arXiv:1710.09500v1 [quant-ph] 26 Oct 2017

$Q|SI\rangle$: A Quantum Programming Environment

Shusen Liu,[*] Xin Wang, Li Zhou, Ji Guan, Yinan Li, Yang He, Runyao Duan,[†] and Mingsheng Ying[‡]
Centre for Quantum Software and Information, Faculty of Engineering and Information Technology, University of Technology Sydney, NSW 2007, Australia

This paper describes a quantum programming environment, named $Q|SI\rangle$ [a]. It is a platform embedded in the .Net language that supports quantum programming using a quantum extension of the **while**-language. The framework of the platform includes a compiler of the quantum **while**-language and a suite of tools for simulating quantum computation, optimizing quantum circuits, and analyzing and verifying quantum programs. Throughout the paper, using $Q|SI\rangle$ to simulate quantum behaviors on classical platforms with a combination of components is demonstrated. The scalable framework allows the user to program customized functions on the platform. The compiler works as the core of $Q|SI\rangle$ bridging the gap from quantum hardware to quantum software. The built-in decomposition algorithms enable the universal quantum computation on the present quantum hardware.

Keywords: Quantum Programming, Quantum Compilation, Quantum Simulation, Quantum Program Analysis, Quantum Program Verification

[a] The software of $Q|SI\rangle$ is available at `http://www.qcompiler.com`.
[*] Shusen.Liu@student.uts.edu.au
[†] Runyao.Duan@uts.edu.au
[‡] Mingsheng.Ying@uts.edu.au; Also at Department of Computer Science and Technology, Tsinghua University, Beijing, China; Also at State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

I. INTRODUCTION

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 [1], Grover's search algorithm [2] and more recently Harrow, Hassidim and Lloyd's algorithm for systems of linear equations [3]. In recent years, governments and industries around the globe have been racing to build quantum computers. As happened 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 implement quantum algorithms in a quantum system is somewhat counterintuitive. Using a platform like $Q|SI\rangle$ could help programmers understand some of these features, which may help to (partially) avoid some of the errors.

Several quantum programming platforms have been developed in the last two decades. The first quantum programming language, QCL, was proposed by Ömer [4, 5] in 1998. It was implemented in C++. A very similar quantum programming language, Q language, was defined by Bettelli et al. [6] in 2003, which was implemented as a C++ library. In 2000, qGCL was introduced by Sanders and Zuliani [7] 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. [8] in 2013. This was implemented using Haskell as its host language. LIQU$i|\rangle$was developed in 2014 by Wecker and Svore from QuArc (the Microsoft Research Quantum Architecture and Computation team) [9] as a modern tool-set and is embedded in another functional programming language F#. In the same year, the quantum programming language Scafford was defined by JavadiAbhari et al. [10]. Its compilation system ScaffCC was developed in the article [11]. Smelyanskiy et al. [12] at Intel built a parallel quantum computing simulator qHiPSTER that can simulate up to 40 qubits on a supercomputer with very high performance.

Contributions of this paper: This paper presents a powerful and flexible new quantum programming environment called $Q|SI\rangle$ [1], named after our research center[2]. The *core* of $Q|SI\rangle$ is *a quantum programming language* and *its compiler*. This language is a quantum extension of the **while**-language. It was first defined in [13] along with a careful study of its operational and denotational semantics (see also [14], 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 [15]. A feedback instruction has been added that allows the efficient implementation of measurement-based case and loop statements. A compiler then transforms the quantum **while**-program into a sequence of f-QASM instructions and further generates a corresponding quantum circuit equivalent to the program (i.e., a sequence of executable

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

[2] http://www.qsi.uts.edu.au

quantum gates). $Q|SI\rangle$ also has a module for optimizing the quantum circuits as well as a module to simulate its quantum programs on a classical computer. Two novel features set $Q|SI\rangle$ apart from other existing quantum programming environments:

- *A quantum program analyzer*. Several algorithms for termination analysis and for computing the average running time of quantum programs were developed by one of the authors in [17, 26]. In addition, a semi-definite programming (SDP) algorithm generates invariants of quantum **while**-loops was developed by one of the authors in [18]. These algorithms have been implemented in $Q|SI\rangle$ for the static analysis of quantum programs. In turn, this program analyzer helps the compiler to optimize the implementation of quantum programs.

- *A quantum program verifier*. A logic in the Floyd-Hoare style was established in [13] (see also [14], Chapter 4). This logic, which reasons about the correctness of quantum programs, has been written in the quantum **while**-language. Recently, a theorem prover was implemented by Liu et al. [19] for quantum Floyd-Hoare logic based on Isabelle/HOL. We plan to link $Q|SI\rangle$ with the quantum theorem prover presented in [19] and provide this facility in our platform for the verification of quantum programs.

II. 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 $q_0, q_1, q_2, \ldots$. However, in practice, almost all existing quantum algorithms involve elements of both classical and quantum computation. Therefore, $Q|SI\rangle$ 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 [13] and Chapter 3 of [14]. 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 initialization 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.

III. THE STRUCTURE OF $Q|SI\rangle$

This section provides an introduction to the basic structure of $Q|SI\rangle$, leaving the details to be described in subsequent sections. $Q|SI\rangle$ 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 $Q|SI\rangle$ as a deeply embedded domain-specific platform for quantum programming using the host language C#.

$Q|SI\rangle$'s framework is shown in the Fig. 1.

A. Basic features of $Q|SI\rangle$

The main features of $Q|SI\rangle$ are explained as follows:

Language supporting: $Q|SI\rangle$ is the first platform to support the quantum **while**-language. In particular, 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, $Q|SI\rangle$ supports quantum types beyond pure qubit states, such as density operators, mixed states, etc. These types have unified operations and can be used in different scenarios. This feature provides high flexible usability and facilitates the programming process.

Dual mode: $Q|SI\rangle$ 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) [15], f-QASM is essentially a quantum circuit description language that can be adapted for a variety purpose. In this language, every line has only one command. f-QASM's 'goto' structure contains more information than the original QASM [15] or space efficient QASM-HL [11]. f-QASM can also be used for further optimization and analysis.

