$Q|SI\rangle$: A QUANTUM PROGRAMMINGENVIRONMENT

Manual Author: Shusen Liu
cqsi.service@gmail.com
June-30-2017
Beta Version 0.1 (Draft)

Abstract

This document is the $Q|SI\rangle$ user guide. By following this guide, users should be able to setup their own software environment to support all the functions of $Q|SI\rangle$. This guide also provides the APIs for advanced user application along with several illustrated examples.

We would like to thank the students of $Q|SI\rangle$ for their assistance with the testing process, and ** for comments that greatly improved the manual.

We would also like to show our gratitude to Paul Pham for sharing his version for Python and the Solovay-Kitaev algorithm for improving the programming environment. Modified versions of both were integrated into our platform.

This manual is an addition to the platform article and the termination module article. All feedback is welcome. Please refer to Appendix B for further details on how best to submit your feedback.

Contents

1 An introduction to $Q|SI\rangle$

1.1 About this guide and $Q|SI\rangle$

This guide describes how to use the $Q|SI\rangle$ quantum programming environment. $Q|SI\rangle$ is used for quantum programming in a quantum extension of the while-language, which is embedded in the .Net language. It includes a compiler for the quantum **while**-language and useful tools for simulating quantum computation, analyzing and verifying quantum programs, and for optimizing quantum circuits. We would like to introduce the platform by giving you a brief overview of its main features and requirements.

What can be done by $Q|SI\rangle$: $Q|SI\rangle$ is a quantum programming platform that allows you to program quantum algorithms. Once your algorithm has been coded, you have several powerful options. You can: execute it with an inherent mathematical calculation engine; compile it using f-QASM (Quantum Assembly Language with feedback) and some predefined gates; debug it through the inbuilt static analysis mode which includes both termination and average running time modules, or verify the correctness of your programming according to your requirements.

What can **NOT** be done by $Q|SI\rangle$: Before getting started, please note that $Q|SI\rangle$ is solely a quantum programming platform. Any physics experiments will still require a cloud physics platform, such as IBMQ. Additionally, how many quantum variables $Q|SI\rangle$ can emulate is limited by the capacity of your home PC.

Why you need $Q|SI\rangle$: $Q|SI\rangle$ provides a very significant quantum environment. While other platforms do have their advantages, the advantage of $Q|SI\rangle$ is its potential to connect different platforms together to explore quantum algorithms.

How to use $Q|SI\rangle$: All you need to host $Q|SI\rangle$ is a PC. Most modules have been developed for Windows and, therefore, we recommend Windows 10 Pro. We are currently developing a cross-platform version. This document includes all you need to know to set up your environment and begin programming in $Q|SI\rangle$. The corresponding authors also welcome any feedback and questions. You can submit your suggestions and comments via Github or email. Feel free to contact us.

1.2 Platform requirements

The programming environment has been developed in Visual Studio for Windows. The hardware and software requirements are based on a PC platform, and the minimum recommendations for executing $Q|SI\rangle$ are provided below. Users may use similar configurations at their own discretion.

1.2.1 Hardware Requirement

CPU: Intel Core i5 or above and/(or compatible above CPU). For decomposition gate function, you may need Aan Intel Core i7 is recommended for decomposition gate functions to reach a reasonable and acceptable execution speed.

Memory: 4GB DDR4 or above. For termination analysis, 4GB memory is the basic requirement. Note that quantum emulation and simulation (execution) require exponentially increasing memory for extra quantum objects.

Hard Disk: A 2GB hard disk is required for basic functions. Predefined gates for compiling requires an extra 2GB (for length 1-16 predefined gates).

Laptop: A standard Surface Pro i5/i7 or equivalent is suitable for $Q|SI\rangle$.

1.2.2 Software requirements

OS: Windows 10 Professional x86-64. Other Windows versions (Windows 7 and Windows 8) with .Net Framework 4.6.1 are NOT recommended as the libraries downloaded by NuGet cause potential conflicts.

Visual Studio: An up-to-date version of Visual Studio 2017 Enterprise is strongly recommended. It includes many exciting features for advanced development. DGML (Directed Graph Markup Language), which is used for drawing single qubit quantum circuits, is only supported by Visual Studio Professional versions and higher. Also note that the next release of $Q|SI\rangle$ will support multi-qubits circuits. Visual Studio 2017 Community supports all required features except for DGML drawing.

Matlab: Matlab 2017a is required for analysis of the input matrix and some of the matrix calculations. Since MATLAB is commercially licensed software, a compiled .Net assembly has been provided in the installation folder to support these components of $Q|SI\rangle$. If you do not have an up-to-date MATLAB license, click on “Support Matlab Assembly.exe to complete your installation.

Python: Python 2.7 is required for the decomposition algorithm in the compiler. Luckily, we have been able to package all essential Python functions into the execution file using the PyInstaller. Users should not have to worry about the Python environment and related installation details.

1.2.3 Virtual machine support

For convenience, we have provided a mirror of the environment. All the components have been configured as described in this guide. Users need only download the **Oracle VM VirtualBox** and execute it with our mirror.

1.2.4 Cloud computation

Cloud computation will be provided soon. Please keep an eye on `http://www.qcompiler.com` for further information.

1.3 Setting up the environment step by step

1.3.1 Choosing a suitable installation file

Online setup file `QSI_Online_Setup` is a 35 MB compressed file that includes all the setup components and fetches the remaining components during installation. The folder structure is shown in Figure 1. The main folder includes four components: `Addons, Installation, QSIMain and Manual`. The `Addons` folder is used for non-core components. Currently, it only includes the decomposition addon. The Installation folder consists of `vs_community.exe (Online)` and `Matlab_Runtime.exe (Online)`. All of them are used for setting executing environment. `QSIMain` is the main project folder. The Manual folder includes all the information researchers may need.

Users in Australia and China BGP with an ADSL+(NBN)-equivalent internet connection are encouraged to use this portable setup file.

Offline setup file `QSI_Offline_Basic_Setup` is an offline version equipped with all the essential dlls. Installation can be completed without an Internet connection. The setup file is approximately 300MB, which can easily be distributed with a portable device. This installation does assume that you already have Visual Studio 2017 and MATLAB 2016a (or later) installed.

The only difference between `QSI_Offline_Basic_Setup` and `QSI_Online_Setup` is that all the files downloaded by NuGet during the `QSI_Online_Setup` have already been prepared. Both installation packages have the same folder structure.

Most education users will benefit from `QSI_Offline_Basic_Setup`.

Complete offline setup file `QSI_Offline_Full_Setup` is an offline version equipped with all the essential dlls, plus a full installation of Visual Studio 2017 and a full installation of MATLAB Runtime. The package size is EXCEEDS 5GB, but installation of $Q|SI\rangle$, Visual Studio 2017, and MATLAB can be completed without an Internet connection. It consists of `vs_community.exe (offline)`, `Matlab_Runtime.exe (offline)`, and other folders as `QSI_Online_Setup`. This installation file is useful for users with an unstable internet connection. The user in China Education Network (CERNET) can benefit from this all-in-one setup file.

