Computer Science • 16 (1) 2015 http://dx.doi.org/10.7494/csci.2015.16.1.103

Joanna Patrzyk
Bartłomiej Patrzyk
Katarzyna Rycerz
Marian Bubak

TOWARDS A NOVEL ENVIRONMENT FOR SIMULATION OF QUANTUM COMPUTING

Abstract *In this paper, we analyze existing quantum computer simulation techniques and their realizations to minimize the impact of the exponential complexity of simulated quantum computations. As a result of this investigation, we propose a quantum computer simulator with an integrated development environment – QuIDE – supporting the development of algorithms for future quantum computers. The simulator simplifies building and testing quantum circuits and understanding quantum algorithms in an efficient way. The development environment provides flexibility of source code edition and ease of the graphical building of circuit diagrams. We also describe and analyze the complexity of algorithms used for simulation as well as present performance results of the simulator as well as results of its deployment during university classes.*

Keywords quantum computation, quantum computer simulators, development environment, quantum algorithms, SUS survey

Citation Computer Science 16 (1) 2015: 103–129

1. Introduction

In recent years, the field of quantum computing has developed significantly. In [25] and [30], the results of experimental realizations of Shor's Prime Factorization Algorithm [44] are presented. Moreover, the first prototypes of a quantum central processing unit were built: one realizing the von Neumann's architecture [28] and another exploiting quantum annealing [21, 24].

Quantum computers can offer an exponential speedup compared to conventional computers [7]. However, universal quantum computers are not yet available. Also, as it has been proven by Richard Feynman that a quantum system can be efficiently simulated only by another quantum system [17]. Conventional computers require exponentially-more time and memory to perform quantum computations.

Despite these restrictions, there is a need for quantum computer simulators to help learn and develop algorithms for future quantum computers. In this paper, we present the evaluation results of the most important simulation techniques in existence as well as their realization. Up to our knowledge, the decribed simulators do not fully support convenient and efficient way of learning quantum algorithms. Regarding the support for convenient learning criterion, each of the simulators provided either a complicated console interface, a raw library Application Programming Interface (API), or an oversimplified graphical user interface. Therefore, as a result of our research, we propose a new environment that combines the flexibility of library API with the ease of use of a graphical Integrated Development Environment (IDE). This makes it convenient to learn, develop, and analyze quantum circuits. Regarding the efficiency criterion, we investigate different approaches to minimize the impact of the exponential complexity of the simulator. We describe data structures used for storing quantum state and analyze algorithms for using these data structures. Next, we present the performance results of the simulator.

This paper is organized as follows. In Section 2, we introduce the basic terms of Quantum Computation Theory. In Section 3, we review existing solutions for modeling quantum computations on conventional computers. We briefly describe the data structures and algorithms used for performing simulations. In Section 4, we review the existing software for simulating quantum computing and classify them in terms of their interfaces and capabilities. We also explain why we decided to build a new simulator. In Section 5, we present the requirements for the new simulator and propose our solution to meet these requirements – the Quantum Integrated Development Environment (QuIDE). Section 6 describes the architectural components of QuIDE, explaining their responsibilities and interactions. In Section 7, we focus on the core simulation module of QuIDE. We describe the algorithms and data structures used for performing the actual simulations. In Section 8, we estimate their space and time complexity. In Section 9, we demonstrate the capabilities offered by QuIDE's user interface. In Section 10, we present the results of functionality and performance evaluation of QuIDE. We also describe the results of deploying the QuIDE simulator in academic classes. We present the usability evaluation and list the quantum algorithms

implemented on QuIDE. In Section 11, we summarize our work and propose ideas for further steps.

2. Basics of quantum computing

The Quantum Information and Computation Theory is a wide field which cannot be easily explained in this short section. An exhaustive explanation can be found in the referenced books [35, 32].

Conventional computers operate on bits (classical bits – cbits). The quantum computer performs computations on quantum bits (qubits). The classical bit can be in one of the two states – either 0 or 1. The qubit can be in the superposition of the states $|0\rangle$ and $|1\rangle$. The notation $|\cdot\rangle$ is called the **Dirac notation** and is the standard notation for states in quantum mechanics. The special states $|0\rangle$ and $|1\rangle$ are known as **computational basis states** and form an orthonormal basis for this vector space.

Qubits can be realized by many different physical systems. For example, in the atom model, the electron can exist in either the so-called 'ground' or 'excited' states, which can be denoted as $|0\rangle$ and $|1\rangle$ respectively.

The state of a single qubit can be described as $|\Phi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where α and β are complex numbers ($\alpha, \beta \in \mathbb{C}$). The α and β are called amplitudes and have to fulfill the normalization condition $|\alpha|^2 + |\beta|^2 = 1$. We say that the state $e^{i\theta}\,|\Phi\rangle$ is equal to $|\Phi\rangle$, up to the **global phase factor** $e^{i\theta}$, where θ is a real number [35].

For the two-qubit system, the computational basis is formed by four states: $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. We also denote them as $|0\rangle_2$, $|1\rangle_2$, $|2\rangle_2$, and $|3\rangle_2$, where the subscript represents the number of qubits.

Similar to classical bits, qubits can form quantum registers. The state of the n-bit classical register is one of the values between 0 and $2^n - 1$. The state of the n-qubit quantum register is represented as

$$|\Psi\rangle_n = \alpha_0\,|0\rangle_n + \alpha_1\,|1\rangle_n + \ldots + \alpha_{2^n-1}\,|2^n-1\rangle_n\,, \quad \text{where} \quad \sum_{j=0}^{2^n-1} |\alpha_j|^2 = 1. \qquad (1)$$

The interpretation of such a state is that the register is simultaneously in multiple-basis states. This phenomenon is called the **quantum superposition**. This is the fundamental property of quantum computers. This is utilized in many algorithms, such as Shor's Factorization Algorithm [44], in order to achieve exponential speedup over classical algorithms. However, there is one drawback – the actual state of the register cannot be learned. It is only possible to **measure** this state. The measurement yields only one label j with probability $|\alpha_j|^2$. After the measurement, the information about the other possible states and their amplitudes is lost. This model of measurement is called **projective** or **von Neumann** measurement [35].

The state of the quantum register formed by several individual qubits can be computed as a **tensor product** of their states. However, not every state of the n-qubit quantum register can be split into the n states of the individual qubits. This

phenomenon is called **quantum entanglement**. The entangled qubits form a tightly-coupled system where an operation on one qubit may affect the other qubits.

The operations on qubits are preformed using quantum gates. All quantum operations are reversible, except the measurement. A quantum gate can operate on a single qubit or on multiple qubits. The quantum gates are described by unitary matrices.

The n-qubit quantum register is described by an 2^n-element vector (called the state vector). The application of a quantum gate is multiplying this vector by the gate's matrix.

The quantum computation consists of quantum gates applied to quantum registers. Similar to conventional computers built from electrical circuits containing wires and logic gates, a quantum computer is build from a **quantum circuit**. A quantum circuit contains wires and elementary quantum gates to manipulate quantum information. The wires are not the physical wires; rather, they correspond to the passage of physical particles such as photons. The computations are executed from left to right, simultaneously for all wires.

3. Overview of quantum computing simulation techniques

The simulation of quantum computations is very time and memory consuming. Memory consumption is especially crucial because even simple systems can exceed available memory, even in supercomputers. The trivial approach is to use vectors to represent quantum registers and matrices for the operations. The n-qubit quantum system is represented by a 2^n-element vector. The whole quantum circuit is represented by $2^n \times 2^n$ matrix, which is constructed by multiplying and performing a Kronecker product of the matrices representing the quantum gates. The result of the computation is obtained by multiplying the matrix by the vector. Such operations require a large amount of memory, which is presented in Table 1. Below, we present techniques which are more efficient than the trivial approach.

Table 1

Memory usage of quantum computing simulation system based on matrix-vector representation.

Number of qubits	5	10	20	21
Memory Usage (state vector)	512 B	16 kB	16 MB	32 MB
Memory Usage (operation matrix)	16 kB	16 MB	16 TB	64 TB

Numerical Linear Algebra Methods is the most general technique for simulating the time evolution of a quantum system based on solving the Schrödinger's equation. For this purpose, it exploits methods such as matrix diagonalization, Chebyshev Polynomial Algorithm, Short-Iterative Lanczos Algorithm [26], or Suzuki-Trotter Product-Formula Algorithm [47]. These methods are reviewed and compared in [40]. Depending on the method, it requires from $O(2^n)$ to $O(2^{2n})$ memory. The latter, with memory and computational complexity of $O(2^n)$, is used by `Quantum Computer`

`Emulator` [41]. However, these techniques should be chosen only when an exact simulation of quantum system time evolution is needed.

