [60]. The Jordan decomposition process is often unacceptable as the reason of using symbolic calculation for Jordan form to avoid the discontinuous property of matrix entries.

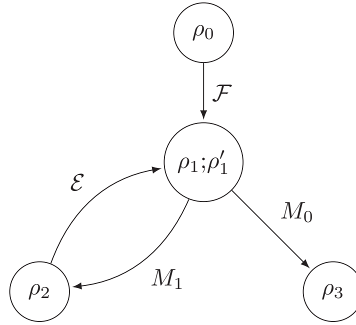


FIGURE 3.1: Flowchart of a Qloop

Compared to Algorithm 3.1, Algorithm 3.2 can achieve a far more reasonable time complexity because it checks for termination without relying on a Jordan decomposition. Input G and the outputs *State* and *Instead* are defined as per Algorithm 3.1. Line 2 generates a maximally entangled state. The key difference is Lines 03-Line 05 where we focus on Proposition 3.20 and use the eigenvector set of $G^\dagger$ to check the termination . This avoids the need for a Jordan decomposition. It is worth noting that even though we have proven the two algorithms to be equivalent, the process of calculating the eigenvalues and eigenvectors is a theoretical implementation of a polynomial process and other than the time complexity of a Jordan decomposition.

One of the necessary steps in both Algorithm 3.1 and Algorithm 3.2 is *CalSteps*, which decides k to output *Instead*. k can be decided easily with Theorem 3.21.

3.3.4 Termination Examples- Qloop

A Qloop is a simplified version of a quantum random walk - a powerful quantum computation algorithm. The flow chart of a Qloop is shown in Figure 3.1. Unlike the 1-dimensional quantum walk, a Qloop does not include a direction translation operator. The direction Left or Right of ρ_1 is only determined by the measurement result. If the result is 0, the status moves to ρ_3, whereas if the result is 1, the status moves to ρ_2. After an operation $\mathcal{E}$ in a single quantum or a sub-program, the current status would move to ρ_1' and a measurement M would be projected on to it. The new measurement result may lead to a Left or a Right route. The following two termination examples show two code-segments of a fixed value with a Qloop.

Qloop 3.3 Qloop with Hadamard gate as a sub-program

Require: $\rho = \rho_1 = |1\rangle$, $M = |0\rangle\langle 0|$, $HGate = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$.
Ensure: ρ

```
1: function QLOOP(ρ)
2:     while M(ρ) do
3:         HGate(ρ)
4:     return ρ
```

Qloop 3.4 Qloop with bit flip gate as a sub-program

Require: $\rho = \rho_1 = |1\rangle$, $M = |0\rangle\langle 0|$, $XGate = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.
Ensure: ρ

```
1: function QLOOP(ρ)
2:     while M(ρ) do
3:         XGate(ρ)
4:     return ρ
```

Although the difference between Qloops 3.3 and 3.4 is only a small section of the sub-program, each program has completely different behaviours. The quantum state ρ after the first measurement is equivalent to $|1\rangle$, but after the sub-programs it would be $\rho = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $\rho = |0\rangle$, respectively. This could lead to different results when ρ is measured again. For instance, in Qloop 3.3, which is an almost-surely terminating program, the program would immediately terminate if the measurement result is 0 with a probability of 50%. However, if the result is 1 with a probability of 50%, the subprogram would continue executing and become $\rho = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ in the next loop. In Qloop 3.4, which is a terminating program, the program would immediately terminate 100% of the time.

We implemented Algorithms 3.1 and 3.2 in MATLAB and tested the speed of both with: a Qloop case; a Hadamard gate; and a bit-flip gate, respectively. Figure 3.2 shows the results.

3.3.5 Conclusion and Discussion

In Section 3.3, we extended the results in the article [12] (see also, Chapter 5 in the book [2]). Two computational algorithms for quantum **while**-language were presented,

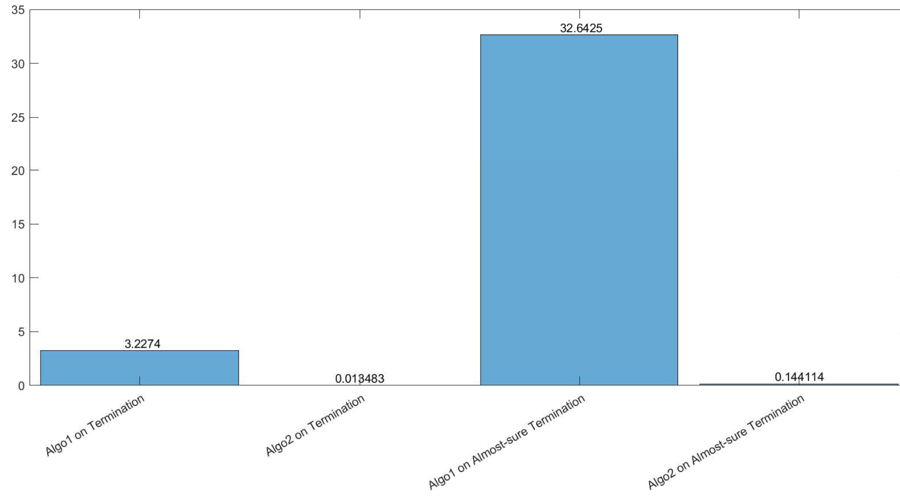


FIGURE 3.2: Algorithm 3.1 and Algorithm 3.2 were programmed in MATLAB 2017b. The experiment was conducted on a PC with Core i7 processor and 16 GB memory. Qloops 3.3 and 3.4 were executed 1000 times with one setting and statistics over the total running time. The figure clearly illustrates Algorithm 3.2 which does not include a Jordan decomposition accelerates termination analysis 300 times faster than Algorithm 3.1 in both termination and almost-surely termination scenario.

along with the lower bound of the terminating steps for outputting a matrix representation with terminating quantum programs. The termination analysis experiments illustrate that the algorithm without a Jordan decomposition can significantly speed up the calculation process. Moreover, both algorithms have been integrated into QSI 's Termination Module [43] to provide extra information about the characteristics of quantum programs for the quantum compiler with some practical success.

3.4 Quantum Control Module

3.4.1 Introduction

QuGCL is a quantum programming language with quantum control. It is a high-level language which can describe quantum algorithms efficiently. For connecting this language to quantum circuits model, we exploited the counterparts of syntax and implementation. Here, we establish that the $[\![\mathbf{QIF}]\!]$ clause is a quantum multiplexor (QMUX) and can be implemented with the help of construction arbitrary 2-dimensional controlled-unitary using Z-Y rotation using LIQU$i|\rangle$. Following this work, it is possible to construct quantum circuits automatically without knowing every detail of unitary transformation is possible.

3.4.2 QMUX Preliminaries

3.4.2.1 QMUX

In classical computation, the simplest multiplexor is a conditional - It is best described as the "**if** ... **then** ... **else** ..." construction , i.e., if a condition is true, then perform the specified action, "**else**" the condition is false. Quantum conditionals are a quantum analog of the classical formed by replacing the condition after "**if**" with a qubit, i.e., true and false is replaced with the basis states $|1\rangle$ and $|0\rangle$.

Definition 3.22. Let $\bar{p} = p_1, \ldots, p_m$ and $\bar{q} = q_1, \ldots, q_n$ be quantum registers, and for each $x \in \{0,1\}^m$, let $C_x = U_x[\bar{q}]$ be a quantum gate. The quantum multiplexor (QMUX) "$\bigoplus C_x$" is a gate on $m + n$ qubits $\bar{p}, \bar{q}$, having the first m qubits $\bar{p}$ as the selected qubits and the remaining n qubits $\bar{q}$ as the data qubits. A QMUX preserves any state of the selected qubits, and performs a unitary transformation on the data qubits, depending on their state:

$$(\bigoplus_x C_x)\,|t\rangle\,|\psi\rangle = |t\rangle\,U_t\,|\psi\rangle$$

for any $t \in \{0,1\}^m$ and $|\psi\rangle \in \mathcal{H}_{\bar{q}}$

The matrix representation of a QMUX is a diagonal:

$$\bigoplus_x C_x = \bigoplus_{x=0}^{2^m-1} U_x = \begin{bmatrix} U_0 & & & \\ & U_1 & & \\ & & \ddots & \\ & & & U_{2^m-1} \end{bmatrix}$$

When there is a single selected qubit, the matrix of the QMUX is a block diagonal $U = \begin{bmatrix} U_0 & \\ & U_1 \end{bmatrix}$. We denote this as $U = U_1 \oplus U_2$. The multiplexor will apply U_0 or U_1 to the data qubits according to the select qubit, i.e.,

$$\mathbf{qif}\,|0\rangle\ \mathbf{then}\,U_0\,\mathbf{else}\,U_1.$$

Shende [47] was the first to show a 2-qubit multiplexor can be decomposed. Subsequently, he proved any qubit multiplexor can be decomposed.

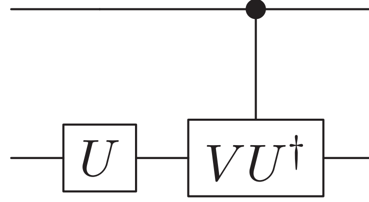


FIGURE 3.3: Decomposition of QMUX

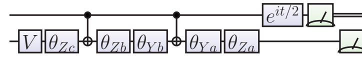


FIGURE 3.4: QMUX gate rotation decomposition, $\theta_{Zc} = R_z(\frac{\theta-\beta}{2})$, $\theta_{Zb} = R_z(\frac{\theta+\beta}{2})$, $\theta_{Yb} = R_z(\frac{\gamma}{2})$,$\theta_{Ya} = R_z(\frac{\gamma}{2})$, $\theta_{Za} = R_z(\beta)$.

Lemma 3.23. *For $\forall U'$ where $U' = U \oplus V$, U and V are one-qubit gates, there exists a decomposition $U' = U \oplus VU^{\dagger}U$.*

The last theorem can be implemented as circuit model as shown in Figure 3.3.

The idea is to first unconditionally apply U on the second qubit, and then apply an $A = VU^{\dagger}$ condition on the first significant qubit. Decomposition for such control operator is well known [23]. If we write $A = e^{i\alpha}R_z(\beta)R_y(\gamma)R_z(\theta)$, then $U \oplus V$ can be implemented with the following two lemmas:

Lemma 3.24. *A 2-dimensional unitary matrix can be written as*

$$\begin{aligned} U &= e^{i\alpha}\begin{bmatrix} e^{-i\beta/2} & 0 \\ 0 & e^{i\beta/2} \end{bmatrix}\begin{bmatrix} \cos\delta/2 & -\sin\delta/2 \\ \sin\delta/2 & \cos\delta/2 \end{bmatrix}\begin{bmatrix} e^{-i\gamma/2} & 0 \\ 0 & e^{i\gamma/2} \end{bmatrix} \\ &= e^{i\alpha}R_z(\beta)R_y(\gamma)R_z(\delta). \end{aligned}$$

Lemma 3.25. *For any special unitary matrix W ($W \in \mathrm{SU}(2)$),there exist matrices A, B and $C \in \mathrm{SU}(2)$ such that $A \cdot B \cdot C = I$ and $A \cdot \sigma_x \cdot B \cdot \sigma_x \cdot C = W$ where $A \equiv R_z(\beta)R_y(\gamma/2)$, $B \equiv R_y(-\gamma/2)R_z(-(\delta+\beta)/2)$ and $C \equiv R_z((\delta-\beta)/2)$ and α, β, γ, δ are in Equation 3.24.*

With Lemmas 3.24 and 3.25, any arbitrary controlled-unitary operator can be constructed as Figure 3.5 shows.

Thus, the problem of producing two-qubit quantum circuits with "**if**" syntax becomesa problem of producing arbitrary controlled-unitary operators automatically.

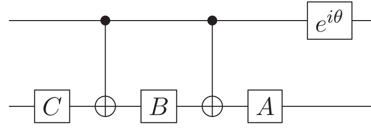


FIGURE 3.5: Decomposition of arbitrary controlled unitary gate

LIQU$i|\rangle$ [18] offers a wide range of operations that can be performed. For physical modelling, the rotation gates R_x, R_y, R_z are supplied as inherent gates, but they cannot automatically generate a universal unitary gate. Users must provide a matrix corresponding to the unitary gate before compilation. The ability to automatically generate arbitrary unitary gates is very important because we rarely have the matrix details for an $A = VU^\dagger$ in a QMUX, nor do we want to be concerned with such detail when simulating a unitary gate in a large quantum network. Therefore, constructing an arbitrary controlled-unitary matrix is essential to the simulation process.

3.4.3 Solving Equations

Given a 2-dimensional unitary gate U, where

$$U = \begin{bmatrix} a & b \\ c & d \end{bmatrix},$$

the goal is to derive the corresponding α, β, γ, δ.

The first inclination is to find the answers by solving equations, i.e.,

$$\begin{aligned} e^{-i\varphi_1} \cos \delta/2 &= a\,, \\ -e^{-i\varphi_2} \sin \delta/2 &= b\,, \\ e^{i\varphi_2} \sin \delta/2 &= c\,, \\ e^{i\varphi_1} \cos \delta/2 &= d\,. \end{aligned}$$

However, even when using an out-of-line procedure, MATLAB, or Math.net (a math library in .net Framework), solving these transcendental equations is hard. “Fsolve” and “Fzero” functions, which work as inherent functions in MATLAB can solve the problem using a non-fixed probability while other unitary matrices can be decomposed at random.

Fortunately, we have found another method to solve this issue.

$$\begin{bmatrix} e^{i(\alpha-\beta/2-\delta/2)}\cos\gamma/2 & -e^{i(\alpha-\beta/2+\delta/2)}\sin\gamma/2 \\ e^{i(\alpha+\beta/2-\delta/2)}\sin\gamma/2 & e^{i(\alpha+\beta/2+\delta/2)}\cos\gamma/2 \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

When all a, b, c, d are not zero,

$$\Rightarrow \begin{cases} e^{i\beta}\tan\frac{\gamma}{2} = \frac{c}{a} \\ -e^{-i\beta}\tan\frac{\gamma}{2} = \frac{b}{d} \\ -e^{i\delta}\tan\frac{\gamma}{2} = \frac{b}{a} \\ e^{-i\delta}\tan\frac{\gamma}{2} = \frac{c}{d} \\ e^{i2\alpha} = \det(U) \end{cases}.$$

We can implement that,

$$e^{i(2\beta)} = -\frac{cd}{ab} \Rightarrow \beta = \text{angle}(-\frac{cd}{ab})/2 + 2k\pi,$$

$$e^{i(2\delta)} = -\frac{bd}{ac} \Rightarrow \delta = \text{angle}(-\frac{bd}{ac})/2 + 2k\pi,$$

$$\tan^2\frac{\gamma}{2} = -\frac{cb}{ad} \Rightarrow \gamma = 2\arctan(\sqrt{-\frac{cb}{ad}}) + \pi,$$

$$\det U = e^{i2\alpha} \Rightarrow \alpha = \text{angle}(\det U)/2,$$

where $k \in \mathbb{Z}$ and angle(z) in $x + yi$ is defined as,

$$\begin{cases} \arctan(\frac{y}{x}) & \text{if } x > 0 \\ \arctan(\frac{y}{x}) + \pi & \text{if } x < 0 \text{ and } y \geq 0 \\ \arctan(\frac{y}{x}) - \pi & \text{if } x < 0 \text{ and } y < 0 \\ \frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0 \\ \text{indeterminate} & \text{if } x = 0 \text{ and } y = 0. \end{cases}$$

When $a = 0$, then $d = 0$ and $b, c \neq 0$, i.e.

$$U = \begin{bmatrix} 0 & b \\ c & 0 \end{bmatrix},$$

Since $\|b\|^2 = \|c\|^2 = 1$, then $\gamma = \pi$. $\det U = -bc = e^{2i\alpha}$ and $\alpha + \beta/2 - \delta/2 = \text{angle}(c)$, $\alpha + \beta/2 = \text{angle}(c) + \delta/2$. We can always assume $\delta = 0$, $\beta = 2(\text{angle}(c) - \alpha)$.

When $b = 0$, then $c = 0$ and $a, d \neq 0$, i.e.,

$$U = \begin{bmatrix} a & 0 \\ 0 & d \end{bmatrix},$$

Since $\|a\|^2 = \|d\|^2 = 1$, then $\gamma = 0$. $\det U = ac = e^{2i\alpha}$ and $\alpha - \beta/2 - \delta/2 = \text{angle}(a)$, $\alpha - \beta/2 = \text{angle}(a) + \delta/2$. We can always assume $\delta = 0$, $\beta = 2(\alpha - \text{angle}(a))$.