1.3.2 Installation

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

If you do not have Visual Studio and MATLAB installed, follow the additional setup procedure below.

1. Double-click `vs_community.exe`. Even though we recommend the commercially licensed Enterprise version of Visual Studio because it

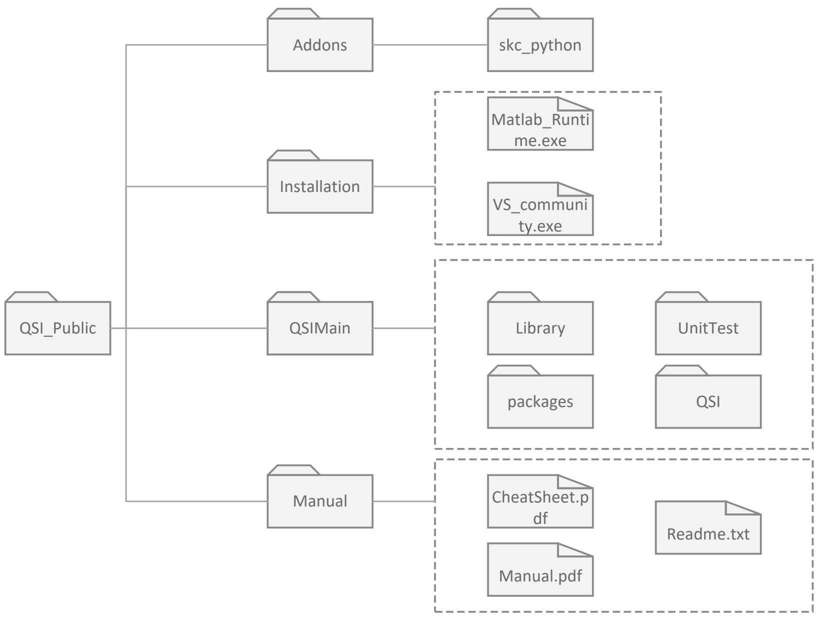


Figure 1: QSI_Offline_Basic_Setup Folder Structure

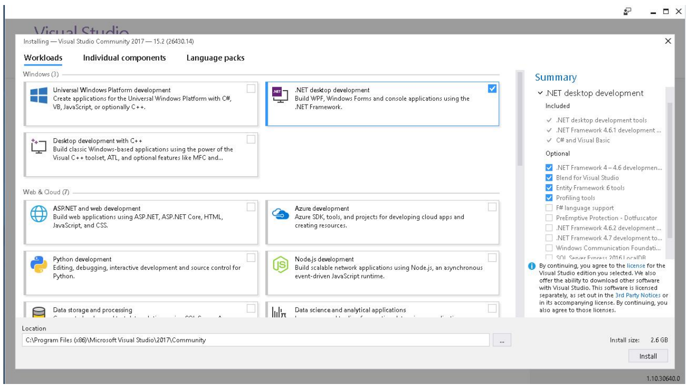
includes many productive toolkits, the Community version is also acceptable. However, note that the DGML component of Visual Studio Community is not able to draw one qubit quantum circuits.

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

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

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

5. Menu → Debug → Start Debugging. Eureka!


Installing — Visual Studio Community 2017 — 15.2 (26430.14)

Workloads | Individual components | Language packs

Windows (3)

Universal Windows Platform development
Create applications for the Universal Windows Platform with C#, VB, JavaScript, or optionally C++.

.NET desktop development [✓]
Build WPF, Windows Forms and console applications using the .NET Framework.

Desktop development with C++
Build classic Windows-based applications using the power of the Visual C++ toolset, ATL, and optional features like MFC and...

Web & Cloud (7)

ASP.NET and web development
Build web applications using ASP.NET, ASP.NET Core, HTML, JavaScript, and CSS.

Azure development
Azure SDK, tools, and projects for developing cloud apps and creating resources.

Python development
Editing, debugging, interactive development and source control for Python.

Node.js development
Build scalable network applications using Node.js, an asynchronous event-driven JavaScript runtime.

Data storage and processing

Data science and analytical applications

Summary

.NET desktop development

Included
- ✓ .NET desktop development tools
- ✓ .NET Framework 4.6.1 development ...
- ✓ C# and Visual Basic

Optional
- [✓] .NET Framework 4 – 4.6 developmen...
- [✓] Blend for Visual Studio
- [✓] Entity Framework 6 tools
- [✓] Profiling tools
- [ ] F# language support
- [ ] PreEmptive Protection - Dotfuscator
- [ ] .NET Framework 4.6.2 development ...
- [ ] .NET Framework 4.7 development to...
- [ ] Windows Communication Foundati...
- [ ] SQL Server Express 2016 LocalDB

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

Location
C:\Program Files (x86)\Microsoft Visual Studio\2017\Community

Install size: 2.6 GB

Install

1.10.30640.0

Figure 2: .NET desktop development

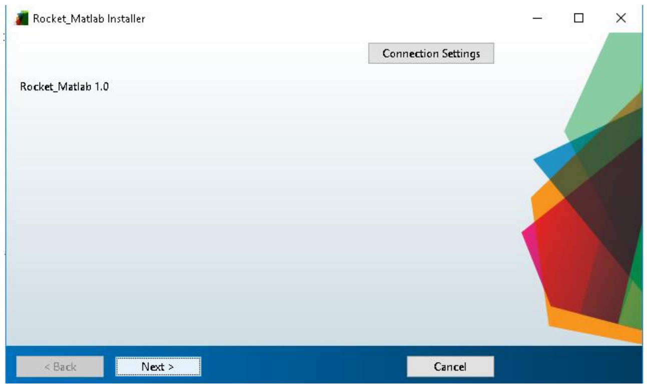


Figure 3: Support MATLAB Assembly

1.3.3 Installing the advanced compiler

Advanced functionality in the form of the Solovay-Kitaev decomposition algorithm is provided as an add-on for the compiler. This is included as a binary file in the Addon folder. Note that you do not need to install or configure this.

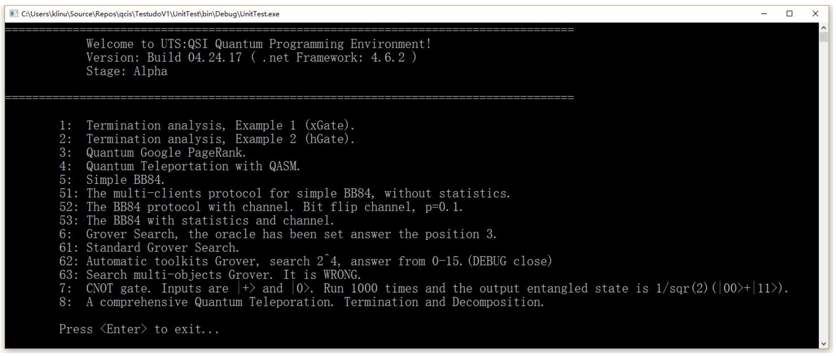
2 Runtime examples

2.1 Getting started from the Console interface

After configuring $Q|SI\rangle$, the best place to start is with the examples provided in the software packages.

