
$Q|SI\rangle$: A QUANTUM PROGRAMMING ENVIRONMENT

Manual Author: Shusen Liu

shusen.liu88@gmail.com

June-30-2017

Alpha Version(Draft)— Version 1

Abstract

This document describes the user guide of $Q|SI\rangle$. Following this guide, users could setup their own software environment supporting all functions of $Q|SI\rangle$. Besides, this guide describes the APIs (Application Program Interface) for user's advanced application. Also, several examples are illustrated in the manual.

This manual is an addition to the platform article and the termination module article. Any feedbacks are welcomed. Please find Appendix B to see how to submit the feedback.

Contents

1	Introduction	3
1.1	$Q SI\rangle$ intro	3
1.2	Environment Requirement	3
1.3	Setup the Environment, Step by Step	4
2	Run Examples	6
2.1	Getting start from Console Interface	6
2.2	Choose One Example and Find the Result	6
2.3	Details of Examples	6
3	Serious Coding	21
3.1	$Q SI\rangle$ Structure (expert)	21
3.2	Main Components of $Q SI\rangle$	22
3.3	From UnitTest to HelloWorld	23
3.4	Using Quantum while -language	27
3.5	A Simple Example	29
3.6	Check the Terminating	29
3.7	Compilation	30
4	Explore APIs	33

Appendix	34
A Contributors	34
B Contact Information	34

1 Introduction

1.1 $Q|SI\rangle$ intro

This guide describes the usage of a quantum programming environment, named $Q|SI\rangle$. $Q|SI\rangle$ is used for quantum programming in a quantum extension of **while**-language and it is embedded in .Net language. It includes a compiler of the quantum **while**-language and useful tools for the simulation of quantum computation, analysis and verification of quantum programs, and the optimization of quantum circuits.

We would like first to illustrate an overview of this platform.

- What can be done by $Q|SI\rangle$: $Q|SI\rangle$ is a quantum programming platform that allows you to program quantum algorithms. After programming, you can do several related works: performing it via inherent mathematical calculation engine; compiling it to f-QASM (Quantum Assembly Language with feedback) and pre-defined gates; debugging it by static analysis modules (Termination and Average running time module) and verifying the correctness of programming related to your requirement.
- What can **NOT** be done by $Q|SI\rangle$: Before getting start, you should notice that $Q|SI\rangle$ is only a quantum programming platform. It cannot perform any physics experiment without cloud physics platform such as IBMQ. Besides, it is still on the way to develop emulator that it cannot accept too much quantum variables on home PC.
- Why you need $Q|SI\rangle$: $Q|SI\rangle$ provides the most significant quantum environment. We believe other platforms do have their advantages. Our advantage is providing the possibilities connecting different platforming together to research the quantum algorithms.
- How to use $Q|SI\rangle$: You just need a personal computer (PC) to perform most functions of $Q|SI\rangle$. Most modules are developed on Windows, we recommend the Windows 10 Professional. We are under developing the Linux version.
- If you need perform your algorithm on IBMQ, you need an account of IBM and get a personal token from it.
- You can set up your environment and begin to program by following this guide. This document includes all your needs to perform programming on $Q|SI\rangle$. The corresponding authors also welcome any feedbacks and questions. You can submit your voice via Github or Emails. Feel free to contact with us.

1.2 Environment Requirement

The programming environment is developed by Visual Studio on the Windows Platform. The hardware and software requirements are based on the Windows system. Some recommendations are given for executing $Q|SI\rangle$. Other similar configurations are required to be checked by users.

1.2.1 Hardware Requirement

- CPU: You need an Intel Core i5 and/or compatible above CPU. For decomposition gate function, you may need an Intel Core i7 to reach a reasonable and acceptable execution speed.
- Memory: 4GB and above DDR4 memory are recommended. For termination analysis, 4GB memory is the basic requirement. Note that quantum emulation

and simulation (execution) require an exponential increasing memory with extra quantum objects.

Hard Disk: 2GB hard disk is required for basic functions. Pre-defined gates for compiler require about extra 2GB (for Length 1-16 pre-defined gates).

Laptop: Typical Surface Pro i5/i7 is suitable for $Q|SI\rangle$.

1.2.2 Software Requirement

OS: Windows 10 Professional x86-64 version is recommended for $Q|SI\rangle$. Other Windows version (Windows 7 and Windows 8) with .Net Framework 4.6 are also acceptable.

Visual Studio: The up-to-date Visual Studio 2017 Enterprise is strongly recommended. It includes many exciting features for further developing. Besides, DGML (Directed Graph Markup Language) component is only supported by Visual Studio Professional version. DGML is used for drawing single qubit quantum circuit (We will support multi-qubits circuits in next stage). Visual Studio 2017 community version can support all the features except DGML drawing.

Matlab: Matlab 2017a is required for analysis of input matrix and some matrix calculations. We notice that Matlab is a commercial research software and therefore we provide a compiled .net assembly from Matlab to support $Q|SI\rangle$. If you do not have the up-to-date Matlab license, you can setup the "Support_Matlab_Assembly.exe".

Python: Python 2.7 is required for decomposition in compiler. We recommend the anaconda environment which includes Numpy and Scipy package. In section 1.3, we introduce how to setup the conda environment and other related packages.

1.2.3 Virtual Machine Supported

For users' convenience, we provide a mirror of the environment. All the configures have been configured as what described. You only need to download **Oracle VM VirtualBox** and execute it with our mirror.

1.2.4 Cloud computation

The cloud computation will be provided soon. Please keep an eye on <http://www.qcompiler.com> for further information.

1.3 Setup the Environment, Step by Step

1.3.1 Choose a Suitable File

Online Setup File **QSI_Online_Setup** is a compressed file including online setup. Its size is only around 50MB. It consists **vs_community.exe** (Online) and **Support_Matlab_Assembly.exe** (Online) and **QSI Folder** (Local). Users in Australia (including NBN and ADSL 2+) and China BGP with a good network can benefit from this portable setup file.

Offline Setup File **Offline Setup File** is a compressed file including offline setup. Its size is MORE than 5GB that it can perform without the Internet. It consists **vs_community.exe** (offline) and **Support_Matlab_Assembly.exe** (offline) and **QSI Folder** (Local). This file for installation is useful for the users with an unstable network connection. The user in China Education Network (CERNET) can benefit from this all-in-one setup file.

1.3.2 Installation

1. Double-click `VS_community.exe`. Even though we recommend enterprise version which is a commercial software package and includes many productive toolkits, visual studio community version can be accepted. The only remark is that it can not draw one qubit quantum circuits with DGML component.
2. Choose **.NET desktop development** (Figure 1) and press install button. It may cost about 15 minutes depending on your network.
3. Double-click `Support_Matlab_Assembly.exe` (Figure 2). Install matlab support components and re-distribution supports from Internet. It may takes about 15 minutes depending on your network.
4. Unzip your downloaded Project $Q|SI\rangle$ file, <https://github.com/klinus9542/QSI> and open `QSI.sln` with Visual Studio 2017 Community.
5. Menu \rightarrow Debug \rightarrow Start Debugging. Eureka!

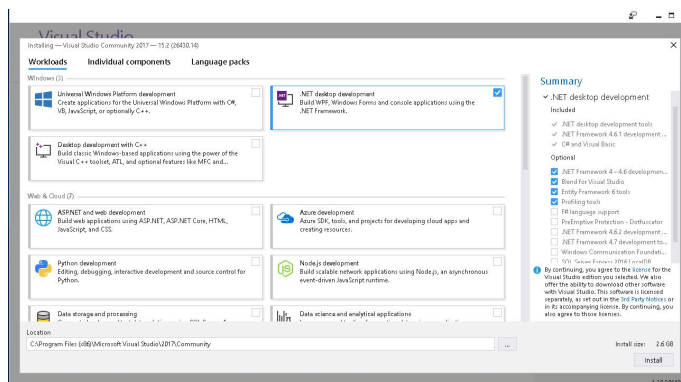


Figure 1: .NET desktop development



Figure 2: Support Matlab Assembly

1.3.3 Install Advanced Compiler and Configure Conda

Coming Soon. . .

