## Contents

# Quantum Computing Library Manual (version 0.1)

***Wajahat Mahmood Qazi***

## 1. Installation and System Requirement

QCL is shipped using standard windows installation program, which can be downloaded from github.com/wmqazi. The runtime requirement for the framework is MS Dot NET 4.0, whereas the hardware and operating system requirements are given below.

Due to computational complexity of quantum algorithms and circuits, it is recommended to run simulations on at least 1.8 GHz processors with minimum 2 GB RAM (4 GB RAM is recommended). It is ideally preferred for normal performance to use Xeon class processors, Quad Core, Core i3, Core i5 or Core i7 processors with 6-8 GB RAM. It is preferred to use MS Windows 7 or Vista, Window XP professional edition can also be used but it is not recommended.

## 2 QCL Design

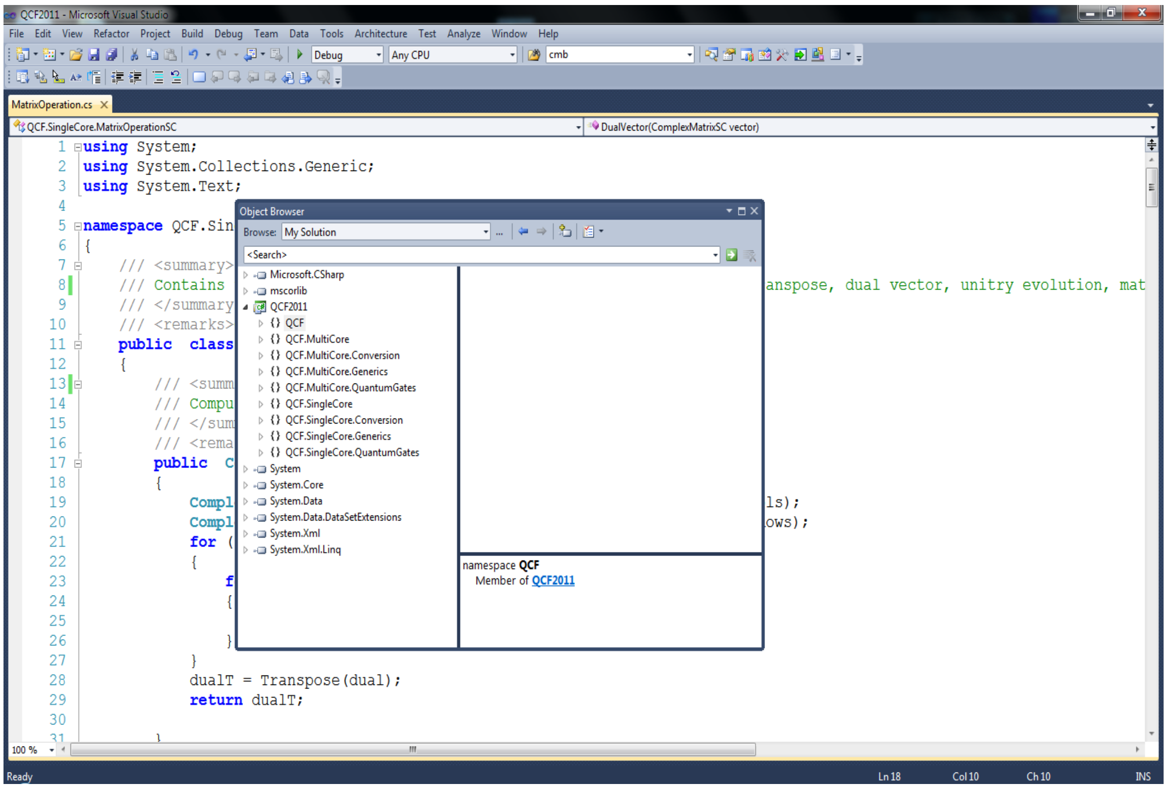
The design of QCL, shown in figure 1 takes advantage of object orientation, delegates and event handling provided by DOT NET. It provides support for executing parallel operation on multicore processor. Its parallel execution is on task parallelism method of programming. Mathematically every quantum artifact can be decomposed to matrices of complex numbers. Therefore, the central core components of QCL consist of classes to represent complex numbers and matrices. These include *ComplexNumber*, *ComplexMatrix*, *MatrixBase*, *MatrixOperation*, *QRegister*, *RandomComplexNumber*, *SpecialOperator* classes along with other classes to represent quantum gates and a couple of delegates and enums. *MatrixBase* in QCL is an abstract class. It is extended by *ComplexMatrix* and *Matrix<T>* class. *ComplexMatrix* class provides a data structure to represent matrix of a complex number. Whereas *Matrix<T>* is generic based matrix representation for any data type as per requirement. *ComplexMatrix* is further extended by some core quantum computing classes which includes *QRegister*, *HardmardGate*, *IdentityGate*, *ToffoliGate*, *PauliXGate*, *PauliY*, *PauliZ*, *ControlledNot* (for more classes check QCL documentation). Indeed, the latest version of MS Visual Studio 2010 provides a separate class for complex numbers, which is of value type object. Value type objects have to undergo boxing/ unboxing when transferring contents between the stack and heap memories, which lead to the poor performance and it, can be extended to create variant versions.

The design of QCL is based on certain assumptions. These assumptions are similar to COVE (Purkeypile, 2009). This similarity is due to the inherent characteristics of classical computers, when performing simulation of quantum and underlying programming paradigm used for simulation. In physical quantum computers the information retained in a qubit collapse due to decoherence. In proposed framework, by default there no time limit for the completion of quantum computation. QCL provide a framework to simulate this very important aspect of quantum mechanics. In QRegister class we have implemented Decoher() method. This method internally calls the DecoherDelegate. Any user defined implementation of decoherence method that is bind with delegate can be used to simulate this concept.