Please note that the examples are not unified into a pure second generation language. Some coding is programmed within the engine layer (basic) language for testing and debugging reasons. These code segments are only for preview purposes.

1. Ensure the system environment has been configured correctly.
2. Menu → Debug → Start Debugging (or press F5).
3. Wait a few seconds. (The first execution of a project may take more than 2 minutes to build depending on your hardware.)
4. Eureka! Your display should look like Figure 4. Welcome to the quantum world.



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

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


      1:  Termination analysis, Example 1 (xGate).
      2:  Termination analysis, Example 2 (hGate).
      3:  Quantum Google PageRank.
      4:  Quantum Teleportation with QASM.
      5:  Simple BB84.
      51: The multi-clients protocol for simple BB84, without statistics.
      52: The BB84 protocol with channel. Bit flip channel, p=0.1.
      53: The BB84 with statistics and channel.
      6:  Grover Search, the oracle has been set answer the position 3.
      61: Standard Grover Search.
      62: Automatic toolkits Grover, search 2^4, answer from 0-15.(DEBUG close)
      63: Search multi-objects Grover. It is WRONG.
      7:  CNOT gate. Inputs are |+> and |0>. Run 1000 times and the output entangled state is 1/sqr(2)(|00>+|11>).
      8:  A comprehensive Quantum Teleporation. Termination and Decomposition.

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

Figure 4: Welcome to $Q|SI\rangle$

2.2 Choose an example and find the result

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

Note that some options may require slight modifications to the code, such as "Option 62". The following subsections explain and illustrate details of the experiments.

2.3 Details of the examples

Each of the following subsections displays and analyzes the details of the examples. For your convenience, the coding for all the examples has been assembled in the Project "UnitTest" under the root Solution, as shown in Figure 5.

2.3.1 CNOT gate showcase-"Option 1"

Input and output

1. Enter the number 1 to start the CNOT Gate Showcase.

2. The console should display “The experiment begins…”. Be patient. The experiment will execute about 1000 times and show the statistic result as Figure 6.

Behind the Console (Expert) This is a very simple and standard example of $Q|SI\rangle$. This code segment is so trivial that it is listed 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 $q2 = |0\rangle\langle 0|$. The CNOT gate (from $q1$ to $q2$) is defined as CNOT $= [1000; 0100; 0001; 0010]$. After the CNOT

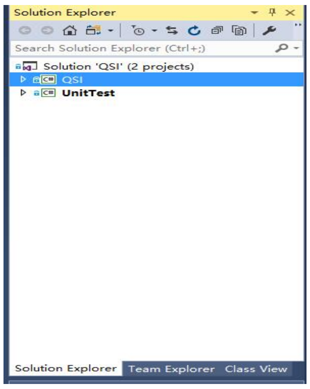


Figure 5: Examples in Project “UnitTest”

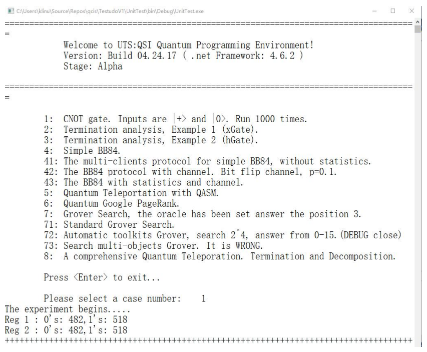

```
C:\Users\klinu\Source\Repos\qcis\TestudoV1\UnitTest\bin\Debug\UnitTest.exe
==============================================================================
=
          Welcome to UTS:QSI Quantum Programming Environment!
          Version: Build 04.24.17 ( .net Framework: 4.6.2 )
          Stage: Alpha

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

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

        Press <Enter> to exit...

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

Figure 6: CNOT Experiment

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

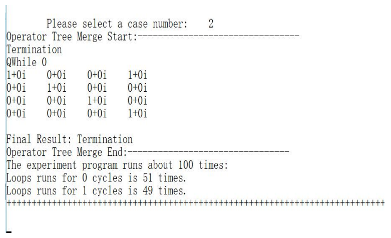
- only consider $q2$, And about half the time the result should be 0 and half the time should be 1.
- Consider results for both $q1$ and $q2$. The 0 result for $q1$ should be the same as the 0 result for $q2$.

Figure 6 illustrates the results just as we predicate.

2.3.2 Termination Analysis-"Option 2,3"

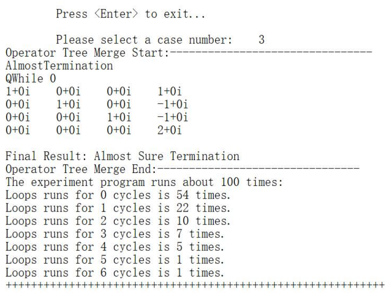
Input and Output

1. Enter the number 2 (or 3) to start the termination analysis example.
2. The Console should display the termination ability of the corresponding example. You can see the "Final Result: Termination" 7 from Example 2 and "Final Result: Almost Sure Termination" 8 from Example 3.
3. You also can find the real execution results from the Console.



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

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

_
```

Figure 7: 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 8: Almost Sure Termination

Behind the Console (Expert) A loop raises the question: "Does the quantum program terminate?" The Termination Analysis module partly answers this question. See the article [] for details.

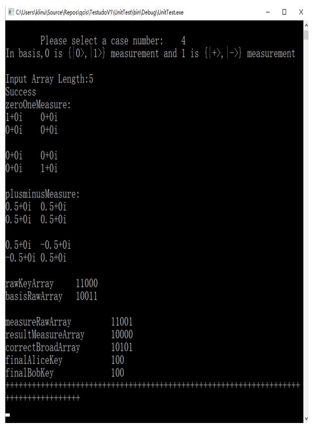
2.3.3 BB84-"Option 4, 41, 42, 43"

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

Simple BB84-"Option 4",

Input and Output

1. Enter the number 4 to start the simple BB84 protocol.
2. The Console will ask for an "Input Array Length". This is the raw initial key length. Input 5 as a test number.
3. The Console will then provide a hint "In basis, 0 is $\{|0\rangle, |1\rangle\}$ measurement and 1 is $\{|+\rangle, |-\rangle\}$" along with other information.
   Let's interpret Figure 9 as an example. The protocol is trying to find consensus for a key length of 5 bits. The initial bits (raw keys) are 11000 and the basis are $\{|+\rangle, |-\rangle\}, \{|0\rangle, |1\rangle\}, \{|0\rangle, |1\rangle\}, \{|+\rangle, |-\rangle\}, \{|+\rangle, |-\rangle\}$. With these two conditions combined, we know that Alice has generated the states $\{|-\rangle, |1\rangle, |0\rangle, |+\rangle, |+\rangle\}$. On the other hand, Bob chooses the basis $\{|+\rangle, |-\rangle\}, \{|+\rangle, |-\rangle\}, \{|0\rangle, |1\rangle\}, \{|0\rangle, |1\rangle\}, \{|+\rangle, |-\rangle\}$. to measure the qubits he received, and he broadcasts these basis. After Alice receives Bob's basis, she checks her chosen basis and finds that the correct positions are $1, 3$ and 5. She broadcasts the correct positions. Bob only keeps positions $1, 3$ and 5 of the measurement results. Both Bob and Alice reach an agreement key of 100.



```
C:\Users\klinu\Source\Repos\qcis\TestudoV1\UnitTest\bin\Debug\UnitTest.exe

        Please select a case number:    4
In basis,0 is {|0>,|1>} measurement and 1 is {|+>,|->} measurement

Input Array Length:5
Success
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     11000
basisRawArray   10011

measureRawArray         11001
resultMeasureArray      10000
correctBroadArray       10101
finalAliceKey           100
finalBobKey             100
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++
```