Qubit-wise Multiplication approach allows us to store the 2^n-element state vector instead of computing the whole $2^n \times 2^n$ matrix representing the quantum circuit. The quantum gates are applied one by one. To obtain the result of applying the gate, only the state vector and gate matrix are needed. This approach reduces memory complexity from $O(2^{2n})$ to $O(2^n)$, and it can be also parallelized [37]. This method is widely used by quantum computer simulators: IT Java Mathematics Library [56], `qclib` [36], or `QCSim` [8].

P-blocked State Representation [22] reduces the amount of memory for storing the state vector from $O(2^n)$ to even $O(n)$ in the best case. The state of the n-qubit quantum system is represented by the so-called *p-blocked* state. The state is *p-blocked* if it is a tensor product of k non-entangled states of at most p qubits. Such representation requires $O(k2^{2p})$ memory. However, this approach needs an algorithm to keep track of entangled states, and after each operation, the number p have to be recomputed. The worst case is when all qubits are entangled; this is very common.

Binary Decision Diagrams (BDD) can be used instead of storing the 2^n-element state vector. In the best case, memory complexity is reduced even to $O(1)$; but in the worst case, it still requires $O(2^n)$ memory. Using BDD instead of simple vectors is very complicated. The application of even the simplest gates requires the complicated transformation of decision diagrams, while the most complex operations require many such transformations and, thus, are very time consuming. This method was applied in the `QDD` simulator [18] and further developed in `QuiDDPro` [53, 54].

Hash Table State Representation method, proposed in the `libquantum` simulation library [11], uses the simple state vector to represent a quantum state. However, the whole 2^n-element vector does not have to be stored. Only the non-zero values of the vector are stored in a hash table, as key-value pairs. The keys are the basis states (for example, $|0\rangle_n$) and the values are their amplitudes which makes it possible to store the state vector using from $O(2^n)$ of memory in the worst case to even $O(1)$ in the best case. The advantage of this method is that the application of quantum gates is very simple.

The quantum state was also simulated via **Bayesian networks** in the `Quantum Fog` simulator [49], where the results were calculated by exploiting Monte Carlo methods. There is also a proposition to model the quantum states with the **tensor network** data structure [29] – a graph of tensors which are multidimensional generalization of matrices. In this method, memory and time complexity grow exponentially with the dimension of the largest tensor. Another simulation technique uses the **Schmidt decomposition** and applies it to represent a quantum state [16, 55]. However, this method is reasonably memory-efficient only when there is a small number of entangled qubits.

Many of these techniques, such as *p-blocked state representation*, *Bayesian networks*, *tensor networks*, or *Binary Decision Diagrams* are mostly theoretical. Also, the complexity of operating on them often obscures their better memory results. The linear-algebra-based methods are good for for modeling the evaluation of quantum systems, but inappropriate for simulation of quantum computations which can be realized in a more efficient way. Therefore, we decided to build a new simulator using a variant of the **Hash Table State Representation** method. It is one of the most memory-efficient techniques, and it is relatively easy to implement all needed operations on this representation.

4. Evaluation of representative quantum computer simulators

Existing simulators can be divided into different types: simulation libraries, Quantum Programming Languages, interpreters, graphical simulators, and toolboxes.

Simulation Libraries group includes the libraries for standard programming languages such as C, Java, C#, and Python. They provide functions for creating qubits and operating on them. The simulation libraries provide significant flexibility due to the fact that quantum operations can be integrated into classical programs. The users can take advantage of all of the special features of the programming language, such as classes, loops, or exception handling. On the other hand, simulation libraries can be difficult to use. They can be used only by users with programming skills. They require the knowledge of a specific programming language and usually cannot be used in a different language. Another drawback of this group is that the user has to take care of displaying the output of the simulation and implementing the stepping execution. Each of these features has to be implemented manually. The examples of such simulation libraries are: `libquantum` [11], `Eqcs` [4], `QDD` [18], `Q++` [13] (for C/C++); `m@th IT Java Mathematics Library` [56] for Java, `Cove` [39] for .NET, `QuTiP` [20] and `qclib` [36] for Python, and even `Haskell Simulator of Quantum Computer` [45] for Haskell and `qlambda` [48] for Scheme.

Quantum Programming Languages are designed especially for describing quantum computations. Some of them are purely theoretical, but the others provide interpreters or compilers. Quantum Programming Languages (QPLs) are much easier to learn than *Simulation Libraries*. Also, they better represent the domain of quantum computing. However, the QPLs offer less flexibility and a smaller set of directives than classical languages. They cannot be integrated with any other libraries. Also, their capabilities of displaying the output of the simulation are limited to the functions included in the QPL. The examples of QPLs are `qMIPS` [51], `CHP` [1], `QCL` [38], `LanQ` [34], `kulka` [33] `LIQUid` [59] or `QuIDDPro` [54].

Interpreters are simulators which provide the interactive Command Line Interface (CLI). They are as easy as the language which they interpret – therefore, their ease is comparable to the *QPLs* or *Simulation Libraries*. Their biggest advantage is their interactivity, which enables users to execute simulations step by step and to preview

the quantum state at any point during the simulation. Their drawbacks are similar to the *QPLs*: they provide a limited set of function and syntax constructions, and they are inconvenient for performing a batch execution. The simulators that offer this type of interface are `CHP` [1], `QCL` [38], `LanQ` [34], `kulka` [33] and `QuIDDPro` [54].

Graphical Simulators are the simulators which provide the Graphical User Interface (GUI). They are convenient to use, even for users with no programming experience. The GUI makes it easier to construct and analyze the algorithms. However, many such tools enable users to simulate only a single algorithm or a very limited set of operations. Also, creating a quantum circuit using GUI editor is often more time consuming than writing source code, especially for complex algorithms with repeated subroutines. The examples of graphical quantum circuit simulators are `QCAD` [58], `Quantum Computer Emulator` [41], `SimQubit` [27], `jQuantum` [57], `Qubit101` [52], `Zeno` [12], Online Quantum Computer Simulator [31] and Javascript Quantum Circuit Simulator [60]. There are also graph-based quantum computation simulators: `Quantum Fog` [49] and `Quantomatic` [15]. In addition, there are many simulation tools that allow users to model only a single quantum phenomenon. These are `Java Quantique Simulator` [9], Grovers Quantum Search Simulation Applet [50], or Bloch Sphere Simulation [43].

Toolboxes for the Scientific Software This group includes packages for MATLAB, Octave, Mathematica, etc. They are a special kind of *Simulation Libraries* and, thus, have similar advantages and drawbacks. However, an additional drawback is their requirement for such scientific software.

The review of existing simulators showed that available software is insufficient for convenient scientific usage, especially for educational needs. Many available simulators only focus on some subfield of the Quantum Information Theory: they only allow users to simulate a concrete quantum algorithm or effect, such as Quantum Key Distribution [5] or Quantum Walks [23]. Moreover, most existing simulators provide only command-line textual interfaces, which are difficult to read unless the simulated system is very simple. For bigger systems, the results need to be visualized in external tools, since the simulators do not provide such visualization methods. On the other hand, GUI simulators usually have very limited functionality. Most of them do not support building custom subroutines. Moreover, GUI simulators allow us to use a limited (and rather small) number of qubits. In existing GUI simulators, there is also no possibility of combining classical and quantum computations. Of course, the Simulation Libraries provide all of these functions. They could be difficult to use, however, especially for people with poor programming experience. What is more, in such a case, users are responsible for programming every action during the simulation, even the stepping execution or printing the visualization of the simulated quantum state. These observations confirmed that there is a need for new simulation software. In the following section, we present our idea of a quantum simulator that will not suffer from the presented drawbacks.

5. A Concept of an innovative quantum computing simulator

In this paper, we propose the Quantum Integrated Development Environment that supports learning, understanding, and analyzing quantum algorithms. The simulator should be understandable and usable for any person with basic knowledge of quantum information and computation theory. Additionally, its performance has to be sufficient to execute the algorithms in a reasonable time on standard PCs. The simulation environment has to include a set of example applications, such as the most important quantum algorithms (like Shor's Factoring algorithm [44] and Grover's Database Search [19]).