Figure 1: QCL class diagram

## 3 QCL Application Program Interface

QCL consist of two packages,” QCF.SingleCore” and “QCF.MultiCore”, respectively each tuned for single-core and multi-core computer systems. The namespace hierarchy presenting various sub-namespaces is shown in figure 2, whereas member classes and their distribution across various namespaces are shown in figure 3 and 4 respectively for single- and multi-core systems and in figure 5, presenting classes that are common for single- and multi- core systems.



```
using System;
using System.Collections.Generic;
using System.Text;

namespace QCF.Sin
{
    /// <summary>
    /// Contains                                          anspose, dual vector, unitry evolution, mat
    /// </summary
    /// <remarks>
    public  class
    {
        /// <summ
        /// Compu
        /// </sum
        /// <rema
        public  C
        {
            Compl                                         ls);
            Compl                                         ows);
            for (
            {
                f
                {

                }
            }
            dualT = Transpose(dual);
            return dualT;

        }
```

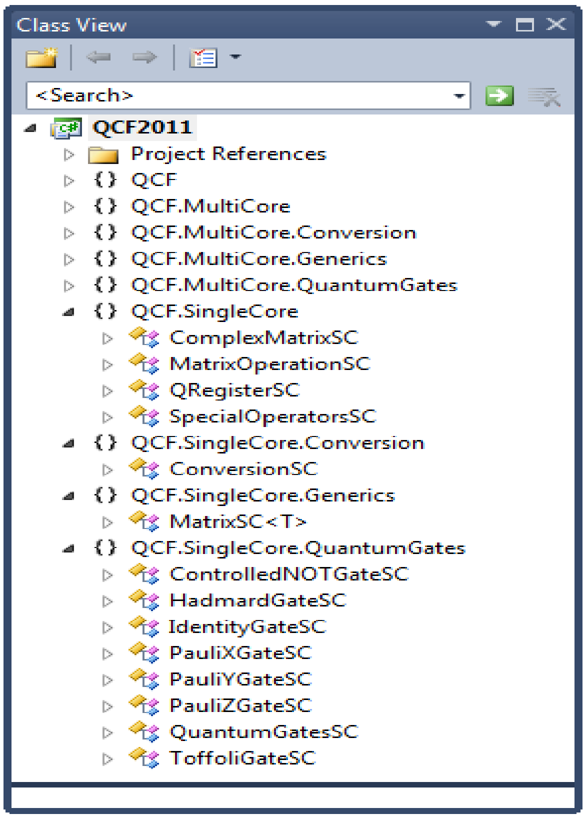
Object Browser

Browse: My Solution

- Microsoft.CSharp
- mscorlib
- QCF2011
  - {} QCF
  - {} QCF.MultiCore
  - {} QCF.MultiCore.Conversion
  - {} QCF.MultiCore.Generics
  - {} QCF.MultiCore.QuantumGates
  - {} QCF.SingleCore
  - {} QCF.SingleCore.Conversion
  - {} QCF.SingleCore.Generics
  - {} QCF.SingleCore.QuantumGates
- System
- System.Core
- System.Data
- System.Data.DataSetExtensions
- System.Xml
- System.Xml.Linq

namespace **QCF**
Member of QCF2011

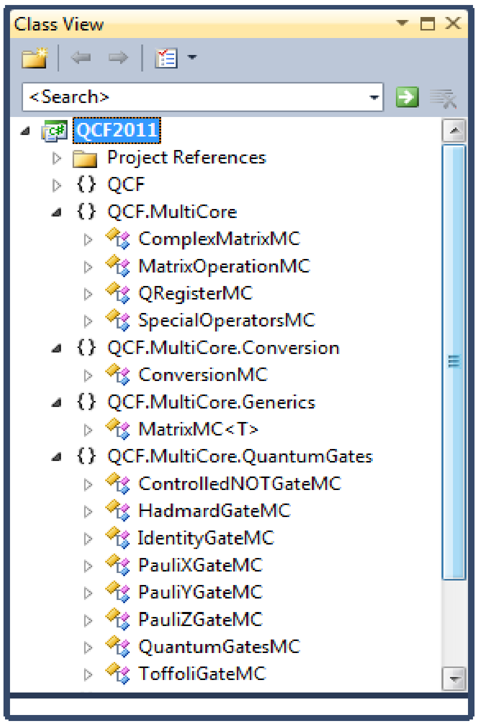
Figure 2: Namespace Hierarchy



Class View

- QCF2011
  - Project References
  - {} QCF
  - {} QCF.MultiCore
  - {} QCF.MultiCore.Conversion
  - {} QCF.MultiCore.Generics
  - {} QCF.MultiCore.QuantumGates
  - {} QCF.SingleCore
    - ComplexMatrixSC
    - MatrixOperationSC
    - QRegisterSC
    - SpecialOperatorsSC
  - {} QCF.SingleCore.Conversion
    - ConversionSC
  - {} QCF.SingleCore.Generics
    - MatrixSC<T>
  - {} QCF.SingleCore.QuantumGates
    - ControlledNOTGateSC
    - HadmardGateSC
    - IdentityGateSC
    - PauliXGateSC
    - PauliYGateSC
    - PauliZGateSC
    - QuantumGatesSC
    - ToffoliGateSC

Figure 3: Class Distribution across Single-core Namespace



```
QCF2011
  Project References
  {} QCF
  {} QCF.MultiCore
    ComplexMatrixMC
    MatrixOperationMC
    QRegisterMC
    SpecialOperatorsMC
  {} QCF.MultiCore.Conversion
    ConversionMC
  {} QCF.MultiCore.Generics
    MatrixMC<T>
  {} QCF.MultiCore.QuantumGates
    ControlledNOTGateMC
    HadmardGateMC
    IdentityGateMC
    PauliXGateMC
    PauliYGateMC
    PauliZGateMC
    QuantumGatesMC
    ToffoliGateMC
```

Figure 4: Class Distribution across Multi-core Namespace

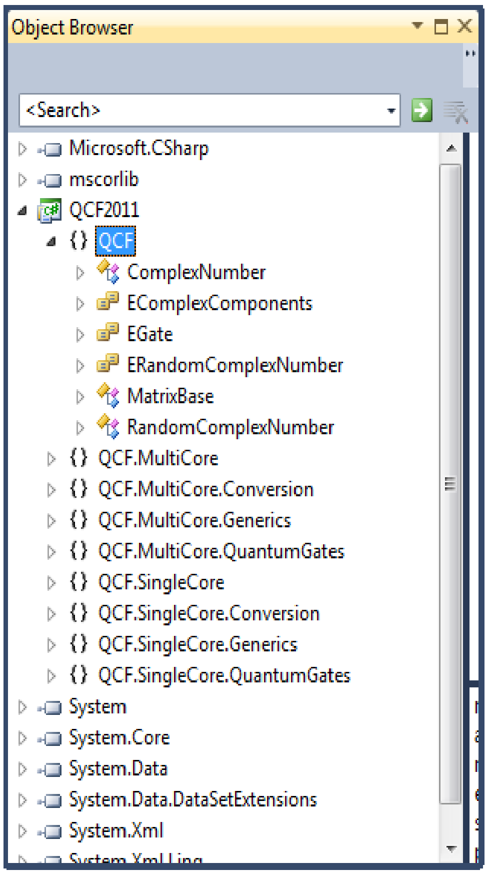


Figure 5: Common classes for Single and Multi-core Systems

## 4 QCF Namespace

This namespace consists of three classes (see figure 6) that are complex number, random complex number and abstract class that act as parent of matrix classes and three enumerated types for representing the components of complex number, standard quantum gates and the selection method for generating complex random number.