2 Run Examples

2.1 Getting start from Console Interface

After configuration of $Q|SI\rangle$, you could start to run the examples which are provided in the software packages.

Please notice that the examples are not unified into the pure second generation language. Some codes are programmed within engine layer (basic) language for testing and debugging reason. These code segments are only for preview purpose.

1. Ensure the system environment has been configured correctly.
2. Menu \rightarrow Debug \rightarrow Start Debugging (or press F5).
3. Wait several seconds. (It may consume more than 2 minutes to build the project at the first time of execution depending on your hardware.)
4. Eureka! You should find the display like Figure 3 shows. Welcome to the quantum world.

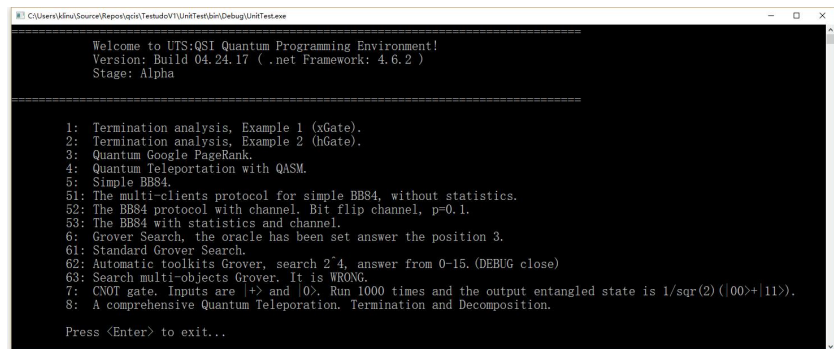


Figure 3: Welcome to $Q|SI\rangle$

2.2 Choose One Example and Find the Result

At console interface, you can choose one example from the list. You only need to input a number and press “Enter” key. The result would be shown after execution.

You may need to notice that some options require a little modification of the code, such as “Option 62”. In the following subsections, the details of the experiment will be displayed and explained.

2.3 Details of Examples

In this subsection, the details of examples would be displayed and analyzed. For your convenience, all the codes of examples are assembled in the Project “UnitTest” under the root Solution as Figure 4 shows.

2.3.1 CNOT Gate Showcase-“Option 1”

Input and Output

1. Firstly, enter the number 1 to start CNOT Gate Showcase.

2. Console displays “The experiment begins. . .”. Just be patient with time, the experiment will execute about 1000 times and show the statistic result as Figure 5.

Behind the Console (Expert) This is a very simple and standard example of $Q|SI\rangle$. The code segment is so trivial that we list it here for your reference.

```
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 $q2 = |0\rangle\langle 0|$. CNOT gate (from $q1$ to $q2$) is defined as $\text{CNOT} = [1000; 0100; 0001; 0010]$. After CNOT gate, the state

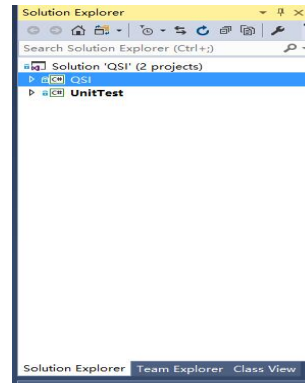


Figure 4: Examples in Project “UnitTest”

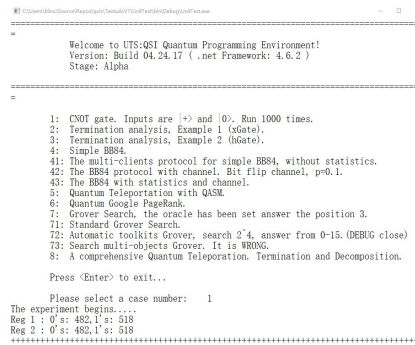


Figure 5: CNOT Experiment

will be $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$. If $\{|0\rangle, |1\rangle\}$ measurement are performed on $q1$ and $q2$, the result should includes that:

- Only consider the second quantum register $q2$, almost half times get result 0 and half times get result 1.
- Consider results of $q1$ and $q2$, the times of result 0 of $q1$ should be the same as the result 0 of $q2$.

Figure 5 illustrates the results as what we predicate.

2.3.2 Termination Analysis-“Option 2,3”

Input and Output

1. First enter the number 2 (or 3) to start termination analysis example.
2. Console displays the termination ability of corresponding example. You can see “Final Result: Termination” 6 from Example 2 and “Final Result: Almost Sure Termination” 7 from Example 3.
3. You also can find the real executing 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 6: 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 7: Almost Sure Termination

Behind the Console (Expert) Loop brings the question that “Does the quantum program terminate?” The termination analysis module partly answers this question. See the article [] for details.

2.3.3 BB84-“Option 4, 41, 42, 43”

BB84 is a quantum key distribution (QKD) protocol developed by Bennett and Brassard in 1984[Bennett and Brassard, 2014]. The protocol is an already-proved security protocol [Shor and Preskill, 2000] relying on no-cloning theorem.

“Option 4”, Simple BB84

Input and Output

1. First enter the number 4 to start the simple BB84 protocol.
2. Console asks “Input Array Length”. This is the raw initial keys length. Input 5 as a test number.
3. After then, Console prints the hint that “In basis, 0 is $\{|0\rangle, |1\rangle\}$ measurement and 1 is $\{|+\rangle, |-\rangle\}$ ” and other information. Let us interpretation the Figure 8 as an example. We can get to know that the protocol tries to consensus a key with length 5 bits. The initial bits (raw keys) are 11000 and the basis are $\{|+\rangle, |-\rangle\}, \{|0\rangle, |1\rangle\}, \{|0\rangle, |1\rangle\}, \{|+\rangle, |-\rangle\}, \{|+\rangle, |-\rangle\}$. Combined these two conditions, we can know Alice generates 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 broadcast these basis. After Alice receives these basis, she checks her chosen basis, finds that the correct positions are 1, 3, 5 and broadcast the correct positions. Finally, Bob keeps only the position 1, 3, 5 of the measurement result. Both of them reach an agreement of key 100.

```

Please select a case number: 4
In basis, 0 is  $\{|0\rangle, |1\rangle\}$  measurement and 1 is  $\{|+\rangle, |-\rangle\}$  measurement
Input Array Length: 5
Success
randomMeasure:
1=0: 0=0
0=0: 0=0
0=0: 0=0
0=0: 1=0
0=0: 1=0