The main functional requirements of the simulator include providing and managing elementary quantum gates and quantum registers described in Section 2 and performing actual computations. The proposed simulator should support: building custom computation subroutines out of the elementary gates, combining quantum and classical computations, previewing of the internal state of the simulated quantum system, and its step-by-step execution.

The simulator has to be easy to obtain, install, and run, even by users with no programming experience. In Section 4, we divided existing simulators into different groups: GUI-based simulators, Interpreters, and Simulation Libraries. QuIDE integrates features from all of these groups, as illustrated in Figure 1.

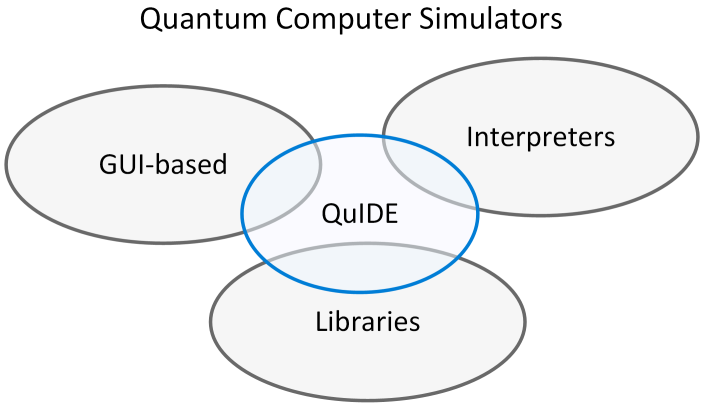

Figure 1. QuIDE integrates the approaches from different groups of existing simulation software.

Figure 2 presents the most important modules and features of the QuIDE simulator. The simulator provides the Code Editor, which enables the user to program simulations using quantum and classical operations. QuIDE also provides the Circuit Designer, in which the quantum circuits can be interactively constructed. The user is able to switch between the source code and graphical circuit at any time of the

designing process. It is possible to execute the simulation step-by-step. The internal quantum state is presented in the Run-Time Preview during the simulation.

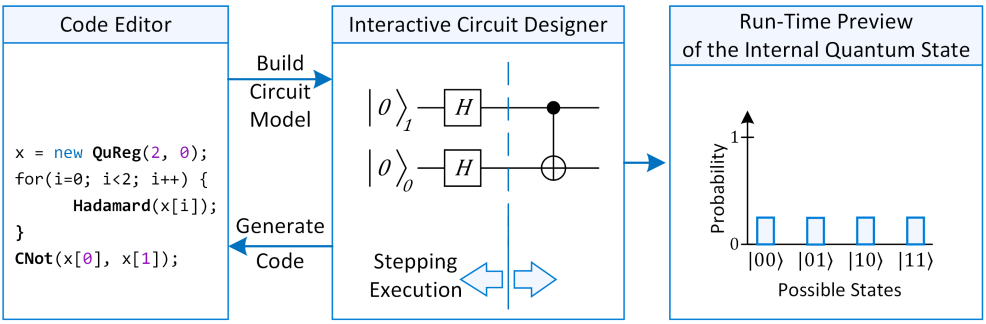

Figure 2. The key modules and capabilities of QuIDE simulator.

6. QuIDE architecture

QuIDE consists of the layers which separate the application logic from the interface exposed to the user. This is achieved by the Model View ViewModel (MVVM) architectural design pattern [46], which is presented in Figure 3. Each layer includes components that are responsible for providing specific application functions.

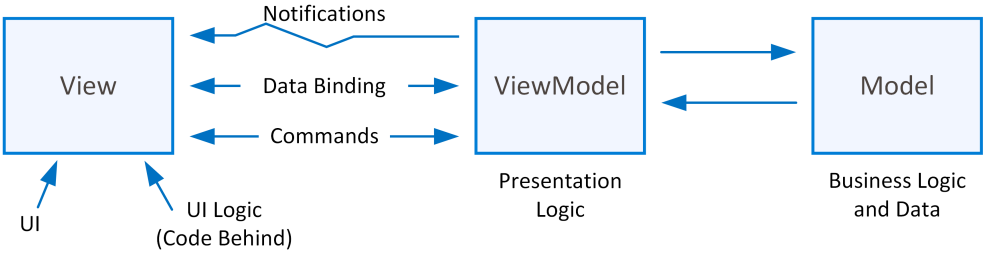

Figure 3. The Model View ViewModel (MVVM) architectural pattern. In MVVM, the View layer is concerned only about the graphical user interface, while the Model layer only about the business logic. All communication between them is realized by the ViewModel layer.

The **View** layer contains no business logic. It is responsible for displaying the GUI based on the data from the **ViewModel** layer as well as passing the user's actions on to the **ViewModel** layer. The **ViewModel** updates the **Model**, based on the user's actions passed on from the **View** layer. It also translates data from the **Model** so it can be displayed by the **View**. The **Model** is the application logic layer. It is responsible for representing the quantum circuits and performing computations. It also implements all of the supporting functions of the simulator, such as source code generation from the quantum circuit.

QuIDE components with respect to the Model-View-ViewModel layers are shown in the Figure 4.

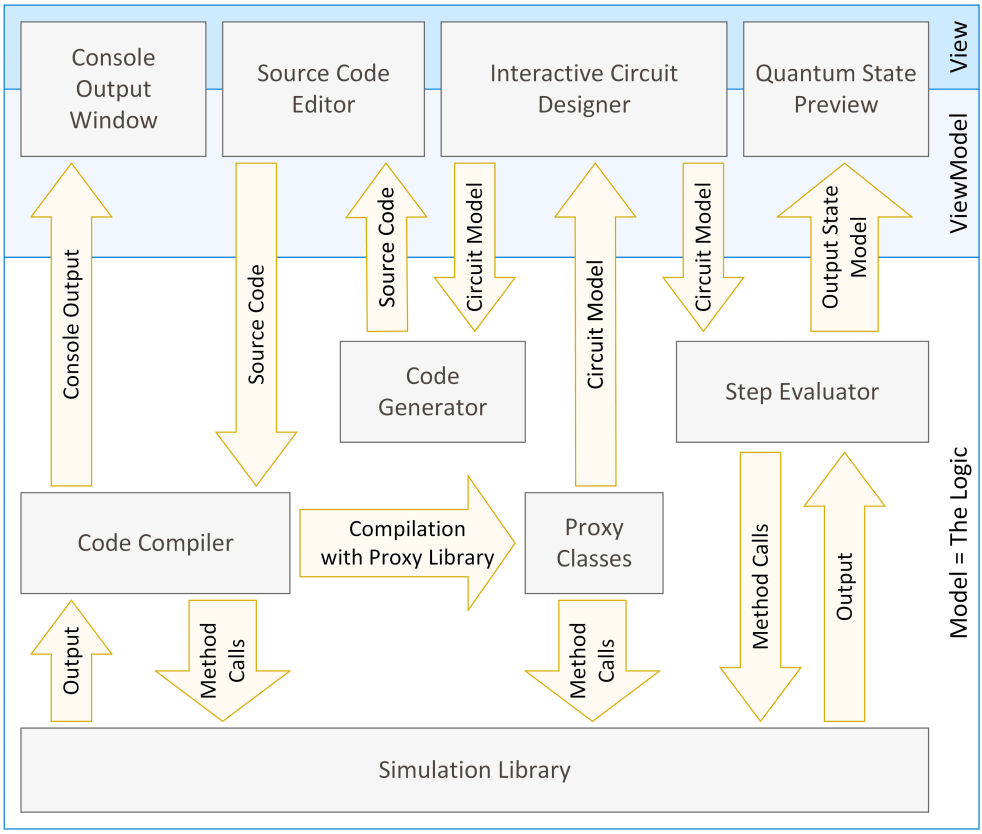

Figure 4. The architecture of QuIDE with respect to Model-View-ViewModel layers. The core is formed by the Simulation Library, which can be used independently to simulate quantum computations.