## 5 ComplexNumber Class

The ComplexNumber class as shown in figure 6 encapsulates various properties, indexers and methods to program complex numbers. The constructor of this class provides four overload constructors to create a complex number object. Class also provides overloaded version of operators that could be used for various type of arithmetic operations on complex numbers.

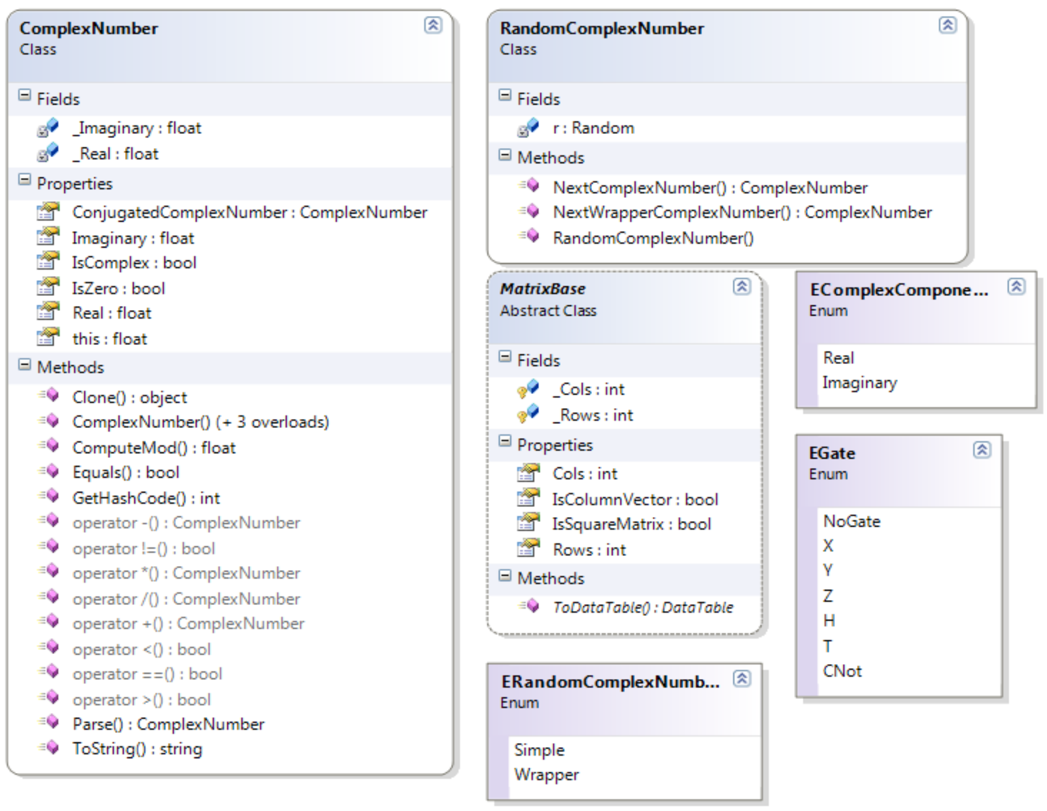


Figure 6: Classes and Enumerated Data types in QCF Namespace

As mentioned above that ComplexNumber class provides four construction methods to create a complex number. This gives a flexibility to developers to construct objects as per to their requirements. Following C# instructions illustrates the object construction.

1. Creating complex number with a specified value for real component of complex number. In this case, imaginary part is by default treated as zero. This approach further provides two overloads, one for float type value for real part and second is for specifying integer type value for real part of the complex number.
   ```
   ComplexNumber complexNumber1 = new
   ComplexNumber(2.2f);
   ```

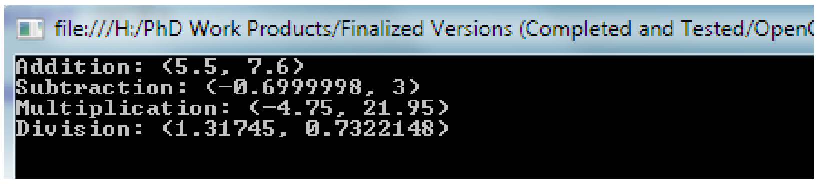
2. Creating complex number with string type argument
   ```
   ComplexNumber complexNumber1 = new
   ComplexNumber("2,3");
   ```

3. Creating complex number with specified real and imaginary values using float data type.

```
ComplexNumber complexNumber1 = new
ComplexNumber(2.4f, 5.3f);
```

ComplexNumber class expose overloaded operators that can be used to perform arithmetic operation as shown below and the output of the sample is shown in figure 7

```
ComplexNumber complexNumber1 = new
ComplexNumber(2.4f, 5.3f);
ComplexNumber complexNumber2 = new
ComplexNumber(3.1f, 2.3f);
ComplexNumber complexNumber3;

complexNumber3 = complexNumber1 +
complexNumber2;
Console.WriteLine("Addition: " +
complexNumber3);
complexNumber3 = complexNumber1 -
complexNumber2;
Console.WriteLine("Subtraction: " +
complexNumber3);
complexNumber3 = complexNumber1 *
complexNumber2;
Console.WriteLine("Multiplication: " +
complexNumber3);
complexNumber3 = complexNumber1 /
complexNumber2;
Console.WriteLine("Division: " +
complexNumber3);
```

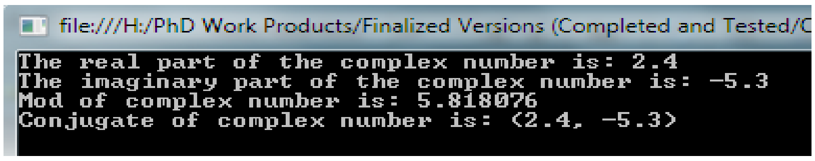

file:///H:/PhD Work Products/Finalized Versions (Completed and Tested/Open

```
Addition: (5.5, 7.6)
Subtraction: (-0.6999998, 3)
Multiplication: (-4.75, 21.95)
Division: (1.31745, 0.7322148)
```