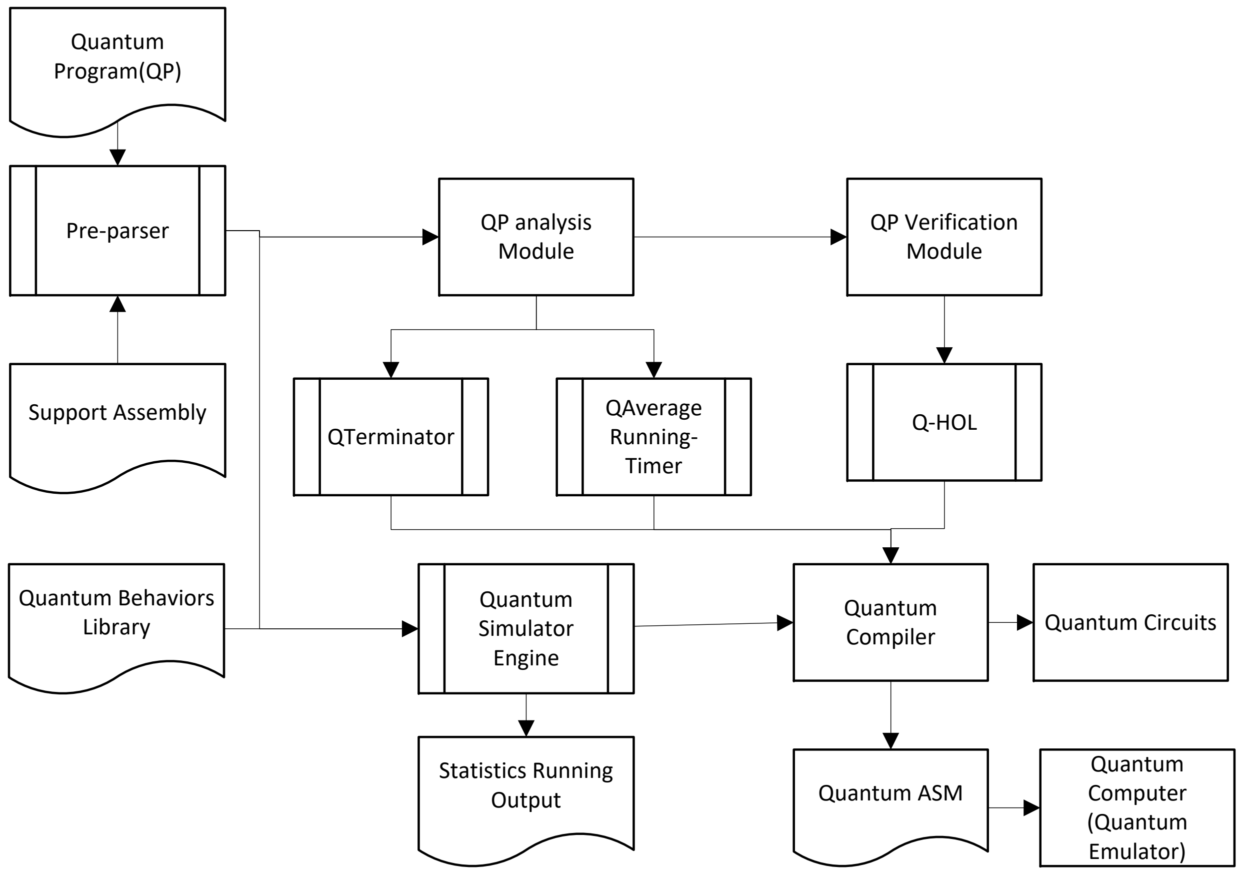


FIG. 1. Framework of $Q|SI\rangle$

Quantum circuits generation: Similar to modern digital circuits in classical computing, quantum circuits provide a low-level representation of quantum algorithms [15]. Our compiler can produce a quantum circuit from a program written in the high-level quantum **while**-language.

Arbitrary unitary operator implementation: The $Q|SI\rangle$ platform includes the Solovay-Kitaev algorithm [20] together with two-level matrix decomposition algorithm [21] and a quantum multiplexor (QMUX) algorithm [22]. 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, $Q|SI\rangle$ has a gate-by-gate description feature. Some basic quantum gates are provided inherently 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 two-dimensional controlled-unitary gates for emulation feasibility.

B. Main components of $Q|SI\rangle$

The $Q|SI\rangle$ platform mainly consists of four parts.

Quantum Simulation Engine: This component includes some support assemblies, a quantum mechanics library and a quantum simulator engine. The support assemblies provide supporting 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 which can be executed on a classical computer.

Quantum Program (QP) Analysis Module: This module currently contributes two sub-modules to 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 outputs are sent to the quantum compiler at the next stage for further usage.

QP Verification Module: This module is a tool for verifying the correctness of quantum programs. It is based on quantum Hoare logic, which was introduced by one of the authors in [13] and is still under development. One possibility for its future advancement is to link $Q|SI\rangle$ to the quantum theorem prover developed by Liu et al [19].

Quantum Compiler: The compiler consists of a series of tools to map a high-level source program representing a quantum algorithm into a quantum device related language [15], e.g., f-QASM and further into a quantum circuit. Our target 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. A tool to optimize the quantum circuits will be added to the compiler in the future.

C. Implementation of $Q|SI\rangle$

One of the basic problems during implementation is how to use probabilistic and classical algorithms to simulate quantum behaviors. To support quantum operations, $Q|SI\rangle$ has been enriched with data structures from a quantum simulation engine. Fig. 2 shows the procedure for simulating a quantum engine. Three types of languages are supported: pure quantum **while**-language, classical **while**-language and a mixed language. The engine starts a support flow path when it detects the quantum part of a program. Then, the engine checks the quantum type for each variable and operator and executes the corresponding support assembly. As mentioned, one of the main features of $Q|SI\rangle$ is that it supports programming in the quantum **while**-language. This feature is provided by the quantum **while**-language support assemblies. All of the quantum behaviors are explained by probabilistic algorithms on a classical computer. The outputs are extended C# languages which can be run on a .Net framework or can be explained in f-QASM and quantum circuits by the compiler.

The quantum simulation engine involves numerous matrix computations and operations. In $Q|SI\rangle$, 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