The top components interact directly with the users and offer them specific functions. However, the actual realization of these functions is done by the cooperation of multiple components; both in the UI and the core layer. The **Interactive Circuit Designer** enables users to graphically construct the quantum circuits. It provides easy methods for adding qubits or quantum gates as well as editing, copying, or deleting them. Moreover, the user is enabled to build custom, complex blocks by composing simpler blocks or built-in quantum gates. When the user wants to generate source code from the circuit, the **Interactive Circuit Designer** passes the circuit model to the **Code Generator**. Also, it displays the circuit model received from the **Proxy Classes** when the user chooses to generate the circuit from the code. In order to evaluate the circuit, it passes the circuit model to the **Step Evaluator**. The **Console**

Output Window displays the standard output if the user decides to execute the simulation in the console mode. This component shows the data received from the **Code Compiler**. The **Source Code Editor** enables the user to write the program code of the simulation. The source code written in the **Source Code Editor** can be directly executed within QuIDE, saved or restored. If the user wants to execute the written program, the source code is passed on to the **Code Compiler**. Also, the **Source Code Editor** displays code generated by the **Code Generator**. The **Quantum State Preview** displays the current quantum state of the simulated circuit. It receives data to be shown from the **Step Evaluator**.

The **Simulation Library** is a core module, responsible for computing the results of simulations. It is described in detail in Section 7. The **Code Compiler** processes the code from the **Source Code Editor**. It compiles the code dynamically at runtime. When the user wants to execute the code in the console mode, the **Code Compiler** uses the **Simulation Library** for the compilation. In order to generate the quantum circuit from the source code, the **Code Compiler** compiles the code using the **Proxy Classes**. The **Proxy Classes** are used when QuIDE generates the circuit from the source code. The **Proxy Classes** are mock libraries which are executed instead of the **Simulation Library** in order to build the circuit. The **Code Generator** takes the circuit model from the **Interactive Circuit Designer** and generates the source code, which is then passed on to the **Source Code Editor**. The **Step Evaluator** is used when the user wants to execute the quantum circuit. It takes the circuit model from the **Interactive Circuit Designer**. Then, it executes each of its elements using the **Simulation Library**. Then, it updates the **Quantum State Preview**.

7. Quantum Simulation Library

QuIDE is based on the **Simulation Library**, as shown in Figure 4. The library has been developed especially for QuIDE. It is based on Hash Table State Representation, a simulation technique described in Section 3. The library is written in the C# language. It is also available as an independent library.

In this section, we describe the data structures used for representing the quantum state and the implementations of quantum operations. Then, we analyze the complexity of the implemented algorithms.

7.1. Data structures

The simulation library is based on Hash Table data structures, because they are the most memory efficient. We decided to use the Dictionary data structure, available in C#, which implements the Hash Table concept.

Quantum registers are represented by a **Register** class. It has an attribute **Width**, which stands for the number of qubits within the register. The internal quantum state of the register is represented by the Dictionary data structure. As presented in Figure 5, the dictionary's keys are unsigned long integers representing the basis states. The value for the given key is the complex amplitude of that basis state.

The user can define multiple **Register** instances. The **Quantum Computer** is a singleton class which manages the lifetime of the registers and is responsible for performing any cross-register operations.

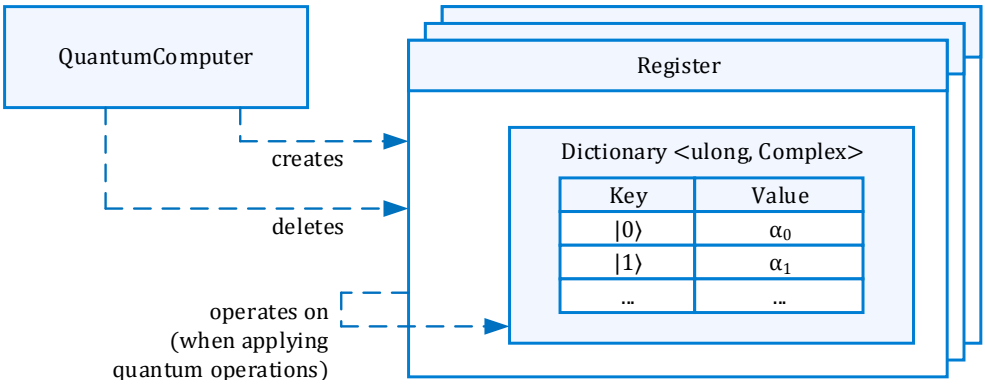

Figure 5. In QuIDE simulation library, the quantum register is represented by the `Register` class. Many registers can be used at the same time, as they are allocated and deallocated by a singleton `QuantumComputer` class. The data structure used for storing the information about the internal quantum state is the dictionary.

7.2. Implementation of Quantum Operations

In this section, we present how to operate on the Dictionary representation of quantum state to ensure maximum memory efficiency. We describe the algorithms for applying the Controlled-Not (C-Not) gate, the universal U gate represented by the matrix, as well as the measurement of quantum state.

7.2.1. The C-Not Gate

The C-Not gate is the simplest controlled quantum operation. The equation (2) shows the C-Not applied to the 2-qubit register, where the 0th qubit is the control and the 1st is the target. The C-Not operation swaps the amplitudes of the basis states, if and only if the control bit is set to 1:

$$\begin{aligned} \mathbf{C}_{01}(\alpha_0 |00\rangle + \alpha_1 |01\rangle + \alpha_2 |10\rangle + \alpha_3 |11\rangle) = \\ = \alpha_0 |00\rangle + \alpha_1 |11\rangle + \alpha_2 |10\rangle + \alpha_3 |01\rangle . \end{aligned} \tag{2}$$

The implementation of this gate is presented in Algorithm 1. For each basis state, we have to exchange its amplitude with the state with reversed target bit. We use a set S to perform this swapping only once for every such pair. Let us consider applying the C-Not from equation (2). Without using the set S, the *for* loop from the line 3 would firstly encounter the state $|01\rangle$ and exchange its amplitude with the state $|11\rangle$,

and then swap those amplitudes again when encountering the state $|11\rangle$. We prevent this by storing in S the states already exchanged (line 11) and checking before each potential swap (line 4). The *if* statement in line 5 is responsible for performing the swap only if the control bit is set to 1 – this is a way in which the controlled gates act. Since the states with zero amplitude are not stored in D, the *if-then-else* block was needed (lines 8–15). In the *if* block the amplitudes are simply interchanged. In the *else* clause, the state with reversed target bit had formerly zero amplitude. Thus, after swapping, the *state* gets amplitude equal to zero, so it has to be removed from D (line 14).

Algorithm 1 C-Not implementation for the dictionary-based state representation

Require: A dictionary D representing the state vector. Its keys are the basis states, and its values are the corresponding amplitudes. **If any basis state has an amplitude equal to 0, it should not be stored in the dictionary.** To access a value for the given key, we use the notation $D[key]$.

Ensure: The actualized D, after the C-Not gate application.

```
 1: procedure C-Not
 2:     S ← ∅
 3:     for all state ∈ keys(D) do
 4:         if state ∉ S then
 5:             if ControlBitIsSet(state) then
 6:                 amplitude ← D[state]
 7:                 reversedTargetState ← ReverseTargetBit(state)
 8:                 if reversedTargetState ∈ keys(D) then
 9:                     D[state] ← D[reversedTargetState]
10:                     D[reversedTargetState] ← amplitude
11:                     S ← S ∪ {reversedTargetState}
12:                 else
13:                     D[reversedTargetState] ← amplitude
14:                     Delete(D[state])
15:                 end if
16:             end if
17:         end if
18:     end for
19: end procedure
```

The function $ControlBitIsSet(state)$ in line 5 checks whether the control bit in the given *state* from the computational basis is set to 1. In equation (2) the 0th qubit is the control, so in this example $ControlBitIsSet(|00\rangle)$ returns false, $ControlBitIsSet(|01\rangle)$ returns true, and so on. The bits are counted starting from the rightmost (least significant) bit. The function $ReverseTargetBit(state)$ in line 7 reverses the target bit in the given *state* from the computational basis. For example, $ReverseTargetBit(|01\rangle) = |11\rangle$ and $ReverseTargetBit(|11\rangle) = |01\rangle$, where the first qubit is the target. The $Delete(D[state])$ function in line 14 removes the entry with key *state* from the dictionary D.

7.2.2. The matrix-defined universal gate

QuIDE enables the user to define and apply any 1-qubit quantum operation, represented by an arbitrary unitary matrix. The Algorithm 2 presents how it could be computed without any additional memory using Dictionary state representation. The similar method was proposed and implemented in **libquantum** simulation library [11].