Figure 7: Arithmetic Operations on Complex Numbers

ComplexNumber class also provide method for computing mod, a property that represents the conjugate of the complex number and an indexer to access the real and imaginary components as per requirement; following illustration presents the programming of these features and figure 8 show the output of arrangement.

```
ComplexNumber complexNumber1 = new
ComplexNumber(2.4f, 5.3f);

float realPart =
complexNumber1[EComplexComponents.Real];
float imaginaryPart =
complexNumber1.ConjugatedComplexNumber[EComplexComp
onents.Imaginary];

Console.WriteLine("The real part of the complex
number is: " + realPart.ToString());

Console.WriteLine("The imaginary part of the
complex number is: " + imaginaryPart.ToString());

float mod = complexNumber1.ComputeMod();
Console.WriteLine("Mod of complex number is: " +
mod.ToString());

Console.WriteLine("Conjugate of complex number is:
" +
complexNumber1.ConjugatedComplexNumber.ToString());
```



file:///H:/PhD Work Products/Finalized Versions (Completed and Tested/C

```
The real part of the complex number is: 2.4
The imaginary part of the complex number is: -5.3
Mod of complex number is: 5.818076
Conjugate of complex number is: (2.4, -5.3)
```

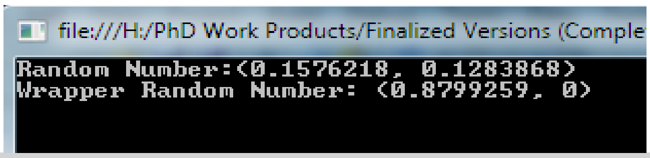
Figure 8: Output of Mod, Conjugate and Access to Components

## 6 RandomComplexNumber Class

The class provides two methods, NextComplexNumber() and NextWrapperComplexNumber() to generate complex numbers at random. NextComplexNumber() method generates a complex number with real and imaginary components having random values; whereas the NextWrapperComplexNumber() method generates a random complex number having imaginary component set to 0. Following code exemplify the class and its methods, whereas, figure 9 shows the output.

```
RandomComplexNumber rndcn = new
RandomComplexNumber();
```

```
ComplexNumber cn = rndcn.NextComplexNumber();

Console.WriteLine("Random Number:" +
cn.ToString());
cn = rndcn.NextWrapperComplexNumber();
Console.WriteLine("Wrapper Random Number: "+
cn.ToString());
```



file:///H:/PhD Work Products/Finalized Versions (Comple

```
Random Number:(0.1576218, 0.1283868)
Wrapper Random Number: (0.8799259, 0)
```

Figure 9: Output of RandomComplexNumber Class Program

## 7 MatrixBase Class

This abstract class is used in QCL to implement extended versions of matrixes, which represent complex number matrix, and a generic implementation of matrix both for multi-core and single-core platforms as shown in figure 10. This abstraction exposes (see figure A.5) properties to access the number of rows, columns along with the type of matrix, which could be square matrix or column vector type along with a ToDataTable() method that prepares the tabular representation of matrix.

Figure 10: Classes derived from MatrixBase Class

## 8 QCF.SingleCore and QCF.MultiCore Namespace

These namespaces and their sub-units provide constructs for simulating the components of any quantum-computing algorithm and circuits. Figure 11 shows the class hierarchy for single-core namespace, whereas, multi-core namespace also have same identical hierarchy such that each class is tagged with a postfix “MC” in their names instead of “SC”, where MC metaphor multi-core and SC symbolize single-core. These classes represent abstract types for a matrix of complex number, matrix operator, quantum register and a special matrix operator and these are discussed in detail below.

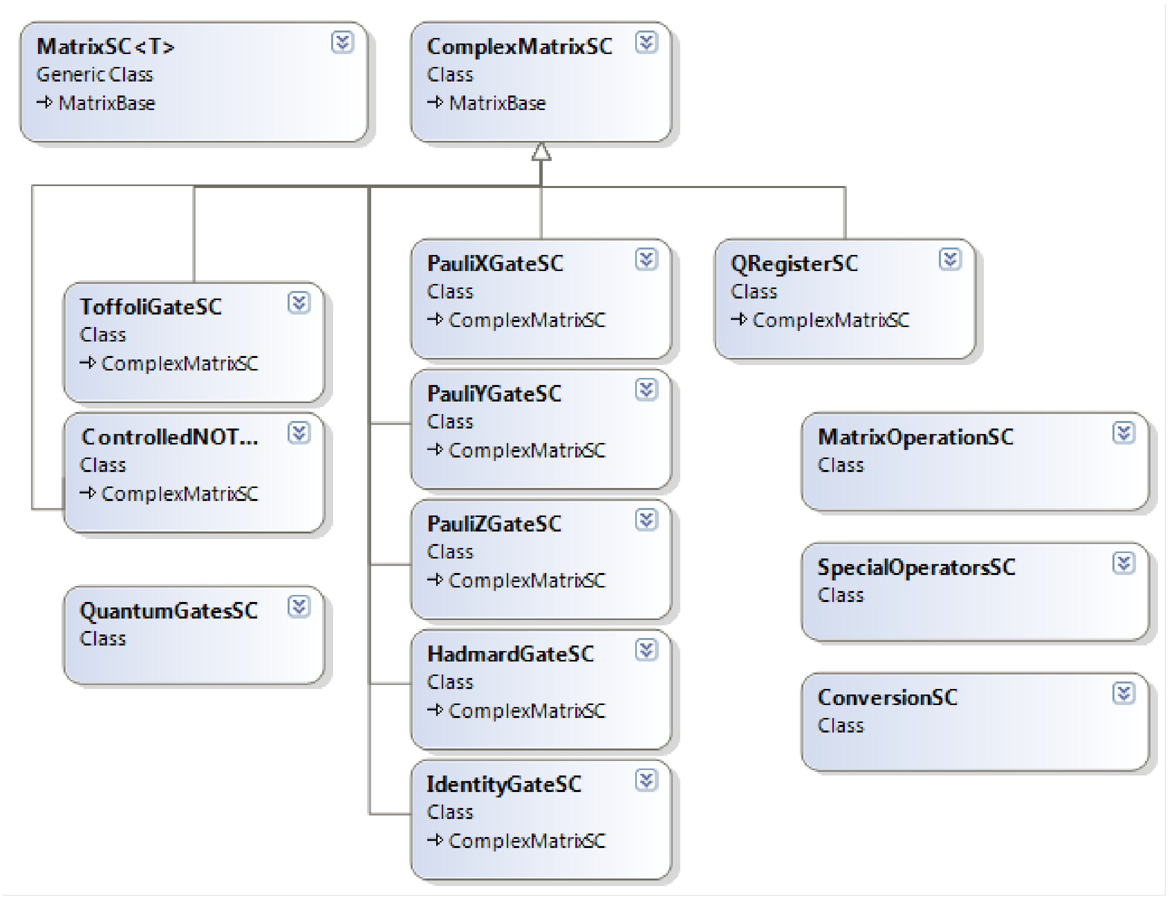


Figure 11: Classes Diagram of QCF.SingleCore Namespace including its Sub-namespaces

## 9 Matrix of a Complex Number

The afore-mentioned namespaces provides classes to implement matrix object with complex number elements entitled as ComplexMatrixSC for single core and ComplexNumberMC for multi-core as shown in figure 12

Figure 12: Member Classes in Single and Multi-core at Base level of Namespace

The complex number matrix class exposes three constructors, arithmetic operators and an index to fetch an element from a give row and column as shown in figure 13 where ComplexMatrixSC and ComplexMatrixMC are respective members of single and multi-core.