This procedure can be implemented easily in MATLAB and Math.net to find the decomposition. The results of the equations can be transferred as parameters (α, β, γ, δ) to reconstruct the controlled-unitary matrix with LIQU$i|\rangle$.

Chapter 4

Experiments

4.1 Experiments with QSI

4.1.1 Configuration

Download the `QSI_Online_Setup` or `QSI_Offline_Basic_Setup` from `http://www.qcompiler.com`, and execute the setup package. The project will be installed on desktop by default. If Visual Studio and MATLAB are already installed, the user can skip the following instructions and launch `QSI.sln` immediately.

If Visual Studio and MATLAB are not installed, follow the additional setup procedure below.

1. Double-click `vs_community.exe`. We recommend the commercially licensed Enterprise version of Visual Studio because it includes many productive toolkits, the Community version is also acceptable. Note that the DGML component in Visual Studio Community is not able to draw one qubit quantum circuits.

2. Choose `.NET desktop development` (Figure 4.1) and press the **Install** button. This may take about 15 minutes to complete depending on the user network.

3. Double-click `Matlab_Runtime.exe` (Figure 4.2). Install the MATLAB runtime components from the Internet. This may take about 15 minutes depending on the user network.

4. Open `QSI.sln` from the `QSIMain` folder with Visual Studio 2017 Community.

5. Menu → Debug → Start Debugging.

Screen captures of the procedure follow.

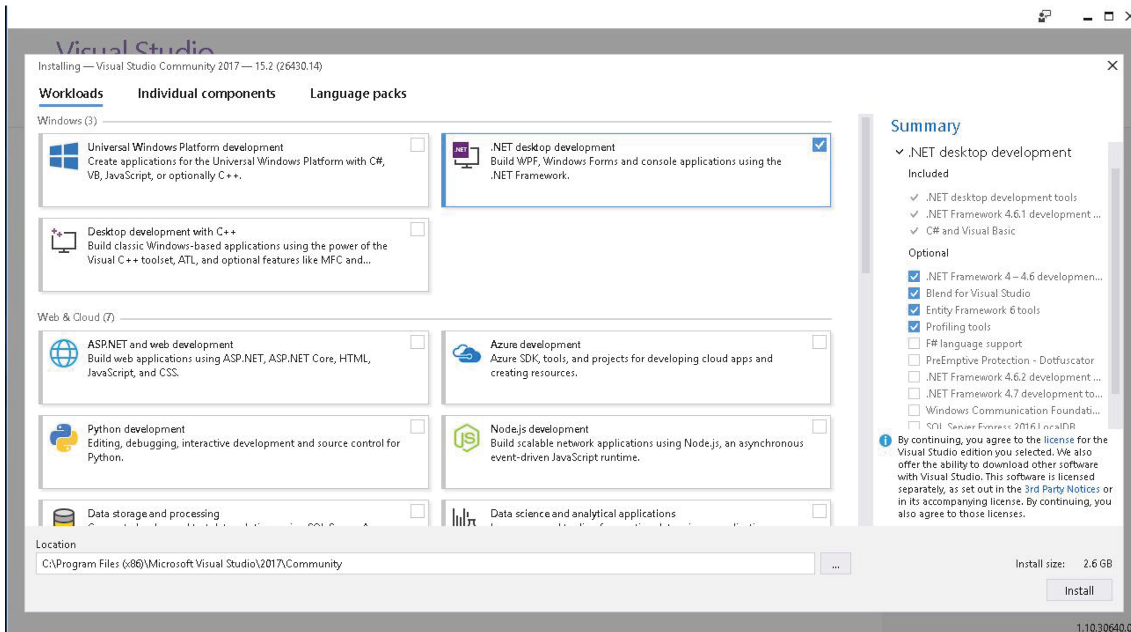


FIGURE 4.1: .NET desktop development

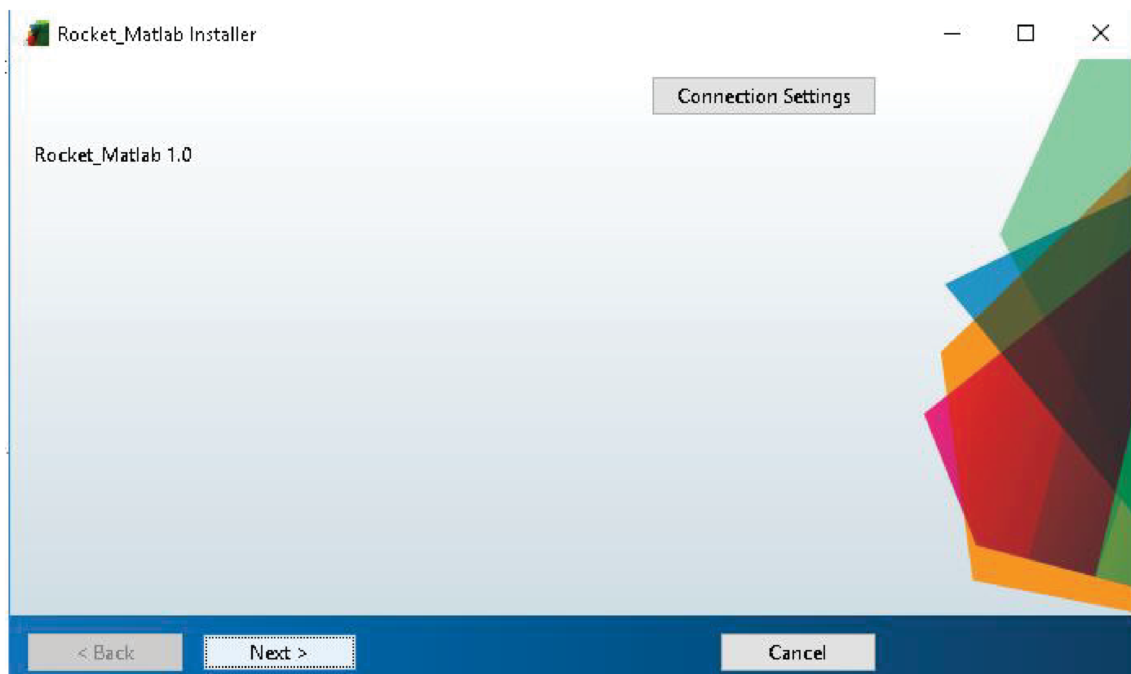


FIGURE 4.2: MATLAB support assembly

4.1.2 Experiments

The following subsections describe and provide an analysis of each example experiments. For user convenience, the coding for all the examples has been assembled in Project "UnitTest" under the Root Solution, as shown in Figure 4.3.

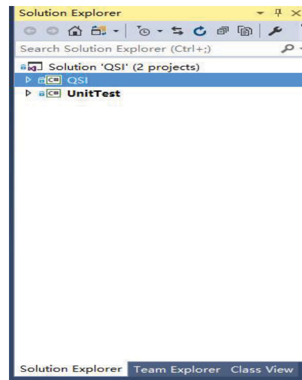


FIGURE 4.3: Examples in Project "UnitTest"

4.1.2.1 CNOT gate showcase-"Option 1"

Input and output

1. Enter the number 1 to start the CNOT Gate Showcase.
2. The console should display "The experiment begins...". The experiment will execute about 1000 times and show the statistic result as Figure 4.4.

Behind the console This is a very simple and standard example of QSI . This code segment is so trivial that it is listed here directly for reference.

```
C:\Users\klinu\Source\Repos\qcis\TestudoV1\UnitTest\bin\Debug\UnitTest.exe
================================================================================
=
          Welcome to UTS:QSI Quantum Programming Environment!
          Version: Build 04.24.17 ( .net Framework: 4.6.2 )
          Stage: Alpha

================================================================================
=

        1:  CNOT gate. Inputs are |+> and |0>. Run 1000 times.
        2:  Termination analysis, Example 1 (xGate).
        3:  Termination analysis, Example 2 (hGate).
        4:  Simple BB84.
        41: The multi-clients protocol for simple BB84, without statistics.
        42: The BB84 protocol with channel. Bit flip channel, p=0.1.
        43: The BB84 with statistics and channel.
        5:  Quantum Teleportation with QASM.
        6:  Quantum Google PageRank.
        7:  Grover Search, the oracle has been set answer the position 3.
        71: Standard Grover Search.
        72: Automatic toolkits Grover, search 2^4, answer from 0-15.(DEBUG close)
        73: Search multi-objects Grover. It is WRONG.
        8:  A comprehensive Quantum Teleporation. Termination and Decomposition.

        Press <Enter> to exit...

        Please select a case number:    1
The experiment begins.....
Reg 1 : 0's: 482,1's: 518
Reg 2 : 0's: 482,1's: 518
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

FIGURE 4.4: CNOT experiment

```
class TestQuantMulti0 : QEnv
{
public Reg r1 = new Reg("r1");
public Reg r2 = new Reg("r2");
public Quantum q1 = MakeDensityOperator(2, "{[0.5 0.5;0.5 0.5]}");
public Quantum q2 = MakeDensityOperator(2, "{[1 0;0 0]}");
U.Emit CNot = MakeU("{[1 0 0 0;0 1 0 0;0 0 0 1;0 0 1 0]}");//1->2 Cnot
M.Emit m = MakeM("{[1 0;0 0],[0 0;0 1]}");

protected override void run()
{
CNot(q1, q2);
Register(r1, m(q1));
Register(r2, m(q2));
}
```

We can see that the quantum object $q1$ is a quantum state $q1 = |+\rangle\langle+|$ and the quantum object $q2$ is $q2 = |0\rangle\langle 0|$. The CNOT gate (from $q1$ to $q2$) is defined as CNOT = [1 0 0 0;0 1 0 0;0 0 0 1;0 0 1 0]. After the CNOT gate, the state will be $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$. If measurement $\{|0\rangle, |1\rangle\}$ are performed on $q1$ and $q2$, the result should:

- Only consider $q2$, And about half the time the result should be 0 and half the time should be 1.
- Consider results for both $q1$ and $q2$. The 0 result for $q1$ should be the same as the 0 result for $q2$.

Figure 4.4 illustrates the results just as we predicated.

4.1.2.2 Termination Analysis-"Option 2,3"

Input and output

1. Enter the number 2 (or 3) to start the termination analysis example.

2. The Console should display the termination ability of the corresponding example. the user can see the "Final Result: Termination"in Figure 4.5 from Example 2 and "Final Result: Almost Sure Termination" in Figure 4.6 from Example 3.

3. The user can also see the real execution results from the Console.

```
        Please select a case number:    2
Operator Tree Merge Start:--------------------------------
Termination
QWhile 0
1+0i    0+0i    0+0i    1+0i
0+0i    1+0i    0+0i    0+0i
0+0i    0+0i    1+0i    0+0i
0+0i    0+0i    0+0i    1+0i

Final Result: Termination
Operator Tree Merge End:--------------------------------
The experiment program runs about 100 times:
Loops runs for 0 cycles is 51 times.
Loops runs for 1 cycles is 49 times.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

_
```

FIGURE 4.5: Termination

```
        Press <Enter> to exit...

        Please select a case number:    3
Operator Tree Merge Start:--------------------------------
AlmostTermination
QWhile 0
1+0i    0+0i    0+0i    1+0i
0+0i    1+0i    0+0i    -1+0i
0+0i    0+0i    1+0i    -1+0i
0+0i    0+0i    0+0i    2+0i

Final Result: Almost Sure Termination
Operator Tree Merge End:--------------------------------
The experiment program runs about 100 times:
Loops runs for 0 cycles is 54 times.
Loops runs for 1 cycles is 22 times.
Loops runs for 2 cycles is 10 times.
Loops runs for 3 cycles is 7 times.
Loops runs for 4 cycles is 5 times.
Loops runs for 5 cycles is 1 times.
Loops runs for 6 cycles is 1 times.
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

FIGURE 4.6: Almost-surely termination

Behind the console Loops raise the question: "Does the quantum program terminate?" The Termination Analysis module partly answers this question. See Chapter 3 for details.

4.1.2.3 BB84-"Option 4, 41, 42, 43"

BB84 is a quantum key distribution (QKD) protocol developed by Bennett and Brassard in 1984 [61]. The protocol is an already-proven security protocol [3] that relies on a no-cloning theorem.

Simple BB84-"Option 4",

Input and output

1. Enter the number 4 to start the simple BB84 protocol.

2. The Console will ask for an "Input Array Length". This is the raw initial key length. Input 5 as a test number.

3. The Console will then provide a hint "In basis, 0 is $\{|0\rangle, |1\rangle\}$ measurement and 1 is $\{|+\rangle, |-\rangle\}$" along with other information.
   Let's interpret Figure 4.7 as an example. The protocol is trying to find consensus for a key length of 5 bits. The initial bits (raw keys) are 11000 and the basis are $\{|+\rangle, |-\rangle\}, \{|0\rangle, |1\rangle\}, \{|0\rangle, |1\rangle\}, \{|+\rangle, |-\rangle\}, \{|+\rangle, |-\rangle\}$. With these two conditions combined, we know that Alice has generated the states $\{|-\rangle, |1\rangle, |0\rangle, |+\rangle, |+\rangle\}$. On the other hand, Bob chooses the basis $\{|+\rangle, |-\rangle\}, \{|+\rangle, |-\rangle\}, \{|0\rangle, |1\rangle\}, \{|0\rangle, |1\rangle\}, \{|+\rangle, |-\rangle\}$. to measure the qubits he received, and he broadcasts these basis. After Alice receives Bob's basis, she checks her chosen basis and finds that the correct positions are $1, 3$ and 5. She broadcasts the correct positions. Bob only keeps positions $1, 3$ and 5 of the measurement results. Both Bob and Alice reach an agreement key of 100.

FIGURE 4.7: Simple BB84 example

Behind the console In this case, a client-server model is used as a prototype for a multi-user communication protocol. A "quantum type converter" is used to convert a '*Ket*' into a density operator. Also, in this case, we use the Quantum Type '*Ket*' and do not consider a quantum channel or an Eve. The flow path is shown in Figure 4.8.

The entire flow path follows.

1. Alice randomly generates a sequence of classical bits called a *rawKeyArray*. Candidates from this raw key sequence are chosen to construct the final agreement key. The sequence length is determined by user input.

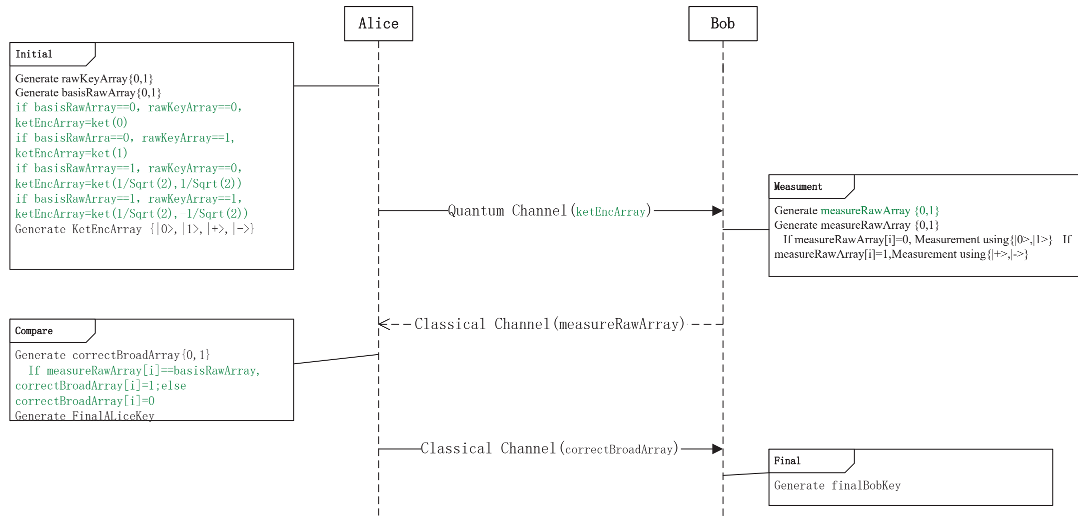


FIGURE 4.8: Simple BB84 protocol

2. Alice also randomly generates a sequence of classical bits called *basisRawArray*. This sequence indicates the chosen basis to be used in next step. Alice and Bob share a rule before the protocol:

   - They use $\{|+\rangle, |-\rangle\}$ or $\{|0\rangle, |1\rangle\}$ as the basis to encode the information.
   - A classical bit 0 indicates a $\{|0\rangle, |1\rangle\}$ basis while a classical bit 1 indicates $\{|+\rangle, |-\rangle\}$. This rule is used to generate Alice's qubits and to check Bob's chosen basis.