Figure 9: Simple BB84 Example

Behind the Console (Expert) In this case, a client-server model is used as a prototype for a multi-user communication protocol. A "quantum type converter" is used to convert a '*Ket*' into a density operator. Also, in this case, we use the Quantum Type '*Ket*' and do not consider a quantum channel or an Eve. The flow path is shown in Figure 10.

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

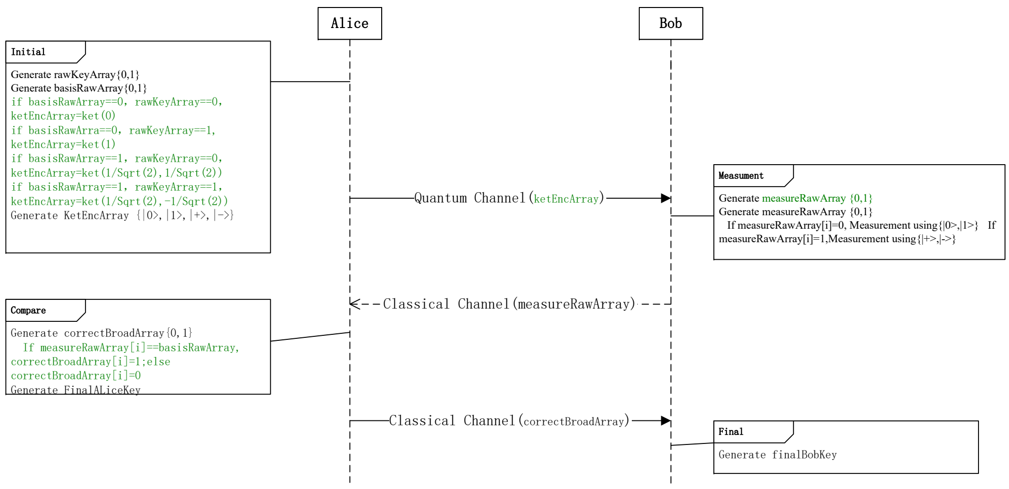

Alice
Bob
Initial
Generate rawKeyArray{0,1}
Generate basisRawArray{0,1}
if basisRawArray==0, rawKeyArray==0, ketEncArray=ket(0)
if basisRawArra==0, rawKeyArray==1, ketEncArray=ket(1)
if basisRawArray==1, rawKeyArray==0, ketEncArray=ket(1/Sqrt(2),1/Sqrt(2))
if basisRawArray==1, rawKeyArray==1, ketEncArray=ket(1/Sqrt(2),-1/Sqrt(2))
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

Figure 10: Simple BB84 protocol

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,

- Quantum Computation Engine Direct Call (QCEDC). This example used some low-level components of $Q|SI\rangle$ (i.e., some functions directly from the quantum computation engine). This mode can be used to design and code quantum communication protocols that cannot easily be described in formal quantum language.
- Client-server Mode. The process uses a client-server model to simulate the BB84 protocol. The model includes many powerful features such as waiting thread and concurrent communication which are 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 length of raw keys.

Multi-clients BB84-"Option 41"

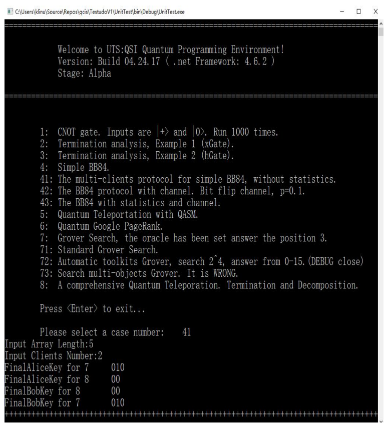
A multi-client BB84 model is a more attractive and practical example. In a multi-client BB84 model, only one Alice generates the raw keys, but there are several Bobs construct an agreement key with Alice.

Input and output

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

Behind the Console (Expert) In this example, users are able to specify the number of clients. Also, a typical BB84 flow path, such as the one outlined in the previous example, would occur for every client-server pair of this model.