Figure 13: Members of Single and Multi-core Complex Matrix Classes

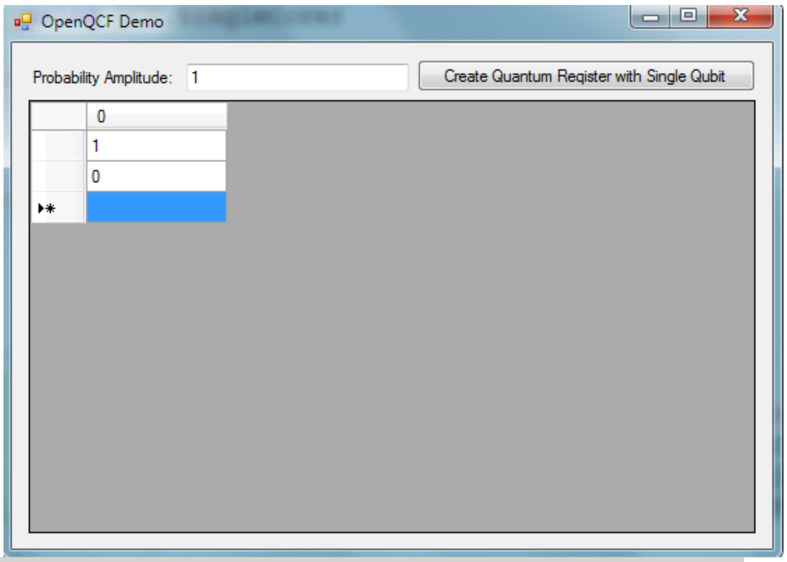
## 10 Quantum Register Class in Single and Multi-core Namespace

In quantum-computing smallest piece of information is stored in quantum bit (qubit), whereas the collection of these quantum qubit forms a quantum register (q-register). QCL provide class to represent information stored in q-register and its member methods are shown in figure 14 that shows QRegisterSC and QResgisterMC both derived from respective complex matrix class.

Figure 14: Class Diagram for Quantum Register

Quantum register class exposes two constructors two create an object. One method is to pass a column vector form of quantum information and second to pass Dirac form of quantum information. Moreover, it also provides a method to compute the probability amplitude in order to access the validity of quantum information. Following C# illustrations provides a programmatic view of the construction of single-qubit register and 4-qubit registers and their respective outputs are shown in figure 15 and 16

```
QRegisterSC qr1 = new QRegisterSC("|0>");
dataGridView1.DataSource = qr1.ToDataTable();
float pb = qr1.ComputeProbabilityAmplitude();
txtProbabilityAmplitude.Text = pb.ToString();
```



Probability Amplitude: 1

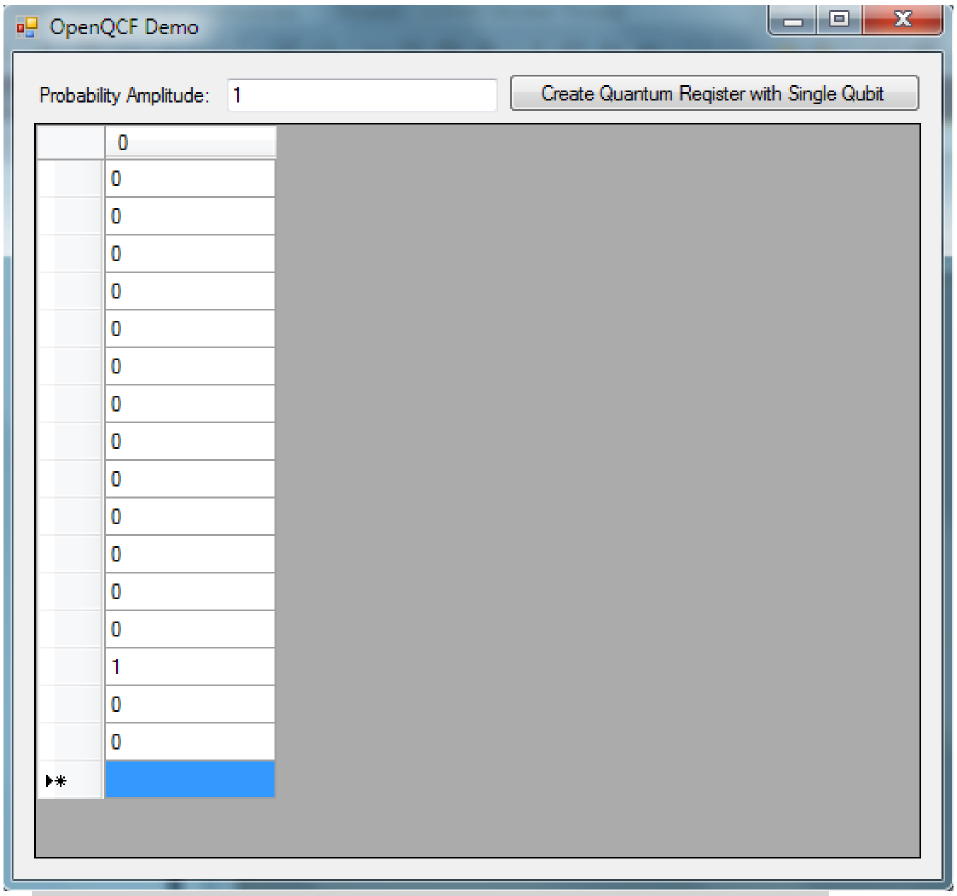
|   |   |
|---|---|
|   | 0 |
|   | 1 |
|   | 0 |
| ▸* |   |

Figure 15: Construction of 1-qubit Quantum Register

```
QRegisterSC qr1 = new QRegisterSC("|1101>");
dataGridView1.DataSource = qr1.ToDataTable();
float pb = qr1.ComputeProbabilityAmplitude();
txtProbabilityAmplitude.Text = pb.ToString();
```