3. Alice generates a sequence of quantum bits called *KetEncArray*, one by one according to the rules below,

   - If the *basisRawArray[i]* in position [i] is 0 and the *rawKeyArray[i]* in position [i] is 0, *KetEncArray[i]* would be $|0\rangle$.
   - If the *basisRawArray[i]* in position [i] is 0 and the *rawKeyArray[i]* in position [i] is 1, *KetEncArray[i]* would be $|1\rangle$.
   - If the *basisRawArray[i]* in position [i] is 1 and the *rawKeyArray[i]* in position [i] is 0, *KetEncArray[i]* would be $|+\rangle$.

- If the *basisRawArray[i]* in position [i] is 1 and the *rawKeyArray[i]* in position [i] is 1, *KetEncArray[i]* would be $|-\rangle$.

4. Alice sends the *KetEncArray* through a quantum channel. In this case, she sends it through the I channel.

5. Bob receives the *KetEncArray* through the quantum channel.

6. Bob randomly generates a sequence of classical bits called *measureRawArray*. This sequence indicates the chosen basis to be used in next step.

7. Bob generates a sequence of classical bits called *tempResult*, using quantum measurement according to the rules:

   - If the *measureRawArray[i]* in [i] position is a classical bit 0, Bob uses a $\{|0\rangle, |1\rangle\}$ basis to measure the *KetEncArray[i]* while a classical bit 1 indicates using a $\{|+\rangle, |-\rangle\}$ basis.

8. Bob broadcasts the *measureRawArray* to Alice using a classical channel.

9. Alice generates a sequence of classical bits called *correctBroadArray*, by comparing Bob's basis *measureRawArray* and her basis *basisRawArray*. If the position [i] is correct, the *correctBroadArray[i]* would be 1, otherwise would be 0.

10. Alice sends the sequence *correctBroadArray* to Bob.

11. Alice generates a sequence of classical bits called $FinalALiceKey$ using the rule:

    - If position [i] in *correctBroadArray[i]* is 1, she keeps *rawKeyArray[i]* and copies it to *FinalALiceKey* , else she discards *rawKeyArray[i]*.

12. Bob generates a sequence of classical bits called $FinalBobKey$ using the rule:

    - If position [i] in *correctBroadArray[i]* is 1, he keeps *tempResult[i]* and copies it to *FinalBobKey[i]*, else he discards *tempResult[i]*.

13. GlobalView: We use a function compare whether every position [i] in *FinalALiceKey* and *FinalBobKey[i]* are equal.

This case shows some useful features,

- Quantum Computation Engine Direct Call (QCEDC). This example uses several low-level components of QSI (i.e., the functions directly from the quantum computation engine). This mode can be employed to design and code quantum communication protocols that cannot easily be described in formal quantum language.
- Client-server Mode. The process uses a client-server model to simulate the BB84 protocol. The model includes many powerful features such as waiting thread and classical concurrent communication in the next example.
- Measurement. According to the theoretical result, choosing a random measurement basis may arrive at half of the correct result. As a result, the agreement of classical shared bits should be almost half length of raw keys.

Multi-clients BB84-"Option 41" A multi-client BB84 model is a more attractive and practical example. In a multi-client BB84 model, only one Alice generates the raw keys, but several Bobs construct an agreement key with Alice.

Input and output

1. Enter the number 41 to start the simple BB84 protocol.
2. The Console will ask for an "Input Array Length". This is the raw initial keys length. Input 5 as a test number.
3. The Console will ask the user to "Input Numbers of Clients". This is the number of "Bobs" to be generated. Input 2 as a test number.
4. The Console will then print the final Alice Key for Thread number 7 is 010 and for Thread 8 is 00 which are the same as final Bob Key for each Bob.

BB84 with a noise channel-"Option 42", A very interesting and practical topic for QSI is to use the BB84 model to consider noisy quantum channels. Because no quantum system is ever perfectly closed, quantum operations are key tools for describing the dynamics of open quantum systems.

Input and output

Figure 4.9: Multi-clients BB84 console

1. Enter the number 42 to start the simple BB84 protocol.
2. The Console will ask for the "Input Array Length". This is the raw initial key length. Input 5 as a test number.
3. Then, the Console will print a hint that "In basis, 0 is $\{|0\rangle, |1\rangle\}$ measurement and 1 is $\{|+\rangle, |-\rangle\}$" along with other information.

Let's interpret Figure 4.10 and 4.11 as an example. In Figure 4.10, the protocol is trying to find consensus for a key length of 5 bits. The initial bits (raw keys) are 10100 and the basis are $\{|+\rangle, |-\rangle\}$,$\{|0\rangle, |1\rangle\}$,$\{|+\rangle, |-\rangle\}$,$\{|0\rangle, |1\rangle\}$,$\{|0\rangle, |1\rangle\}$. With these two conditions combined, we know that Alice has generated the quantum states $\{|-\rangle, |0\rangle, |-\rangle, |0\rangle, |0\rangle\}$ and sends them through a **quantum channel**. On the other hand, Bob chooses the basis $\{|+\rangle, |-\rangle\}$,$\{|+\rangle, |-\rangle\}$,$\{|+\rangle, |-\rangle\}$,$\{|0\rangle, |1\rangle\}$,$\{|+\rangle, |-\rangle\}$ to measure the qubits he received, and he broadcasts these basis. After Alice receives these basis, she checks her chosen basis, finds the correct positions are 1 and 2. She broadcasts the correct positions. Bob only keeps positions 1 and 2 of the measurement results. Bob and Alice reach an agreement of key 10; however, the key agreement is not successful. In Figure 4.11, Alice and Bob follow the protocol outlined above, but, the BB84 fails due to interference in the quantum channel.

Behind the console In this example, different channels, such as the bit flip, depolarizing, amplitude damping, and I-identity channels are described by a set of Kraus operator.

FIGURE 4.10: BB84 with a quantum channel, success

FIGURE 4.11: BB84 with a quantum channel, failure

Alice and Bob use these quantum channels to communicate with each other via the BB84 protocol as Figure 4.8 shows. To ensure successful communication, verification steps also need to be considered.

Some quantum channels $\mathcal{E}$ that can be used during communication are defined below:

Deplarizing channel, where $p = 0.5$,

$$\mathcal{E} := \left\{ \begin{bmatrix} \frac{\sqrt{5}}{\sqrt{8}} & 0 \\ 0 & \frac{\sqrt{5}}{\sqrt{8}} \end{bmatrix}, \begin{bmatrix} 0 & \frac{1}{\sqrt{8}} \\ \frac{1}{\sqrt{8}} & 0 \end{bmatrix} \begin{bmatrix} 0 & \frac{-i}{\sqrt{8}} \\ \frac{i}{\sqrt{8}} & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{8}} & 0 \\ 0 & -\frac{1}{\sqrt{8}} \end{bmatrix} \right\}.$$

Amplitude damping channel, where $\gamma = 0.5$,

$$\mathcal{E} := \left\{ \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ 0 & 0 \end{bmatrix} \right\}.$$

And three kinds of bit flip channel:

Bit flip channel, where $p = 0.25$,

$$\mathcal{E} := \left\{ \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix}, \begin{bmatrix} 0 & \frac{\sqrt{3}}{2} \\ \frac{\sqrt{3}}{2} & 0 \end{bmatrix} \right\}.$$

Bit flip channel, where $p = 0.5$,

$$\mathcal{E} := \left\{ \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 \\ 0 & \frac{1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & 0 \end{bmatrix} \right\}.$$

Bit flip channel, where $p = 0.75$,

$$\mathcal{E} := \left\{ \begin{bmatrix} \frac{\sqrt{3}}{2} & 0 \\ 0 & \frac{\sqrt{3}}{2} \end{bmatrix}, \begin{bmatrix} 0 & \frac{1}{2} \\ \frac{1}{2} & 0 \end{bmatrix} \right\}.$$

The program flow path follows the simple BB84 protocol. The only differences are in Step 4 and an added sampling step.

- Alice sends the *KetEncArray* through a quantum channel. In this case, it is one of the channels mentioned above.

BB84 with statics and channels-"Option 43" From above example, we know that both the quantum channel and the verification (check) affects the quantum state. In this example, we want to answer the question: What is a suitable proportion for a fixed quantum channel?

Input and output

1. Enter the number 43 to start the BB84 protocol with statistics and channel.

2. The Console will display "x-axis..." and "y-axis...".

3. We assume that, in the verification step, any one-bit difference will cause a failure in that run. We can send different lengths of data packets and use the criteria to check the percentage of the protocol.

   A one-time experiment is shown in Figure 4.12. Let's interpret Figure 4.12. The BB84 protocol is in an amplitude damping quantum channel (defined as $\mathcal{E} := \{\begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ 0 & 0 \end{bmatrix}\}$). In the output, the number in Column 1 and Row 1 means the protocol has an 85% chance of success if it performs in the amplitude damping quantum channel defined above with a 32 qubit data packet and 10% of the results are used for verification.

```
C:\Users\klinu\Source\Repos\qcis\TestudoV1\UnitTest\bin\Debug\UnitTest.exe
        4:  Simple BB84.
        41: The multi-clients protocol for simple BB84, without statistics.
        42: The BB84 protocol with channel. Bit flip channel, p=0.1.
        43: The BB84 with statistics and channel.
        5:  Quantum Teleportation with QASM.
        6:  Quantum Google PageRank.
        7:  Grover Search, the oracle has been set answer the position 3.
        71: Standard Grover Search.
        72: Automatic toolkits Grover, search 2^4, answer from 0-15.(DEBUG close)
        73: Search multi-objects Grover. It is WRONG.
        8:  A comprehensive Quantum Teleporation. Termination and Decomposition.

        Press <Enter> to exit...

        Please select a case number:    43
The x-axis is the DATA package length, from 32qbit-512qbit.
The y-axis is the sample percentage, from 10%-100%.
For every case with different parameters, the case will run about 100 times and add
up the success times.

The statistical process begins:

80      50      30      15      0
63      25      9       0       0
40      16      1       0       0
29      7       0       0       0
17      3       1       0       0
10      4       0       0       0
8       1       0       0       0
6       1       0       0       0
5       0       0       0       0
4       1       0       0       0
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

FIGURE 4.12: BB84 with statistics and quantum channel

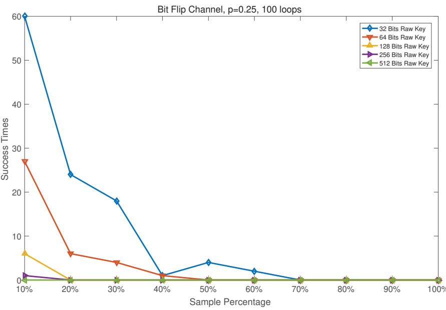
Behind the console The program flow path follows the simple BB84 protocol. The only differences are in Step 4 and an added sampling step is added,

- Alice performs the *KetEncArray* through a quantum channel. In this case, it is one of the channels mentioned above.
- Sampling check step: Alice randomly publishes some sampling positions of the bits. Bob checks these bits against his own key strings. If all the bits in these sampling strings are the same, he believes the key distribution is a success; Otherwise the connection fails.

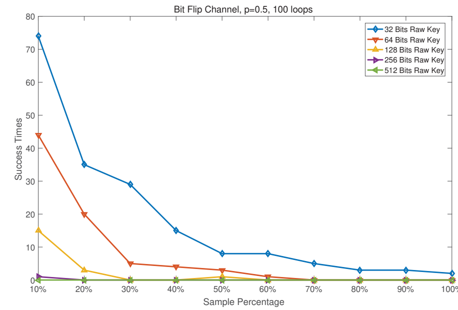
For a statistical quantity characterising success in a channel in the BB84 protocol, this experiment should be executed as 100 shots for each channel. In every shot for each channel, different sampling percentages and package lengths should be considered. Tables

and figures are provided that show the trade-off between success times, different sampling proportions, and package lengths in each of the quantum channels.

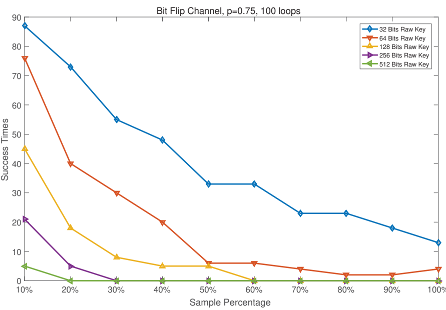
Different channels and different proportion for BB84 Success times for different sampling percentages in different channels over 100 shots are shown in Figure 4.13.



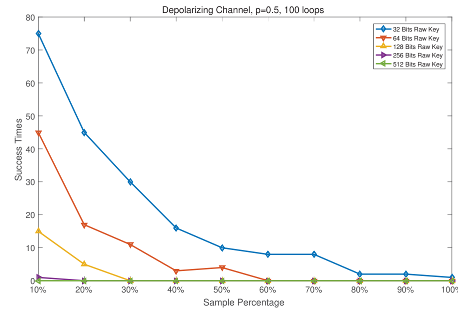
(a) Bit Flip Channel, $p = 0.25$, $loops = 100$



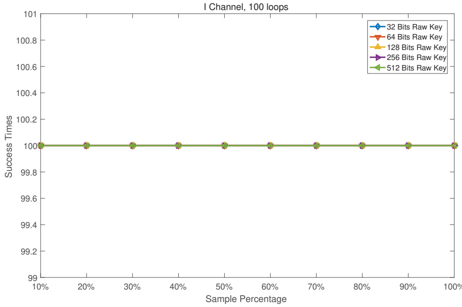
(b) Bit Flip Channel, $p = 0.5$, $loops = 100$



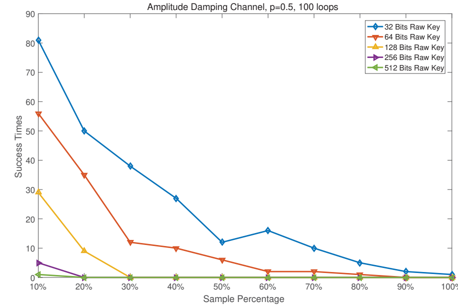
(c) Bit Flip Channel, $p = 0.75$, $loops = 100$



(d) Depolarizing Channel, $p = 0.5$, $loops = 100$



(e) I-channel, $loops = 100$



(f) Amplitude Damping Channel, $p = 0.5$, $loops = 100$

FIGURE 4.13: Statistics of success communication via BB84 with channels

Features and analysis The example generates some 'error' bits during communication due to the properties of quantum channels and cause a failed connection. However, not all error bits can be found in the sampling step because, in theory, almost half the bits

are invalid in the measurement step. Additionally, the sampling step is a probability verification step, which means it does not use all the agreed bits to verify the communication procedure.

Sub-figures (a), (b) and (c) in Figure 4.13 are bit flip channels with different probabilities. Overall, the number of successful shots increases as p and the raw key length shortens. This is because p is a reflection of the percentage of information remaining in bit-flip channel, and an increase in p means fewer communication errors. A shorter length ensures fewer bits are sampled. Sub-figures (d), (e) and (f) illustrate the communication capacity of the BB84 protocol in the other three channels. Note that the I-identity channel has a 100% success rate which means it is a noiseless channel and can keep information intact during the transfer procedure.

4.1.2.4 Quantum teleportation with QASM-"Option 5"

Input and output

1. Enter the number 5 to start Quantum Teleportation with QASM.
2. The Console will display that "Quantum Teleportation begins..." and show the experiment result.
3. Meanwhile, the Console pops up a notebook and displays the generated "QASM".
4. The teleportation experiment will run 1000 times and the statistic results will be shown on the Console.

Behind the console The key code segment is trivial enough to show here.

In Lines 01 to 16, we define: one classical register $r3$; three quantum registers $Alice$, $Bob1$ and $Bob2$; four quantum gates $hGate$, $CNot$, $xGate$ and $zGate$; and one measurement m. Lines 19 to Line 21 define how the quantum gates perform on the quantum registers. The most interesting section is from Lines 22 to 35 which includes two advanced $[\![QIF]\!]$ clauses in the code segment. A measurement in Line 37 illustrates how to ensure that the received quantum state is real the $Alice$ state.