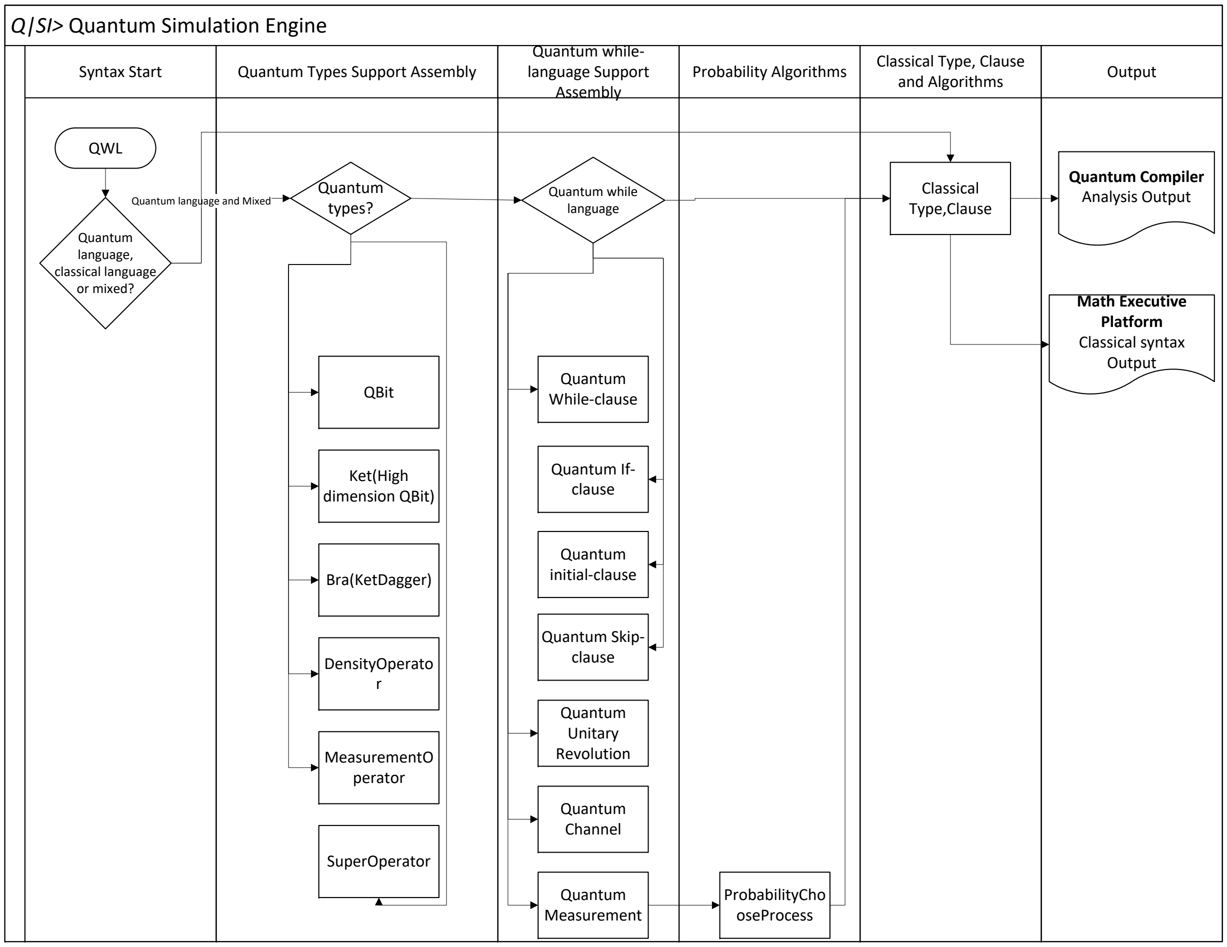


FIG. 2. $Q|SI\rangle$ Pre-parser

everyday and the advanced needs of .Net developers[3]. 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 auxiliary code analysis tool. Roslyn is a set of open-source compilers and code analysis APIs for C# and Basic languages. Since 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.

IV. THE QUANTUM COMPILER

A 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

[3] https://www.mathdotnet.com

or virtual quantum processor. It provides quantum circuits for quantum chip design. It also optimizes quantum circuits.

The $Q|SI\rangle$ compiler is heavily dependent on other modules. It collects data structures from the quantum simulation engine and splits the program into several parts: variables, quantum gates, quantum measurements, entry and exit points for each clause 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 $\{U_1, U_2, \dots, U_m\}$ where $U_1, U_2, \dots, U_m \in U(2)$ (qubit gate). This corresponds to a scenario in quantum device development: people need universal computation in spite of only a few of gates, which can be produced by the manufacturers. 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 computing 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.

A. f-QASM

QASM (Quantum Assembly Language) is widely used in modern quantum simulators. It was first introduced in [15] 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 [16] carefully characterizes its theoretical properties. In 2014, A.JavadiAbhari et al. [11] defined a space-consuming flat description and denser hierarchical description QASM, called QASM-HL. Recently, Smith et al. [25] 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, i.e., so there is only one command in every line or period. However, a further motivation is to solve an issue raised by the IBM QASM 2.0 list and provide the ability to have conditional operations to implement feedback based on measurement outcomes.

B. Basic definition of f-QASM

Let us first define the registers:

- Define $\{r_1, r_2, \dots\}$ as the finite classical registers.
- Define $\{q_1, q_2, \dots\}$ as the finite quantum registers.

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

Then we define two kinds of basic operations:

- Define the command "$op(q)$" as $q := op(q)$, where op is a unitary operator and q is a quantum register.

- Define the command "$\{op\}(q)$" as $r := \{op\}(q)$, where $\{op\}$ is a set of measurement operators, q is a quantum register, and r a is 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 $\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$.

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

- Define "$MOV(r_1, r_2)$", r_1 and r_2 are 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 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.

C. f-QASM examples

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

1. Initialization

$\mathbf{q} := |\mathbf{0}\rangle$ means the program initializes 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. Unitary transformation

$\bar{\mathbf{q}} = \mathbf{U}[\bar{\mathbf{q}}]$ means the program performs a unitary transformation on the register q. The compiler will check whether the unitary matrix is a basic gate. An example program segment of unitary transformation follows:

```
hGate(q1);
```

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

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

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 $xGate = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, and *zGate* is a phase-flip gate $zGate = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$. 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);
```

4. Loop

A loop construct is provided using $QWhile(M(q))$, where $QWhile$ is a key word, 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 which have been described above. It could 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);
```

D. Decomposition of a general unitary transformation

Given a set $\{U_1, U_2, \ldots, U_n\}$ of basic gates. If any unitary operator can be approximated to arbitrary accuracy by a sequence of gates from this set, then the set is said to be universal [21].

In the compiler, there are two kinds of built-in decomposition algorithms. One is the QR method given in [21, 23]. It 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 [20].

The other is the QSD method presented in [22], 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 [20].

V. THE QUANTUM SIMULATOR

A. Quantum types

Data types can be extended from classical computing to quantum computing. For example, quantum generalizations of boolean and integer variables were introduced in [13]. The state space of a quantum boolean variable is the 2-dimensional Hilbert space **Boolean** = $\mathcal{H}_2$, and the state space of a quantum integer variable is the infinite-dimensional Hilbert space **integer** = $\mathcal{H}_\infty$. In $Q|SI\rangle$, every kind of quantum variable has its own initialization method and operation. Currently, $Q|SI\rangle$ contains only finite-dimensional quantum variables, but infinite-dimensional variables will be added in the future. The quantum types used in $Q|SI\rangle$ are presented in Fig. 3.

The entire quantum types are defined as subclasses of one virtual base class called $QuantumTypes\langle T\rangle$. The introduction of the virtual base class is only for the purpose of indicating 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.