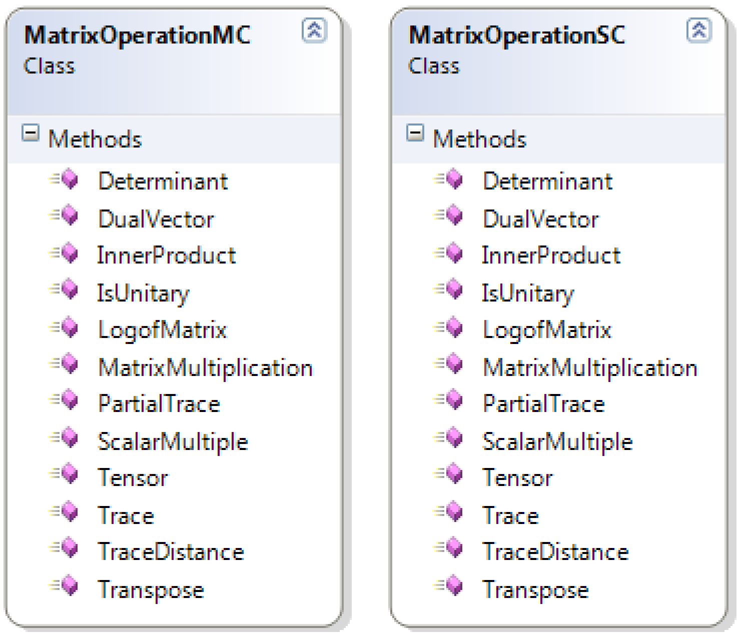
## 11 Matrix Operator Class

The matrix operator class for single and multi-core programming exposes methods for performing and computing scalar multiplication, inner product, dual vector, tensor, trace, trace distance, partial trace, matrix multiplication, determinant and a method to check the unitary nature of matrix (see figure 17).



```
0
0
0
0
0
0
0
0
0
0
0
0
0
0
1
0
0
```

Figure 16: Construction of 1-qubit Quantum Register



MatrixOperationMC
Class

Methods
- Determinant
- DualVector
- InnerProduct
- IsUnitary
- LogofMatrix
- MatrixMultiplication
- PartialTrace
- ScalarMultiple
- Tensor
- Trace
- TraceDistance
- Transpose

MatrixOperationSC
Class

Methods
- Determinant
- DualVector
- InnerProduct
- IsUnitary
- LogofMatrix
- MatrixMultiplication
- PartialTrace
- ScalarMultiple
- Tensor
- Trace
- TraceDistance
- Transpose

Figure 17: Members of Matrix Operation Class

## 12 Special Operator Class

This class provides three utility methods (see figure 18), which are not common in matrix operation but have significant role in simulating quantum operations. These functions are used to convert any binary sequence or real no to corresponding quantum state.

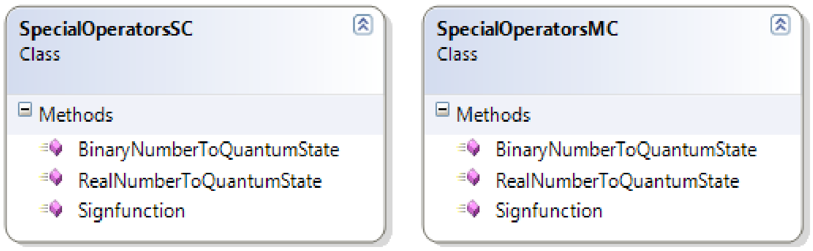


Figure 18: Members of Special Operator Class

## 13 Quantum Gates

QCL API provides single and multi-core implementation of standard single and multi-quantum bit gates like PauliX, PauliY, Hadmard, ControlledNot, ToffoliGate as shown in figure 19. It also provides QuantumGateMC and QuantumGateSC classes, which encapsulate objects of the aforementioned quantum gates accordingly.

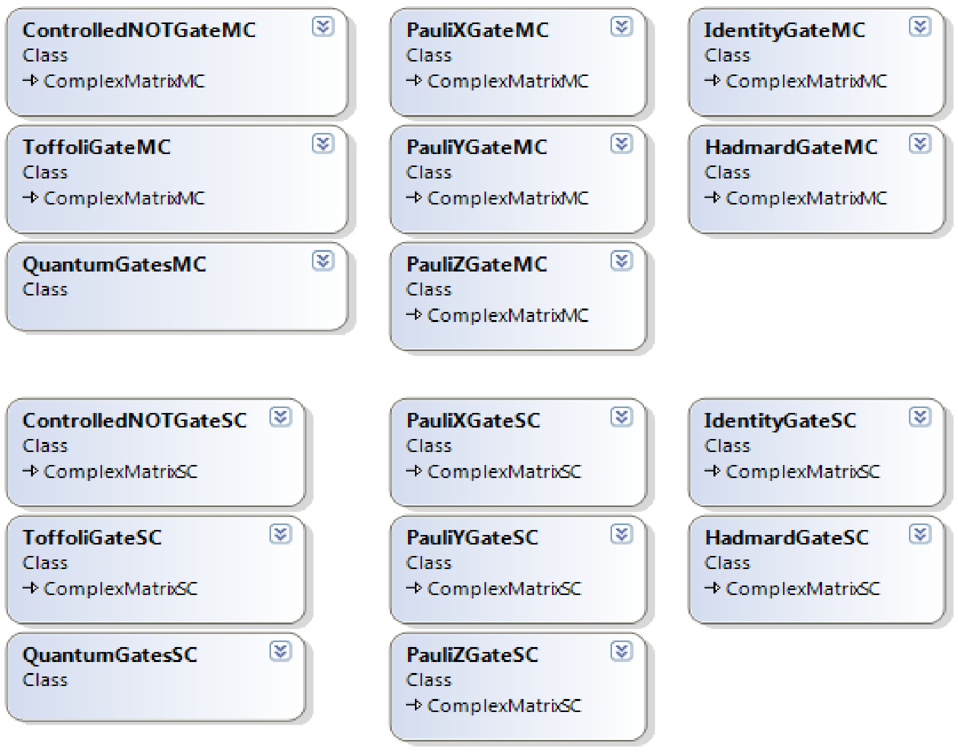


Figure 19: Quantum Gates Implemented in QCL

The functioning of APIs are not limited to afore-mentioned gates, new gates can be created according to requirement. The programing of new gates or quantum circuit is illustrated in example below.