plusMinusMeasure:
0.5=0: 0.5=0
0.5=0: 0.5=0
0.5=0: -0.5=0
-0.5=0: 0.5=0

rawKeyArray      11000
basisRawArray    10011

measureRawArray  11001
resultMeasureArray 10000
correctThroughArray 10101
finalAliceKey    100
finalBobKey      100
*****

```

Figure 8: Simple BB84 Example

Behind the Console (Expert) In this case, a client-server model is used as a prototype for multi-user communication protocol. “Quantum type converter” is used to convert ‘Ket’ to density operator. Also, in this case, we use the Quantum Type ‘Ket’ and do not consider quantum channel or Eve. The flow path is as Figure 9.

The entire flow path is as follows.

1. Alice: She randomly generates a sequence of classical bits, *rawKeyArray*. This sequence is the raw key which would be chosen from to construct as the final agreement key. The sequence length is given by the user’s input.
2. Alice: She randomly generates a sequence of classical bits, *basisRawArray*. This sequence indicates the chosen basis which would be used in next step. Alice and Bob share the rule before the protocol:
 - They use $\{|+\rangle, |-\rangle\}$ or $\{|0\rangle, |1\rangle\}$ to encode information.
 - Classical bit 0 indicates $\{|0\rangle, |1\rangle\}$ basis while classical bit 1 indicates $\{|+\rangle, |-\rangle\}$. This rule is used to generate qubits and

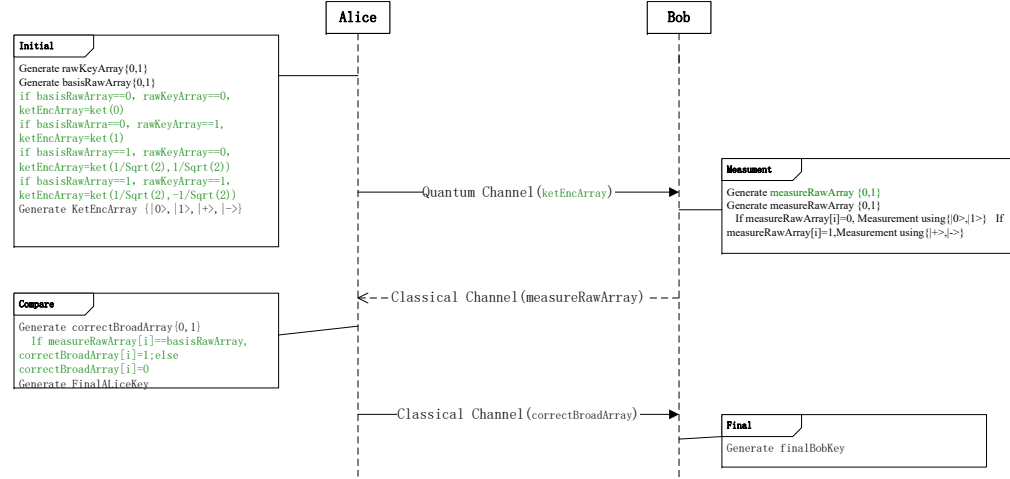


Figure 9: Simple BB84 protocol

to check Bob's basis.

3. Alice: She generates a sequence of quantum bits, *KetEncArray*, one by one according the rule as below:
 - If the *basisRawArray*[*i*] in the position [*i*] is 0 and the *rawKeyArray*[*i*] in the position [*i*] is 0, *KetEncArray*[*i*] would be $|0\rangle$.
 - If the *basisRawArray*[*i*] in the position [*i*] is 0 and the *rawKeyArray*[*i*] in the position [*i*] is 1, *KetEncArray*[*i*] would be $|1\rangle$.
 - If the *basisRawArray*[*i*] in the position [*i*] is 1 and the *rawKeyArray*[*i*] in the position [*i*] is 0, *KetEncArray*[*i*] would be $|+\rangle$.
 - If the *basisRawArray*[*i*] in the position [*i*] is 1 and the *rawKeyArray*[*i*] in the position [*i*] is 1, *KetEncArray*[*i*] would be $|-\rangle$.
4. Alice: She puts the *KetEncArray* through a quantum channel. In this case, it is *I* (identity) channel.
5. Bob: He gets the *KetEncArray* through the quantum channel.
6. Bob: He generates a sequence of classical bits randomly, *measureRawArray*. This sequence indicates the chosen basis which would be used in next step.
7. Bob: He generates a sequence of classical bits, *tempResult*, by quantum measurement using the common sense rules,
 - If the *measureRawArray*[*i*] in [*i*] position is classical bit 0, Bob use $\{|0\rangle, |1\rangle\}$ basis to measure *KetEncArray*[*i*], otherwise classical bit 1 indicates using $\{|+\rangle, |-\rangle\}$ basis.
8. Bob: He broadcasts *measureRawArray* to Alice using a classical public channel.
9. Alice: She generates a sequence of classical bits *correctBroadArray*, by comparing Bob's basis *measureRawArray* and her basis *basisRawArray*. If in the position [*i*] is correct, *correctBroadArray*[*i*] would be 1, otherwise it would be 0.
10. Alice: She sends this sequence *correctBroadArray* to Bob.

11. Alice: She generates a sequence of classical bits, *FinalAliceKey* using this rule:
 - If the position [i] in *correctBroadArray[i]* is 1, she would keep the *rawKeyArray[i]* and copy it to *FinalAliceKey*, else she would drop the *rawKeyArray[i]*.
12. Bob: He generates a sequence of classical bits, *FinalBobKey* using this rule:
 - If the position [i] in *correctBroadArray[i]* is 1, he keeps the *tempResult[i]* and copy it to *FinalBobKey[i]*, otherwise he drops the *tempResult[i]*.
13. GlobalView: We use a function to compare every position [i] in *FinalAliceKey* and *FinalBobKey[i]*. If they are the same, this communication is success, otherwise it fails.

This case shows some useful features,

- Quantum Computation Engine Directly Call (QCEDC). In this example, we use low-level component (Functions from quantum computation engine direct) of $Q|SI\rangle$, this mode can be used to design and code the quantum communication protocol which is not easily described by formal quantum language.
- Client-server Mode. The process is using a client-server model to simulate BB84 protocol. The model includes many potential features such as waiting thread and concurrent communication which is used in next example.
- Measurement. According to theory, choosing randomly measurement basis may get half correct result. As a result, the agreement of classical shared bits should be almost half length of raw keys.

“Option 41”, Multi-clients
BB84

A more attractive and practical example is multi-clients BB84 model. In the multi-clients BB84 model, there is one Alice to generate raw keys and some Bobs to make an agreement with Alice.

Input and Output

1. First enter the number 41 to start the simple BB84 protocol.
2. Console asks “Input Array Length”. This is the raw initial keys length. Input 5 as a test number.
3. Console asks “Input Clients Numbers”. This means how many clients will be produces. Input 2 as a test number.
4. After then, Console prints 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.

Behind the Console (Expert) In this case, users can specify the number of clients. Also in every client-server pair of this model would be a typical BB84 flow path.

- Quantum Computation Engine Directly Call (QCEDC). In this example, we use low-level component (Functions from quantum computation engine direct) of $Q|SI\rangle$, this mode can be used to design and code the quantum communication protocol which is not easily described by formal quantum language.

```

Welcome to UTS-QST 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 (Gate).
3: Termination analysis, Example 2 (Gate).
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 PaperFun.
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 Teleportation. Termination and Decomposition.

Press (Enter) to exit...

Please select a case number: 41
Input Array Length:5
Input Clients Number:2
FinalAliceKey for 7 010
FinalAliceKey for 8 00
FinalBobKey for 8 00
FinalBobKey for 7 010

```

Figure 10: Multi-clients BB84 Console

- Waiting thread. Many clients would be generated and communicate with Alice. Each of them finally reaches an agreement.
- Measurement threads. In this case, Alice generates the raw keys and Bob measures the quantum bits. A serious question is that when a client is considered to generate raw key while the server measures: how to ensure the correctness and fairness of measurement for the server.

“Option 42”, BB84 with Noise Channel

A very interesting and practical topic for using our $Q|SI\rangle$ is to use BB84 model considering quantum noise channel. Because no quantum system is ever perfectly closed, quantum operations are the key tools for description of the dynamics of open quantum systems.

Input and Output

1. First enter the number 42 to start the simple BB84 protocol.
2. Console asks “Input Array Length”. This is the raw initial keys length. Input 5 as a test number.
3. After then, Console prints the hint that “In basis, 0 is $\{|0\rangle, |1\rangle\}$ measurement and 1 is $\{|+\rangle, |-\rangle\}$ ” and other information.

Let us interpretation the Figure 11 and Figure 12 as an example. In Figure 11, we can get to know that the protocol tries to consensus a key with length 5 bits. The initial bits (raw keys) are 10100 and the basis are $\{|+\rangle, |-\rangle\}, \{|0\rangle, |1\rangle\}, \{|+\rangle, |-\rangle\}, \{|0\rangle, |1\rangle\}, \{|0\rangle, |1\rangle\}$. Combined these two conditions, we can know Alice generates the quantum states $\{|-\rangle, |0\rangle, |-\rangle, |0\rangle, |0\rangle\}$ and send 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 broadcast these basis. After Alice receives these basis, she checks her chosen basis, finds that the correct positions are 1,2 and broadcast the correct positions. Finally, Bob keeps only the position 1,2 of the measurement result. Both of them reach an agreement of key 10. However, Alice and Bob do not reach an agreement on the keys. In Figure 12, Alice and Bob does the same way as above. But, the BB84 fails due to the interference of the quantum channel.

```

Press <Enter> to exit...