- Quantum Computation Engine Direct Call (QCEDC). This example used some low-level components of $Q|SI\rangle$ (i.e., some functions directly from the quantum computation engine). This mode can be used to design and code quantum communication protocols that cannot easily be described in formal quantum languages.



```
C:\Users\klinu\Source\Repos\qcis\TestudoV1\UnitTest\bin\Debug\UnitTest.exe
==========================================================================

          Welcome to UTS:QSI Quantum Programming Environment!
          Version: Build 04.24.17 ( .net Framework: 4.6.2 )
          Stage: Alpha

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


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

       Press <Enter> to exit...

       Please select a case number:    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 11: Multi-clients BB84 Console

- Waiting thread. Many clients are 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. However, this raises a serious question. When a client generates a raw key while the server measures, how can we ensure the server correctly and fairly conducts the measurement?

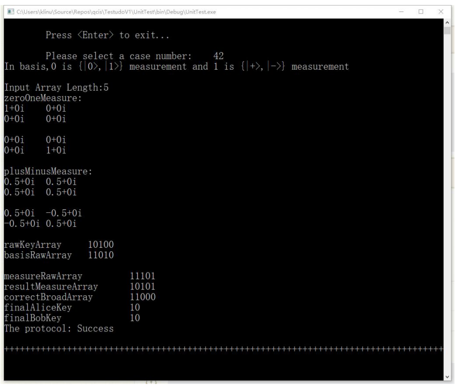
BB84 with a noise channel-"Option 42",

A very interesting and practical topic for the $Q|SI\rangle$ is to use the BB84 model to consider noisy quantum channels. Because no quantum system is ever perfectly closed, quantum operations are key tools for describing the dynamics of open quantum systems.

Input and output

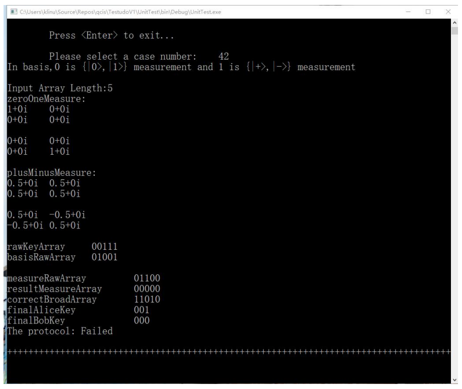
1. Enter the number 42 to start the simple BB84 protocol.
2. The Console will ask for the "Input Array Length". This is the raw initial key length. Input 5 as a test number.
3. Then, the Console will print a hint that "In basis, 0 is $\{|0\rangle, |1\rangle\}$ measurement and 1 is $\{|+\rangle, |-\rangle\}$"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

Let's interpret the Figure 12 and 13 as an example. In Figure 12, the protocol is trying to find consensus for a key length of 5 bits. The initial bits (raw keys) are 10100 and the basis are $\{|+\rangle, |-\rangle\}$,$\{|0\rangle, |1\rangle\}$,$\{|+\rangle, |-\rangle\}$,$\{|0\rangle, |1\rangle\}$,$\{|0\rangle, |1\rangle\}$. With these two conditions combined, we know that Alice has generated the quantum states $\{|-\rangle, |0\rangle, |-\rangle, |0\rangle, |0\rangle\}$ and sends them through a **quantum channel**. On the other hand, Bob chooses the basis $\{|+\rangle, |-\rangle\}, \{|+\rangle, |-\rangle\}, \{|+\rangle, |-\rangle\}, \{|0\rangle, |1\rangle\}, \{|+\rangle, |-\rangle\}$ to measure the qubits he received, and he broadcasts these basis. After Alice receives these basis, she checks her chosen basis, finds the correct positions are 1 and 2. She broadcasts the correct positions. Bob only keeps positions 1 and 2 of the measurement results. Bob and Alice reach an agreement of key 10; however, the keey agreement is not successful. In Figure 13, Alice and Bob follow the protocol outlined above, but, the BB84 fails due to interference in the quantum channel.


C:\Users\klinu\Source\Repos\qcis\TestudoV1\UnitTest\bin\Debug\UnitTest.exe

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

        Please select a case number:    42
In basis,0 is {|0>,|1>} measurement and 1 is {|+>,|->} 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     10100
basisRawArray   11010

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

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 12: BB84 with a Quantum Channel, Success


C:\Users\klinu\Source\Repos\qcis\TestudoV1\UnitTest\bin\Debug\UnitTest.exe

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

        Please select a case number:    42
In basis,0 is {|0>,|1>} measurement and 1 is {|+>,|->} 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 13: BB84 with a Quantum Channel, Failure

Behind the Console (Expert) In this example, different channels, such as the bit flip, depolarizing, amplitude damping, and I-identity channels are described by a Kraus operator. Alice and Bob use these quantum channels to communicate with each other via the BB84 protocol as Figure 10 shows. To ensure successful communication, verification steps also need to be considered.

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

Deplarizing channel, where $p = 0.5$,

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

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

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

And three kinds of bit flip channel:

Bit flip channel, where $p = 0.25$,

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

Bit flip channel, where $p = 0.5$,

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

Bit flip channel, where $p = 0.75$,

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

The program flow path follows the simple BB84 protocol. The only differences are to 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.

BB84 with statics and channels-"Option 43"

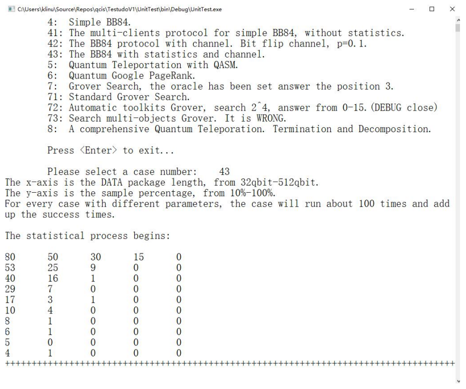
From above example, we know that both the quantum channel and the verification (check) affects the quantum state. In this example, we want to answer the question: What is a suitable proportion for a fixed quantum channel?

Input and output

1. Enter the number 43 to start the BB84 protocol with statistics and channel.
2. The Console will display "x-axis..." and "y-axis...".

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

   A one-time experiment is shown in Figure 14. Let's interpret Figure 14. The BB84 protocol is in an amplitude damping quantum channel (defined as $\mathcal{E} := \{\begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{\sqrt{2}} \end{bmatrix}, \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ 0 & 0 \end{bmatrix}\}$.) In the output, the number in Column 1 and Row 1 means the protocol has an 85% percent chance of success if it performs in the amplitude damping quantum channel defined above with a 32 qubit data packet and 10% of the results are used for verification.



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

        Press <Enter> to exit...

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

The statistical process begins:

80      50      30      15      0
53      25      9       0       0
40      16      1       0       0
29      7       0       0       0
17      3       1       0       0
10      4       0       0       0
8       1       0       0       0
6       1       0       0       0
5       0       0       0       0
4       1       0       0       0
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 14: BB84 with Statistics and Quantum Channel

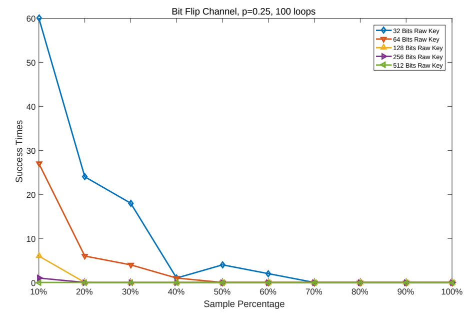
Behind the Console (Expert) The program flow path follows the simple BB84 protocol. The only differences are in Step 4 and a sampling step is added,

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

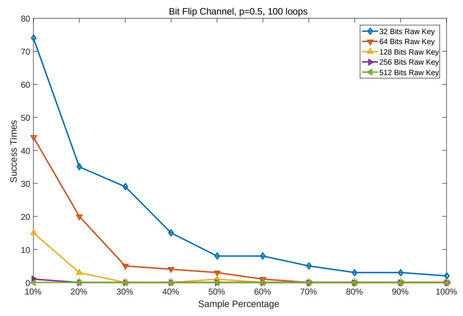
For a statistical quantity characterizing success in a channel in a channel in the BB84 protocol, this experiment should be executed 100 shots for each channel. In every shot for each channel, different sampling percentages and package lengths should be considered. Tables and figures are provided that show the trade-off between success times, different sampling proportions, and package lengths in each of the quantum channels.

Different channels and different proportion for BB84 (Expert) Success times for different sampling percentages in different channels over 100 shots are shown in Figure 15.

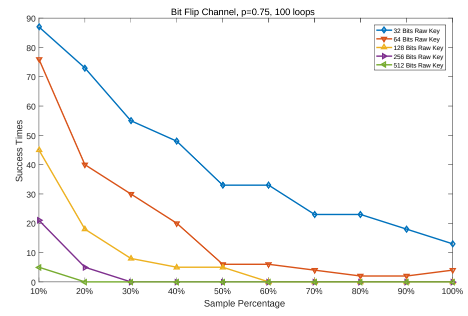