```
01 class TestQuantMulti1 : QEnv //Quantum Teleportation
02 {
03     public Reg r3 = new Reg("r3");
04     public Quantum Alice=MakeQBit(2,"{[1/sqrt(5); sqrt(4)⤸
05                     /sqrt(5)]}");
06     public Quantum Bob1 = MakeDensityOperator(2, "{[0.5 0.5;⤸
07                          0.5 0.5]}");//|0>+|1>
08     public Quantum Bob2 = MakeDensityOperator(2, "{[1 0;0 0]}⤸
09                          ");
10     U.Emit hGate = MakeU("{[1/sqrt(2) 1/sqrt(2); 1 / sqrt(2) ⤸
11                    -1 / sqrt(2)]}");
12     U.Emit CNot = MakeU("{[1 0 0 0;0 1 0 0;0 0 0 1;0 0 1 0]}⤸
13                   ");//1->2 Cnot
14     U.Emit xGate = MakeU("{[0 1; 1 0]}");
15     U.Emit zGate = MakeU("{[1 0;0 -1]}");
16     M.Emit m = MakeM("{[1 0;0 0],[0 0;0 1]}");
17     protected override void run()
18     {
19         CNot(Bob1, Bob2); //Prepare |00>+|11> for Bob
20         CNot(Alice, Bob1);
21         hGate(Alice);
22         QIf(m(Bob1),
23             () =>
24         { },
25         () =>
26         {
27             xGate(Bob2);
28         });
29         QIf(m(Alice),
30             () =>
31         { },
32         () =>
33         {
34             zGate(Bob2);
35         });
36
37         Register(r3, m(Bob2));
38     }
39 }
```

Figure 4.14: Quantum teleporation with f-QASM

4.1.2.5 Quantum Google PageRank-“Option 6”

Input and output

1. Enter the number 6 to start the Quantum Google Rank.
2. The Console will ask “Please enter the length. . . ”. the user can just press Enter the key to use the default configuration.
3. By default, the Console will display the adjacency matrix (see the corresponding graph in Figure 4.15) and show the steps of the quantum transformation.
4. After a while, the program will show the averaged quantum pagerank.

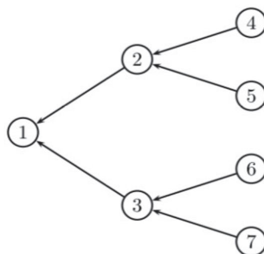


Figure 4.15: Default network structure

4.1.2.6 Quantum Grover search-“Options 7,71,72 & 73”

Grover search algorithm is a representative quantum algorithm. It solves search problems in databases consisting of N elements, indexed by number $0, 1, \ldots, N-1$ with an oracle providing the answer as a position. The algorithm is able to find solutions with a probability $O(1)$ within $O(\sqrt{N})$ steps.

Grover Search, with a pre-defined oracle - “Option 7”

Input and output

1. Enter the number 7 to start Grover search.
2. The Console will display “The default oracle ...” and the result.
3. Of course, the user can check the results of the algorithm. However the key component of this example is the source code corresponding to the experiment.

“Option 71”- Standard Grover Search

Input and output

1. Enter the number 71 to start a standard Grover search.
2. The Console will ask ”How large ...”. Type 7 for an instance. We recommend not making the user instance larger than 9 on PC as it consumes too much time and memory during the measurement stage.
3. The Console will ensure the volume of the database and display the number. Then, it will ask ”where is the correct answer...”. Type 56 for an instance. The oracle will be set up in this step.
4. After the oracle has been called 8 times ($R \leq \frac{\pi}{4}\sqrt{\frac{N}{M}}$), the result will be displayed.

Behind the console Suppose $|\alpha\rangle = \frac{1}{\sqrt{N-1}} \sum_x'' |x\rangle$ is not the solution but rather $|\beta\rangle = \sum_x' |x\rangle$ is the solution where $\sum_x'$ indicates the sum of all the solutions. The initial state $|\psi\rangle$ may be expressed as

$$|\psi\rangle = \sqrt{\frac{N-1}{N}}\, |\alpha\rangle + \sqrt{\frac{1}{N}}\, |\beta\rangle \ .$$

Every rotation makes the θ to the solution where

$$\sin\theta = \frac{2\sqrt{N-1}}{N}.$$

When N becomes larger, the gap between the measurement result and the real position number is less than $\theta = \arcsin\frac{2\sqrt{N-1}}{N} \approx \frac{2}{\sqrt{N}}$. It is almost impossible to have a wrong answer within r times.

4.1.2.7 Grover search with automatic toolkits - "Option 72"

Input and output

1. Before executing go to *TestGroverH.cs* file, find `//#undef DEBUG` and uncomment it as `#undef DEBUG` at the second line of the file.
2. Then press "F5" to execute the program and enter the number 72 to start the Automatic Toolkits Grover Search.
3. The Console will run the experiment 16 times and set the Oracle from 0 to 15.
4. Do not forget to restore the uncomment line to comment line (`#undef DEBUG` to `//#undef DEBUG`).

Behind the console This experiment is an automatic toolkit the user can use to verify an algorithm. This functionality can be quite useful when the user consider generating many oracles in a program.

4.1.2.8 Standard Grover search -"Option 73"

Input and output

1. Enter the number 73 to start a standard Grover search.
2. The Console will ask "How large...". Type 7 as an instance. We recommend not making the user instance larger than 9 on PC as it consumes too much time and memory during the measurement stage.

3. The Console will ensure the volume of the database and display the number. Then, it asks "Where are the correct answers…". Type 56 and 57 for an instance. The oracle will be set up in this step.

4. The Algorithm will try to search for the target numbers.

Behind the console This experiment is a multi-object Grover's search algorithm that supposes the multi-answer (position) of the oracle. We use the strategy of adding that a blind box to reverse the proper position of the answer.

This experiment reveals that Grover's algorithm leads to an avalanche of errors in a multi-object setting, indicating that the algorithm needs to be modified in some way. A new blind box (a unitary gate) is added that reverses the proper position of the answer. In short, the oracle is a matrix where all the diagonal elements are 1, but all the answer positions are −1. Thus, the blind box is a diagonal matrix where all the elements are 1, and all the answer position that have been found are −1. When these two boxes are combined, we create a new oracle with the answers to all the questions except the ones that were found in previous rounds.

The measurement shows different probabilities of the final result. Theoretical result holds that if we have multiple answers, the state after r times oracles and phase gates should become the state near both of them. For example, if the answers are $|2\rangle, |14\rangle \in \mathcal{H}_{64}$, the state before the measurement is expected to be almost $\frac{1}{\sqrt{2}}(|2\rangle + |14\rangle)$. We should get the $|2\rangle$ or $|14\rangle$ the first time and the other one the next time. However, we actually get results other than $|2\rangle$ and $|14\rangle$ with high probability, which indicates that the multi-object search strategy is not very good.

It worth noting that due to multi-objects, the real state after using Grover's search algorithm becomes a $a(|2\rangle + |14\rangle) + b(|1\rangle + |3\rangle + |4\rangle + |5\rangle + ...)$, where a, b are complex scalars that satisfy $|a|^2 + |b|^2 = 1$. However, b cannot be ignored even it is very small. An interesting issue occurs when the wrong position index is found. If the wrong index is obtained by the measurement, the algorithm creates an incorrect blind box and reverses the wrong position of the oracle, i.e., it adds a new answer to the questions. In next round, the proportion of correct answers is further reduced. For example, in the last example, we would have

gotten the wrong answer by a measurement of, say $|5\rangle$. After another procedure, the state would become: $a(|2\rangle + |14\rangle + |5\rangle) + b(|1\rangle + |3\rangle + |4\rangle + |5\rangle + ...)$. It becomes harder and harder to find the correct answer with this state.

In short, one wrong answer in a round causes an avalanche of errors with Grover's search algorithm.

4.1.3 Serious Coding

After a taste of some examples with QSI , we are going to illustrate user-defined code.

Sub-section 4.1.3.1 figures out the user-defined components insisting of entry point, body code structure and quantum **while**-language in practice.

Sub-section 4.1.3.2 demonstrates "Qloop", a very impressive "HelloWorld" example of the quantum realm that illustrates the power of our platform. This example will familiarise the user with some of the advanced components available in QSI .

Sub-section 4.1.3.3 demonstrates the way to use the termination module.

Sub-section 4.1.3.4 is designed to teach the user details of the quantum compiler and related configures.

4.1.3.1 From UnitTest to HelloWorld

This section provides a brief tutorial for writing a standard quantum code segment. The tutorial includes two main parts: quantum programming and related classical function programming.

Entry to the user code Start the user code in a QSI project. QSI (Project) is designed for user code that does not include too many assumptions or definitions compared with UnitTest (Project).

PATH: `Solution 'Testudo' → QSI(Project)`(Figure 4.16)

Writing Quantum Code

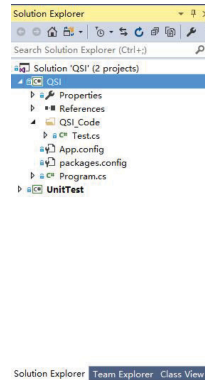


FIGURE 4.16: Entry to the user code

Code file `Test.cs` under the QSI_Code (Folder) is a good file name for the user quantum code.

PATH: `Solution 'Testudo'` → `QSI(Project)` → `QSI_Code` → `Test.cs`

the user can give the user quantum code any file name. We only suggest naming the user code 'Test.cs' because some modules, such as Termination Analysis, require the user to designate a file path to the quantum code and, in this example, we have designated the target file for termination analysis as 'Test.cs' in 'Program.cs' under the QSI (Project).

Component quantum libraries Two .Net assemblies are provided that relate to the quantum code:

Assemblies: `QuantumRuntime, QuantumToolkit` .

These two assemblies conclude analogous namespace:

Namespace: `QuantumRuntime, QuantumToolkit` .

Each has several different sub-namespaces:

QuantumRuntime Sub-namespace: `Operator`

QuantumToolkit Sub-namespace: `Parser, Runtime, Type`

Suitable namespace A suitable Namespace is an essential part of quantum programming development. In fact, it is also necessary for developing most C# programs. Unless,

the user figure out all the functions and objects in the namespace, we suggest the user introduce following namespaces:

Default Namespaces for Development: `QuantumRuntime, QuantumToolkit, QuantumToolkit.Parser`
.

the user can use the code segment to introduce our recommended namespaces:

```
using QuantumRuntime;
using QuantumToolkit;
using QuantumToolkit.Parser;
```

Don't forget to introduce the other assemblies required in classical programming. A full example has been given in the 'Test.cs'.

Exploit namespace The user can exploit our provided assemblies using 'Reflection' inherited in Visual Studio.

PATH: `Menu → View → Object Browser`

Import static functions To allow the user to access static members of a type without having to qualify the access with the type name, 'using static' is a good option. Please import commonly used static functions like code segment shows.

```
using static QuantumRuntime.ControlStatement;
using static QuantumRuntime.Operator.E;
using static QuantumRuntime.Operator.M;
using static QuantumRuntime.Operator.U;
using static QuantumRuntime.Quantum;
using static QuantumRuntime.Registers;
```

Inheriting is essential for reducing code complexity A parent class called `QEnv` has been provided to reduce code complexity. In fact, the `QEnv` is the base class in the quantum programming environment. It inherited from `MarshalByRefObject` to ensure a clean and separate environment for each experiment.

In our `Test.cs`, we have provided the following example. Please do not delete it:

Table 4.1: Quantum variable types and corresponding initial methods

Type	**Keywords**	**Example**	**Initialization**
Quantum Register	Quantum	Quantum Bob1	MakeDensityOperator & MakeQBit
Unitary Gate	U.Emit	U.Emit hGate	MakeU
Quantum Channel	E.Emit	E.Emit BitFlip	MakeE
Measurement	M.Emit	M.Emit m	MakeM
Classical Register	Reg	Reg r1	new Reg

Table 4.2: Quantum variable initialisation matrix form

Function	**Component**	**Example**
MakeDensityOperator	Matrix	MakeDensityOperator(“[1 0;0 0]”)
MakeQbit	Matrix	MakeQbit(“[1; 0]”)
MakeU	Matrix	MakeU(“[1 0;0 -1]”)
MakeM	Matrix	MakeM(“[1 0;0 0],[0 0;0 1]”)
MakeE	Matrix	MakeE(“[1 0;0 0],[0 0;0 1]”)

```
class Text:QEnv
{
}
```

Declaring registers, quantum registers, quantum gates and quantum measurement A variable is simply a name given to a storage area that our programs can manipulate. Each variable in C# has a specific type, which determines the size and the access way to variable's memory. The basic quantum variable types and initial methods provided in QSI can be categorised as per Table 4.1.

Both kinds of registers, the classical and the quantum registers, need a number to indicate the dimensions of the object and a matrix for filling the object during the initialization function. In quantum code, QSI derives this dimension implicitly so users can ignore this aspect of the register. Conceptually, unitary gates, quantum channels, and measurements do not rely on dimension information, so all inputs need only be in matrix form as Table 4.2 shows.

run() as a container for quantum circuits (workflow) After defining the “input”, Gates (Unitary Matrix), measurement and the initial quantum state, the user can write the self-defined quantum circuits inside the function `run()`.

A very brief example follows to illustrate the `run()` function.

```
protected override void run()
{
hGate(q1);
Register(r1, m(q1));
}
```

The gates should match the number of qubits. In most scenarios, a single-qubit universal gate set and a two-qubit gate, CNOT, is enough to construct a quantum circuit. But QSI does support other dimensions of a unitary matrix. Any given multi-qubit gates (excluding a CNOT) could be translated into single unitary gates and CNOTs.

Writing classical code A standard program should include both classical and quantum coding. In the previous section, we showed how to write a quantum code segment as the key to the experiment, but classical coding is also essential. As a programming environment, only a few high-level functions have been provided for classical code, such as analysis, execution, compilation and so on. Also, as a flexible environment, being able to control everything just the way the user wants is a cool setting for a programmer. We suggest that all classical coding should be placed in '`Program.cs`', which is also the entry point for the project in the C# language.

At the very beginning, take note that QSI has two modes: static and dynamic. Static mode is used for analysing the code structure, termination, and compilation, whereas dynamic mode is used to execute the code.

Create, initialise, run, and collect

Creat The first code step is to create an instance of the quantum code the user wrote and saved as `Test.cs`.

Creat:
```
var test = QEnv.CreateQEnv<QSI_Code.Test>();
```

The default configuration allows the user to display any changes about classical registers. If the user do not need any modification of registers trigger display, the user can use the following command to stop the outputting status:

Reg Display Control: `test.DisplayRegisterSet = false;`

Initilise Initialisation is a preparation step that makes all the quantum objects available to the classical code. It also initialises the background variables and the input matrix.

Init: `test.Init();`

Run The initialisation process will come to a halt, and it is time for the key steps to take over.

Run: `test.Run();`

Note that `test.Init();` only needs to be executed once, while `test.Run();` can be executed many times. In fact, most quantum experiments rely on statistical results, which means that the user will probably need to execute `test.Run();` many times depending on the user's requirements.

Collect Results will need to be "collected" in the classical part of the coding. All classical registers are NOT and can be viewed or indexed by functions belonging to classical code by default. Thus, a "public" modifier is a good choice for making collection by other programs possible. A display process is shown here as a collection.

Collect: `Console.WriteLine(test.r1.value);`

Using the quantum while-language After that brief introduction, we can start to code with the quantum while-language. The new clauses are based on the previous syntax. Thus, the only clauses "to be updated" are two new structures: qif and qwhile.

QIf Prototype

```
QIf(Measure(Qubits), () => {Sub_program_1}, () => {Sub_program_2}), ...;
```

Measure is a measurement, which could be a binary output measurement or a multi-bit

output measurement;

Qubits is a set of qubits indexed by a variable; and

Sub_program is a subprogram written in the quantum **while**-language. The behavior of this structure is actually a case statement. First, a measurement is performed on the variable Qubits. Then, a sub program is executed according to the output index, i.e., the output according to the size of the number. For example, Subprogram 1 is executed when the measurement gives the result 0, while Subprogram 2 is executed when the measurement gives the result 1.

Qwhile Prototype

```
QWhile(Measure(Qubits), () => {Sub_program});
```

Measure is a measurement, which should be a binary output measurement;

Qubits is a set of qubits indexed by a variable; and

Sub_program is a subprogram written in the quantum while-language. The behavior of this structure is actually a loop statement. First, a measurement is performed on the variable qubits. Then, based on the output (NOT output index), i.e., result 0 or result 1, a Subprogram may be executed. For example, result 1 means the `Sub_program` is executed, but result 0 leads to *SKIP* the `Sub_program` , and the clause after QWhile is executed instead.

4.1.3.2 A simple example

Let's write a simple example that includes `QIf` and `QWhile`. This example is a modified quantum teleportation that illustrates the power of these new structures.

The code is shown in Figure 4.17 shows.

The code segment is similar to a standard quantum teleportation. Note that a `QWhile` has been added in Line 23. That means the body of the quantum teleportation will only be executed when the measurement result is 1. Another modification has been added at Lines 30 and 34 – a `QIf` that shows an Empty or a unitary matrix `xGate` (or `zGate`) is performed on the qubits according to the result of the measurement.