Please select a case number: 42
In basis, 0 is  $|0\rangle, |1\rangle$  measurement and 1 is  $|+\rangle, |-\rangle$  measurement

Input Array Length: 5
zeroOneMeasure:
0+0i 0+0i
0+0i 0+0i
0+0i 0+0i
0+0i 1+0i

plusMinusMeasure:
0.5+0i 0.5+0i
0.5+0i 0.5+0i
0.5+0i -0.5+0i
-0.5+0i 0.5+0i

rawKeyArray 10100
basisRawArray 11010

measureRawArray 11101
resultMeasureArray 10101
correctBroadArray 11000
finalAliceKey 10
finalBobKey 10
The protocol: Success

```

Figure 11: BB84 with a Quantum Channel, Success

```

Press <Enter> to exit...

Please select a case number: 42
In basis, 0 is  $|0\rangle, |1\rangle$  measurement and 1 is  $|+\rangle, |-\rangle$  measurement

Input Array Length: 5
zeroOneMeasure:
1+0i 0+0i
0+0i 0+0i
0+0i 0+0i
0+0i 1+0i

plusMinusMeasure:
0.5+0i 0.5+0i
0.5+0i 0.5+0i
0.5+0i -0.5+0i
-0.5+0i 0.5+0i

rawKeyArray 00111
basisRawArray 01001

measureRawArray 01100
resultMeasureArray 00000
correctBroadArray 11010
finalAliceKey 001
finalBobKey 000
The protocol: Failed

```

Figure 12: BB84 with a Quantum Channel, Failure

Behind the Console (Expert) In this example, different channels such as bit flip channel, depolarizing channel, amplitude damping channel and I -identity channel are described by Kraus operator. Alice and Bob uses these quantum channels to communicate with each other via the BB84 protocol as figure 9 shows. During the communication, verification steps are also needed to be considered.

In this example, some quantum channels \mathcal{E} defined as below performing on communication:

Depolarizing 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 $p = 0.75$,

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

Program flow path is as Simple BB84 case. The only differences are the step 4 and a sampling step is added,

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

“Option 43”, BB84 with Statics and Channel

From above example, we know that not only the quantum channel has effects on quantum state, but also the verification (check) step does. In this example, we want to answer that what is a suitable proportion for a fixed quantum channel?

Input and Output

1. First enter the number 43 to start the BB84 protocol with statistics and channel.
2. Console displays that “x-axis...” and “y-axis...”.

3. We assume that in the verification step, any one bit difference would cause the failure in that run. We send different lengths of data package and use the criteria to check the percentage of protocol. One time experiment is like Figure 13 shows. Let us interpretation the Figure 13. The BB84 protocol is under an amplitude damping quantum channel (Defined as $\mathcal{E} := \left\{ \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ 0 & 0 \end{bmatrix} \right\}$.) In the output, the number in Column 1 and Row 1 means the protocol is 85% percentage success if it performs under the amplitude damping quantum channel defined above with 32 qubit data package and 10% results used for verification.

```

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 Teleportation. 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
53 25 9 0 0
40 15 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 13: BB84 with Statistics and Quantum Channel

Behind the Console (Expert) Program flow path is as Simple BB84 case. The only differences are the step 4 and a sampling step is added,

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

For statistical quantity of success times which characterizes the channels in BB84 protocol, we execute this experiment 100 shots for each channel. In every shot for each channel, different sampling percentages and package lengths are considered. Finally, tables and figures are given to find the trade-off between success times within different sampling proportions and package lengths on quantum channels.

Different Channels and Different Proportion for BB84 (Expert) Success times within different sampling percentages on different channels in 100 shots are given by figure 14.

Features and Analysis(Expert) The example generates some ‘error’ bits during communication due to quantum channels. These bits cause a fail connection. Meanwhile, not all error bits can be found in the sampling step because almost half bits are invalid in the measurement step in theory. Besides, sampling step is also probability verification step which means it does not use all agreed bits to verify the communication procedure.

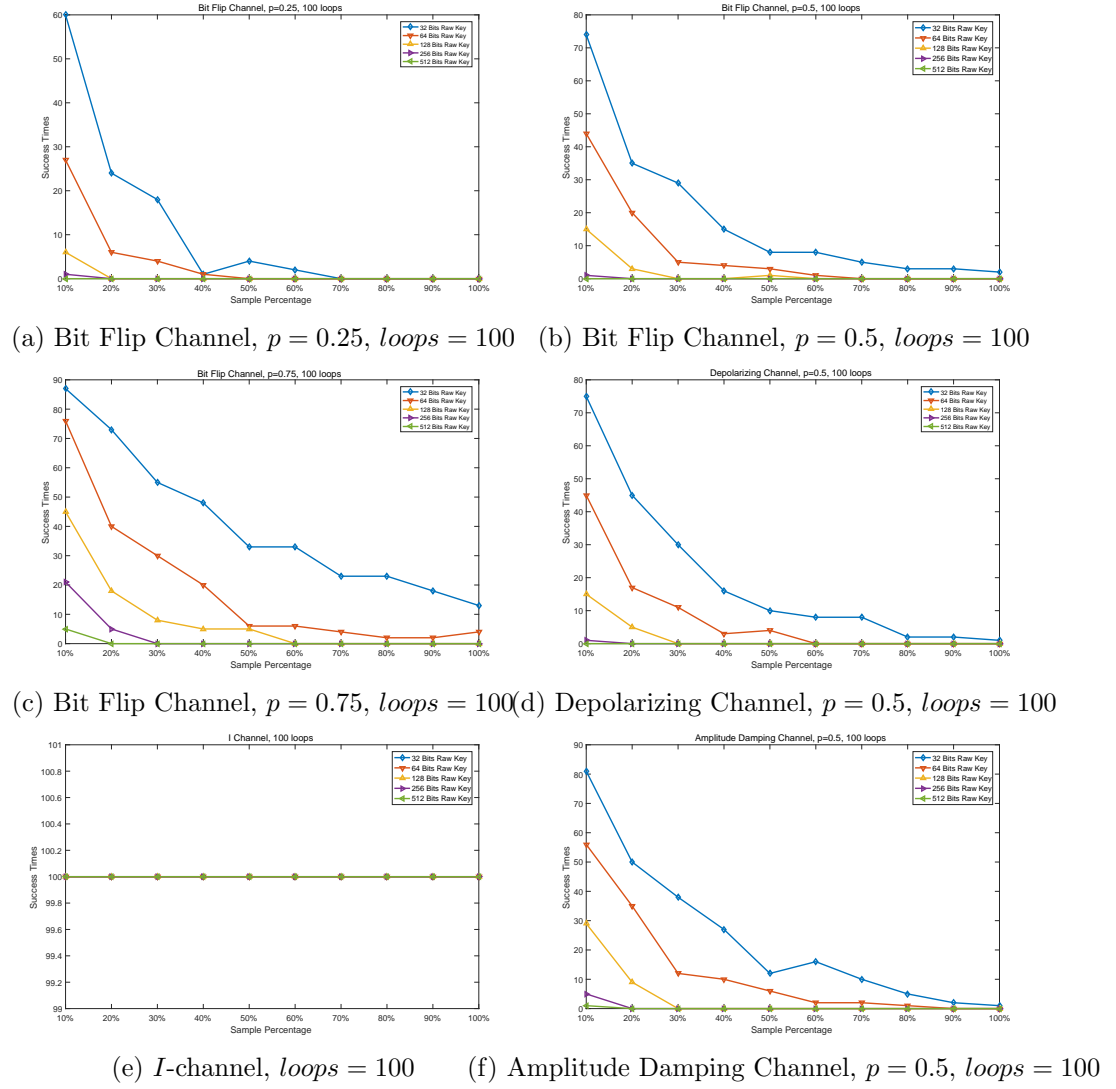


Figure 14: Statistics of success communication via BB84 with channels

Sub-figure (a),(b) and (c) in Figure 14 are bit flip channels with different probabilities. In conclusion, success shots are increasing with the improvement of p and the descent of raw key length. The reason is that p indicates the percentage of information remains in bit flip channel, and the improvement of p means fewer errors in communication. The shorter length of raw key ensures fewer bits are sampled. Sub-figure (d),(e) and (f) illustrate communication ability in the BB84 protocol of other three channels. It should be noticed that the I -identity channel is 100% success which means noiseless channel can keep information intact during transfer procedure.