Similar to the C-Not operation, we need to find pairs of basis states, which differs only in the target bit. Let α be an amplitude of the basis state $|* \cdots * 0 * \cdots *\rangle$ and β an amplitude of $|* \cdots * 1 * \cdots *\rangle$ (the rest of bits – noted as "*" – are exactly the same for both states). The resulting amplitudes for these basis states, α' and β' can be computed as shown in (3). If we perform these computations for each such pair of basis states, we will obtain a final result

$$\begin{bmatrix} \alpha' \\ \beta' \end{bmatrix} = U \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} a\ \alpha + b\beta \\ c\alpha + d\beta \end{bmatrix}. \tag{3}$$

As in algorithm 1, we use a set S to prevent from processing the same pair of states twice (lines 2, 4, 10). In line 5 we start to initialize α and β. After line 11, they hold amplitudes of the currently-processed *state* and the corresponding state with reversed target bit – as in equation (3). In (3) we assume, that the target bit of the processed state is set to 0. The same is checked in line 12. Thus, in the *if* clause we perform the same computations as in (3). For the opposite case, we need to swap computations. The α' and β' become the new amplitudes for the processed basis state (*state*) and the state with reversed target bit ($reversedTargetState$). In lines 19–28, we assign these values by putting them into D. However, we first check, if they are not extremely close to zero, which can be caused by a lack of precision for floating-point operation. In such cases, we assume that these amplitudes are equal to zero and remove them from D.

The functions $ReverseTargetBit(state)$ in line 7 and $Delete(D[state])$ in line 20 are described in Section 7.2.1. The function $TargetBitIsZero(state)$ in line 12 checks whether the target bit in the given *state* from the computational basis is set to 0. For example, if we apply this function to the 2-qubit register and the 1st qubit is the target, $TargetBitIsZero(|00\rangle)$ returns true, $TargetBitIsZero(|10\rangle)$ returns false, and so on.

7.2.3. The general multi-qubit operations

Any reversible quantum operation which is applied to k qubits can be represented by a $2^k \times 2^k$ unitary matrix. However, as explained in Section 3 and Table 1, using such matrices entails inefficient and very high memory usage. QuIDE does not allow us to define any multi-qubit operation by a unitary matrix. Instead, it provides adding control bits to any 1-qubit gate, building composite gates from existing quantum gates and creating subroutines from available gates which can be then accessed and executed as a reusable, multi-qubit gates. It has been proven that any multi-qubit

Algorithm 2 Matrix-defined 1-qubit unitary gate application

Require: A dictionary D representing the state vector (as in algorithm 1); A 2×2 unitary matrix U, consists of numbers a, b, c and d, as in equation (3).

Ensure: The actualized D, after the unitary U gate application.

```
1: procedure UNITARY(U)
2:     S ← ∅
3:     for all state ∈ keys(D) do
4:         if state ∉ S then
5:             α ← D[state]
6:             β ← 0
7:             reversedTargetState ← ReverseTargetBit(state)
8:             if reversedTargetState ∈ keys(D) then
9:                 β ← D[reversedTargetState]
10:                S ← S ∪ {reversedTargetState}
11:            end if
12:            if TargetBitIsZero(state) then
13:                α′ ← a α + bβ
14:                β′ ← cα + dβ
15:            else
16:                β′ ← a β + bα
17:                α′ ← cβ + dα
18:            end if
19:            if |α′|² < ϵ then
20:                Delete(D[state])
21:            else
22:                D[state] ← α′
23:            end if
24:            if |β′|² < ϵ then
25:                Delete(D[reversedTargetState])
26:            else
27:                D[reversedTargetState] ← β′
28:            end if
29:        end if
30:    end for
31: end procedure
```

reversible quantum operation can be built up from a small set of elementary gates [3]. QuIDE provides all of these elementary gates.

Adding control bits to any 1-qubit gate is achieved by integrating Algorithm 1 and Algorithm 2. We check the control bit as in Algorithm 1 and then operate on the amplitudes as in Algorithm 2. As a result, QuIDE allows to add a control bit to any 1-qubit gate. In addition, several gates (e.g., Toffoli gate) can have any number of control bits.

Composite gates are very simple mechanisms for building multi-qubit operations. When users build a quantum circuit, they can select any part of this circuit and

group selected gates into a single, multi-qubit gate. This gate can be then reused and is represented by a single quantum gate. During the execution of such gate, QuIDE simply applies the inner gates of the composite gate, one by one.

Subroutines are the most-flexible method of defining multi-qubit operations. They can be defined in the source code. They can be parametrized and can include classical flow control statements, such as loops and conditions. Subroutines enable the user to integrate classical and quantum operations, which is common in implementing several quantum algorithms. In QuIDE, subroutines are represented as reusable blocks.

7.2.4. The measurement

In QuIDE, there are two methods for measuring the quantum bits. It is possible to measure a single qubit or the whole quantum register. The Algorithm 3 describes the dictionary-based implementation of the quantum register measurement.

Algorithm 3 MEASUREMENT implementation for the dictionary-based state representation

Require: A dictionary D representing the state vector (as in algorithm 1).
Ensure: The result of the measurement – one of the possible basis states; the actualized D, after the measurement operation.

```
 1: function MEASURE
 2:     random ← NextRandomReal(0.0, 1.0)
 3:     sum ← 0
 4:     result ← 0
 5:     for all state ∈ keys(D) do
 6:         result ← state
 7:         α ← D[state]
 8:         sum ← sum + |α|^2
 9:         if sum ⩾ random then
10:             break
11:         end if
12:         DeleteAll(D)
13:         D[result] ← 1
14:     end for
15:     return result
16: end function
```

At first, we pick a random real number between 0 and 1 (line 2) and initialize auxiliary variables (lines 3, 4). The *sum* variable will hold the sum of the probabilities of the subsequent basis states. Once it exceed the random value (line 9), we stop the algorithm, and the currently-processed basis state becomes the result of the measurement. In this way, we ensure that the measurement of the quantum state returns a random state from possible-basis states. Before leaving the function block, the quantum state has to be destroyed – in order to fully simulate the measurement operation. The superposition of the basis states is collapsed; after this, it contains only the measured state with probability 1. This is achieved by removing all values from

D and putting into it only the resulting state (as dictionary key) with its amplitude equal to 1 (as dictionary value) (lines 12, 13).

The function $NextRandomReal(0.0, 1.0)$ in line 2 returns a pseudo-random real number within a given range; in this case the range is $[0.0, 1.0)$. The function $DeleteAll(D)$ in line 12 removes all entries from the dictionary D.

8. Analysis of Simulation Complexity

Space Complexity. The quantum system's state vector is stored in the dictionary. It is the only data structure used for representing quantum computations. Only the basis states for which the amplitudes are non-zero are stored in the dictionary. If the n-qubit quantum system is in one of the pure basis states, the dictionary of size 1 is sufficient to store the state. The space complexity in the best case is, therefore, $O(1)$. In the worst case, all of the 2^n amplitudes of the n-qubit quantum system have to be stored. As a result, space complexity in the worst case equals $O(2^n)$.

Time Complexity. The Algorithms 1, 2 and 3 present the examples of operations on the quantum state. Each quantum operation in QuIDE is implemented in a similar way using the *for* loop. This loop iterates over all of the entries in the dictionary representing the state vector. For this reason, the complexity of any operation depends on the size of this dictionary. As shown in Section 8, the size of the dictionary varies from $O(1)$ to $O(2^n)$. As a result, the time complexity of a single operation on an n-qubit quantum system equals $O(1)$ in the best case and $O(2^n)$ in the worst case.

9. QuIDE User Interface Capabilities

The Graphical User Interface of QuIDE is depicted in Figure 6. The program window consist of three main parts: **Source Code Editor**, **Interactive Circuit Designer** and **Run-Time Preview**.

In the **Source Code Editor**, the user can define the quantum simulation by writing the source code. The user can execute the source code in the **Console Output Window** by clicking the **Run In Console** button **(2)**. The user can be working with multiple source code files opened in tabs.

The **Interactive Circuit Designer** enables us to build the quantum circuit using graphical symbols. The user can generate the circuit diagram from the source code using the **Build Circuit** button **(1)**. The circuit can be translated into the source code using the **To Code** button **(4)**.

QuIDE provides a set of elementary quantum gates. The reusable parts of the circuit can be grouped using the **Group** button **(6)** into a single component. Such compound gates can be ungrouped using the **Ungroup** button **(5)**. There are also predefined component gates available for the user – they can be selected from the **Select Composite Gate** menu **(7)**. All compound gates created by the user are also available in this menu.