Example 1:

Consider a quantum circuit shown in figure 20, which takes two quantum bits and circuit is composed of a Hadmard and identity matrix. In this example, a representing quantum operator/gate will be constructed for the given circuit by apply tensor product between the given gates and then the resultant gate will be applied on quantum register with state given as $|11\rangle$ as shown below using C# code and output of the program is shown in figure 21

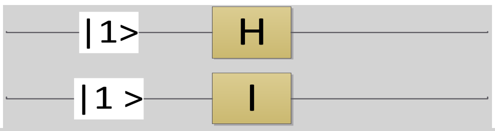
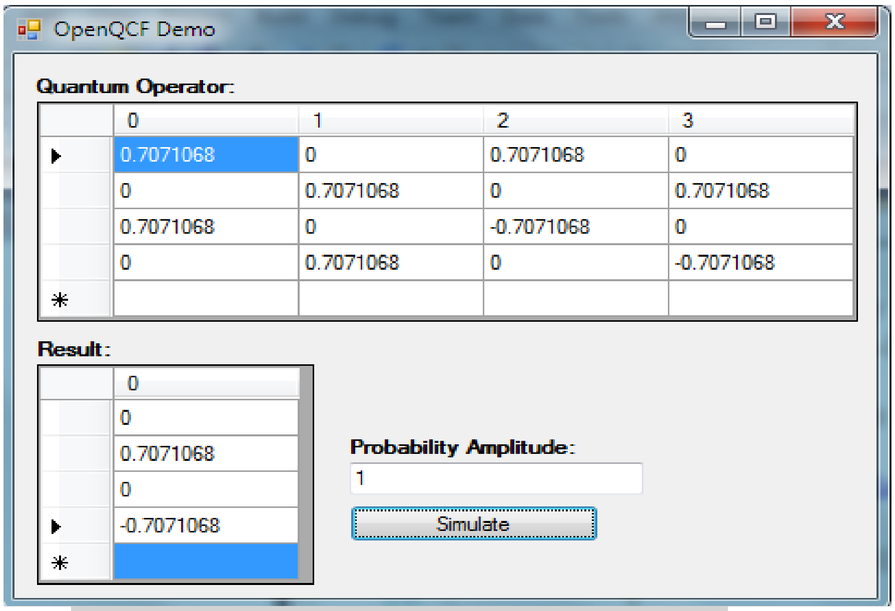


Figure 20: Two-Quantum Bit Circuit with Hadmard and Identity Gate

```
HadmardGateSC hadmardGate = new
HadmardGateSC();

IdentityGateSC identityGate = new
IdentityGateSC();

MatrixOperationSC matrixOp = new
MatrixOperationSC();

ComplexMatrixSC circuit =
matrixOp.Tensor(hadmardGate, identityGate);

dataGridView1.DataSource =
circuit.ToDataTable();

QRegisterSC qr = new QRegisterSC("|11>");
QRegisterSC resultantRegister = new
QRegisterSC(matrixOp.MatrixMultiplication(circu
it, qr));

txtProbabilityAmplitude.Text =
resultantRegister.ComputeProbabilityAmplitude()
.ToString();

dataGridView2.DataSource =
resultantRegister.ToDataTable();
```



| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| ▶ | 0.7071068 | 0 | 0.7071068 | 0 |
| | 0 | 0.7071068 | 0 | 0.7071068 |
| | 0.7071068 | 0 | -0.7071068 | 0 |
| | 0 | 0.7071068 | 0 | -0.7071068 |
| * | | | | |

| | 0 |
|---|---|
| | 0 |
| | 0.7071068 |
| | 0 |
| ▶ | -0.7071068 |
| * | |

Figure 21: Simulation Result for Example 1

Furthermore, if the input and output of a quantum system is known then using following equations quantum gates can also constructed as shown in example 2 and its corresponding code, whereas, figure 22 show the output.

Example 2:

In this example, the matrix operator will be formulated for PauliX gate using the input and output information given below:

$$|0\rangle \rightarrow 1\rangle$$
$$|1\rangle \rightarrow 0\rangle$$

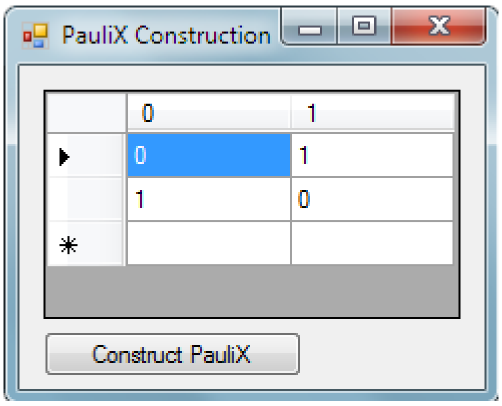
The standard method for formulating the gate from given input and output information is as following Gate = $\sum_i |input_i\rangle\langle output_i|$, therefore, the PauliX gate $U_{PauliX}$ will be formulated as

$$U_{PauliX} = |0\rangle\langle 1| + |1\rangle\langle 0|$$

Hence, the programming of this formulation using QCL is as following

```
QRegisterSC input1, output1;
QRegisterSC input2, output2;

MatrixOperationSC matrixOperator = new
MatrixOperationSC();

input1 = new QRegisterSC("|0>");
output1 = new QRegisterSC("|1>");
```

```
ComplexMatrixSC dualOfOutput1;
dualOfOutput1 = matrixOperator.DualVector(output1);

input2 = new QRegisterSC("|1>");
output2 = new QRegisterSC("|0>");
ComplexMatrixSC dualOfOutput2;
dualOfOutput2 = matrixOperator.DualVector(output2);

ComplexMatrixSC PauliXGate;
ComplexMatrixSC m1, m2;

m1 = matrixOperator.MatrixMultiplication(input1,
dualOfOutput1);
m2 = matrixOperator.MatrixMultiplication(input2,
dualOfOutput2);

PauliXGate = m1 + m2;

dataGridView1.DataSource =
PauliXGate.ToDataTable();
```



| | 0 | 1 |
|---|---|---|
| ▶ | 0 | 1 |
| | 1 | 0 |
| * | | |

Figure 22: Simulation Result for Example 2

Furthermore, if the input and output of a quantum system is known then using following equations quantum gates can also constructed as shown in example 2 and its corresponding code, whereas, figure 22 show the output.

## 14 Simulating Quantum Artifacts using QCL

As mentioned above that, QCL was evaluated by simulating Grover search algorithm and quantum mutual information algorithm for full-scale evaluation of the framework for its application in real world scenarios that required quantum computations.