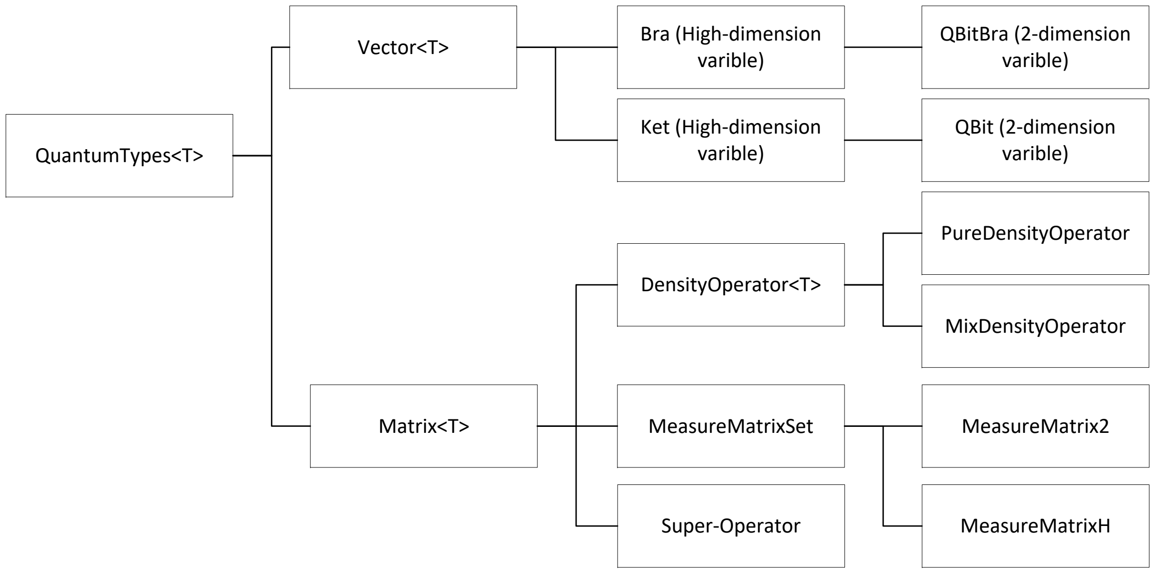


FIG. 3. $Q|SI\rangle$ Quantum types layer

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 specialized (sub)types *QBit* and *QBitBra* are provided for two-dimension quantum variable. Note that they are compatible when we consider the boolean type as a subtype of an integer. Also, these types must accept a few rules:

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

Hidden states: It is well-known that the information of a *QBit* or a *Ket* cannot be extracted unless the state is measured. However, as indicated by Nielsen and Chuang in [21] , "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 our platform $Q|SI\rangle$, 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 gets nothing about this hidden information unless it is measured. Thus, we classify the state of information storage as a "*Protect*" class, which means that the information of a quantum state cannot be touched easily.

The matrix form is widely used in (the semantics of) the quantum **while**-language. There are three categorized of matrix: *DensityOperator*$\langle T\rangle$, *MeasureMatrixSet* and *SuperOperator*. *DensityOperator*$\langle T\rangle$ is also 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; (2) ρ is a positive operator. Every operation of objects is set to trigger the verification of these conditions in order 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 $\sum_i M_i^\dagger M_i = I$. It is very flexible to define a quantum measurement in such a way. Specifically, a plus-minus basis $\{|+\rangle, |-\rangle\}$ and a computation basis $\{|0\rangle, |1\rangle\}$ are two built-in measurements, and a user can easily use their designed 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.

1. *Simulation of quantum behaviors*

The basis of simulating quantum computation is to simulate the quantum behaviors defined by the four basic postulates of quantum mechanics [21]:

- **Postulate 1**: Associated to 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 the platform $Q|SI\rangle$, 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 $Q|SI\rangle$, we have added the function $UnitaryTrans$ to some of our quantum types such as *QBit*, *Ket* and *DensityOperator⟨T⟩* in a closed quantum system. 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 acting 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 $\frac{M_m|\psi\rangle}{\langle\psi|M_m^\dagger M_m|\psi\rangle}$.

  Quantum measurements are simulated with a modified Monte Carlo method. A detailed description is postponed to the next subsection.

- **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 $|\psi\rangle$, then the joint state of the total system is $|\psi_1\rangle \otimes |\psi_2\rangle \otimes \dots$

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

2. Simulating measurement with pseudo-random number sampling

In $Q|SI\rangle$, a pseudo-random number sampling method is employed to simulate quantum measurement. It is the numerical experiment generating pseudo-random numbers are distributed according to a given probability distribution [24].

Let a quantum measurement be described by a collection of bounded linear operators $\{M_m\}$ that satisfy a completeness condition $\sum_m M_m^\dagger M_m = I$. m is used to describe the measurement results and the corresponding probability set is denoted as P, where $P = \{p_1, p_2, \dots, p_m\}$. The indexed variable set is denoted as Y where Y can be settled to the value $\{0, 1\}$. The current system state is assumed to be the quantum state $|\psi\rangle$, the indexed variables are $Y_1, \dots, Y_m$ and the probabilities are $\Pr[Y_i = 1] = p_i$ where $p_i = \langle\psi|M_i^\dagger M_i|\psi\rangle$, $P = \{p_1, \dots, p_m\}$. A uniform distribution X from $Q|SI\rangle$ is used to simulate a random variable Y.

Math.net provides a random variable X called *RandomSource* which is uniformly distributed on $(0,1)$. Then the interval $[0,1]$ is divided into m intervals as $[0,p_1],(p_1,p_1+p_2],\dots,(\sum_{i=1}^{m-1}p_i,1]$. The width of interval i equals the probability p_i.

Finally, measurement triggers the strategy in following steps:

1. Given a measurement $\{M_m\}$ and the current quantum state $|\psi\rangle$, $Q|SI\rangle$ computes the set $P=\{p_1,p_2,\dots,p_m\}$, where $p_i=\langle\psi|M_i^\dagger M_i|\psi\rangle$. This step provides the probability distribution Y: $\Pr[Y=i]=p_i$.

2. $Q|SI\rangle$ checks the elements of P. If there exists any $p_i=0$, discard the index i in the next step. If there exists any $p_i=1$, return the index i as the final result and skip the following steps.

3. Assuming P' is a set having the same quantity as P, $Q|SI\rangle$ accumulates the distribution P to P' with the rules: for each p_i in P', $p_i'=\sum_i p_i$.

4. Draw a number x which is a uniformly pseudo-random number distributed between $(0,1)$.

5. Find p_i', such that $p_{i-1}'\le x$ and $p_i'\ge x$ and return the index i. It should be noted that $i=1$ in the case of $x<p_1'$ and $i=m$ in the case of $x>p_{m-1}'$.