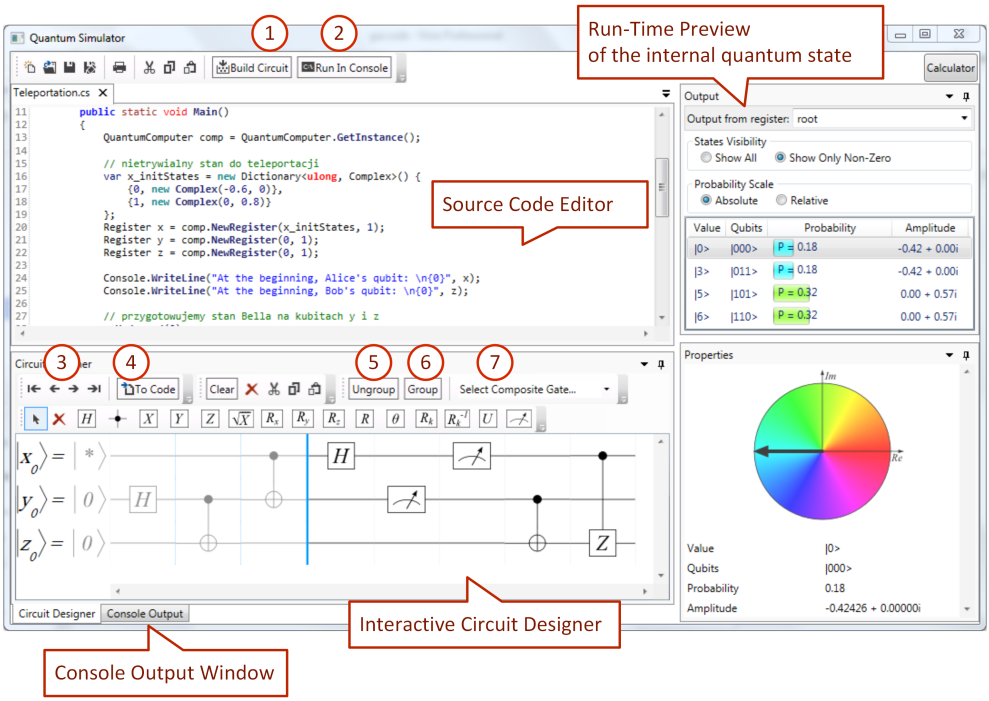

Figure 6. The Graphical User Interface of the QuIDE simulator. Main functionality, marked by numbers in the figure, is described in the text of the paper.

The circuit can be evaluated step-by-step using the **(3)** button group. During the stepping evaluation, the state of the quantum system is presented in the **Run-Time Preview** section. The user can preview the state of the whole quantum system, a specified register, or a range of qubits.

The **Properties** section is used to display detailed information about the selected quantum gate or the quantum state from the **Run-Time Preview**.

10. Evaluation of functionality and performance

QuIDE was tested against its functionality and performance. In this section, we present the results of these tests. We begin with comparing the functions of QuIDE with the existing simulators in Section 10.1. The performance of the simulator is discussed in Section 10.2.

10.1. Results of Functionality Evaluation

The Functionality Evaluation consisted in comparing the functions supported by QuIDE with the functionality of existing simulators. Table 2 presents the comparison

in terms of the most important features described in the Requirements Specification, introduced in Section 5. In the table, points 1 and 2 show the methods for designing a quantum circuit. Criteria 3 and 4 check whether it is possible to switch between graphical circuit representation and a source code. Point 5 stands for the possibility to preview an internal quantum state during the simulation. Feature 6 enable users to execute the simulation in a stepping mode. Criterion 7 shows whether users can build reusable computation blocks (subroutines). The last point stands for the possibility to build an algorithm with both classical and quantum operations.

Table 2

Results of the functionality evaluation.

	QuIDE	QCAD200 [58]	jQuantum [57]	SimQubit [27]	qMIPS [51]	Qubit101 [52]	Zeno [12]	Cove [39]	libquantum [11]	CHP [1]	LanQ [34]	Q++ [13]	QCL [38]	QuIDDPro [54]	Javascript QC Simulator [60]	Online QC Simulator [31]
1. Source code edition	Yes	No	No	No	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No
2. Graphical circuit designing	Yes	Yes	Yes	Yes	No	Yes	Yes	No	No	No	No	No	No	No	Yes	Yes
3. Source code ⇒ Graphical circuit	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
4. Graphical circuit ⇒ Source code	Yes	No	No	No	No	No	No	No	No	No	No	No	No	No	No	No
5. Run-time preview	Yes	No	Yes	Yes	Yes	No	Yes	[a]	[a]	No	[a]	[a]	Yes	Yes	No	No
6. Stepping mode	Yes	No	Yes	No	Yes	Yes	Yes	[b]	[b]	No	No	[b]	Yes	Yes	No	No
7. Reusable subroutines	Yes	No	No	Yes	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes	No	Yes	No
8. Mixing classical & quantum	Yes	No	No	No	Yes	No	No	Yes	Yes	No	Yes	Yes	Yes	Yes	No	No

[a] Implementable – the user could insert the printing command into the source code.

[b] Implementable – the user could write a program able to perform computations interactively, step by step.

The results of the evaluation explain why the QuIDE simulator was created. Although there were existing simulators, they lacked many useful features. We have successfully implemented these features in QuIDE.

10.2. Performance results

In the first performance test, we checked what is the maximum number of qubits that can be simulated in the best and worst cases. The worst case concerns the situation when the simulator has to use the maximum amount of RAM. The results are presented in Table 3.

In this comparison, QuIDE has almost the best results. Only two simulators perform better – `CHP` and `libquantum`. `CHP` has the best result because it is very simplified simulator, able to model only a few elementary quantum operations. It does not allow us to build an arbitrary quantum circuit. `libquantum` is a powerful library written in pure C and makes a use of MPICH functions. QuIDE used as a standalone library, is only slightly worse than `libquantum`. The full GUI-based QuIDE is a little bit worse; however, it surpasses other GUI simulators such as `jQuantum` or `QCAD200`.

Table 3
Maximum number of qubits that can be simulated in the best and worst cases by the quantum computer simulators.

Simulator Name	Maximum number of qubits simulated in the best case	Maximum number of qubits simulated in the worst case
QuIDE	no limits	23
QuIDE.dll	no limits	26
QCAD200	no limits	15
jQuantum	22	22
SimQubit	24	24
qMIPS	32	16
Qubit101	16	16
Zeno	24	20
Cove	13	12
libquantum	no limits	27
CHP	over 10 000	over 10 000
LanQ	7	7
Q++	25	25
QCL	64	25
QuIDDPro	no limits	n/d
Javascript QC Simulator	9	9
Online QC Simulator	9	9

In the second performance test, we precisely measure the amount of RAM used in the worst case. For this test, we have chosen two existing simulation tools – one graphical simulator (`jQuantum`) and one simulation library (`libquantum`). They are compared with QuIDE in both GUI and library modes. The results are presented in Figure 7. The chart shows the relationship between the amount of RAM and the number of qubits.

As we can see in Figure 7, memory consumption grows exponentially for all simulators. These results confirm the theoretical complexity estimated in Section 8. All presented simulators have $O(2^n)$ complexity, which is the best case of the exponential complexity possible for quantum computer simulators. QuIDE in standalone library mode performs almost as well as `libquantum`. In GUI mode, QuIDE needs the most memory; however, it can simulate larger systems than `jQuantum` and offers more functionality.

Figure 8 presents the execution time of QuIDE simulating Grover's Fast Database Search Algorithm [19]. We measured the execution time of a single iteration of the algorithm. The figure shows the relationship between the simulation time and the number of used qubits.

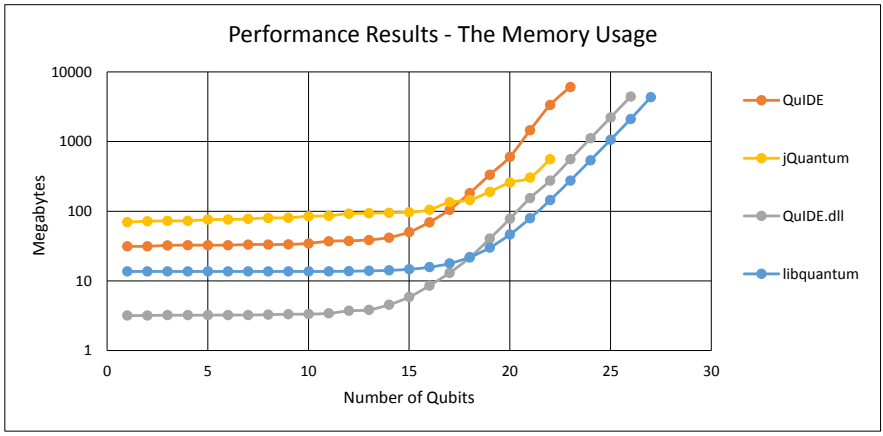

Figure 7. The amount of memory used by the simulators for the worst case – when the qubits are in a flat superposition state. The results are shown on a logarithmic scale. The errors are negligible. For a number of qubits higher than 20, the lines on the chart can be approximated by a common function $y = \alpha e^{0.68x}$, where y is the amount of memory used and x is the number of qubits. The factor α is a constant, different for each of the simulators. Approximately, this function can be expressed also as $y = \alpha 2^x$. Thus, the space complexity for simulating n-qubit quantum system is confirmed to be $O(2^n)$.