```
01 class TestQuantMulti3 : QEnv {
02     public Reg r3 = new Reg("r3");
03
04     public QReg qOutput = new QReg();
05     public Quantum LooperQ = MakeDensityOperator("{[1 0;0 0]}⤸
06                                 ");
07     public Quantum Alice = MakeQBit("{[1/sqrt(5); sqrt(4)⤸
08                             /sqrt(5)]}");
09     public Quantum Bob1 = MakeDensityOperator("{[0.5 0.5;0.5 ⤸
10                             0.5]}");//1/2(|0>+|1>)
11     public Quantum Bob2 = MakeDensityOperator("{[1 0;0 0]}");
12     U.Emit hGate = MakeU("{[1/sqrt(2) 1/sqrt(2); 1 / sqrt(2) ⤸
13                     -1 / sqrt(2)]}");
14     U.Emit CNot = MakeU("{[1 0 0 0;0 1 0 0;0 0 0 1;0 0 1 0]}⤸
15                    ");//1->2 CNot
16     U.Emit xGate = MakeU("{[0 1; 1 0]}");
17     U.Emit zGate = MakeU("{[1 0;0 -1]}");
18
19     M.Emit m = MakeM("{[1 0;0 0],[0 0;0 1]}");
20
21     protected override void run() {
22         hGate(LooperQ);
23         QWhile(m(LooperQ),
24         () => {
25             hGate(LooperQ);
26
27             CNot(Bob1, Bob2); //|00>+|11> for Bob
28             CNot(Alice, Bob1);
29             hGate(Alice);
30             QIf(m(Bob1),
31             () => {},
32             () => {
33                 xGate(Bob2); });
34             QIf(m(Alice),
35             () => {},
36             () => {
37                 zGate(Bob2); }); }
38               );
39         Register(r3, m(Bob2));
40
41     } }
```

Figure 4.17: Teleportation with loops

4.1.3.3 Checking for termination

Termination abstract Here, the user may notice that although loop structures give true power to program, a serious issue is raised: termination. When a loop is added, a program may become non-terminating, meaning the program never halts. Users may also know that the non-halting problem is a hot topic in classical computing. However, quantum scenarios are more complex than classical ones. A quantum state could be in a superposition that, in simple terms, means it could execute in a non-halting way across several workflows. Given loop structure is also an influencing factor, this type of termination problem is non-trivial in most cases.

For more details on the termination analysis module, please refer to the articles listed in the references.

Checking for termination using the modules provided To check the whether a given program is properly terminated, the user needs to specify the path of the input file and the class name of the target program. An example follows.

```
var exeDir = Path.GetDirectoryName(
Process.GetCurrentProcess().MainModule.FileName);
```

```
var inputFile = Path.Combine(exeDir, @"..\..\QSI_Code\Test.cs");
var generator = new Generator(File.ReadAllText(inputFile));
generator.Parse("Test");
generator.MatRepANDAnalysis(false);
```

Lines 01 and 02 denote the path of the target class. Line 03 produces a generator, which is core to static analysis. Line 04 then tries to analyse the input class file. Please do not forget to mention the target class if there is more than one class in the file. Line 05 attempts to find the corresponding termination function and outputs the result.

The termination analysis tool only supports pure quantum codes. That means users should NOT mix any classical code in classes with quantum code.

4.1.3.4 Compilation

QASM (Quantum Assembly Language) is widely used in modern quantum simulators. We propose another format of QASM called f-QASM (Quantum Assembly Language with feedback). The most significant motivation behind this variation is to translate the inherent logic in high-level programming languages into a simple command set, i.e., so there is only one command in every line or period. However, a further motivation is to solve an issue raised on the IBM QASM 2.0 list and provide conditional operations for implementing feedback based on measurement outcomes.

f-QASM (Quantum Assembly Language with feedback) f-QASM is an improved quantum assembly language that can be used with the compiler in quantum simulators and real quantum processors. A set of measurement-based operations is defined to execute if and while structures. Using f-QASM, the compiler can compile a high-level quantum language that includes loop and case-statements into a low-level device-independent or device-dependent instruction set.

Compile quantum program using compile module

Compile modules Prototype

```
QAsm.Generate("Parameter_1", Parameter_2, Parameter_3, Parameter_4,
              Parameter_5(generator.OperatorGenerator.OperatorTree));
```

$QAsm.WriteQAsmText(Parameter_1);$

The first command is the quantum compilation command.

Parameter_1 is the class name to be compiled.

Parameter_2 is a number from 0-3 indicating how "deep" the user want the quantum circuits to be generated.

Parameter_3 is an integer from 1-100 indexed to the accuracy of decomposition for the arbitrary gate U which is performed on a single qubit. In fact, given any $\epsilon > 0$, the Solovay-Kitaev theorem can generate a series of gates from a fixed finite set to approximate U with a precision ϵ of $\Theta(\log^c(1/\epsilon))$, where c is a small constant approximately equal to 2 [50]. In the article [24], accuracy is determined by a recursion number, and that number can be set here.

Parameter_4 is an enumeration type consisting of two options:

1. `QuantumMath.PreSKMethod.OrginalQSD`,
2. `QuantumMath.PreSKMethod.OrginalQR`.

The first option, `QuantumMath.PreSKMethod.OrginalQSD`, uses the algorithm from the article [47]. The second option uses the algorithm from Section 4.5 of the book [50].

Parameter_5 is a fixed OperatorTreeNode Type which is generated by the static analysis module.

The second command writes f-QASM into a file. **Parameter_1** is a Boolean value: `true` or `false`. `true` means the f-QASM file will open automatically after program compilation.

Compiled results The compiled results can be found under the folder, *bin\Debug\temp* or *bin\Release\temp*.

The command