The P distribution of the Y variable where $p_i=\Pr(0<Y\le p_i')=\sum_i p_i'$ is the simulated distribution using the uniform distribution variable X. This method of pseudo-random number sampling was developed for Monte-Carlo simulations and its quality is determined by the quality of the pseudo-number.

After i is randomly chosen with the distribution $P=\{p_1,\dots,p_m\}$, the function returns the value of i and the quantum state is modified as an atom operation. According to quantum mechanics, the state $|\psi\rangle$ would be changed into $|\psi'\rangle=\frac{M_i|\psi\rangle}{\sqrt{\langle\psi|M_i^\dagger M_i|\psi\rangle}}$.

*3. Simulating operational semantics of 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, quantum **if**-clause are denoted as **cif** and quantum **while**-clause are denoted as **cwhile** in quantum simulation engine. To simulate these two functions, the related function methods are encapsulated in Quantum Mechanics Library.

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

Definition V.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 $Q|SI\rangle$.

Initialization:

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

where

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

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

In $Q|SI\rangle$, 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 revolution:

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

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 initialization clause and a unitary transformation clause. However, for convenience, $Q|SI\rangle$ provides a flexible state claim to initialize a *QBit*, or a *Ket* using a vector, and to initialize 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:

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

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 $Q|SI\rangle$ 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, $Q|SI\rangle$'s strategy is to automatically skip that clause if the outcome index exceeds $Func\langle T\rangle$ number. In fact, nothing to be done on variables excepted a measurement in this case.

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

$$int\; Measu2ResultIndex(MeasureMatrixSet)$$

to return the measurement result and go to the correct subprogram, then it would call the $void\, StateChange(int)$ inherently, which 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 $Q|SI\rangle$, 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 calls the $void\, StateChange(int)$ inherently as per the case statement.

VI. EXPERIMENTS

Here, we present three experiments to show the power of our quantum programming environment: Qloop, BB84 and Grover's search algorithm. Readers can find more details in the Appendices.

Qloop: Qloop case is a "Hello world" example that includes a quantum channel, a quantum measurement, a quantum **while**-clause and some quantum variables. Basically, it can be regarded as a simplified quantum walk. This test illustrates three main features of the $Q|SI\rangle$ platform, super-operators, unitary transformations and quantum measurement.

The basic idea of a Qloop is to perform a super-operator on a quantum state and leave the state changed. A counter is used to record the number of times the state enters into different branches. A measurement is taken in every shots and the counter should show the predicted probability for the state.

BB84: BB84 is a quantum key distribution (QKD) protocol developed by Bennett and Brassard in 1984 [27]. The protocol is an already-proven security protocol [**?**] that relies on the no-cloning theorem. Using this protocol Alice and Bob reach an agreement about a classical key string that can be used to encrypt classical bits.

Several different scenarios are considered in this experiment. The simple BB84 case outlines the basic communication procedure between two clients: Alice and Bob. The multi-client BB84 case illustrates a more practical example where one Alice generates the raw keys, and many Bobs make an agreement key with Alice. The most interesting case is the BB84 protocol in a channel with quantum noise. Because no real quantum systems are ever perfectly closed, super-operators can serve as a key tool for describing the dynamics of open quantum systems. In this case, influencing factors for QKD are explored. The package length and sampling percentages are crucial to real QKD protocol under quantum noise. With $Q|SI\rangle$, different parameters are tested in different channels which can be adjusted for practical use of this protocol.

Grover's search algorithm: Grover's search algorithm is an impressive algorithm in the quantum domain. It solves search task in disorderly databases consisting of N elements, indexed by number $0, 1, \ldots, N-1$ with an oracle. The oracle returns it answers according to position and can find solutions with a high probability within $O(1/N)$ error and $O(\sqrt{N})$ steps.

A more general multi-object Grover's search is also considered that supposes there is more than one answer (position) for the oracle to find. In this case, we use a blind box strategy that reverses 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 be modified in some way.

VII. CONCLUSIONS

This paper presents a new software platform, $Q|SI\rangle$, for programming quantum computers. $Q|SI\rangle$ includes an embedded quantum **while**-language, a quantum simulator, and quantum program analysis and verification toolkits. The platform can be used to simulate quantum algorithms, analyze the termination and average running time of quantum programs, and verify program correctness.

Throughout the paper, we demonstrate how to use $Q|SI\rangle$ to simulate quantum behaviors on classical platforms using a combination of components. We discuss simulating measurement with pseudo-random number sampling, and how to generate the syntax and semantics of the quantum **while**-language.

Active development of $Q|SI\rangle$ is ongoing. The tensor product is a clumsy way of emulating quantum circuits. We may need to consider timing and entanglement analysis inspired by [11] to extend $Q|SI\rangle$'s quantum computing power. The Termination and Average Running Time modules need to be unified into one format for syntax, and we are considering how to split classical and quantum coding for verification purposes.

Interfaces for different quantum computation programs, such as LIQU$i|\rangle$, ScaffCC and even the real quantum computation platform from IBM's quantum experiment also need to be considered. These diversified platforms often can provide different views of one quantum program.

ACKNOWLEDGMENTS

We were grateful to Michael Blumenstein, Ian Burnett, Yuan Feng, and Glenn Wightwick for their helpful discussions and support to this project.

[1] Peter W Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26(5):1484, 1997.

[2] 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.

[3] Aram W Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Physical Review Letters*, 103(15):150502, 2009.

[4] Bernhard Ömer. A procedural formalism for quantum computing. 1998.

[5] Peter Selinger. A brief survey of quantum programming languages. In *International Symposium on Functional and Logic Programming*, pages 1–6. Springer, 2004.

[6] 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.

[7] Jeff W Sanders and Paolo Zuliani. Quantum programming. In *International Conference on Mathematics of Program Construction*, pages 80–99. Springer, 2000.

[8] 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.

[9] Dave Wecker and Krysta M Svore. Liquid: A software design architecture and domain-specific language for quantum computing. *arXiv preprint arXiv:1402.4467*, 2014.

[10] 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.

[11] 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.

[12] Mikhail Smelyanskiy, Nicolas PD Sawaya, and Alán Aspuru-Guzik. qhipster: the quantum high performance software testing environment. *arXiv preprint arXiv:1601.07195*, 2016.

[13] Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):19, 2011.

[14] Mingsheng Ying. *Foundations of Quantum Programming*. Morgan Kaufmann, 2016.

[15] 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.

[16] Mingsheng Ying and Yuan Feng. A flowchart language for quantum programming. *IEEE Transactions on Software Engineering*, 37(4):466–485, 2011.