2.3.4 Quantum Teleportation with QASM-“Option 5”

Input and Output

1. First enter the number 5 to start the Quantum Teleportation with QASM.
2. Console displays that “Quantum Teleportation begins...” and shows the experiment result.
3. Meanwhile, console pops up a notebook and displays the generated “QASM”.
4. The teleportation experiment will be run 1000 times and the statistic results will be shown on the console.

Behind the Console (Expert) The key code segment is trivial that we show it here.

```

01 class TestQuantumHilbert : 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 1 0;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         Register(r3, m(Bob2));
37     }
38 }
39 }

```

Figure 15: Quantum Teleportation with f-QASM

Form Line 01 to Line 16, we define one classical register $r3$, three quantum registers $Alice$, $Bob1$, $Bob2$, four quantum gates $hGate$, $CNot$, $xGate$ and $zGate$ and one measurement m . Line 19 to Line 21 define performing quantum gates on quantum registers. The most interesting part is from Line 22 to Line 35, there are two advanced clause QIf in the code segment. To illustrate the received quantum state is real the $Alice$ state, we perform a measurement in Line 37.

2.3.5 Quantum Google PageRank-“Option 6”

Input and Output

1. First enter the number 6 to start the Quantum Google Rank.
2. Console asks “Please enter the length. . .”, you can just press Enter key to use the default configure.
3. By default, the console displays the adjacency matrix (the corresponding graph is as Figure 16) and shows the step of quantum evolution.
4. After a while, the program shows the averaged quantum pagerank.

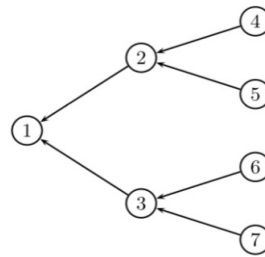


Figure 16: Default Network Structure

Behind the Console (Expert) Coming soon. . .

2.3.6 Quantum Grover Search-“Option 7,71,72,73”

Grover search algorithm is a representative quantum algorithm. It solves the searching problem through a database consisting of N elements, indexed by number $0, 1, \dots, N - 1$ with an oracle answering the position. The algorithm can find the solution with probability $O(1)$ within $O(\sqrt{N})$ steps.

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

Input and Output

1. First enter the number 7 to start Grover Search.
2. Console displays “the default oracle . . .” and the result.
3. On one hand, you can check the result by Grover search algorithm; on the other hand, you should be noticed that the key component is the source code corresponding to this experiment. Do not forget to check the source code.

Behind the Console (Expert) Coming soon. . .

“Option 71”- Standard
Grover Search

Input and Output

1. First enter the number 71 to start Standard Grover Search.
2. The console asks "How large ...". Type 7 for an instance (We advise the number should not be more than 9 on PC, it consume too much memory and time on measurement).
3. Console will ensure the volume of the database and display the number. After that, it asks "where is the correct answers...". Type 56 for an instance. The oracle will be set up in this step.
4. After 8 times oracle called up ($R \leq \frac{\pi}{4} \sqrt{\frac{N}{M}}$), the result has been found as display.

Behind the Console (Expert) The Grover Search algorithm can be described as Figure .

Suppose $|\alpha\rangle = \frac{1}{\sqrt{N-1}} \sum_x |x\rangle$ is not the solution and $|\beta\rangle = \sum'_x |x\rangle$ is the solution where \sum'_x indicates a sum of over all 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 more 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.

“Option 72”- Automatic
toolkits Grover Search

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 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 (Expert) This experiment is an automatic toolkit used for programmers to verify the algorithm. It is quite useful when the programmer consider to generate many oracles.

“Option 73”- Standard
Grover Search

Input and Output

1. First enter the number 73 to start Standard Grover Search.
2. The console asks “How large...”. Type 7 as an instance. (We advise the number should not be more than 9 on PC, it consume too much memory and time on measurement.)
3. Console will ensure the volume of the database and display the number. After that, it asks “where is 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 the target numbers.

Behind the Console (Expert) experiment: multi-objects Grover search algorithm. It supposes more answers (positions) of the right answers in the oracle. We use the strategy that a blind box is added reversing the position where it belongs. In this experiment, we find snow-slide influence of multi-objects Grover search algorithm indicating the algorithm should be modified in some sense.

In the experiment, we added a new blind box (a unitary gate) which reverses the position of the answer where it belongs. In short, Oracle is the matrix with all diagonal elements are 1 except all answer positions are -1 . Thus, the blind box is the diagonal matrix with all elements are 1 except answer position which have been found are -1 . Combine these two boxes, we create a new Oracle which answers all the questions except ones we have found in last rounds.

The measurement shows different probabilities of final result. The theory shows that if we have multi-answers, the state after r times oracles and phase gate 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 as almost $\frac{1}{\sqrt{2}}(|2\rangle + |14\rangle)$. We should get the $|2\rangle$ or $|14\rangle$ in the first time and get the other one in next time. However, we actual get other results with high probability besides $|2\rangle$ and $|14\rangle$ which indicates that the multi-objects search algorithm is not a good one.

It should be noticed that due to multi-objects, the real state after Grover search algorithm becomes $a(|2\rangle + |14\rangle) + b(|1\rangle + |3\rangle + |4\rangle + |5\rangle + \dots)$ 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 wrong index is measured, the algorithm would create a wrong blind box. It reverses the wrong position of Oracle, i.e., it adds a new answer to the questions. In next round, the proportion of right answer would reduce further. For example, in last example we get a wrong answer by measurement, say $|5\rangle$. After new procedure, the state would become: $a(|2\rangle + |14\rangle + |5\rangle) + b(|1\rangle + |3\rangle + |4\rangle + |5\rangle + \dots)$. It becomes more harder to find the correct answer with this state.

In conclusion, the wrong answer in one round causes avalanche effect errors in Grover search algorithm.

2.3.7 A comprehensive Quantum Teleportation-“Option 8”

Input and Output Coming soon...

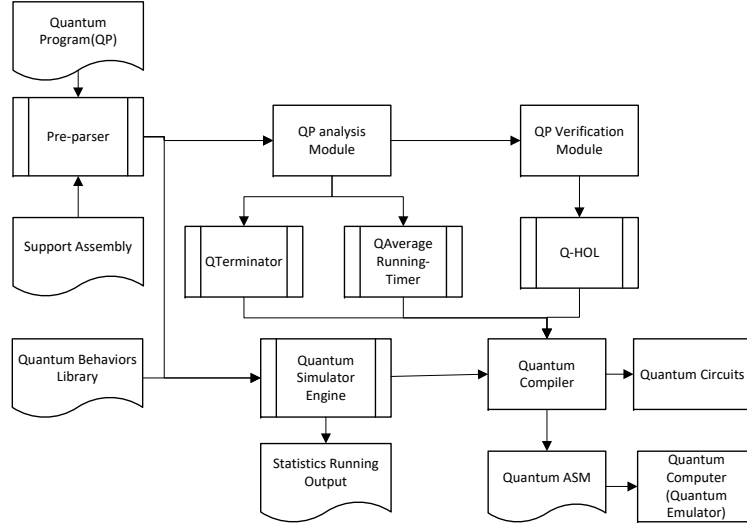


Figure 17: $Q|SI\rangle$ framework

3 Serious Coding

After a taste of examples on $Q|SI\rangle$, you may want to have a try on your code. Please be patient, a good start is following this section.

In this section, we will firstly instruct $Q|SI\rangle$ structure. $Q|SI\rangle$ is different from other quantum platform. You may need some background both in quantum information, computer science and the platform information. $Q|SI\rangle$ structure can help you to find what is the code workflow and how to adjust code segment responding to your requirement.

In the second subsection, a very impressive example “Qloop” as the first “HelloWorld” in quantum world will be showed to illustrate the power of our platform. Following this example, you can acknowledge some components in $Q|SI\rangle$.

In the last subsection, you could learn the details of quantum **while**-language. Quantum **while**-language is an elegant programming language which can enhance you with the loop power.