```
QAsm.Generate("Parameter_1", Parameter_2, Parameter_3, Parameter_4,
              Parameter_5(generator.OperatorGenerator.OperatorTree));
```

generates the compiled files. The compiled level is decided by the **Parameter_2**.

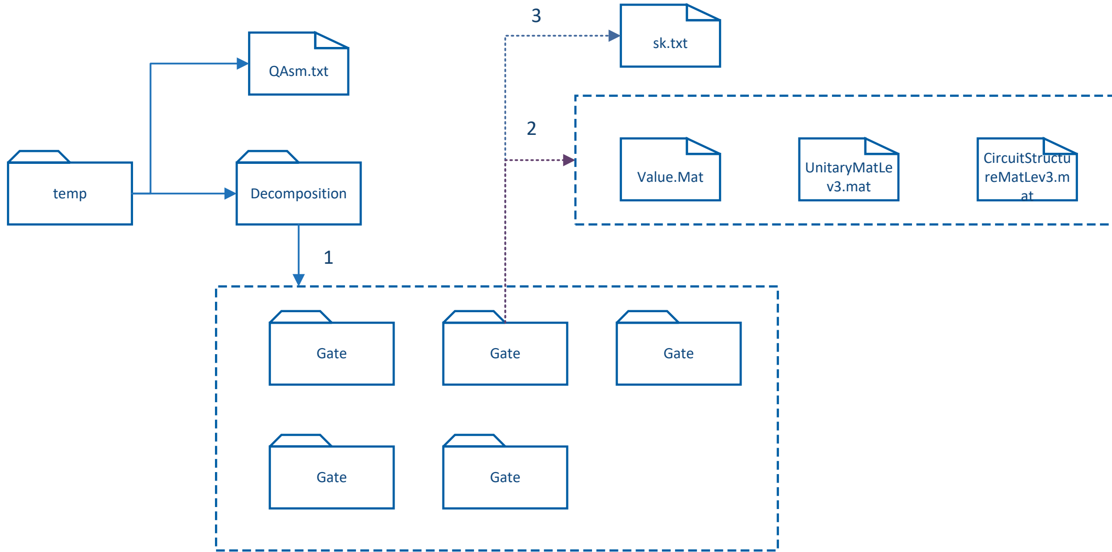


FIGURE 4.18: Compiled file structure

1. **Parameter_2=0** would do nothing to the compilation.
2. **Parameter_2=1** would convert all the super-operator $\mathcal{E}$ into the system environment model with the U matrix. This number is currently ignored and will be developed in the next stage.
3. **Parameter_2=2** would convert all the U gates into qubit gates and CNot gates.
4. **Parameter_2=3** would convert all qubit gates into $H, TInv, T$ gates using the Solovay-Kitaev algorithm, where $H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$, $T = \begin{bmatrix} 1 & 0 \\ 0 & \exp^{i\pi/4} \end{bmatrix}$ and $TInv = \begin{bmatrix} 1 & 0 \\ 0 & \exp^{-i\pi/4} \end{bmatrix}$.

The accuracy for approximating of qubit gates and CNOT gates to arbitrary gate is decide by **Parameter_3**. A large integer means higher accuracy. For general usage, an integer of 3 is enough for approximation. Accuracy can reach $\exp^{-6}$ accuracy, i.e., $||U - U'||_2 < \exp^{-6}$ where U' is constructed with $H, T, TInv$ gates.

4.2 Experiments with QISKit and QSI

4.2.1 Implementing the Perfect Discrimination of Unitary Operators on IBMQ Cloud Quantum Computer

4.2.1.1 Introduction

Any two unitary operators from a fixed finite set can be discriminated is a useful property of quantum information realm. Recently, IBMQ has released their public accessible quantum computers based on superconductor which have high fidelity both on single qubit (Gate error is under 7‰) and paired qubits (Multqubits Error is under 12.28%). We experimentally demonstrate discrimination procedure by programming the schemes on *ibmqx4* quantum computer with QISKit and QSI modules. Sequential and parallel discrimination schemes are described and both of them reach success at least 86% probability. The results show that the sequential scheme is still more accurate than the parallel one under the condition of superconductor quantum computer. From the procedure, we can see that experimental efficiency can be improved significantly by the programming tools.

4.2.1.2 Description of the Experiments

The Discrimination Schemes

The Parallel Schemes As described in [36, 62], the perfect discrimination protocols for arbitrary two unitary operators require an N-partite ($N < \infty$) quantum states $|\Psi\rangle$ as the input. $|\Psi\rangle$ is chosen to satisfy that $U^{\otimes N}|\Psi\rangle$ and $V^{\otimes N}|\Psi\rangle$ are orthogonal. To identify the unknown unitary operator, we perform the measurement $\{M_0 = U^{\otimes N}|\Psi\rangle\langle\Psi|(U^{\otimes N})^\dagger, M_1 = V^{\otimes N}|\Psi\rangle\langle\Psi|(V^{\otimes N})^\dagger\}$. If the outcome is 0, we assert that the unknown operator is U; otherwise the unknown operator is V.

In our setting, we set $U = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{2}{3}i\pi} \end{bmatrix}$ and $V = I_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. N can be chosen as 2 and the input state is chosen as

$$|\Psi\rangle = (\frac{1}{\sqrt{3}}|0\rangle + \frac{1}{\sqrt{6}}|1\rangle) \otimes |0\rangle + (-\frac{1}{\sqrt{6}}|0\rangle + \frac{1}{\sqrt{3}}|1\rangle) \otimes |1\rangle \ . \tag{4.1}$$

It is easy to verify that

$$U^{\otimes 2}|\Psi\rangle = (\frac{1}{\sqrt{3}}|0\rangle + \frac{e^{\frac{2}{3}i\pi}}{\sqrt{6}}|1\rangle)\otimes|0\rangle + (-\frac{e^{\frac{2}{3}i\pi}}{\sqrt{6}}|0\rangle + \frac{e^{\frac{4}{3}i\pi}}{\sqrt{3}}|1\rangle)\otimes|1\rangle\ ,$$

and $\langle\Psi|U^{\otimes 2}|\Psi\rangle = 0$. Thus, the measurement $\{M_0 = U^{\otimes 2}|\Psi\rangle\langle\Psi|(U^{\otimes 2})^\dagger, M_1 = |\Psi\rangle\langle\Psi|\}$ is sufficient to achieve the perfect discrimination.

The Sequential Schemes As described in [38], arbitrary two unitary operators U and V can be distinguished without entanglement, albeit additional unitary operators are required. Explicitly, an input state $|\Phi\rangle$ is needed, and a finite number of ancillary unitary operators $X_1, \dots, X_{N-1}$ are required, such that $UX_1U\cdots UX_{N-1}U|\Phi\rangle$ and $VX_1V\cdots VX_{N-1}V|\Phi\rangle$ can be distinguished perfectly.

In our settings, only 1 ancillary operator is needed, which is

$$X = \begin{bmatrix} \frac{\sqrt{2}}{\sqrt{3}} & -\frac{1}{\sqrt{3}} \\ \frac{1}{\sqrt{3}} & \frac{\sqrt{2}}{\sqrt{3}} \end{bmatrix}.$$

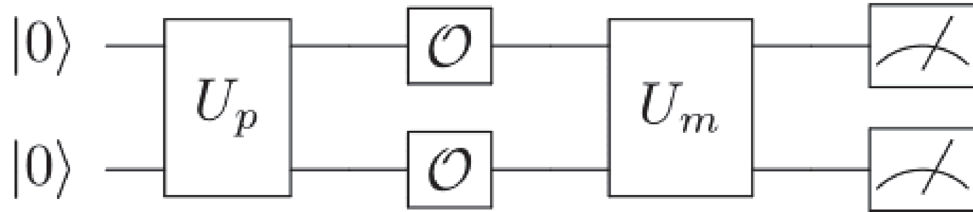
Notice that X is indeed the *rotation matrix* $\begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}$ with $\alpha = \arctan(1/\sqrt{2})$. Moreover, we can choose the input as

$$|\Phi\rangle := \frac{1}{\sqrt{2}}(|\varphi_0\rangle + |\varphi_1\rangle)\,, \tag{4.2}$$

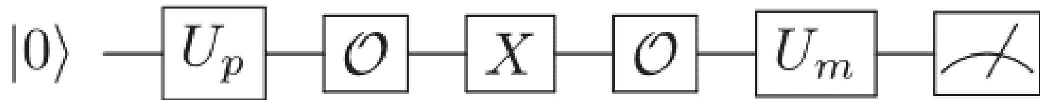
where $|\varphi_0\rangle$ and $|\varphi_1\rangle$ are the eigenvectors of

$$\begin{aligned} X^\dagger UXU &= \begin{bmatrix} \frac{2}{3} + \frac{1}{3}e^{\frac{2}{3}i\pi} & -\frac{\sqrt{2}}{3}e^{\frac{2}{3}i\pi} + \frac{\sqrt{2}}{3}e^{\frac{4}{3}i\pi} \\ -\frac{\sqrt{2}}{3} + \frac{\sqrt{2}}{3}e^{\frac{2}{3}i\pi} & \frac{1}{3}e^{\frac{2}{3}i\pi} + \frac{2}{3}e^{\frac{4}{3}i\pi} \end{bmatrix} \\ &= \begin{bmatrix} \frac{1}{2} + \frac{\sqrt{3}i}{6} & -\frac{\sqrt{6}i}{3} \\ -\frac{\sqrt{2}}{2} + \frac{\sqrt{6}i}{6} & -\frac{1}{2} - \frac{\sqrt{3}i}{6} \end{bmatrix}. \end{aligned}$$

Eventually, we perform the measurement $\{M_0 = UXU|\Phi\rangle\langle\Phi|U^\dagger X^\dagger U^\dagger, M_1 = X|\Phi\rangle\langle\Phi|X^\dagger\}$. As result of 0 implies the unknown operator is U, while a result of 1 implies the unknown operator is I.



(A) The parallel scheme to distinguish the unknown operator $\mathcal{O} \in \{U, I\}$, where U_p and U_m indicate the state preparation and measurement circuits.



(B) The sequential scheme to distinguish the unknown operator $\mathcal{O} \in \{U, I\}$, where U_p and U_m indicate the state preparation and measurement circuits.

FIGURE 4.19: Parallel and sequential discrimination schemes

Although compute the input states (and ancillary operators) is fundamental with unitary operators for both parallel and sequential discriminations, this is not a straightforward process with experiments on IBM's quantum processor. On the one side, suitable unitary operators are required to produce the desired input states (Eq. 4.1 and Eq. 4.2) from the state $|0\rangle$, which is the only quantum state initiated by IBM's quantum processor. On the other side, IBM's quantum processor is only able to measure quantum states on computational basis for each register. Thus, we need additional unitary operators to rotate the desired measurements. Specifically, the measurement performed in parallel discrimination needs to be *localised* to fulfil the requirements of IBM's quantum processor.

Implementation details The parallel and sequential discrimination schemes in Figure 4.19a and Figure 4.19b, respectively. Note that the unitary operator $U = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{2}{3}i\pi} \end{bmatrix}$ is a simple phase rotation gate (with the parameter $\lambda = \frac{2}{3}\pi$) and can be generated either by *QISKit* [42] or directly in *Composer* advanced mode of IBM Q Experience. In general, *QISKit* can be used to implement all single-qubit unitary gates, parameterised as

$$\begin{aligned} U(\theta,\phi,\lambda) &\coloneqq R_z(\phi)R_y(\theta)R_z(\lambda) \\ &= \begin{bmatrix} e^{-i(\phi+\lambda)/2}\cos(\theta/2) & -e^{-i(\phi-\lambda)/2}\sin(\theta/2) \\ e^{i(\phi-\lambda)/2}\sin(\theta/2) & e^{i(\phi+\lambda)/2}\cos(\theta/2) \end{bmatrix}, \end{aligned}$$

on the quantum processor with gate fidelity of around 99.9% (depending on the status of the processor). Whereas, IBM's quantum processor can only provide the ground (qubit) state $|0\rangle$, and measures the output with respect to a computational basis of $\{|0\rangle\langle 0|, |1\rangle\langle 1|\}$ for each register. Such limitations require us to generate the input state preparation circuits (U_p's in Figure 4.19a and 4.19b) and rotate the measurement on a computational basis (U_m's in Figure 4.19a and 4.19b). In the sequential scheme (Figure 4.19b, U_p and U_m can be chosen as two qubits rotation gates: To convert $|0\rangle$ to the input $|\Phi\rangle$, computed by Eq. 4.2, one can choose $U_p = U(1.1503, 6.4850, 2.2555)$ and $U_m = U(0.7854, 6.0214, 6.1913)$ [1]. Implementing the circuit in Figure 4.19b and measuring the output state, we say $\mathcal{O}$ is U if the (measurement) output is 0, and $\mathcal{O}$ is I if the output is 1.

While in the parallel scheme, to prepare the input state $|\Psi\rangle$, computed in Eq. 4.1, we use the circuit shown in Figure 4.20. However, in the measurement step, as the two possible outputs are *bipartite* qubit states, we cannot directly perform an *entangled measurement* to distinguish them, due to the limitation of IBM's quantum processor. Fortunately, any two multipartite orthogonal quantum states can always be distinguished locally [63]. We use a circuit for this with the computational basis, shown in Figure 4.21. Implementing the circuit in Figure 4.19a and measuring the output state, we say $\mathcal{O}$ is U if the (measurement) output is 01 or 10, and $\mathcal{O}$ is I if the output is 00 or 11.

[1] We mention that the software platform *QSI* provides a toolbox to compute the parameters for arbitrary qubit unitary matrices.

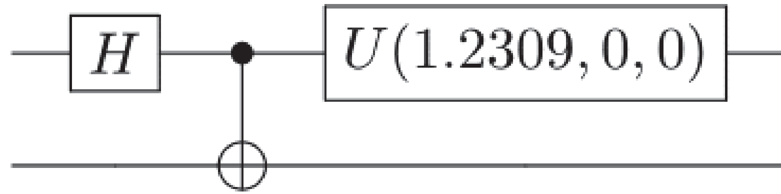


FIGURE 4.20: The quantum circuit (U_p) generates $|\Psi\rangle$ from $|0\rangle \otimes |0\rangle$.

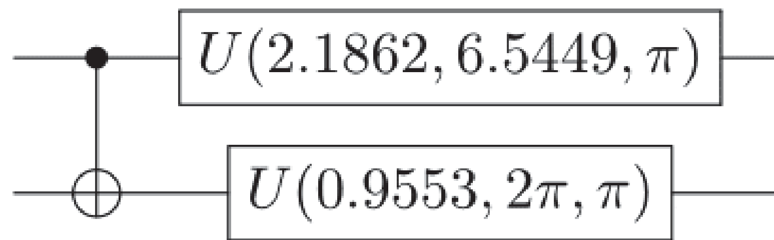


FIGURE 4.21: The quantum circuit (U_m) distinguishes $U^{\otimes 2}|\Psi\rangle$ and $|\Psi\rangle$.

4.2.1.3 The experiments

We performed a series of discrimination experiments with IBM's quantum processor *ibmqx4* and generated the circuits with *QSI* (the key code segments can be found at (`https://github.com/klinus9542/UnitaryDistIBMQ`)) . First, we generated a uniformly random bit to select the identity of I and U, which *QSI* can easily accomplish. Then, we generated the discrimination protocols, as shown in Figure 4.19a and Figure 4.19b, replacing the gate $\mathcal{O}$ with the chosen gate with respect to the random bit. *QSI* provides all the single-qubit parameters for each gate along with access to the application programming interface (API) for IBM's quantum cloud service by transforming the quantum program into QASM. Prior to executing the experiment, *QSI* provides simulation and verification toolkits to check whether the programs are written correctly. For each random bit, we executed the schemes on *ibmqx4* 1024 times and gathered the measurement results.

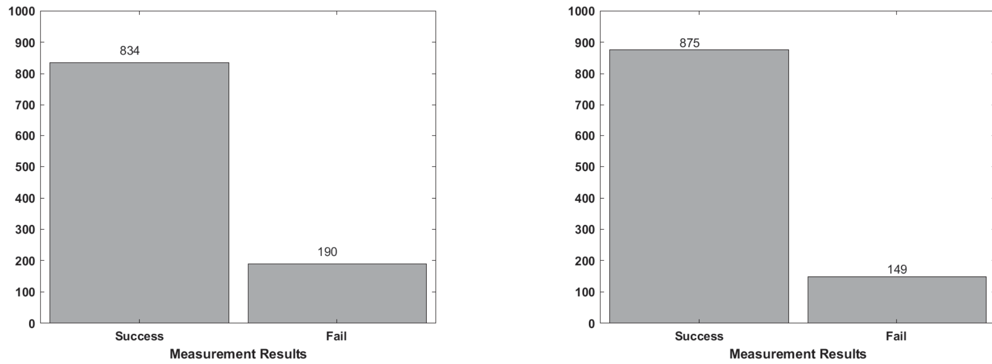
Based on the theoretical calculations, the identity of the unknown unitary operations should be uniquely determined. For instance, with the parallel scheme (Figure 4.19a) and $\mathcal{O}$ as the chosen U, the measurement outputs should only be 01 and 10 and appear an equal number of times. However, current quantum technologies may not be able to achieve theoretical performance standards. As mentioned, the fidelity of a single-qubit gate is 99.9%, which creates an unavoidable error. But another type of error arises from the state preparation circuits and measurement circuits. This is because the theoretical input states and the measurements contain irrational parameters presented by the float type in the software, which cannot be created accurately.

Also note that even measurement on real quantum computers must be synchronous for each single qubit because unused qubits measurement results would also be reported while only the results of the working qubits would be taken into account. For statistical purposes, other changes on unused qubits would be ignored.

Having applied the parallel and sequential discrimination schemes shown in 4.19a and Figure 4.19b, Figure 4.22a and Figure 4.22b show the statistical measurement results for the parallel discrimination scheme, and Figure 4.23a and Figure 4.23b show the statistical measurement results for sequential discrimination scheme. The results need to be sorted, as some 'impossible' results might appear. In principle, the statistical results might be $xy000$ when using the 5-qubit *ibmqx4* chip. However, the outputs may actually be arbitrary 5-bit strings given the errors between used qubits and unused qubits. Here, we ignore the unused qubits and sort the final results. In addition, the optimisation module in QISKit prevents results from objective gates and measurement error conditions, which may automatically combine neighbour gates into rotation gates depending on the strategy. We also use barrier functions to force the program to execute on the original circuits without combining gates.

Figure 4.24 illustrates the box-plot of success probabilities with the parallel and sequential schemes. In the experiment, replacing $\mathcal{O}$ by U or I depends on the value of the random bit by the classical computer. We performed each scheme 10 times with a randomly chosen $\mathcal{O}$, each of which includes 1024 repeating experiments. Both the worst (85.83%) and the best (98.63%) success probabilities came from the discrimination experiments using the sequential scheme with a standard deviation of $\sigma = 0.061$. The best success probability was

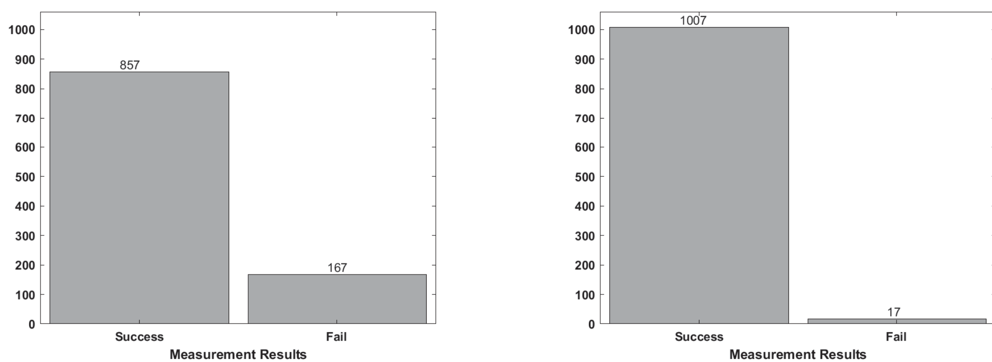
achieved when $\mathcal{O}$ was replaced by I. Hence, the discrimination scheme (in Figure 4.19b) contained only 3 qubit gates to perform. The worst success probability was achieved when $\mathcal{O}$ was replaced by U, where 5 (rather complicated) qubit gates needed to be performed on the inputs. With the parallel scheme, the success probabilities ranged from 88% to 92%, with not insignificant differences (standard deviation $\sigma = 0.017$).



(A) Perform the circuit in Figure 4.19a by replacing $\mathcal{O}$ by U for 1024 times. After sorting the outputs, 834 rounds output either 01 or 10 (indicates $\mathcal{O}$ is U), and 190 rounds output either 00 or 11 or other results (indicates $\mathcal{O}$ is not U).

(B) Perform the circuit in Figure 4.19a by replacing $\mathcal{O}$ by I for 1024 times. After sorting the outputs, 875 rounds output either 00 or 11 (indicate $\mathcal{O}$ is I), and 190 rounds output either 01 or 10 or other results (indicate $\mathcal{O}$ is not I).

FIGURE 4.22: Statistical results in the parallel discrimination experiments.



(A) Perform the circuit in Figure 4.19b by replacing $\mathcal{O}$ by U for 1024 times. After sorting the outputs, 857 rounds output 0 (indicate $\mathcal{O}$ is U), and 167 rounds output either 1 or other results (indicate $\mathcal{O}$ is not U).

(B) Perform the circuit in Figure 4.19b by replacing $\mathcal{O}$ by I for 1024 times. After sorting the outputs, 1007 rounds output 1 (indicate $\mathcal{O}$ is I), and 17 rounds output either 1 or other results (indicate $\mathcal{O}$ is not I).

FIGURE 4.23: Statistical results in the sequential discrimination experiments.

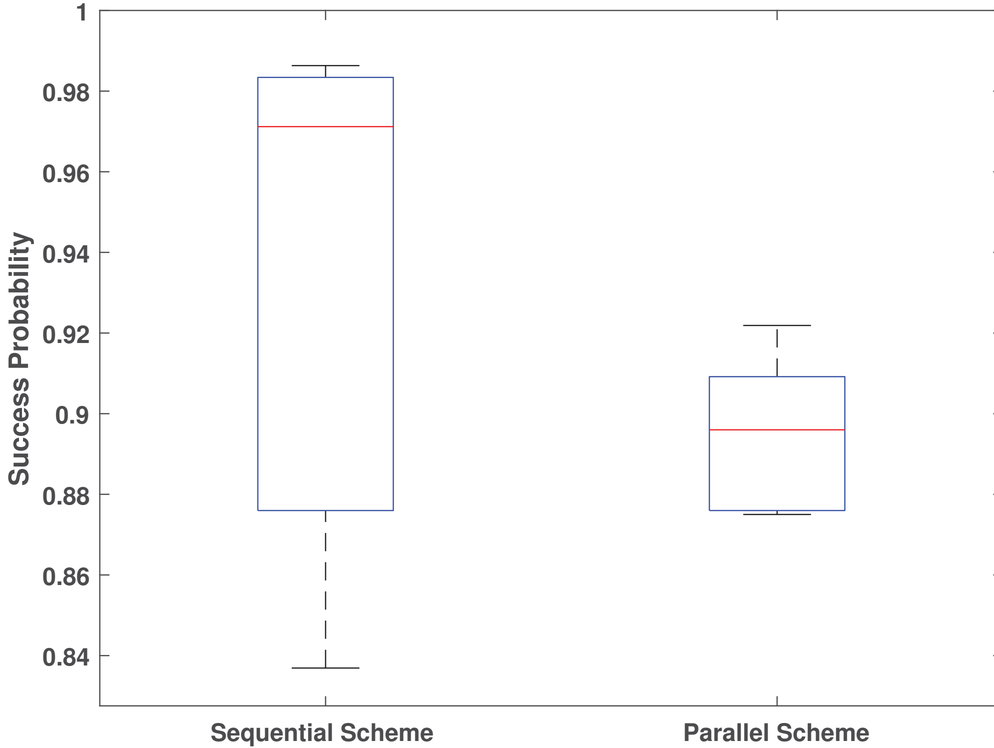


FIGURE 4.24: The discrimination success probability distributions. Both the sequential and parallel discrimination are shown here. For each round in each scheme, U and I was chosen depending from a random coin-flip. We execute the experiment for each scheme with 10 randomly chosen $\mathcal{O}$, and repeated the experiment 1024. In each box, the central mark indicates the median, and the top and the bottom indicate the 75% and 25% percentiles, respectively.