[17] Mingsheng Ying, Nengkun Yu, Yuan Feng, and Runyao Duan. Verification of quantum programs. *Science of Computer Programming*, 78(9):1679–1700, 2013.

[18] Mingsheng Ying, Shenggang Ying, and Xiaodi Wu. Invariants of quantum programs: characterisations and generation. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 818–832. ACM, 2017.

[19] 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.

[20] Christopher M Dawson and Michael A Nielsen. The solovay-kitaev algorithm. *arXiv preprint quant-ph/0505030*, 2005.

[21] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge University Press, 2010.

[22] 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.

[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] Luc Devroye. Sample-based non-uniform random variate generation. *Proceedings of the 18th conference on Winter simulation*, pages 260–265, 1986.

[25] Robert S Smith, Michael J Curtis, and William J Zeng. A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355*, 2016.

[26] Mingsheng Ying and Yuan Feng. Quantum loop programs. *Acta Informatica*, 47(4):221–250, 2010.

[27] Charles H Bennett and Gilles Brassard. Quantum cryptography: Public key distribution and coin tossing. *Theoretical computer science*, 560:7–11, 2014.

Appendix A: Setup and configuration of $Q|SI\rangle$

$Q|SI\rangle$ mainly relies on IDE (Visual Studio) to provide the details of the program. After completing a program using $Q|SI\rangle$, the programmer needs to build and compile it. This feature is considered to be an essential component because a smarter IDE is a basic way of ensuring the syntax is correct as programs grow in size and complexity. This feature is unlike IScasMC or QPAT which are not able to execute a program.

NuGet is a part of the .Net development platform and it is used in $Q|SI\rangle$ to manage the packages. All packages used to provide functions, such as matrix computation, random number generation, and Roslyn, etc., can be automatically controlled by *NuGet*. To add all the essential packages, a user needs only add the NuGet feed v3 "https://api.nuget.org/v3/index.json" to the Visual Studio 2017 configuration. This will provide the package resources and automatically configure them for the platform.

$Q|SI\rangle$ is compatible with any version of Visual Studio 2015 and later. However, we recommend the Enterprise version of Visual Studio 2017 because of some of its premium features, such as the ability to draw quantum circuits with the DGML tools, the most up-to-date Math.net, etc. Examples are stored in the sub-folder UnitTest. All entry-level examples can be found in the 'Program.cs' file in UnitTest.

Appendix B: Experiment-Qloop case

The first example showcases the Qloop case. It uses quantum channels, measurement, quantum **while**-clause and quantum variables. The Qloop case can also be treated as a simplified quantum walk. The flow path is shown in Fig. 4.

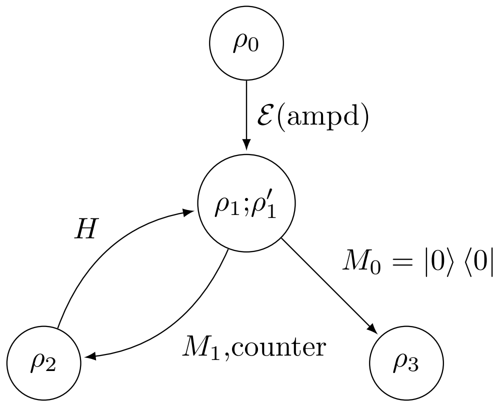


FIG. 4. Qloop

1. Input and output

Input:

- $\rho_0 := |+\rangle\langle +|$;
- $\mathcal{E} := \{E_0 = |0\rangle\langle 0| + |1\rangle\langle 1|/\sqrt{2}, E_1 = |0\rangle\langle 1|/\sqrt{2}\}$;

- $M := \{M_0 = |0\rangle\langle 0|, M_1 = |1\rangle\langle 1|\}$;
- $H := |+\rangle\langle 0| + |-\rangle\langle 1|$;
- $Counter := 0$.

Output:

- num: the number of circles is num.

2. Results

The Qloop experiment is executed about 100000 shots and the results are shown in the Fig. 5.

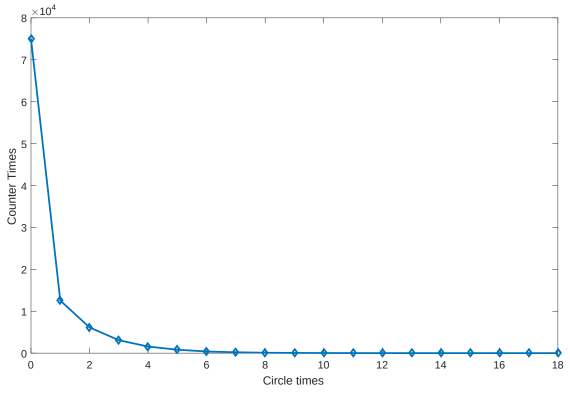


FIG. 5. Qloop data

3. Features and analysis

After calculation, it is clear that $\rho_1 = \mathcal{E}(\rho_0) = \frac{3}{4}|0\rangle\langle 0| + \frac{1}{4}|1\rangle\langle 1| + \frac{1}{2\sqrt{2}}|0\rangle\langle 1| + \frac{1}{2\sqrt{2}}|1\rangle\langle 0|$, $\rho_2 = |1\rangle\langle 1|$, $\rho_1' = |+\rangle\langle +|$ and $\rho_3 = |0\rangle\langle 0|$.

The three main features of this experiment include super-operators, unitary transformation, and measurement operation. In addition, processes that consider a qubit's collapse and measurement probability are involved as part of quantum mechanics.

- Super-operator operation. The initial state passes through a quantum channel and becomes ρ_1. Let M be performed on the state ρ_1 in each shot. There is a $\frac{3}{4}$ probability that the state would change to ρ_3 and then terminate. Likewise, there is a $\frac{1}{4}$ probability of moving in a circle and having the process recorded by the counter. So if the program is executed many times, such as in a 100000 shot experiment, the counter should show the state enters the circle about 25000 times.