Features and analysis(Expert) The example generates some 'error' bits during communication due to quantum channels. These bits cause a fail connection. Meanwhile, not all error bits can be found in the sampling step because, in theory, almost half bits are invalid in the measurement step. Additionally, the sampling step is also probability verification step



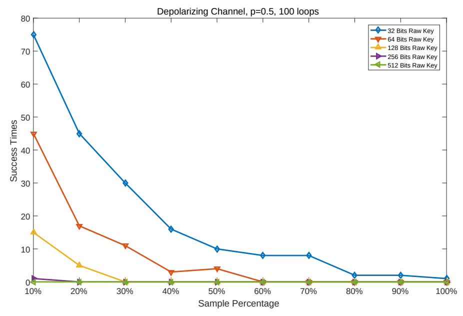
(a) Bit Flip Channel, $p = 0.25$, $loops = 100$



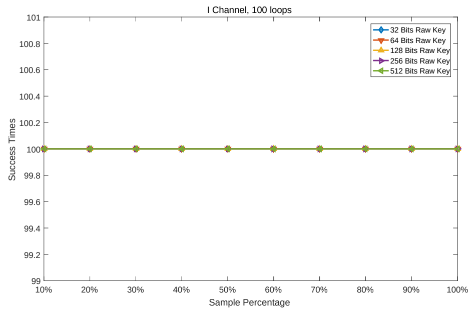
(b) Bit Flip Channel, $p = 0.5$, $loops = 100$



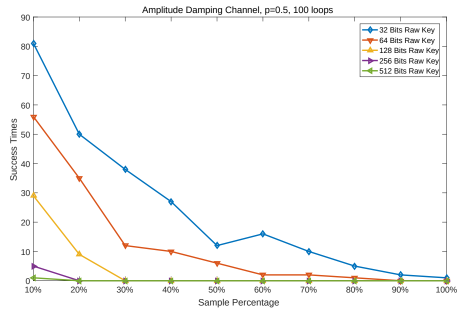
(c) Bit Flip Channel, $p = 0.75$, $loops = 100$



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



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



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

Figure 15: Statistics of success communication via BB84 with channels

which means it does not use all the agreed bits to verify the communication procedure.

Sub-figure (a),(b) and (c) in Figure 15 are bit flip channels with different probabilities. Overall, successful shots increase as p and the raw key length shortens. This is because p is a reflection of the percentage of information remains in bit flip channel, and an increase in p means fewer errors in communication. A shorter length ensures fewer bits are sampled. Sub-figures (d),(e) and (f) illustrate the communication capacity of the BB84 protocol in the other three channels. Note should be noticed 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.

2.3.4 Quantum teleportation with QASM-"Option 5"

Input and output

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

Behind the Console (Expert) The key code segment is trivial enough to show here.

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

Figure 16: Quantum Teleporation with f-QASM

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

2.3.5 Quantum Google pagerank-“Option 6”

Input and output

1. Enter the number 6 to start the Quantum Google Rank.
2. The Console will ask “Please enter the length...”. You can just press Enter the key to use the default configuration.
3. By default, the Console will display the adjacency matrix (see the corresponding graph in Figure 17) and show the steps of the quantum transformation.
4. After a while, the program will show the averaged quantum pagerank.

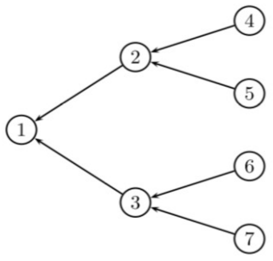


Figure 17: Default Network Structure

Behind the Console (Expert) Coming soon...

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

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

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

Input and output

1. Enter the number 7 to start Grover search.
2. The Console will display “The default oracle ...” and the result.
3. Of course, you can check the results of the algorithm. However the key component of this example is the source code corresponding to the experiment. Don’t forget to check the source code!

Behind the Console (Expert) Coming soon...

“Option 71”- Standard Grover Search

Input and output

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

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

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 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.

Grover search with automatic toolkits - "Option 72"

Input and output

1. Before executing go to *TestGroverH.cs* file, find `//#undef DEBUG` and uncomment it as `#undef DEBUG` at the second line of the file.
2. Then press "F5" to execute the program and enter the number 72 to start the Automatic Toolkits Grover Search.
3. ! $\rightarrow$ 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 you can use to verify an algorithm. This functionality can be quite useful when you consider generating many oracles in a program.

Standard Grover search -"Option 73"

Input and output

1. Enter the number 73 to start a standard Grover search.
2. The Console will ask "How large…". Type 7 as an instance. We recommend not making your instance larger than 9 on PC as it consumes too much time and memory during the measurement stage.
3. The Console will ensure the volume of the database and display the number. Then, it asks "Where are the correct answers…". Type 56 and 57 for an instance. The oracle will be set up in this step.
4. The Algorithm will try to search for the target numbers.

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

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

It worth noting that due to multi-objects, the real state after using Grovers search algorithm becomes a $a(|2\rangle+|14\rangle)+b(|1\rangle+|3\rangle+|4\rangle+|5\rangle+....)$ 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. For example, in the last example, we would have gotten a wrong answer by a measurement of, say, $|5\rangle$. After another procedure, the state would become: $a(|2\rangle+|14\rangle+|5\rangle)+b(|1\rangle+|3\rangle+|4\rangle+|5\rangle+....)$. It becomes harder and harder to find the correct answer with this state.

In conclusion, one wrong answer in a round causes an avalanche of errors with Grovers search algorithm.

2.3.7 A comprehensive quantum teleportation-"Option 8"

Input and output Coming soon…

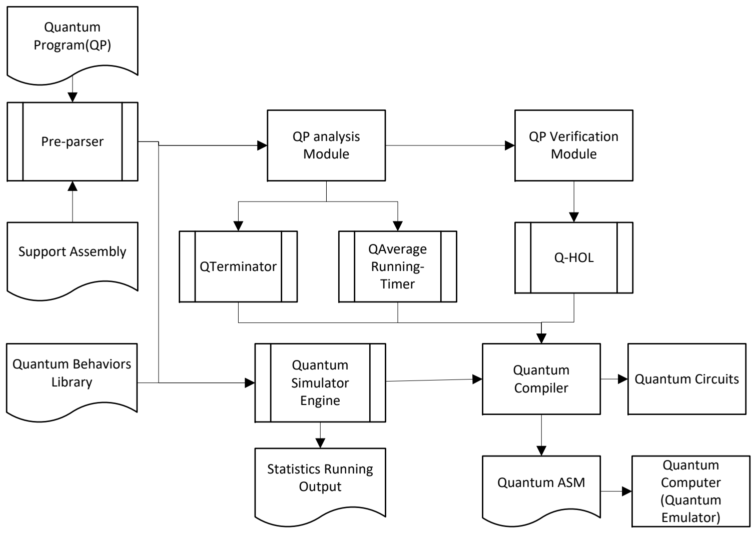


Figure 18: $Q|SI\rangle$ framework

3 Serious Coding

After a taste of some examples with $Q|SI\rangle$, you may want to try your own code. Please be patient. This section will help you get off to a good start.

Section 3.1 explains $Q|SI\rangle$ s structure. $Q|SI\rangle$ is different from other quantum platforms. You may need some background in quantum information, computer science, and with the platform before you can generate more advanced code. $Q|SI\rangle$ s structure can help you find the code workflow and adjust the code segments to respond to your requirements.

Section 3.2 demonstrates “Qloop, a very impressive “HelloWorld example of the quantum realm that illustrates the power of our platform. This example will familiarize you with some of the advanced components available in $Q|SI\rangle$.

Section 3.3 is designed to teach you the details of the quantum while-language. The quantum **while**-language is an elegant programming language that enhances loop power.

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

This section introduces $Q|SI\rangle$ s basic structure. $Q|SI\rangle$ 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 Figure 18.

The basic features of $Q|SI\rangle$

The main features of $Q|SI\rangle$ are

Language supporting : $Q|SI\rangle$ is the first platform to support the quantum **while**-language. In particular, it allows us to program with a measurement-based case statement and a while-loop. These two program structures provide more efficient and clearer descriptions of some quantum algorithms, such as quantum walks and Grovers search algorithm.

Enriched quantum types : Compared to other simulators and analysis tools, $Q|SI\rangle$ supports several kinds of quantum types, such as "Ket" and "Qbit", in a quantum computation engine. These types have unified operations and can be used in different scenarios. This feature provides a highly flexible programming environment to facilitate the coding 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 program compilation, analysis, and verification.program compilation, analysis and verification.