4.2.1.4 Conclusion and Discussion

In this sub-section, we used IBM 's quantum processor (*ibmqx4*) to discriminate between unitary operators, with the help of the $QISKit$ and QSI modules. The results shown in Figure 4.22 and Figure 4.23 indicate that both schemes were able to distinguish the qubit unitary gates U and I with rather high success probabilities, although not perfectly. In addition, we used QSI modules to perform 10 random experiments, where U and I were chosen at random and uniformly executed in each discrimination scheme. The result in Figure 4.24 suggests that both schemes were able to distinguish between the randomly chosen unitary gates with high probability.

Although we only considered two simple qubit unitary gates in these experiments, $U = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{2}{3}i\pi} \end{bmatrix}$ and $I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, the success probability distributions in Figure 4.24 implies that,

the sequential scheme may have produced higher success probabilities than the parallel scheme. However, the success probabilities with the parallel scheme are more robust. From this we conclude that when the set of known unitary operations have rather simple structures, such as an identity or Hadamard gate, a sequential scheme would be more suitable than a parallel scheme. Our assertion is due to the fact that the coherence and fidelity of two-qubit gates are still not ideal in IBM quantum processors. Further, while using a parallel discrimination scheme is more robust, it may not achieve a 99% success probability, but the success probabilities do not differ too much.

As the first implementation to distinguish quantum operations in a real quantum processor, we believe that our experiments provide evidence of how to choose parallel and sequential schemes in discrimination tasks. If the known quantum operation sets are not too complicated, sequential schemes are preferred and should generate higher success probabilities. However, if more robust results are required, parallel discrimination schemes may be more suitable. We have left the implementation of discrimination in general quantum operations as a further research direction.

4.2.2 Distinguishing Bell States with Local Measurement on IBMQ

4.2.2.1 Introduction and Preliminaries

It is well known that Bell state cannot be distinguished using local operations and classical communication (LOCC) in one-shot by one-copy. Nonlocality lays the heart of quantum information as it ensures that, even if a compound quantum system consisting of two parts and the quantum states are orthogonal, it is not always possible to reliably discover which state the system is in through local measurement or even LOCC. Bennett et al. [64]demonstrate that there are orthogonal sets of product states that cannot be reliably distinguished by a pair of separated observers no matter which of the states has been sent to them. Even if LOCC is allowed, these authors prove that a finite gap exists between mutual information by a joint measurement on the state and a measurement in which only local actions are permitted. Since their paper was published, some further studies have been conducted in the hope of distinguishing various orthogonal Bell states [63, 65, 66]. Virmani et al. [66] showed that if two copies of a Bell state are provided, they can be

distinguished by LOCC with certainty. Therefore, in this section, we demonstrate how to distinguish Bell states using two copies on an IBMQ quantum computer.

Bell States We assume two parties, Alice and Bob, share one side of a Bell entanglement state. The candidate Bell states are assumed to be $|\Phi_0\rangle_{AB} = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, $|\Phi_1\rangle_{AB} = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$, $|\Phi_2\rangle_{AB} = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$, $|\Phi_3\rangle_{AB} = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$.

Obviously, there is no method to distinguish $|\Phi_1\rangle$, $|\Phi_2\rangle$, $|\Phi_3\rangle$ and $|\Phi_4\rangle$ with LOCC with one copy.

Let's consider two copies. Let two copies of Bell states be defined as

$$|\Psi_0\rangle = |\Phi_0\rangle_{A_1,B_1} \otimes |\Phi_0\rangle_{A_2,B_2} = \frac{1}{2}(|0000\rangle + |0101\rangle + |1010\rangle + |1111\rangle)_{A_1A_2B_1B_2},$$

$$|\Psi_1\rangle = |\Phi_1\rangle_{A_1,B_1} \otimes |\Phi_1\rangle_{A_2,B_2} = \frac{1}{2}(|0011\rangle + |0110\rangle + |1001\rangle + |1100\rangle)_{A_1A_2B_1B_2},$$

$$|\Psi_2\rangle = |\Phi_2\rangle_{A_1,B_1} \otimes |\Phi_2\rangle_{A_2,B_2} = \frac{1}{2}(|0000\rangle - |0101\rangle - |1010\rangle + |1111\rangle)_{A_1A_2B_1B_2},$$

$$|\Psi_3\rangle = |\Phi_3\rangle_{A_1,B_1} \otimes |\Phi_3\rangle_{A_2,B_2} = \frac{1}{2}(|0011\rangle - |0110\rangle - |1001\rangle + |1100\rangle)_{A_1A_2B_1B_2}.$$

4.2.2.2 Protocol

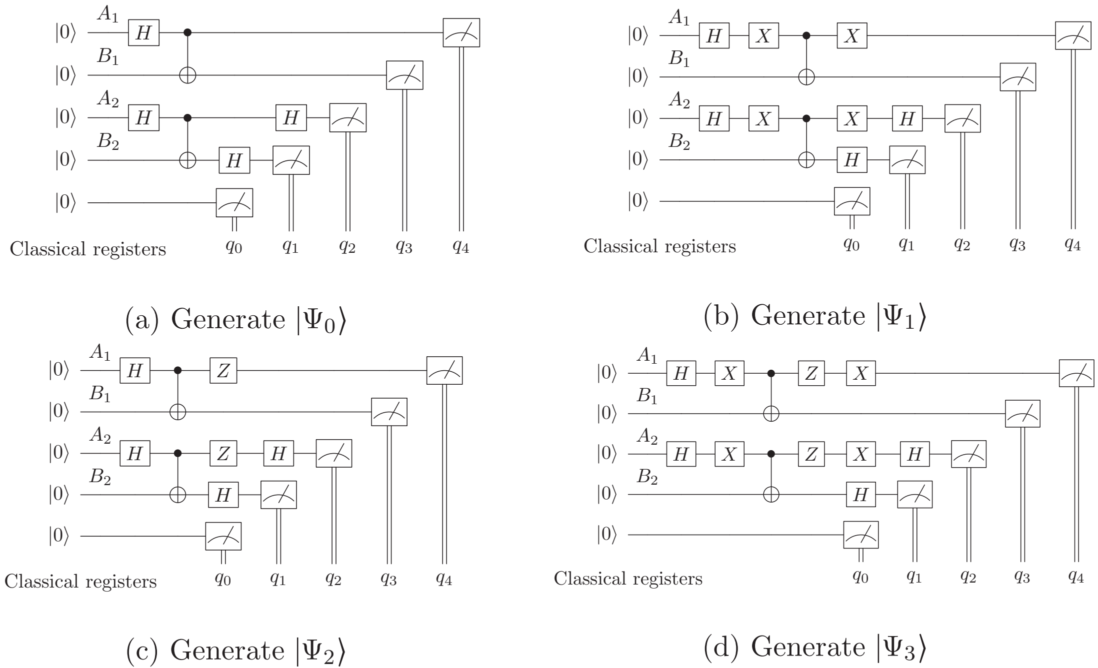
Charlie generates two copies one of the prepared states secretly, and then sends A_1 and A_2 to Alice and B_1 and B_2 to Bob. As only computational measurement can be performed on an IBMQ quantum computer, Alice performs a $\{|0\rangle, |1\rangle\}$ project measurement on A_1 and $\{|+\rangle, |-\rangle\}$ project measurement on A_2 while Bob performs a $\{|0\rangle, |1\rangle\}$ project measurement on B_1 and a $\{|+\rangle, |-\rangle\}$ project measurement on B_2.

4.2.2.3 Experiment

Quantum circuits on IBMqx4 We use the following circuits to generate the targeted Bell state.

4.2.2.4 Ideal Results

For convenience, we re-order the system from A_1, B_1, A_2, B_2 to A_1, B_1, A_2, B_2. The measurement results can easily be seen as binary results. The result for $\{|0\rangle, |+\rangle\}$ is denoted



(a) Generate $|\Psi_0\rangle$ (b) Generate $|\Psi_1\rangle$

(c) Generate $|\Psi_2\rangle$ (d) Generate $|\Psi_3\rangle$

FIGURE 4.25: Circuits for generating Bell states

TABLE 4.3: Ideal measurement results on two copies of a Bell state

	Possible Results			
$\lvert\Psi_0\rangle = \frac{1}{2}(\lvert 0000\rangle + \lvert 0101\rangle + \lvert 1010\rangle + \lvert 1111\rangle)_{A_1A_2B_1B_2}$	0011	0000	1100	1111
$\lvert\Psi_1\rangle = \frac{1}{2}(\lvert 0011\rangle + \lvert 0110\rangle + \lvert 1001\rangle + \lvert 1100\rangle)_{A_1A_2B_1B_2}$	0100	0111	1000	1011
$\lvert\Psi_2\rangle = \frac{1}{2}(\lvert 0000\rangle - \lvert 0101\rangle - \lvert 1010\rangle + \lvert 1111\rangle)_{A_1A_2B_1B_2}$	0001	0010	1110	1101
$\lvert\Psi_3\rangle = \frac{1}{2}(\lvert 0011\rangle - \lvert 0110\rangle - \lvert 1001\rangle + \lvert 1100\rangle)_{A_1A_2B_1B_2}$	0110	0101	1010	1001

the integer 0, and the result for $\{|1\rangle, |-\rangle\}$ is denoted as 1. The ideal results can therefore be expressed as a four-bit-length string set, as shown in Table 4.3.

4.2.2.5 Experimental Results

Two Copies The experiment was performed on *ibmqx4* universal quantum computer. Figure 4.26 shows the experimental results.

As shown, four surges exist for each state, even though the results are not balanced. Additionally, half the strings (16 of 32) have significant peaks, which indicates a corresponding quantum state. Moreover, as the errors may come from gate and measurement fidelity issues on a real quantum computer, half of the strings that should not emerge have also been provided within a small proportion.

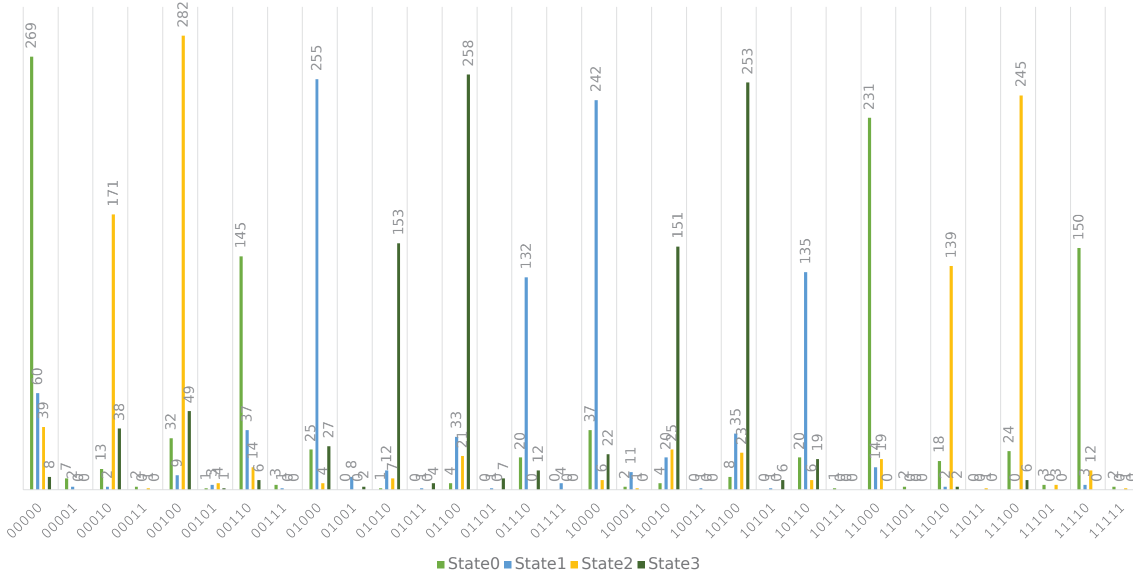


FIGURE 4.26: Measurement results with two copies of a Bell state. Charlie uniformly chooses one of Bell states from $|\Psi_1\rangle$, $|\Psi_2\rangle$, $|\Psi_3\rangle$ and $|\Psi_4\rangle$. In this example, Charlie exactly chooses 1024 shots for $|\Psi_1\rangle$, $|\Psi_2\rangle$, $|\Psi_3\rangle$ and $|\Psi_4\rangle$. For each state, there are four significant peaks of results and thus the results reflect the state Charlie chose.

One Copy One copy is not enough to distinguish which Bell state is present with LOCC. Therefore, we directly perform the measurement on two copies.

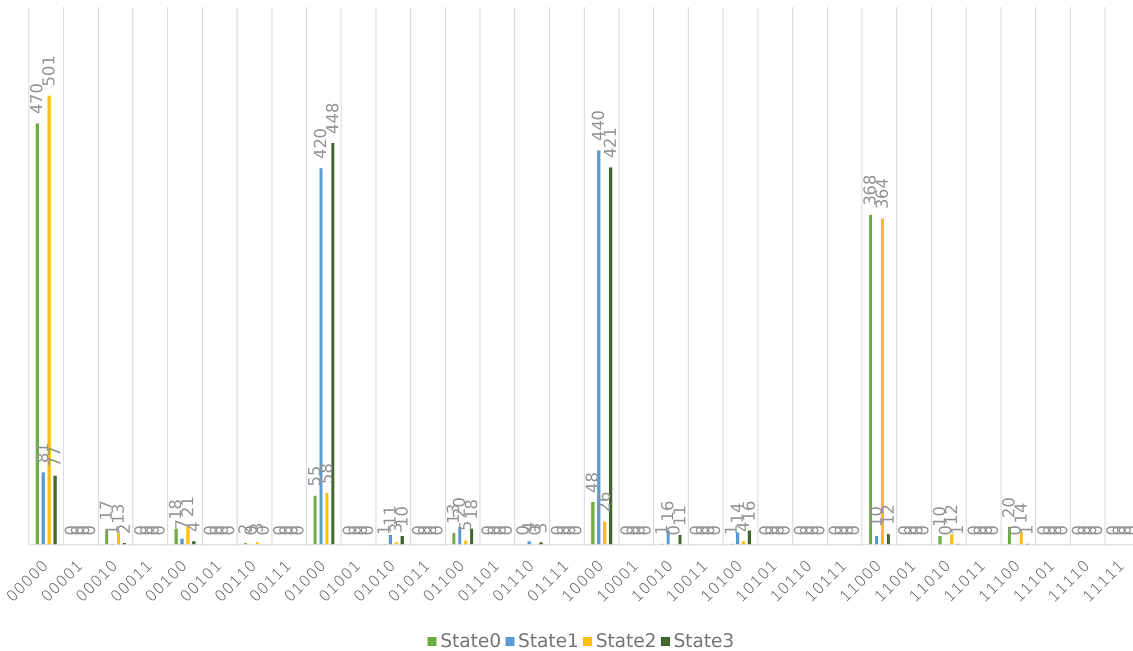


FIGURE 4.27: One copy experiment and computational measurement.

As Figure 4.27 shows, all four strings have two peaks which indicate the corresponding

state. Hence, when a result is given, we are still unsure about the corresponding state. For example, if Charlie secretly chooses $|\Psi_0\rangle$ as a black box and permits Alice and Bob to perform their measurements and classical communications, they still cannot ensure the prepared state is either $|\Psi_0\rangle$ or $|\Psi_2\rangle$ according to Figure 4.27.

4.3 Experiments with QISKit and LIQU$i|\rangle$

In this section, we focus on constructing QMUX for $[\![\mathbf{QIF}]\!]$. As mentioned in the last chapter, $\wedge_1(U)$ can be employed to simulate QMUX behaviour with a CNOT gate and a Z-Y rotation. Moreover, in the Hamilton mode of LIQU$i|\rangle$, R_y (Z rotation) and R_z (Y rotation) gates can be used as basic elements. The task is to denote the corresponding rotation angles and construct phase gates, say E_{phase} gate,

$$E_{\text{phase}} = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\alpha} \end{bmatrix}.$$

We have used the circuit shown Figure 4.28 to emulate any given circuit. Hence, once a controlled-unitary is simulated by rotation successfully, it should illustrate the ability to simulate any original unitary matrix that represents a $[\![\mathbf{QIF}]\!]$ behaviour.

4.3.1 Typical Cases

4.3.1.1 Hadamard Gate Case (H Gate)

Hadamard gates (H Gate) play an important role in quantum networks and algorithms. An H gate can be defined as converting $|0\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle \mapsto \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. We constructed a controlled-H gate with the input $|0\rangle$ or $|1\rangle$, ran the simulation several times and measured it using the computational basis M, in LIQU$i|\rangle$ to reflect the circuits. When the control qubit has the state $|1\rangle$ and the circuit works properly, the measurement results for target qubit should be almost half of 0 and half of 1.

After decomposition with the QSI module, the parameters were $\alpha = \pi/2, \beta = 0.000000, \gamma = -\pi/2, \delta = 0.000000$. The quantum circuit in Figure 4.28 includes a loop body to measure

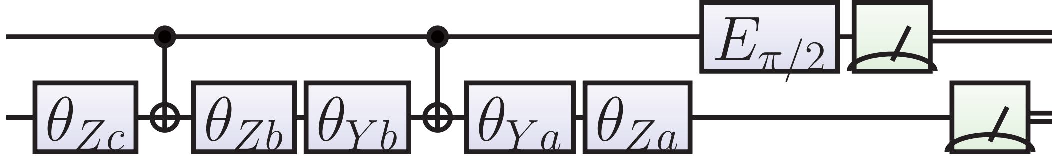


FIGURE 4.28: Controlled-unitary gate decomposition

the output state for 1000 shots. The experiment results are shown in Table 4.4 and clearly show the controlled-H gate was successfully simulated.

TABLE 4.4: Measurement results of controlled-Hadamard gate

Input State	Measurement Results of the Second Qubit
$\|0\rangle$, $\|0\rangle$	1000 times with a result of 0, no times with a result of 1
$\|0\rangle$, $\|1\rangle$	1000 times with a result of 1, no times with a result of 0
$\|1\rangle$, $\|0\rangle$	599 times with a result of 0, 401 times with a result of 1
$\|1\rangle$, $\|1\rangle$	470 times with a result of 0, 530 times with a result of 1

4.3.1.2 Identity Gate Case (I Gate)

An identity gate (I gate) should, theoretically, not have any effect on the quantum state in theoretical computation, for the specific computational basis, $|0\rangle \mapsto |0\rangle$ and $|1\rangle \mapsto |1\rangle$. We performed $|0\rangle$ or $|1\rangle$ through a simulated quantum circuit in Figure 4.28 for 1000 shots and then measured the result. In principle, the result should be similar to the input state exactly regardless the input state.

After decomposition with the QSI module, the parameters were $\alpha = 0.000000, \beta = 0.000000, \gamma = 0.000000, \delta = 0.000000$. The experiment results appear in Table 4.5. As shown, the I gate composited by the rotation gates worked as accurately as the original form.

TABLE 4.5: Measurement results of controlled-identity gate

Input State	Measurement Result of the Second Qubit
$\|0\rangle$, $\|0\rangle$	1000 times with a result of 0, no times with a result of 1
$\|0\rangle$, $\|1\rangle$	1000 times with a result of 1, no times with a result of 0
$\|1\rangle$, $\|0\rangle$	1000 times with a result of 0, no times with a result of 1
$\|1\rangle$, $\|1\rangle$	1000 times with a result of 1, no times with a result of 0

4.3.1.3 Bit-flip Gate Case (X Gate)

A bit-flip gate (X gate) flips the state in quantum circuit as well as flipping the computational basis, e.g., $|0\rangle \mapsto |1\rangle$ and $|1\rangle \mapsto |0\rangle$. We perform $|0\rangle$ or $|1\rangle$ through the simulated quantum circuit in Figure 4.28 for 1000 shots and measured the results. In principle, once a controlled-qubit is state $|1\rangle$, the result should be exactly the opposite to the input state.

After decomposition with the QSI module, the parameters were $\alpha = \pi/2, \beta = -\pi, \gamma = \pi, \delta =$ 0.000000. The results are shown in Table 4.6.

TABLE 4.6: Measurement result of controlled-Bit flip gate

Input State	Measurement Results of the Second Qubit
$\|0\rangle$, $\|0\rangle$	1000 times with a result of 0, no times with a result of 1
$\|0\rangle$, $\|1\rangle$	1000 times with a result of 1, no times with a result of 0
$\|1\rangle$, $\|0\rangle$	1000 times with a result of 1, no times with a result of 0
$\|1\rangle$, $\|1\rangle$	1000 times with a result of 0, no times with a result of 1

4.3.2 Extended 2-qubit QMUX

n-qubit QMUXs and its equivalences can be multiplexed with the result from the articles [67, 68]. The results from the Theorem 8 in Shende et al.'s work [47] can also be applied.

Theorem 4.1 (Demultiplexing R_k)**.** *Demultiplexing multiplexed rotation gates (R_k gates, where $k = y, z$) with the circuit as shown in Figure 4.29.*

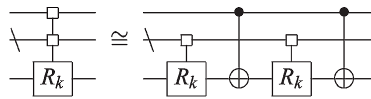


FIGURE 4.29: Demultiplexing multiplexed rotation gates

Chapter 5

Summary

5.1 Summary of Contributions

In this thesis, we show an explicit approach to a quantum programming environment including its architecture and implementation.

First, we researched and developed the primary framework as the key fundamental component. A new layered embedding language was presented based on a flexible structure that adapts to the most powerful quantum language – the quantum while-language. We designed a new compiler for connecting software to real or virtual quantum hardware and researched an inherited low-level instruction set called f-QASM. An algorithm within the framework for simulating quantum programs, including loop behaviours, is also presented.

Second, we developed several modules for the programming environment. The proposed parallel modules and approximate decomposition protocols for extensions to the separation unitary problem. The protocol enables the parallel computation under the approximate meaning. The termination modules were transformed from pure theoretical estimating rules into actual efficiency modules for checking a quantum program's ability to terminate. The modules include a new and improved algorithm to significantly accelerate the procedure. The quantum control modules ensure that QuGCL is feasible on current quantum hardware.

Lastly, we prepared a range of experiments to demonstrate the capabilities and flexibility of the QSI programming environment. For example, we show how the platform is able to support BB84 protocol, Grover's search algorithm, etc. Several experiments on unitary discrimination and Bell-state discrimination are included by connecting the QISKit with IBMQ. We also demonstrate how a quantum language can be combined with classical functional programming, i.e., Microsoft's LIQU$i|\rangle$ as an illustration of QMUX's construction, which lays at the heart of QuGCL.