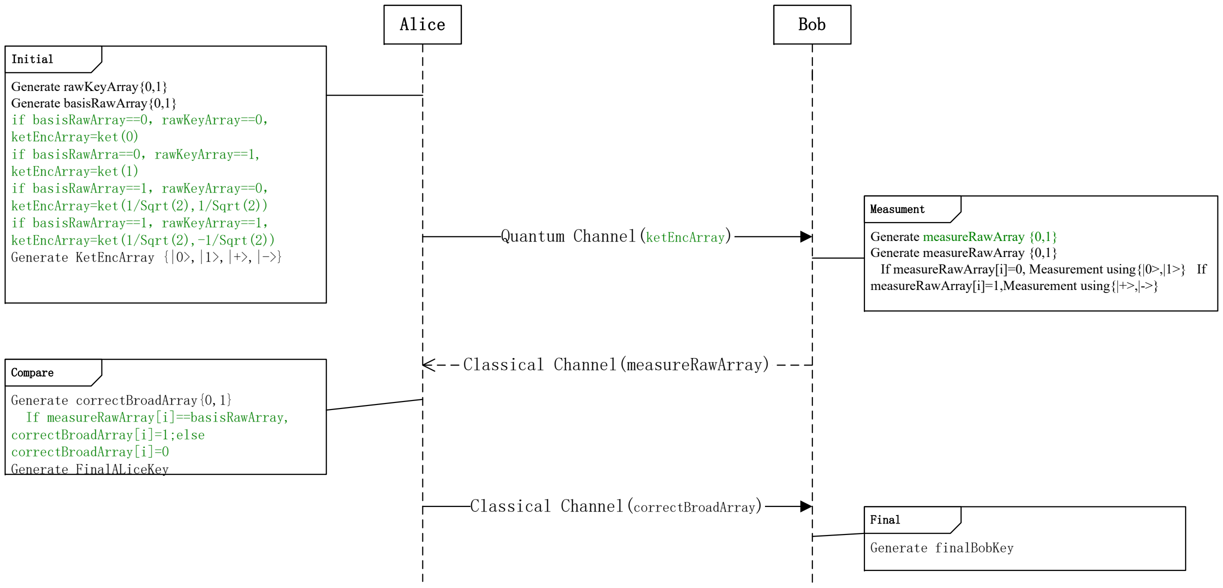
- Measurement operation and unitary transformation. After the first measurement, ρ_1 would change to ρ_2 and continue or it would change to ρ_3 and terminates. If the state becomes ρ_2, after a Hadamard operator which is a unitary transformation, it becomes $\rho_1' = |+\rangle\langle+|$ and counter records the circle once. When a measurement M is performed on the state, we can assert that almost half the time ρ_1' becomes $|0\rangle\langle 0|$ and half the time it becomes $|1\rangle\langle 1|$. If the result is $|1\rangle\langle 1|$, it will enter into the loop body again and is recorded by our counter. In total, the counter number shows how many circles the state enters into. Obviously, this decreases at almost half the rate of a geometric progression, as in say $1 - 12556$, $2 - 6140$, $3 - 3095$, ...

Appendix C: BB84 case

BB84 is a basic quantum key distribution (QKD) protocol developed by Bennett and Brassard in 1984 [27].

1. Simple BB84 case

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. For simplicity and clarity, this example only '*Ket*' quantum types is considered, not quantum channels or Eves. The flow path is shown in Fig. 6.



Alice
Bob
Initial
Generate rawKeyArray{0,1}
Generate basisRawArray{0,1}
if basisRawArray==0, rawKeyArray==0,
ketEncArray=ket(0)
if basisRawArra==0, rawKeyArray==1,
ketEncArray=ket(1)
if basisRawArray==1, rawKeyArray==0,
ketEncArray=ket(1/Sqrt(2),1/Sqrt(2))
if basisRawArray==1, rawKeyArray==1,
ketEncArray=ket(1/Sqrt(2),-1/Sqrt(2))
Generate KetEncArray {|0>,|1>,|+>,|->}
Quantum Channel(ketEncArray)
Measument
Generate measureRawArray {0,1}
Generate measureRawArray {0,1}
If measureRawArray[i]=0, Measurement using{|0>,|1>} If measureRawArray[i]=1,Measurement using{|+>,|->}
Classical Channel(measureRawArray)
Compare
Generate correctBroadArray{0,1}
If measureRawArray[i]==basisRawArray,
correctBroadArray[i]=1;else
correctBroadArray[i]=0
Generate FinalALiceKey
Classical Channel(correctBroadArray)
Final
Generate finalBobKey

FIG. 6. Simple BB84 protocol

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.

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 $\{|0\rangle, |1\rangle\}$ 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 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 the same.

This case shows some useful features,

- Client-server mode. The process uses a client-server model to simulate the BB84 protocol. The model includes many implicit features, such as waiting threads and concurrent communications which are also used in the next example.
- Measurement. According to theory, 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 the length of the raw keys.

2. BB84 case, multi-client

The multi-client BB84 model offers a more attractive and practical example. In this model, there is one Alice to generate the raw keys and many Bobs to construct an agreement key with Alice.

In this case, users can specify the number of clients. Also, a typical BB84 flow path would occur for every client-server pair of this model.

- Threads Model. Many clients are generated and communicate with Alice. Each of them finally reaches an agreement.
- Measurement threads. In this case, Alice generates raw keys, and Bob measures the quantum bits. However, this raises a serious question that when a client is considered to generate a raw key while the server measure, how can we ensure the server correctly and fairly conducts the measurement for the server.

3. BB84 case with noise

A practical topic for the $Q|SI\rangle$ is to consider the BB84 model with noisy quantum channels. Noisy quantum operations are the key tools for the description of the dynamics of open quantum systems.

In this example, different channels such as bit flip, depolarizing, amplitude damping and I-identity channels are described by quantum operations performing as the evolution of quantum systems in a wide variety of circumstances. Alice and Bob use these quantum channels to communicate with each other via the BB84 protocol as Fig. 6 shows. During communication, verification steps also need to be considered.

a. Input and output

In this example, the basic quantum channels are defined as follows:

deplarizing channel with noise parameter $p = 0.5$,

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

amplitude damping channel with noise parameter $\gamma = 0.5$,

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

bit flip channel with noise parameter $p = 0.25$,

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

bit flip channel with noise parameter $p = 0.5$,

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

bit flip channel with noise parameter $p = 0.75$,

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

The flow path follows the simple BB84 protocol shown in Fig. 6. The only differences are in Step 4 and a sampling step is added.

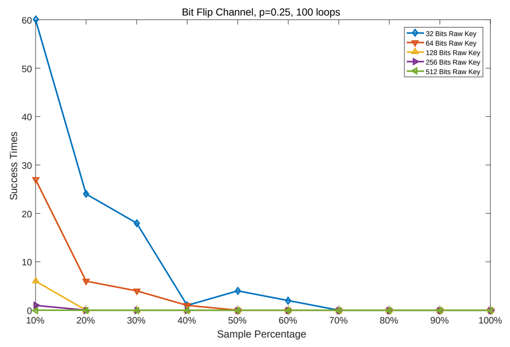
- Alice sends the *KetEncArray* through a quantum channel. In this case, it is one of the channels mentioned above.

- Sampling check step: Alice randomly publishes some sampling positions randomly with the bits against these positions in her own key string. 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.