3.1 $Q|SI\rangle$ Structure (expert)

In this subsection, we introduce the basic structure of $Q|SI\rangle$. $Q|SI\rangle$ is designed to offer a unified general-purpose programming environment supporting the quantum **while**-language. It includes a compiler for quantum **while**-programs, a simulator of quantum computation, and a module for analysis and verification of quantum programs. We implement $Q|SI\rangle$ as a deeply embedded domain-specific platform for quantum programming with host language C#. The framework of $Q|SI\rangle$ is showed in the figure 17.

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

Language Supporting : $Q|SI\rangle$ is a the first platform supporting the quantum **while**-language. In particular, it allows us to program with the measurement-based case statement and **while**-loop. These two program structures provide us with the more efficient and clearer descriptions

of some quantum algorithms such as quantum walks and Grover search.

- Quantum Type Enriched : Comparing with other simulators and analysis tools, $Q|SI\rangle$ supports several kinds of quantum types such as “Ket” and “Qbit” in quantum compute engine. These types have unified operations and can be used in different scenarios. This feature provides high flexible usability and facilitates the programming process.
- Two Modes : $Q|SI\rangle$ has two executable modes. One is the “Running-time execution”, which shows one-shot experiment by simulating quantum behaviors. The other is “Static execution”, which is designed mainly for quantum program compilation, analysis and verification.
- Instruction Set f-QASM : We define an extension of QASM (Quantum Assembly Language), namely f-QASM. It is essentially a quantum circuit description language that can be adapted to different purposes. In this language, there is only one command in every line. Compared with the original QASM [Svore et al., 2006] and space compact QASM-HL [JavadiAbhari et al., 2014], f-QASM contains more information with its ‘goto’ structure. It also can be used for further optimization and analysis.
- Quantum Circuits Generated : Similar to modern digital circuits in classical computing, quantum circuits provide a low-level representation of quantum algorithms [Svore et al., 2006]. Our compiler produces a quantum circuit from an program written in the high-level quantum **while**-language.
- Arbitrary Unitary Operator : In our platform, we implement the Solovay-Kitaev algorithm [Dawson and Nielsen, 2005] together with two-level matrix decomposition [Nielsen and Chuang, 2010] and quantum multicomplexer [Shende et al., 2006]. Therefore, an arbitrary unitary operator can be transferred into a quantum circuit consisting of quantum gates from a small pre-defined set of basic gates that will be provided by the future quantum chip manufactures.

3.2 Main Components of $Q|SI\rangle$

The platform is mainly constituted by four parts:

- Quantum Simulation Engine : This component includes Support Assemblies, Quantum Behaviors Library and Calculation Engine. Support Assemblies provide supporting of quantum types and quantum language. More precisely, it provides a series of quantum objects and reentrant wrapped functions to play the role of quantum program constructs **if** and **while**. Quantum Behaviors Library provides the behaviors of quantum objects such as unitary transformation and measurement including the result and post-state. Calculation Engine is designed as an execution engine. It accepts quantum objects and the rules from Quantum Behaviors Library and transfers them into probability programming running steps.
- Quantum Program (QP) Analysis Module : Currently, this module contributes two sub-modules for static analysis: QTerminator and QAverage Running-Timer. The former provides the termination information, and the latter evaluates running time of the given program. Their outputs are sent to quantum compiler at the next stage for further usage.
- QP Verification Module : (Coming Soon...) This module is a tool for verification of the correctness of quantum programs, based on the quantum Hoare

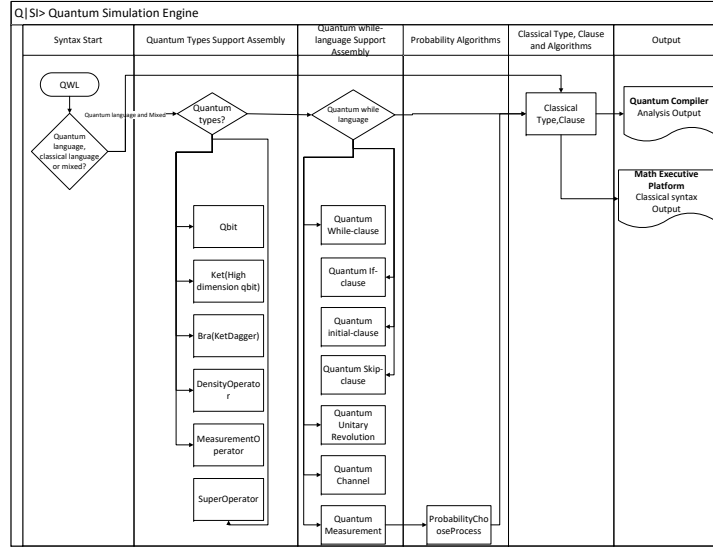


Figure 18: $Q|SI\rangle$ Pre-parser

logic introduced by one of the authors in [Ying, 2011]. It is still under development. One possibility is to link $Q|SI\rangle$ to the quantum theorem prover developed by Liu et al [Liu et al., 2016].

Quantum Compiler : The compiler consists of a series of tools mapping a high-level source program representing a quantum algorithm onto a quantum device related language [Svore et al., 2006], e.g. f-QASM and further to a quantum circuit. Our target is to implement any source codes without considering the details of devices, i.e. to automatically construct quantum circuits based on the source codes. Also, a tool for optimization of quantum circuits will be added into the compiler in the future.

Implementation of $Q|SI\rangle$ One of the basic problems in the implementation of the platform is how to use probabilistic and classical algorithms to simulate quantum behaviors. In order to support quantum operations, $Q|SI\rangle$ has been enriched with data structures from quantum simulation engine. Figure 18 shows the process and procedure of quantum simulation engine. In this simulation engine, we support two types of languages: one is the pure quantum **while**-language and the other is a mixed quantum **while**-language. The engine starts a support path flow when it detects the quantum part of a program. After that, the engine checks quantum type for every variable and operator and then executes the corresponding support assembly. As said before, one main feature of $Q|SI\rangle$ is that it supports programming in the quantum **while**-language. The feature is provided by the quantum **while**-language support assemblies. All of the quantum behaviors are explained by probabilistic algorithms on a classical computer. Outputs are extended C# languages which can be run on the .net framework or be explained to f-QASM and quantum circuits by the compiler.

3.3 From UnitTest to HelloWorld

This subsection is a brief tutorial for writing a standard quantum code segment. Tutorial includes two main parts: quantum program and related classical function program.

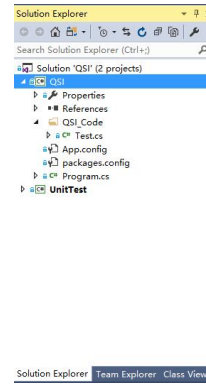


Figure 19: Entry to your code

Entry to your code You could start your code from project QSI. QSI (Project) is designed for user's code that it does not include too many assumptions and definitions compared with UnitTest (Project).

PATH: Solution 'Testudo' → QSI(Project) 19

Writing Quantum Code

3.3.1 Code File

`Test.cs` under the `QSI_Code` (Folder) is a good place to write your code for quantum part.

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

! → You also can write your codes for the quantum part anywhere as you like. But our suggestion is that 'Test.cs' is a good place to start. Some modules such as Termination Analysis module require designating file path (Code for the quantum part). In our example, we designate the targeted file for termination analysis is 'Test.cs' in 'Program.cs' under QSI (Project).

3.3.2 Component Libraries for Quantum

Two .Net assemblies are provided for code for quantum part:

Assemblies: QuantumRuntime, QuantumToolkit .

These two assemblies conclude analogous namespace:

Namespace: QuantumRuntime, QuantumToolkit .

Also, Each of them provides several different sub-namespace:

QuantumRuntime Sub-namespace: Operator

QuantumToolkit Sub-namespace: Parser, Runtime, Type

Suitable Namespace Namespace is an essential part for developing quantum programming. In fact, it is also necessary for developing most C# programs. Unless, you figure out all the functions and objects in namespace, we suggest you introduce following namespaces:

Default Namespaces for Developing: QuantumRuntime, QuantumToolkit, QuantumToolkit.Parser .

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


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

Do not forget to introduce other used assemblies. A full example has been given in the 'Test.cs'. Just try it.

Exploit Namespace (expert) You could exploit our provided assemblies by 'Reflection' giving by Visual Studio.