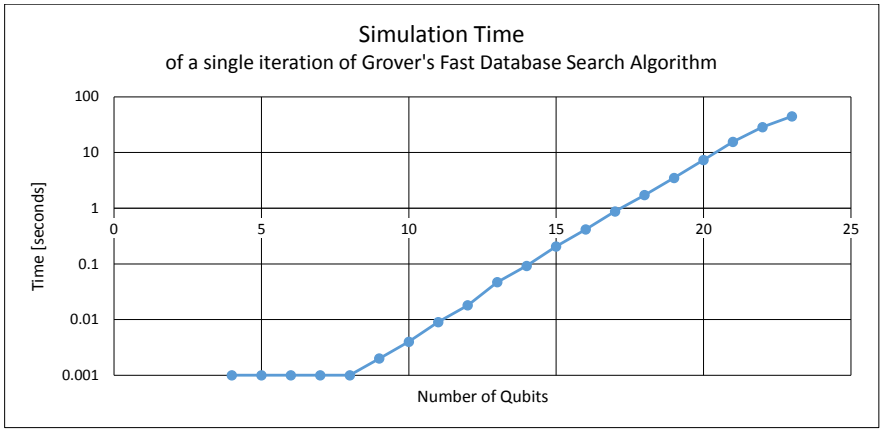

Figure 8. The simulation time of a single iteration of Grover's Fast Database Search Algorithm [19]. The results are achieved by the QuIDE simulator. The points on the chart form a line, which can be approximated by the function $y = \alpha 2^x$. The errors are negligible. The expected $O(2^n)$ time complexity is thus confirmed.

Simulation time grows exponentially with the number of qubits. It confirms the complexity estimated in Section 8. For small amounts of qubits, the simulation time is close to zero. It is convenient for educational and presentational purposes. It confirms the achievement of our goal – QuIDE is fast enough to be used for teaching and learning.

QuIDE was the main simulation software in the course 'Mathematics for Future Computer Systems' for Master of Science in Computer Science at AGH University of Science and Technology [2]. In past years, students were working on the `libquantum` library. Apart from basic quantum gates, we have also implemented the complex subroutines and used them in the following quantum algorithms : Deutsch Problem [14], Bernstein-Vazirani Problem [7] Grover's Fast Database Search Algorithm [19], Shor's Prime Factorization Algorithm [44], Quantum Teleportation [6], Quantum Dense Coding [35].

The usability of QuIDE was evaluated and compared to `libquantum` using the System Usability Scale (SUS) surveys [10]. Students were also asked to give points to the two additional statements *I think that the system make it easier to understand the quantum computations* and *I think that the system is a good tool to design and analyze quantum algorithms*. The obtained average usability score for the simulator was 75 points (out of 100 possible). The obtained average usability score was better then this for libquantum, which obtained 50 points. The average was calculated from answers from about 100 students.

11. Conclusions and future work

In this paper, we present the simulation environment supporting convenient and efficient learning and building quantum computer algorithms. In this study, we evaluated existing quantum computer techniques and their most-popular implementations. As a result of the evaluation, we proposed and built a novel quantum computer simulation platform. The main features of the proposed environment regarding convenient learning are: support for building quantum algorithms via source code and graphically with conversion between both ways of algorithm presentation, step-by-step execution with the step back option and preview of the actual internal quantum state. Regarding the efficiency criterion, we evaluated the performance and functionality of our solution and compared it to existing simulators. QuIDE was used in university classes where students were asked to grade its usability. The results of this work demonstrate that we developed a usable tool, which has been successfully used for educational purposes. The performance and functionality evaluation proved that this is one of the best currently-available tools of this type.

An interesting approach would be to transfer the presented simulator into a Web application distributed in the Software as a Service infrastructure, and the simulation library could be optimized for two targets: local execution on the user's PC and for execution on clouds or grids.

Acknowledgements

The research presented in this paper has been partially supported by AGH grant no 11.11.230.124. It was also partially supported by the European Union within the European Regional Development Fund program no. POIG.02.03.00-12-137/13 as part of the PLGrid Core project.

References

[1] Aaronson S., Gottesman D.: Improved simulation of stabilizer circuits. *Phys. Rev. A*, vol. 70, p. 052328, 2004. http://dx.doi.org/10.1103/PhysRevA.70.052328.

[2] AGH: Syllabus – module Matematyka w informatyce przyszłości, 2012. http://syllabuskrk.agh.edu.pl/2014-2015/en/magnesite/study_plans/stacjonarne-informatyka-systemy-rozproszone-i-sieci-komputerowe/module/iin-2-101-sr-s-matematyka-w-informatyce-przyszlosci.

[3] Barenco A., Bennett C. H., Cleve R., DiVincenzo D. P., Margolus N., Shor P., Sleator T., Smolin J. A., Weinfurter H.: Elementary gates for quantum computation. *Phys. Rev. A*, vol. 52, pp. 3457–3467, 1995. http://dx.doi.org/10.1103/PhysRevA.52.3457.

[4] Belkner P.: Eqcs-0.0.8, 2012. http://home.snafu.de/pbelkner/eqcs/. Accessed May 10, 2014.

[5] Bennett C. H., Brassard G.: Quantum cryptography: Public key distribution and coin tossing. *Proceedings of IEEE International Conference on Computers, Systems, and Signal Processing*, p. 175. India, 1984.

[6] Bennett C. H., Brassard G., Crépeau C., Jozsa R., Peres A., Wootters W. K.: Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Phys. Rev. Lett.*, vol. 70, pp. 1895–1899, 1993. http://dx.doi.org/10.1103/PhysRevLett.70.1895.

[7] Bernstein E., Vazirani U.: Quantum Complexity Theory. In: *SIAM J. Comput.*, vol. 26(5), pp. 1411–1473, 1997. ISSN 0097-5397. http://dx.doi.org/10.1137/S0097539796300921.

[8] Black P. E., Lane A. W.: Modeling Quantum Information Systems. *Proc. Quantum Information and Computation II, Defense and Security Symposium*. 2004.

[9] Bouvarel B., Oudin O., Vallier L.: Simulateur de Cryptographie Quantique, 2009. http://sourceforge.net/projects/simu-quantique/. Accessed Sep 18, 2014.

[10] Brooke J.: SUS – A quick and dirty usability scale. In: P. W. Jordan, B. Thomas, B. A. Weerdmeester, A. L. McClelland, eds, *Usability Evaluation in Industry*. Taylor and Francis, 1996.

[11] Butscher B., Weimer H.: *Simulation eines Quantencomputers*, 2003. http://www.libquantum.de/files/libquantum.pdf.

[12] Cabral G. E.: Zeno – Quantum Circuit Simulator, 2006. http://dsc.ufcg.edu.br/~iquanta/zeno/index_en.html. Accessed Sep 18, 2014.

[13] Cybernet: Q++. http://sourceforge.net/projects/qplusplus/. Accessed May 10, 2014.

[14] Deutsch D.: Quantum theory, the Church-Turing principle and the universal quantum computer. *Royal Society of London Proceedings Series A*, vol. 400, pp. 97–117, 1985. http://dx.doi.org/10.1098/rspa.1985.0070.

[15] Dixon L., Duncan R., Kissinger A.: Quantomatic, 2011. https://sites.google.com/site/quantomatic/. Accessed Sep 18, 2014.

[16] Ekert A., Knight P. L.: Entangled quantum systems and the Schmidt decomposition. *American Journal of Physics*, vol. 63(5), pp. 415–423, 1995. http://dx.doi.org/http://dx.doi.org/10.1119/1.17904.

[17] Feynman R., Shor P. W.: Simulating Physics with Computers. *SIAM Journal on Computing*, vol. 26, pp. 1484–1509, 1982.