f-QASM instruction set : Defined as an extension of QASM (quantum assembly language), f-QASM is essentially a quantum circuit description language that can be adapted for a variety purposes. In this language, every line has only one command. f-QASM's 'goto' structure contains more information than the original QASM [Svore et al., 2006] or space-efficient QASM-HL [JavadiAbhari et al., 2014], f-QASM can also be used for further optimization and analysis.

Quantum circuit generation : 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 a program written in the high-level quantum **while**-language.

Arbitrary unitary operator : The $Q|SI\rangle$ platform also includes 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 could be transferred into a quantum circuit consisting of quantum gates from a small predefined set of basic gates once these are available from quantum chip manufacturers.

3.2 The 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 behaviors library and a calculation engine. The support assemblies provide supporting for the quantum types and quantum languages. More specifically, they provide a series of quantum objects and reentrant wrapped functions to play the role of the quantum program constructs **if** and **while**. The quantum behaviors library provides the behaviors for quantum objects, such as unitary transformation and measurement including the result and post-state. The calculation engine is designed as an execution engine. It accepts quantum objects and their rules from the quantum behaviors library and converts them into probability programming running steps.

Quantum Program (QP) Analysis Module : This module currently contributes two submodules to static analysis mode: 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 the quantum compiler at the next stage for further usage.

QP Verification Module : (Coming Soon. . .) 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 [Ying, 2011]and is

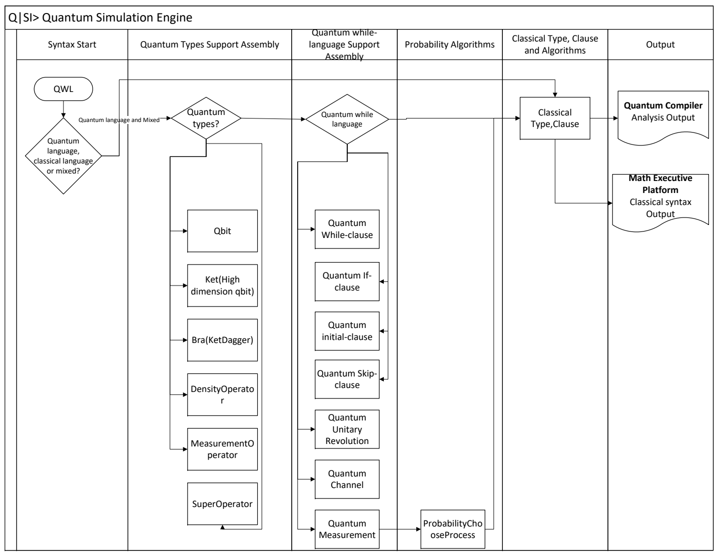


Figure 19: $Q|SI\rangle$ Pre-parser

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. [Liu et al., 2016].

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 [Svore et al., 2006], 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 device 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.

Implementation $Q|SI\rangle$ 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. Figure 19 shows the procedure for simulating a quantum engine. Two types of languages are supported: one is the pure quantum while-language, the other is a mixed quantum **while**-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 every variable and operator and executes the corresponding support assembly. As mentioned before, 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.

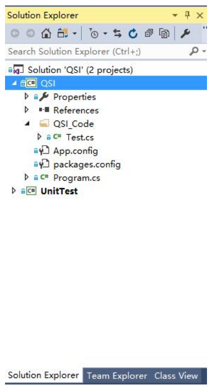


Figure 20: Entry to your code

3.3 From UnitTest to HelloWorld

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

Entry to your code

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

PATH: `Solution ‘Testudo’` → `QSI(Project)` 20

Writing Quantum Code

3.3.1 Code File

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

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

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

3.3.2 Component quantum libraries

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

Assemblies: `QuantumRuntime, QuantumToolkit` .

These two assemblies conclude analogous namespace:

Namespace: `QuantumRuntime, QuantumToolkit` .

Each has several different sub-namespaces:

QuantumRuntime Sub-namespace: `Operator`

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

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

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

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

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

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

Exploit Namespace (expert) You can exploit our provided assemblies using 'Reflection' given by Visual Studio.

PATH: `Menu` $\rightarrow$ `View` $\rightarrow$ `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 commonly used static functions like code segment shows.

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

3.3.4 Inheriting is essential for reducing code complexity

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

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

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

3.3.5 Declaring registers, quantum registers, quantum gates and quantum measurement

A variable is simply a name given to a storage area that our programs can manipulate. Each variable in C# has a specific type, which determines the size and layout of the variable's memory and, hence, 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 variable 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

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

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 a container for quantum circuits (workflow)

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

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

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

! → The gates should match the number of qubits. In most scenarios, a single qubit gate and a CNOT is enough for a quantum circuit. But, $Q|SI\rangle$ does support further dimensions of a unitary matrix. Any given multi-qubit gates (excluding a CNOT) would be translated to single unitary gates and a CNOT.

! → Dont forget to collect your results using a measurement. Without a measure, you have zero information about quantum states. Also, repetitive experiments usually describe general cases in quantum mechanics.

Writing Classical Code

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

! → At the very beginning, take note that $Q|SI\rangle$ has two modes: static and dynamic. Static mode is used for analyzing the code structure, termination, and compilation, whereas dynamic mode is used to execute the code. Remember these modes, as they can help you avoid unknown errors.

3.3.7 Create, initialize, run, and collect

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

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

The default configuration allows you to display 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. It also initializes the background variables and the input matrix.

Init: `test.Init();`

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

Run: `test.Run();`

! $\rightarrow$ Note that `test.Init();` only needs to be executed once, while `test.Run();` can be executed for many times. can be executed many times. In fact, most quantum experiments rely on statistical results, which means that you will probably need to execute `test.Run();` many times depending on your requirements. When executing the function `test.Run();` more than once, DO NOT FORGET TO USE `test.InitSuperRegister();`TO RESET ALL REGISTERS, otherwise the last execution may pollute or disturb the next experiment. A general case is a loop in classical coding.

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

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

3.4 Using the quantum while-language

3.4.1 A brief introduction to the quantum **while**-language

For reader's convenience, this section contains a brief review of the quantum **while**-language. Quantum **while**-programs are generated using the following simple syntax:

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

The quantum **while**-language is a pure quantum language without classical variables. Its only assumptions are a set of quantum variables denoted by symbols $q, q_1, q_2, ...$ 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.*