PATH: Menu → View → Object Browser

3.3.3 Import Static Functions

To allow you 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 static functions in common use 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;
```

3.3.4 Inherit is Essential for Reducing Code Complexity

A parent class called **QEnv** is provided to reduce code complexity. In fact, the **QEnv** is the quantum programming environment base class. It inherited from **MarshalByRefObject** to ensure the separated and clean environment for each experiment.

In our **Test.cs**, we have provided the example as following. Please leave it:

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

3.3.5 Declare Registers, Quantum Registers, Quantum Gates and Quantum Measurement

A variable is nothing but 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 layout of the variable's memory the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The basic quantum related types and initial methods provided in $Q|SI\rangle$ can be categorized as:

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

Two kinds of registers, classical and quantum registers, need a number indicating the dimension of the object and a matrix for filling the object

in initialization function. In the code for the quantum part, the dimension is derived implicitly by $Q|SI\rangle$ and thus it can be ignored by the user. In addition, unitary gate, quantum channel, and measurement do not rely on dimension information in concept, i.e., all inputs can only be matrix,

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]")

3.3.6 run() as Container for Quantum Circuit (Workflow)

After defining the “input”, Gates (Unitary Matrix) measurement and initial quantum state, you could write your quantum circuits inside the function `run()`.

Here we give a very brief example to illustrate the `run()` function.

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

- ! → Gate should match with the number qubits. In most scenario, single qubit gate and CNOT is enough for quantum circuits. But, our $Q|SI\rangle$ also support other dimension of unitary matrix. Any given multi-qubits gates (exclude CNOT) would be translated to single unitary gates and CNOT
- ! → Do not forget to collect results using measurement. Unless doing a measurement, you have zero information about the quantum states. Also, due to quantum mechanics, the repetitive experiment is usually in a general case.

Writing Classical Code

A standard program should include classical and quantum codes. In the last part, we learn that how to write a quantum code is the key to the experiment. But, the classical code is also essential for an experiment. As a programming environment, several high-level functions such as analysis, executing, compilation is only provided in classical code. Also, as a flexible environment, controlling every thing as you described is a cool setting for the programmer. We suggest that all the classical codes should be placed in ‘`Program.cs`’ which is also the entry of the project in C# language.

- ! → At the very beginning, we should notice that there are two internal modes for $Q|SI\rangle$: static mode and dynamic modes. Static mode is used for analysis concerning programming structure such as termination analysis, compilation whereas dynamic mode is used for ‘executing’. Remember this can help you avoid some unknown errors.

3.3.7 Creat, Initilize, Run and Collect

Creat The first step is creating an instance of quantum code which you have written in `Test.cs`.

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

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

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

Initilize Initialization is a prepared step that makes up all the quantum objects available to the classical code. Also, it initializes the background variables and input matrix.

Init: `test.Init();`

Run After all the things poise, the key steps begin to be involved.

Run: `test.Run();`

! → You should notice that `test.Init();` only need to be executed for once while `test.Run();` can be executed for many times. In fact, most quantum experiment needs a statistical results which means `test.Run();` is required for many time according to the case. When the function `test.Run();` is executed for more than one time, DO NOT FORGET TO USE `test.InitSuperRegister();` TO RESET ALL REGISTERS, otherwise, last execution may pollute or disturb the next round experiment. A general case is a loop in classical code part.

Collect Results need to be “collected” in the classical code part. All the classical registers are NOT can be viewed or indexed by the functions belongs to classical code part in default. Thus, a “public” modifier is a good choice that makes the collection by other programs possible.

A display process is shown here as a collection.

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

3.4 Using Quantum while-language

3.4.1 Brief Instruction to Quantum **while**-language

For reader’s convenience, we briefly review the quantum **while**-language. The quantum **while**-program are generated by the following simple syntax:

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

The quantum **while**-language is a pure quantum language without classical variables. It assumes only a set of quantum variables denoted by symbols q, q_1, q_2, \dots . A quantum register is a sequence \bar{q} of distinct quantum variables. Almost all of the existing quantum algorithms involve both classical computation and quantum computation. So, in practical applications, the quantum **while**-language has to be combined with a classical programming language. The execution of a quantum program can be conveniently described regarding transitions between configurations.

Definition 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.

Here, let give some explanations of the quantum program constructs defined above; for more detailed descriptions and examples, see [Ying, 2011] and Chapter 3 of [Ying, 2016].

Skip :

$$\overline{\langle \mathbf{skip}, \rho \rangle} \rightarrow \langle \mathbf{E}, \rho \rangle .$$

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

Initialisation :

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

where

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

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

Unitary transformation :

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

The statement “ $\bar{q} := U[\bar{q}]$ ” means that a unitary transformation (quantum gate) U is performed on quantum register \bar{q} and leave other variables unchanged.

Sequential Composition :

$$\frac{\langle S_1, \rho \rangle \rightarrow \langle S'_1, \rho \rangle}{\langle S_1; S_2, \rho \rangle \rightarrow \langle S'_1; S_2, \rho \rangle} .$$

As in a classical programming language, in the composition $S_1; S_2$, program S_1 is executed firstly. After S_1 terminates, S_2 is executed.

Case statement :

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

for each possible outcome m of measurement $M = \{M_m\}$. In the case statement **if** $(\Box m \cdot M[\bar{q}] = m \rightarrow S_m)$ **fi**, M is a quantum measurement with m standing for its possible outcomes. To execute this statement, M is firstly 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 the the outcome m and executed. Difference between a classical case statement and a quantum case statement is that the state of quantum program variables \bar{q} is changed after performing the measurement.

while-Loop :

$$\begin{aligned} \text{(L0)} \quad & \overline{\langle \text{while}(M[\bar{q}] = 1) \mathbf{do} S \mathbf{od}, \rho \rangle} \rightarrow \langle E, M_0 \rho M_0^\dagger \rangle , \\ \text{(L1)} \quad & \overline{\langle \text{while}(M[\bar{q}] = 1) \mathbf{do} S \mathbf{od}, \rho \rangle} \rightarrow \langle S, M_m \rho M_m^\dagger \rangle . \end{aligned}$$

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. In its execution, to check the loop guard, M is performed on the quantum register \bar{q} . If the outcome is 0, then the program terminates, and if the outcome is 1 the program executes the loop body S and continues. Note that here the state of program variables \bar{q} is also changed after performing measurement M .

3.4.2 Using Quantum **while**-Language

After the brief instruction of quantum, we can start to code with quantum **while**-language. The new clauses are based on the previous syntax. Thus, the only “To be updated” are two new structure: **qif** and **qwhile**.

QIf **Prototype**

$QIf(\text{Measure}(\text{Qubits}), () \Rightarrow \{\text{Sub_program_1}\}, () \Rightarrow \{\text{Sub_program_2}\}), \dots;$

Measure: is a measurement, it can be a binary output measurement or a multi-bits output measurement.

Qubits: is a set of qubits indexed by a variable.

Sub_program: is a sub-program written by quantum **while**-language. The behavior of this structure is actual a case statement. Firstly, a measurement would be performed on the variable **Qubits**. Then based on the output index, i.e., the output in according to the size of the number, the **Sub_program** will be chosen to execute. For example, **Sub_program_1** is chosen to execute when the measurement gives the result 0 while **Sub_program_2** is chosen to execute when the measurement gives the result 1.

Qwhile **Prototype**

$QWhile(\text{Measure}(\text{Qubits}), () \Rightarrow \{\text{Sub_program}\});$

Measure: is a measurement, it should be a binary output measurement.

Qubits: is a set of qubits indexed by a variable.

Sub_program: is a sub-program written by quantum **while**-language. The behavior of this structure is actual a loop statement. At first, a measurement would be performed on the variable **Qubits**. Then based on the output (NOT output index), i.e., result 0 or result 1, the **Sub_program** may be executed. For, example, result 1 will lead to **Sub_program** be executed. And result 0 will lead to *SKIP* the **Sub_program** and the clause after **QWhile** will be executed.

3.5 A Simple Example

Let’s write a simple example including **QIf** and **QWhile**. The example is a modified quantum teleportation that can illustrate the power of our new structures.

The code is as following Figure 20 shows.

The code segment is similar to the standard quantum teleportation. We should notice that a **QWhile** is added in Line 23. That means the the body of quantum teleportation only be executed when the measurement result is 1. Another modification is Line 30 and Line 34, a **QIf** is added to the body. That show an **Empty** or a unitary matrix **xGate** (or **zGate**) is performed on the qubits according to the result of measurement.

3.6 Check the Terminating

Termination Abstract You may notice that the loop structures give the true power to quantum programming coder while an another serious issue is raised: Termination. When a loop is added, the program may be non-terminating that means the program may never halt.

You may also know that halting problem is a hot topic in classical computer. Moreover, quantum scenario is more complex than classical one. Quantum state can be a superposition state that in simple explaining