## 15 Simulating Grover Search Algorithm

Grover algorithm is a quantum version of searching process through unsorted listed of items. The circuit shown in figure 23 represents a Grover search algorithm that searches values from 0-3. The circuit comprise of an oracle and along with 10 hadmard gates and 4 PauliX gates organized as shown in figure 23

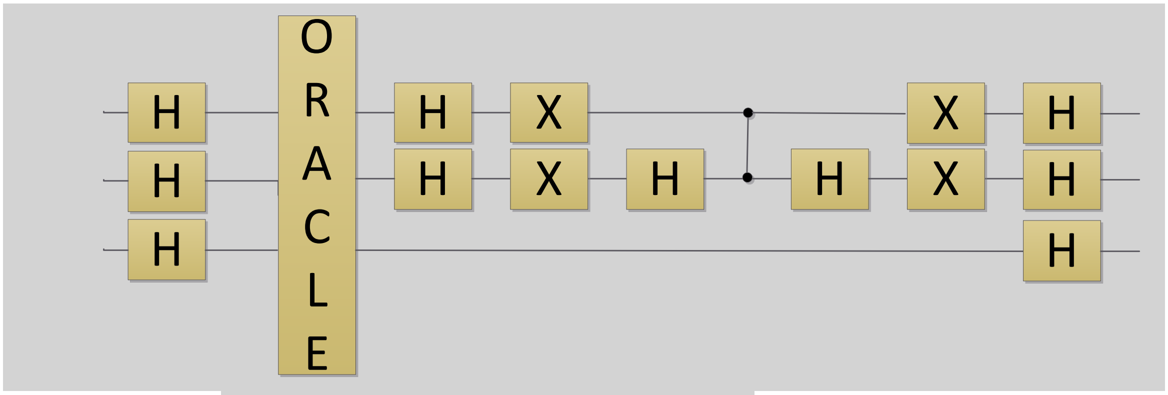


Figure 23: Gover Search Algorithm

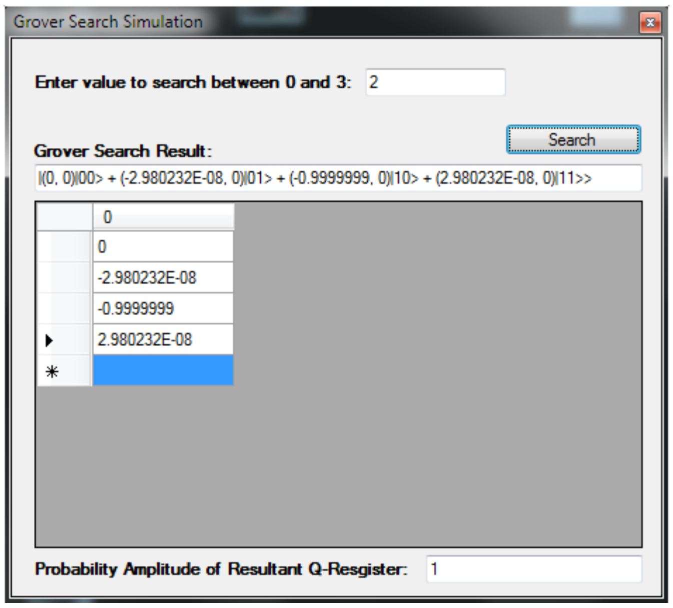
The Grover simulation was designed to take an integer number from classical world and search its presence in quantum world, as the result the quantum representation of the search is generated. The oracle of the Grover algorithm was designed accordingly to cater the numeric value from classical world. C# code to implement to simulate the Grover algorithm is given below that consists of two methods which are invoked according to the sequence diagram given in figure 24, whereas; figure 25 shows the simulation output for search numeric value 2.

Figure 24: Grover Search Simulation 1-level Sequence Diagram

```
private void btnGroverSearch_Click(object sender,
EventArgs e)
{
    int valueToSearchInClassical =
    int.Parse(txtValueToSearch.Text);

    QRegisterSC a = new QRegisterSC("|00>");
    ComplexMatrixSC H2 = new ComplexMatrixSC(4, 4);
    ComplexMatrixSC HI = new ComplexMatrixSC(4, 4);
    ComplexMatrixSC X2 = new ComplexMatrixSC(4, 4);
    MatrixOperationSC operation = new
    MatrixOperationSC();

    H2 = operation.Tensor(QuantumGatesSC.H,
    QuantumGatesSC.H);

    QRegisterSC ha = new
    QRegisterSC(operation.MatrixMultiplication(H2,
    a));

    QRegisterSC haO = new
    QRegisterSC(Oracle(ha,valueToSearchInClassical)
    );

    QRegisterSC haOh = new
    QRegisterSC(operation.MatrixMultiplication(H2,
    haO));

    X2 = operation.Tensor(QuantumGatesSC.X,
    QuantumGatesSC.X);
   QRegisterSC haOhX = new
   QRegisterSC(operation.MatrixMultiplication(X2,
   haOh));

   HI = operation.Tensor(QuantumGatesSC.I,
   QuantumGatesSC.H);

   QRegisterSC haOhXh = new
   QRegisterSC(operation.MatrixMultiplication(HI,
   haOhX));

   QRegisterSC haOhXhCNOT = new
   QRegisterSC(operation.MatrixMultiplication(Quan
   tumGatesSC.CNot, haOhXh));
```

```
    QRegisterSC haOhXhCNOTh = new
    QRegisterSC(operation.MatrixMultiplication(HI,
    haOhXhCNOT));

    QRegisterSC haOhXhCNOThX = new
    QRegisterSC(operation.MatrixMultiplication(X2,
    haOhXhCNOTh));

    QRegisterSC haOhXhCNOThXh = new
    QRegisterSC(operation.MatrixMultiplication(H2,
    haOhXhCNOThX));

    string s1 = haOhXhCNOThXh.ToString();
    txtGroverSearchResult.Text = s1;

    dataGridView1.DataSource =
    haOhXhCNOThXh.ToDataTable();

    txtProbabilityAmplitude.Text =
    haOhXhCNOThXh.ComputeProbabilityAmplitude().ToS
    tring();
}

public ComplexMatrixSC Oracle(ComplexMatrixSC
matrix, int valueToSearchInClassical)
{
    ComplexMatrixSC ResultantMatrix = new
    ComplexMatrixSC(matrix.Rows, matrix.Cols);

    MatrixOperationSC operation=new
    MatrixOperationSC();

    ComplexNumber t= new ComplexNumber(-1,0);

    for (int i = 0; i < matrix.Rows; i++)
    {
        for (int j = 0; j < matrix.Cols; j++)
        {
            if (i == valueToSearchInClassical)
            {
                ResultantMatrix[i, j] =t*
                matrix[i, j];
            }
```

```
                    else
                    {
                        ResultantMatrix[i, j] = matrix[i,
                        j];
                    }
                }
            }
            return ResultantMatrix;
}
```



Grover Search Simulation

Enter value to search between 0 and 3: 2

Grover Search Result:

|(0, 0)|00> + (-2.980232E-08, 0)|01> + (-0.9999999, 0)|10> + (2.980232E-08, 0)|11>>

| | 0 |
|---|---|
| | 0 |
| | -2.980232E-08 |
| | -0.9999999 |
| ▶ | 2.980232E-08 |
| * | |

Probability Amplitude of Resultant Q-Resgister: 1

Figure 25: Gover Search Simulation