Some brief explanations of the quantum program constructs defined above follow; For more detailed descriptions and examples, see [Ying, 2011] and Chapter 3 of [Ying, 2016].

Skip: :

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

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

Initialisation :

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

where

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

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

Unitary transformation :

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

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

Sequential Composition: :

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

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 :

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

for each possible outcome m of measurement $M = \{M_m\}$. In the case statement **if** $(\Box m \cdot M[\bar{q}] = m \to S_m)$ **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 :

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

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

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

QIf **Prototype**

$\boxed{QIf(\texttt{Measure(Qubits)}, () => \{\texttt{Sub\_program\_1}\}, () => \{\texttt{Sub\_program\_2}\}), \ldots;}$

Measure is a measurement, which could be a binary output measurement or a multi-bit output measurement;
Qubits is a set of qubits indexed by a variable; and
Sub_program is a subprogram written in the quantum **while**-language.
The behavior of this structure is actually a case statement. First, a measurement is performed on the variable Qubits. Then, a sub program is executed according to the output index, i.e., the output according to the size of the number. For example, Subprogram 1 is executed when the measurement gives the result 0, while Subprogram 2 is executed when the measurement gives the result 1.

Qwhile **Prototype**

$\boxed{QWhile(\texttt{Measure(Qubits)}, () => \{\texttt{Sub\_program}\});}$

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

3.5 A simple Example

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

The code is shown in Figure 21 shows.

The code segment is similar to a standard quantum teleportation. Note that a `QWhile` has been added in Line 23. That means the body of the quantum teleportation will only be executed when the measurement

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

Figure 21: Teleportation with Loops

result is 1. Another modification has been added at Lines 30 and 34 a `QIf` that shows an Empty or a unitary matrix `xGate` (or `zGate`) is performed on the qubits according to the result of the measurement.

3.6 Check for termination

Termination abstract

Here, you may notice that the loop structures give true power to quantum programming coder, while another serious issue is raised: termination. When a loop is added, a program may become non-terminating. That means the program may never halt. You may also know that a non-halting problem is a hot topic in classical computing. Moreover, quantum scenarios are more complex than classical ones. A quantum state could be in a superposition that, in simple terms, means it could non-halting execute across several workflows. Considering the loop structure is also an influencing factor, this type of termination problem is non-trivial in most cases.

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

Checking for termination using the modules provided

To check the whether a given program is properly terminated, you need to denote the path of the input file and the class name of the target program. An example follows.

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

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

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

3.7 Compilation

QASM (Quantum Assembly Language) is widely used in modern quantum simulators. It was first introduced in [Svore et al., 2006], who defined it as a technology-independent reduced instruction set computing assembly language extended by a set of quantum instructions based on the quantum circuit model. Subsequently, the article [Ying and Feng, 2011] carefully characterized its theoretical properties. In 2014, A.JavadiAbhari et al. [JavadiAbhari et al., 2014] defined a space-consuming flat description and a 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 format of QASM called f-QASM (Quantum Assembly Language with feedback). The most significant motivation behind this variation is to translate the inherent logic in high-level programming languages into a simple command set, i.e., so there is only one command in every line or period. However, a further motivation is to solve an issue raised by the IBM QASM 2.0 list and provide the ability to have conditional operations to implement feedback based on measurement outcomes.

3.7.1 f-QASM (Quantum Assembly Language with feedback)

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

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

3.7.2 Compile Quantum Program using Compile Module

Compile Module **Prototype**

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

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

The first command is the quantum compilation command.
Parameter_1 is the class name to be compiled.
Parameter_2 is a number from 0-3 indicating how "deep" you want the quantum circuits to be generated. The default number is 0. We will explain this in next section.
Parameter_3 is an integer from 1-100 indexed to the accuracy of decomposition for the arbitrary gate U which is performed on a single qubit. In fact, given any $\epsilon > 0$, the Solovay-Kitaev theorem can generate a series of gates to approximate U on a precision ϵ using $\Theta(\log^c(1/\epsilon))$ from a fixed finite set, where c is a small constant approximately equal to 2 [Nielsen and Chuang, 2010]. From the article [Dawson and Nielsen, 2005], the accuracy is decided by a recursion number, and that number can be set here.
Parameter_4 is an enumeration type consisting of two options:

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

The first option, `Matlab.PreSKMethod.OrginalQSD`, uses the algorithm from the article [Shende et al., 2006]. The second option uses the algorithm from Section 4.5 of the book [Nielsen and Chuang, 2010]

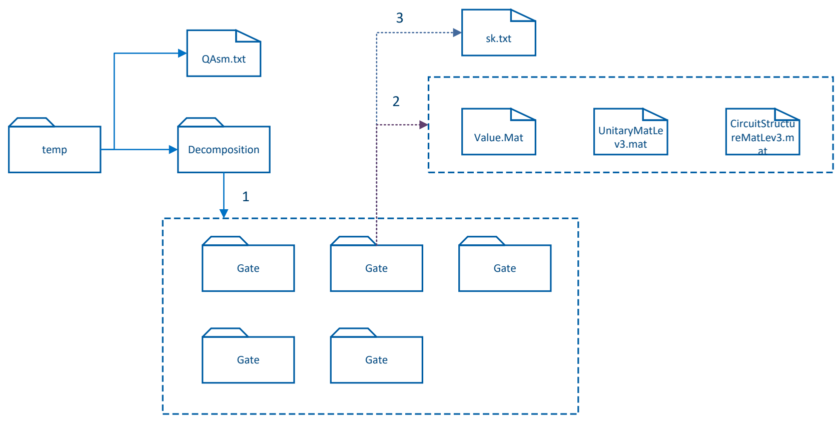


Figure 22: Compiled File Structure

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

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

3.7.3 Compiled Results

The compiled results can be found under the folder, $\boxed{bin\backslash Debug\backslash temp}$ or $\boxed{bin\backslash Release\backslash temp}$.

The command

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

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

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

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

4 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 Members:	Ji Guan	guanji1992@gmail.com
	Li Zhou	zhou31416@gmail.com
	Xin Wang	wangxinfelix@gmail.com

B Contact Information

Any issues and comments are welcome. Feel free to make your voice heard; we are listening! You can send your feedback in two ways.

- Submit your feedback via the project repository. The link is `https://github.com/klinus9542/QSI` .
- Submit your feedback via Email to `mailto:cqsi.service@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). Scaffcc: a framework for compilation and analysis of quantum computing programs. *Proceedings of the 11th ACM Conference on Computing Frontiers*, 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.