```

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     public Quantum Bob1 = MakeDensityOperator("{[0.5 0.5; 0.5 0.5]}");
09     public Quantum Bob2 = MakeDensityOperator("{[1 0; 0 0]}");
10     U.Emit hGate = MakeU("{1/sqrt(2) 1/sqrt(2); 1 / sqrt(2) 0}");
11     U.Emit hGate = MakeU("{1/sqrt(2) 1/sqrt(2); 1 / sqrt(2) 0}");
12     U.Emit CNot = MakeU("{[1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1]}");
13     U.Emit CNot = MakeU("{[1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 1]}");
14     U.Emit xGate = MakeU("{[0 1; 1 0]}");
15     U.Emit zGate = MakeU("{[1 0; 0 -1]}");
16
17     M.Emit m = MakeM("{[1 0; 0 0], [0 0; 0 1]}");
18
19     protected override void run() {
20         hGate(LooperQ);
21         QWhile(m(LooperQ),
22             () => {
23                 hGate(LooperQ);
24                 CNot(Bob1, Bob2); //|00>+|11> for Bob
25                 CNot(Alice, Bob1);
26                 hGate(Alice);
27                 QIf(m(Bob1),
28                     () => {
29                         zGate(Bob2);
30                     });
31                 QIf(m(Alice),
32                     () => {
33                         zGate(Bob2);
34                     });
35                 Register(r3, m(Bob2));
36             });
37     }
38 }

```

Figure 20: Teleportation with Loops

that it can execute on several work-flow. Considering the loop structure is also an influencing factor, the terminating problem is not trivial for most cases.

Reader can find the details for termination analysis module in the article for references.

Check Terminating using
Provided modules

To check the terminating of the given program, you need denote the path of the input file and class name of the targeted program. The following is an example:

```

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);

```

Line 01 and Line 02 denotes the path of the targeted class. Line 03 produces a generator which is a core for static analysis. After that, Line 04 tries to analyze the input class file. Please do not forget to mention the targeted class if there is more than one class in the file. Line 05 is going to find its terminating ability and output it.

The termination analysis only supports the pure quantum codes, i.e., you should NOT mix any classical codes in the class of quantum codes.

3.7 Compilation

QASM (Quantum Assembly Language) is widely used in modern quantum simulators. It was firstly raised in [Svore et al., 2006] 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. After that, the article [Ying and Feng, 2011] carefully characterized its theoretical property. In 2014, A.JavadiAbhari et

al. [JavadiAbhari et al., 2014] defined a space-consuming flat description and denser hierarchical description QASM, called QASM-HL. Recently, Smith et al. [Smith et al., 2016] proposed a hybrid QASM for classical-quantum algorithms and applied it in Quil.

We propose another QASM format called f-QASM (Quantum Assembly Language with feedback). The most significant motivation is to translate the inherent logic in the high-level programming language into a simple command set, i.e., it should be just only one command in every line or period. Another motivation is to solve the following issue raised by IBM QASM 2.0 list: conditional operations to implement feedback based on measurement outcomes.

3.7.1 f-QASM

We introduce an improved quantum assembly language called f-QASM (Quantum Assembly Language with feedback) which can be used for the compiler on the quantum simulator and the real quantum processor. A set of measurement-based operations is defined to execute **if** and **while** structure. Utilizing the improved quantum assembly language, the compiler can compile a high-level quantum language which includes loop and case-statement to a low-level device-independent or device-dependent instruction set. Let us define the registers firstly:

- Define $\{r_1, r_2, \dots\}$ as the finite classical registers.
- Define $\{q_1, q_2, \dots\}$ as the finite quantum registers.
- Define $\{fr_1, fr_2, \dots\}$ as the finite flag registers. They are a special kind of classical registers which are often used to illustrate partial results of the code segment. For 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, i.e. it is 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, it can only be used after decomposing it 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.

3.7.2 Compile Quantum Program using Compile Module

Compile Module **Prototype**

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

```
QAsm.WriteQAsmText(Parameter_1);
```

The first command is the quantum compilation command.

Parameter_1 is the class name which is going to be compiled.

Parameter_2 is a number indicating how “deep” quantum circuits you will generate. The default number is 0. We will explain it in next section.

Parameter_3 is an enumeration type consisting two options:

1. Matlab.PreSKMethod.OriginalQSD,
2. Matlab.PreSKMethod.OriginalQR.

Parameter_4 is fixed OperatorTreeNodeType which is generated by static analysis module.

The second command is to write QASM-f into a file. **Parameter_1** is a bool value: **true** or **false**. **true** means it will open the QASM-f file automatically after program compilation.

4 Explore APIs

Coming soon.

Appendix

A Contributors

Key Members:	Prof. Mingsheng Ying	mingshengying@gmail.com
	Prof. Runyao Duan	runyao@gmail.com
	Shusen Liu	shusen.liu88@gmail.com
	Yang He	wildfire_810@gmail.com
Other Contributor:	Ji Guan	guanji1992@gmail.com
	Li Zhou	zhou31416@gmail.com
	Xin Wang	wangxinfelix@gmail.com

B Contact Information

Any issues and comments are welcomed. Feel free to show your voice, we are hearing. You can update your feedbacks via two methods.

- Submit your voice via the project repository. The link is <https://github.com/klinus9542/QSI>.
- Submit your voice to Email address. <mailto:shusen.liu88@gmail.com>

References

- [Bennett and Brassard, 2014] Bennett, C. H. and Brassard, G. (2014). Quantum cryptography: Public key distribution and coin tossing. *Theoretical computer science*, 560:7–11.
- [Dawson and Nielsen, 2005] Dawson, C. M. and Nielsen, M. A. (2005). The solovay-kitaev algorithm. *arXiv preprint quant-ph/0505030*.
- [JavadiAbhari et al., 2014] JavadiAbhari, A., Patil, S., Kudrow, D., Heckey, J., Lvov, A., Chong, F. T., and Martonosi, M. (2014). Scaffold: a framework for compilation and analysis of quantum computing programs. page 1.
- [Liu et al., 2016] Liu, T., Li, Y., Wang, S., Ying, M., and Zhan, N. (2016). A theorem prover for quantum hoare logic and its applications. *arXiv preprint arXiv:1601.03835*.
- [Nielsen and Chuang, 2010] Nielsen, M. A. and Chuang, I. L. (2010). *Quantum computation and quantum information*. Cambridge University Press.
- [Shende et al., 2006] Shende, V., Bullock, S., and Markov, I. (2006). Synthesis of quantum-logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1000–1010.
- [Shor and Preskill, 2000] Shor, P. W. and Preskill, J. (2000). Simple proof of security of the bb84 quantum key distribution protocol. *Physical review letters*, 85(2):441.
- [Smith et al., 2016] Smith, R. S., Curtis, M. J., and Zeng, W. J. (2016). A practical quantum instruction set architecture. *arXiv preprint arXiv:1608.03355*.
- [Svore et al., 2006] Svore, K. M., Aho, A. V., Cross, A. W., Chuang, I., and Markov, I. L. (2006). A layered software architecture for quantum computing design tools. *IEEE Computer*, 39(1):74–83.

- [Ying, 2011] Ying, M. (2011). Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6):19.
- [Ying, 2016] Ying, M. (2016). *Foundations of Quantum Programming*. Morgan Kaufmann.
- [Ying and Feng, 2011] Ying, M. and Feng, Y. (2011). A flowchart language for quantum programming. *IEEE Transactions on Software Engineering*, 37(4):466–485.