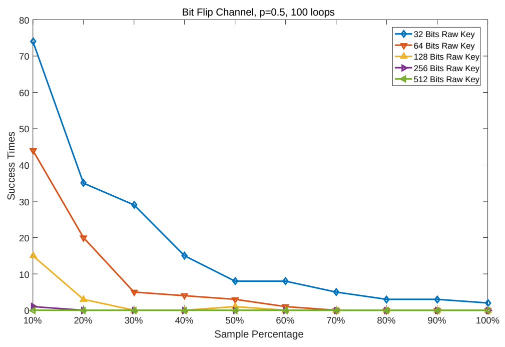
For a statistical quantity characterizing success in a channel with the BB84 protocol, we executed a 100-shot experiment for each channel. In every shot for every channel, different sampling percentages and package lengths were considered. The tables and figures are provided in Fig. 7 that show the trade-off between success times, different sampling proportions and package lengths in each of the quantum channels.

b. Results

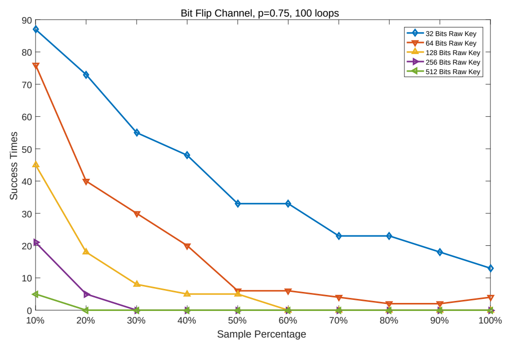
Success times for different sampling percentages in different channels in 100 shots are provided by Fig. 7.



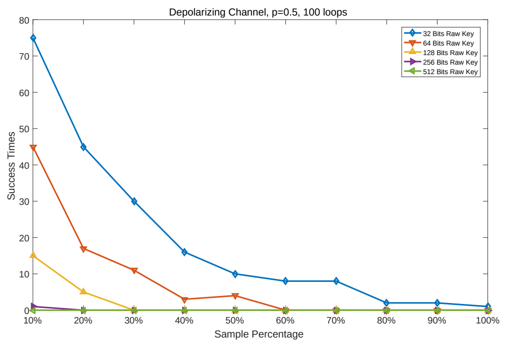
(a) Bit Flip Channel, $p = 0.25$, $loops = 100$



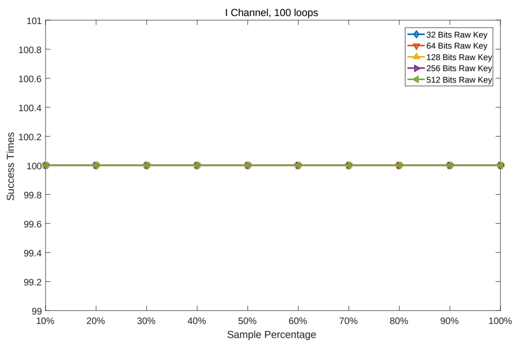
(b) Bit Flip Channel, $p = 0.5$, $loops = 100$



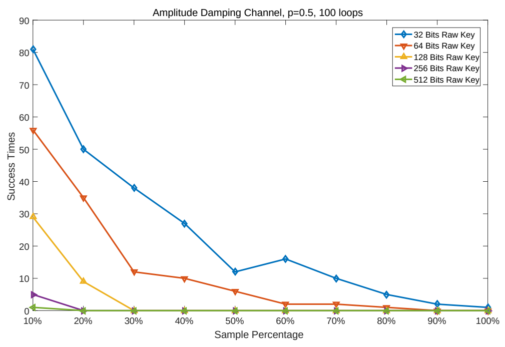
(c) Bit Flip Channel, $p = 0.75$, $loops = 100$



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



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



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

FIG. 7. Statistics of success communication via BB84 with channels

c. *Features and Analysis*

The example generates some 'erroneous' bits during communication due to the quantum channels. These bits cause a connection failure. Meanwhile, 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 also a probability verification step which means it does not use all the agreed bits to verify the communication procedure.

Subfigures (a),(b) and (c) in Fig. 7 are bit flip channels with different probabilities. Overall, successful shots increase as p increases and the raw key length shortens. This is because p is a reflection of the percentage of information that remains in bit flip channel and an increase in p means fewer errors in communication. A shorter raw key length ensures fewer bits are sampled. Sub-figure (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.

Appendix D: Grover's search algorithm

Grover's search algorithm is a well-known quantum algorithm. It solves searching problems in databases consisting of N elements, indexed by number $0, 1, \dots, N-1$ with an oracle provides the answer as a position. This algorithm can find solutions with a probability of $O(1)$ within $O(\sqrt{N})$ steps.

1. A Simple Grover's search algorithm

In this example, we assume there is only one answer to the question, i.e., the oracle will only reverse one position at a time. Further, the oracle is assumed to be working as a black box and can reverse the correct position of the answer. After querying the oracle $r = \frac{\pi}{4}\sqrt{N}$ times with the corresponding phase rotations, the quantum state includes the correct information to answer the question.

a. *Input and output*

Input:

- The total number of space N. For convenience, we restrict $N = 2^n$.
- The correct position of the search. That is used to construct oracle.

Output:

- The final position of the measurement result.
- Oracle time r.

b. *Results*

The simple Grover's search algorithm has only one result, and the final measurement result shows the correct answer to the searching problem.

c. *Features and analysis*

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

2. Multi-object Grover's search algorithm

A more general Grover's search algorithm is considered: a multi-object Grover's search algorithm. This case supposes that there may be more than one correct answer (position) for the oracle to find. We use a strategy that adds a blind box to reverse the proper position of the answer. This experiment reveals that Grover's algorithm leads to an avalanche of error in a multi-object setting, indicating that algorithm needs to be modified in some way.

A new blind box (a unitary gate) is added, which 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 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 ones were found in previous rounds.

a. Input and output

The input is

- The total number of spaces N. For convenience, we restrict $N = 2^n$.
- All correct positions of the search.

The output is

- The final position of the measurement result.
- Oracle time r.

b. Results

The measurement shows different probabilities of the final result. The theory holds that if we have multiple-answers, the state after r times oracles and phase gates 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 $|2\rangle$ or $|14\rangle$ the first time and the other one the next time. However, we get results other than $|2\rangle$ and $|14\rangle$ with high probability, which indicates that the multi-object search algorithm is not very good.

c. Features and analysis

It worth noting that due to multi-objects, the real state after using Grover's search algorithm becomes $a(|2\rangle + |14\rangle) + b(|1\rangle + |3\rangle + |4\rangle + |5\rangle + ...)$ where $a, b \in \mathcal{C}$ and $|a|^2 + |b|^2 = 1$. 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 measured, 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. In the last example, we would have gotten a wrong answer by measurement, say $|5\rangle$. After new 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.