5.2 Future work

Developing a quantum programming environment is a state-of-the-art research topic that combines the most recent theoretical results in the quantum realm with industry requirements. We anticipate that as breakthroughs occur with the development of quantum hardware, physical implementation and linkage toolkits will be in high demand. In addition, there are likely to be many types of quantum computer, each with their own related standards, and each will require compatible toolkits. Hence, there is much work still to do.

Architecture In its current stage of development, QSI is an academic prototype constructed to demonstrate concepts, algorithms, and experimental results. In future, it may need to be reconstructed. This version of QSI is based on several academic articles research group has published over the past few years. However, as new functionality is developed, new modules will be added and QSI will become more and more complicated. Therefore, an amount of time will be required to adjust "past code" when each new module added.

Architecture and design are critical components of long-lasting software. In the field of quantum computing and the integration of classical systems into a programming environment, the following aspects of research require particular attention: mixed quantum language models and compilation techniques, the legitimacy of mixed quantum procedures and other procedural automatic inspection tools, abstract description of new quantum devices, and the characteristics of the simulator.

Future versions of QSI are intended to integrate C# and the quantum while-language. By further embedding specialised languages into the quantum domain, our hope is to provide a platform that is easy-to-program, more efficient, of a higher level, and can support a next-generation language. This next-generation language would focus the representations and semantics of quantum data structures, quantum gate descriptions, quantum control clauses, and quantum computation commands. Interactions with classical language and storage will also be considered.

Additionally, a layered compiler will be required to transplant a singleton model (QASM) written in C# into an open LLVM structure, where a collection of modular and reusable compiler and toolchain technologies could be used to develop both the front-and back-ends. LLVM provides a unified compiler structure, which opens opportunities to use a draw on the contributions of a range of advanced compiler tool-chains for many aspects of a new quantum compiler. Using LLVM also requires a robust quantum intermediate representation (QIR) that is similar to the classical process. The QIR needs to be designed as a general and efficient intermediate language that can completely describe the data and program. All the outputs would depend on a unified data structure in memory. The unified data structure would provide independent program information for each isolated quantum device or quantum hardware, and would include specific device information to generate compatible machine code at a low level.

Computation Engine To improve efficiency, the "computation engine" in QSI needs to be separated from other trivial components. The calculation engine may need to be rewritten in C or C++ using parallel technologies to meet performance standards. We also need to implement a corresponding calculation engine on a supercomputer or in the cloud. Further, vector machine technology, such as CUDA or OpenCL, have shown significant speed improvements when handling massive amounts of data in current AI applications. This technology is also a potential direction to pursue in designing an advanced simulator and engine.

Applications Beyond pure quantum algorithms, such as Grover's search and Shor's factoring algorithm, there are still many applications to be built for different purposes.

Embedding a zoo of quantum algorithms into a solver would combine elegant classical algorithms with a sweep of applications for the quantum era.

Cross-platform Developing QSI also needs to be separated from the .NET Framework or, as a short-term measure, its front-end needs to be separated, given that potential users will be running many different operating systems, e.g., Mac OS (Apple), Linux, UNIX, etc. The .NET Framework is not a good cross-platform solution for this project. Even though it is possible to transplant the programming environment into other platforms, crashes and instability are still common, which seriously detracts from a good user experience. Additionally, the setup configurations required are so tedious and unbearable that they present a solid barrier for most developers. QSI needs to be more flexible. IBMQ is an online version of their quantum computer that offers "zero-configuration software technology" to create a composer that supports "drag and run ". This feature not only has wide market appeal, it also reduces the heavy load of studying syntaxes and semantics. We may need to consider developing this feature when developing QSI into the next stage quantum programming environment.

Bibliography

[1] Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):19, 2011.

[2] Mingsheng Ying. *Foundations of Quantum Programming*. Morgan Kaufmann, 2016.

[3] Peter W Shor and John Preskill. Simple proof of security of the bb84 quantum key distribution protocol. *Physical review letters*, 85(2):441, 2000.

[4] 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. ACM, 1996.

[5] Aram W Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical review letters*, 103(15):150502, 2009.

[6] M Veldhorst, CH Yang, JCC Hwang, W Huang, JP Dehollain, JT Muhonen, S Simmons, A Laucht, FE Hudson, KM Itoh, et al. A two-qubit logic gate in silicon. *Nature*, 526(7573):410–414, 2015.

[7] Cody Jones, Mark F Gyure, Thaddeus D Ladd, Michael A Fogarty, Andrea Morello, and Andrew S Dzurak. A logical qubit in a linear array of semiconductor quantum dots. *arXiv preprint arXiv:1608.06335*, 2016.

[8] IBM QX team. ibmqx2 backend specification. Retrieved from https://ibm.biz/qiskit-ibmqx2, 2017.

[9] IBM QX team. ibmqx3 backend specification. Retrieved from https://ibm.biz/qiskit-ibmqx3, 2017.

[10] Mingsheng Ying and Yuan Feng. Quantum loop programs. *Acta Informatica*, 47(4): 221–250, 2010.

[11] Krista Svore, Andrew Cross, Alfred Aho, Isaac Chuang, and Igor Markov. Toward a software architecture for quantum computing design tools. In *Proceedings of the 2nd International Workshop on Quantum Programming Languages (QPL)*, pages 145–162, 2004.

[12] Mingsheng Ying, Nengkun Yu, Yuan Feng, and Runyao Duan. Verification of quantum programs. *Science of Computer Programming*, 78(9):1679–1700, 2013.

[13] Bernhard Ömer. A procedural formalism for quantum computing. 1998.

[14] Peter Selinger. A brief survey of quantum programming languages. In *International Symposium on Functional and Logic Programming*, pages 1–6. Springer, 2004.

[15] Stefano Bettelli, Tommaso Calarco, and Luciano Serafini. Toward an architecture for quantum programming. *The European Physical Journal D-Atomic, Molecular, Optical and Plasma Physics*, 25(2):181–200, 2003.

[16] Jeff W Sanders and Paolo Zuliani. Quantum programming. In *International Conference on Mathematics of Program Construction*, pages 80–99. Springer, 2000.

[17] Alexander S Green, Peter LeFanu Lumsdaine, Neil J Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *ACM SIGPLAN Notices*, volume 48, pages 333–342. ACM, 2013.

[18] Dave Wecker and Krysta M Svore. Liquid: A software design architecture and domain-specific language for quantum computing. *arXiv preprint arXiv:1402.4467*, 2014.

[19] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. Scaffcc: Scalable compilation and analysis of quantum programs. *Parallel Computing*, 45:2–17, 2015.

[20] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T Chong, and Margaret Martonosi. Scaffcc: a framework for compilation and analysis of quantum computing programs. *Proceedings of the 11th ACM Conference on Computing Frontiers*, page 1, 2014.

[21] Mikhail Smelyanskiy, Nicolas PD Sawaya, and Alán Aspuru-Guzik. qhipster: the quantum high performance software testing environment. *arXiv preprint arXiv:1601.07195*, 2016.

[22] Simon J Devitt. Performing quantum computing experiments in the cloud. *Physical Review A*, 94(3):032329, 2016.

[23] Adriano Barenco, Charles H Bennett, Richard Cleve, David P DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52(5):3457, 1995.

[24] Christopher M Dawson and Michael A Nielsen. The solovay-kitaev algorithm. *arXiv preprint quant-ph/0505030*, 2005.

[25] Juliana Kaizer Vizzotto and Antônio Carlos da Rocha Costa. Concurrent quantum programming in haskell. In *VII Congresso Brasileiro de Redes Neurais, Sessao de Computaçao Quântica*. Citeseer, 2005.

[26] Nengkun Yu and Mingsheng Ying. Reachability and termination analysis of concurrent quantum programs. In *International Conference on Concurrency Theory*, pages 69–83. Springer, 2012.

[27] Simon J Gay and Rajagopal Nagarajan. Communicating quantum processes. In *ACM SIGPLAN Notices*, volume 40, pages 145–157. ACM, 2005.

[28] Yuan Feng, Runyao Duan, Zhengfeng Ji, and Mingsheng Ying. Probabilistic bisimulations for quantum processes. *Information and Computation*, 205(11):1608–1639, 2007.

[29] Mingsheng Ying. π-calculus with noisy channels. *Acta Informatica*, 41(9):525–593, 2005.

[30] Philippe Jorrand and Marie Lalire. Toward a quantum process algebra. In *Proceedings of the 1st conference on Computing frontiers*, pages 111–119. ACM, 2004.

[31] François Bourdoncle. Abstract debugging of higher-order imperative languages. *ACM SIGPLAN Notices*, 28(6):46–55, 1993.

[32] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *ACM SIGPLAN Notices*, volume 41, pages 415–426. ACM, 2006.

[33] Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. T2: Temporal property verification. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 387–393. Springer, 2016.

[34] Yangjia Li, Nengkun Yu, and Mingsheng Ying. Termination of nondeterministic quantum programs. *Acta informatica*, 51(1):1–24, 2014.

[35] Yangjia Li and Mingsheng Ying. Algorithmic analysis of termination problems for quantum programs. *Proceedings of the ACM on Programming Languages*, 2(POPL): 35, 2017.

[36] Antonio Acin. Statistical distinguishability between unitary operations. *Physical review letters*, 87(17):177901, 2001.

[37] G Mauro D'Ariano, Paoloplacido Lo Presti, and Matteo GA Paris. Using entanglement improves the precision of quantum measurements. *Physical review letters*, 87 (27):270404, 2001.

[38] Runyao Duan, Yuan Feng, and Mingsheng Ying. Entanglement is not necessary for perfect discrimination between unitary operations. *Physical review letters*, 98(10): 100503, 2007.

[39] Liu Jian-Jun and Hong Zhi. Experimental realization of perfect discrimination for two unitary operations. *Chinese Physics Letters*, 25(10):3663, 2008.

[40] Pei Zhang, Liang Peng, Zhi-Wei Wang, Xi-Feng Ren, Bi-Heng Liu, Yun-Feng Huang, and Guang-Can Guo. Linear optical implementation of perfect discrimination between single-bit unitary operations. *Journal of Physics B: Atomic, Molecular and Optical Physics*, 41(19):195501, 2008.

[41] Anthony Laing, Terry Rudolph, and Jeremy L O'Brien. Experimental quantum process discrimination. *Physical review letters*, 102(16):160502, 2009.

[42] Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. Open quantum assembly language. *arXiv preprint arXiv:1707.03429*, 2017.

[43] Shusen Liu, Xin Wang, Li Zhou, Ji Guan, Yinan Li, Yang He, Runyao Duan, and Mingsheng Ying. Qsi: a quantum programming environment. *arXiv preprint arXiv:1710.09500*, 2017.

[44] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[45] Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS'05)*, pages 249–258. IEEE, 2005.

[46] Mingsheng Ying, Nengkun Yu, and Yuan Feng. Alternation in quantum programming: from superposition of data to superposition of programs. *arXiv preprint arXiv:1402.5172*, 2014.

[47] V.V. Shende, S.S. Bullock, and I.L. Markov. Synthesis of quantum-logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1000–1010, jun 2006. doi: 10.1109/tcad.2005.855930.

[48] Krysta Marie Svore, Alfred V Aho, Andrew W Cross, Isaac Chuang, and Igor L Markov. A layered software architecture for quantum computing design tools. *IEEE Computer*, 39(1):74–83, 2006.

[49] Tao Liu, Yangjia Li, Shuling Wang, Mingsheng Ying, and Naijun Zhan. A theorem prover for quantum hoare logic and its applications. *arXiv preprint arXiv:1601.03835*, 2016.

[50] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge University Press, 2010.

[51] Mingsheng Ying and Yuan Feng. A flowchart language for quantum programming. *IEEE Transactions on Software Engineering*, 37(4):466–485, 2011.

[52] Robert S Smith, Michael J Curtis, and William J Zeng. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355*, 2016.

[53] Jinchuan Hou. On the tensor products of operators. *Acta Mathematica Sinica*, 2:010, 1993.

[54] John B Conway. *A course in functional analysis*, volume 96. Springer Science & Business Media, 2013.

[55] Fernando Casas Pérez, Ander Murua, and Mladen Nadinic. Efficient computation of the zassenhaus formula. 2012.

[56] Wilhelm Magnus. On the exponential solution of differential equations for a linear operator. *Communications on pure and applied mathematics*, 7(4):649–673, 1954.

[57] Nicholas Wheeler. Quantum theory of 2-state systems. http://www.reed.edu/physics/faculty/wheeler/documents/Quantum

[58] Gabriel Larotonda. Norm inequalities in operator ideals. *Journal of Functional Analysis*, 255(11):3208–3228, 2008.

[59] Vern Paulsen. *Completely bounded maps and operator algebras*, volume 78. Cambridge University Press, 2002.

[60] Theo Beelen and Paul Van Dooren. Computational aspects of the jordan canonical form. *Reliable Numerical Computation*, pages 57–72, 1990.

[61] Charles H Bennett and Gilles Brassard. Quantum cryptography: Public key distribution and coin tossing. *Theoretical computer science*, 560:7–11, 2014.

[62] Runyao Duan, Cheng Guo, Chi-Kwong Li, and Yinan Li. Parallel distinguishability of quantum operations. In *Information Theory (ISIT), 2016 IEEE International Symposium on*, pages 2259–2263. IEEE, 2016.

[63] Jonathan Walgate, Anthony J Short, Lucien Hardy, and Vlatko Vedral. Local distinguishability of multipartite orthogonal quantum states. *Physical Review Letters*, 85 (23):4972, 2000.

[64] Charles H Bennett, David P DiVincenzo, Christopher A Fuchs, Tal Mor, Eric Rains, Peter W Shor, John A Smolin, and William K Wootters. Quantum nonlocality without entanglement. *Physical Review A*, 59(2):1070, 1999.

[65] Sibasish Ghosh, Guruprasad Kar, Anirban Roy, Aditi Sen, Ujjwal Sen, et al. Distinguishability of bell states. *Physical review letters*, 87(27):277902, 2001.

[66] Shashank Virmani, Massimiliano F Sacchi, Martin B Plenio, and Damian Markham. Optimal local discrimination of two multipartite pure states. *Physics Letters A*, 288 (2):62–68, 2001.

[67] Stephen S Bullock and Igor L Markov. Smaller circuits for arbitrary n-qubit diagonal computations. *arXiv preprint quant-ph/0303039*, 2003.

[68] Mikko Möttönen, Juha J Vartiainen, Ville Bergholm, and Martti M Salomaa. Quantum circuits for general multiqubit gates. *Physical review letters*, 93(13):130502, 2004.