[18] Greve D.: QDD: A Quantum Computer Emulation Library. http://thegreves.com/david/QDD/qdd.html. Accessed May 10, 2014.

[19] Grover L. K.: A fast quantum mechanical algorithm for database search. *STOC'96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pp. 212–219. ACM Press, New York, NY, USA, 1996. http://citeseer.ist.psu.edu/175549.html.

[20] Johansson J., Nation P., Nori F.: QuTiP 2: A Python framework for the dynamics of open quantum systems. *Computer Physics Communications*, vol. 184(4), pp. 1234–1240, 2013, ISSN 0010-4655. http://dx.doi.org/10.1016/j.cpc.2012.11.019.

[21] Johnson M. W., Amin M. H. S., et al.: Quantum annealing with manufactured spins. In: *Nature*, vol. 473(7346), pp. 194–198, 2011, ISSN 0028-0836. http://dx.doi.org/10.1038/nature10012.

[22] Jozsa R., Linden N.: On the role of entanglement in quantum-computational speed-up. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, vol. 459(2036), pp. 2011–2032, 2003. http://dx.doi.org/10.1098/rspa.2002.1097.

[23] Kempe J.: Quantum random walks – an introductory overview. *Contemporary Physics*, vol. 44(4), pp. 302–327, 2003. Lanl-arXive quant-ph/0303081.

[24] Lanting T., Przybysz A. J., et al.: Entanglement in a Quantum Annealing Processor. *Phys. Rev. X*, vol. 4, p. 021041, 2014. http://dx.doi.org/10.1103/PhysRevX.4.021041.

[25] Lanyon B. P., Weinhold T. J., Langford N. K., Barbieri M., James D. F. V., Gilchrist A., White A. G.: Experimental Demonstration of a Compiled Version of Shor's Algorithm with Quantum Entanglement. *Phys. Rev. Lett.*, vol. 99, p. 250505, 2007. http://dx.doi.org/10.1103/PhysRevLett.99.250505.

[26] Leforestier C., Bisseling R., et al.: A comparison of different propagation schemes for the time dependent Schrödinger equation. *Journal of Computational Physics*, vol. 94(1), pp. 59–80, 1991.

[27] Lomont C.: SimQubit, Cybernet's quantum circuit simulator, 2005. http://sourceforge.net/projects/simqubit/. Accessed Sep 18, 2014.

[28] Mariantoni M., Wang H., et al.: Implementing the Quantum von Neumann Architecture with Superconducting Circuits. *Science*, vol. 334(6052), pp. 61–65, 2011. http://dx.doi.org/10.1126/science.1208517.

[29] Markov I., Shi Y.: Simulating Quantum Computation by Contracting Tensor Networks. *SIAM Journal on Computing*, vol. 38(3), pp. 963–981, 2008. http://dx.doi.org/10.1137/050644756.

[30] Martin-Lopez E., Laing A., Lawson T., Alvarez R., Zhou X. Q., O'Brien J.: Experimental realisation of Shor's quantum factoring algorithm using qubit recycling. *Lasers and Electro-Optics Europe (CLEO EUROPE/IQEC), 2013 Conference on and International Quantum Electronics Conference*, pp. 1–1. 2013. http://dx.doi.org/10.1109/CLEOE-IQEC.2013.6801701.

[31] Mavridi P., Tsimpouris C.: Demo: Quantum Computer Simulator, 2010. http://www.wcl.ece.upatras.gr/ai/resources/demo-quantum-simulation. Accessed Sep 18, 2014.

[32] Mermin N.: *Quantum Computer Science: An Introduction*. Cambridge University Press, 2007, ISBN 9781139466806. http://books.google.pl/books?id=q2S9APxFdUQC.

[33] Miszczak J. A.: *Probabilistic aspects of quantum programming*. Ph.D. thesis, Institute of Theoretical and Applied Informatics, Polish Academy of Sciences, 2008.

[34] Mlnařík H.: *Quantum Programming Language LanQ*. Ph.D. thesis, Masaryk University, 2007.

[35] Nielsen M. A., Chuang I. L.: *Quantum Computation and Quantum Information*. Cambridge University Press, 2000, ISBN 521635039. http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0521635039.

[36] Nowotniak R.: *Wykorzystanie metod ewolucyjnych sztucznej inteligencji w projektowaniu algorytmów kwantowych*. Master's thesis, Politechnika Łódzka, 2008.

[37] Obenland K. M., Despain A. M.: A Parallel Quantum Computer Simulator, 1997. http://arxiv.org/abs/quant-ph/9804039.

[38] Ömer B.: *Quantum Programming in QCL*. Master's thesis, Technical University of Vienna, 2000.

[39] Purkeypile M. D.: *Cove: A Practical Quantum Computer Programming Framework*. Ph.D. thesis, Colorado Technical University, 2009. http://arxiv.org/abs/0911.2423.

[40] Raedt H. D., Michielsen K.: Computational Methods for Simulating Quantum Computers. M. Rieth, W. Schommers, eds., *Handbook of Theoretical and Computational Nanotechnology*, vol. 3: Quantum and molecular computing, quantum simulations, chap. 1, p. 248. American Scientific Publisher, 2006. http://arxiv.org/abs/quant-ph/0406210.

[41] Raedt K. D., Michielsen K., Raedt H. D., Trieu B., Arnold G., Richter M., Lippert T., Watanabe H., Ito N.: Massively parallel quantum computer simulator. *Computer Physics Communications*, vol. 176(2), pp. 121–136, 2007, ISSN 0010-4655. http://dx.doi.org/http://dx.doi.org/10.1016/j.cpc.2006.08.007.

[42] Samad Abdel W., Ghandour R., Hajj Chéhade M. N.: Memory Efficient Quantum Circuit Simulator Based on Linked List Architecture, 2005. http://arxiv.org/abs/quant-ph/0511079.

[43] Shary S., Cahay M.: Bloch Sphere Simulation, 2012. http://www.ece.uc.edu/~mcahay/blochsphere/. Accessed Sep 18, 2014.

[44] Shor P. W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.*, vol. 26(5), pp. 1484–1509, 1997, ISSN 0097-5397. http://dx.doi.org/10.1137/S0097539795293172.

[45] Skibinski J.: Haskell Simulator of Quantum Computer. http://web.archive.org/web/20010803034527/http://www.numeric-quest.com/haskell/QuantumComputer.html. Accessed May 10, 2014.

[46] Smith J.: WPF Apps With The Model-View-ViewModel Design Pattern. *MSDN Magazine*, 2009.

[47] Suzuki M.: Decomposition formulas of exponential operators and Lie exponentials with some applications to quantum mechanics and statistical physics. *Journal of Mathematical Physics*, vol. 26(4), pp. 601–612, 1985. http://dx.doi.org/http://dx.doi.org/10.1063/1.526596.

[48] Tonder van A.: A Lambda Calculus for Quantum Computation. *SIAM Journal on Computing*, vol. 33(5), pp. 1109–1135, 2004. http://dx.doi.org/10.1137/S0097539703432165.

[49] Tucci R.: Graphical computer method for analyzing quantum systems, 1998. http://www.google.com/patents/US5787236. US Patent 5,787,236.

[50] Vaccaro J.: Quantum computer simulator, 2009. http://www.ict.griffith.edu.au/joan/qucomp/qucompApplet.html. Accessed Sep 18, 2014.

[51] Vázquez J.M.C.d.P.: *qMIPS Quantum processor simulator*, 2013. http://institucional.us.es/qmipsmaster/qMIPS/documentation.php. Accessed September 18, 2014.

[52] Vázquez J.M.C.d.P.: *Qubit101 Quantum circuit simulator*, 2013. http://institucional.us.es/qmipsmaster/Qubit101/documentation.php. Accessed September 18, 2014.

[53] Viamontes G. F., Markov I. L., Hayes J. P.: Improving Gate-Level Simulation of Quantum Circuits. *Quantum Information Processing*, vol. 2(5), pp. 347–380, 2003, ISSN 1570-0755. http://dx.doi.org/10.1023/B:QINP.0000022725.70000.4a.

[54] Viamontes G. F., Markov I. L., Hayes J. P.: *Quantum Circuit Simulation*. Springer, 2009.

[55] Vidal G.: Efficient Classical Simulation of Slightly Entangled Quantum Computations. *Phys. Rev. Lett.*, vol. 91, p. 147902, 2003. http://dx.doi.org/10.1103/PhysRevLett.91.147902.

[56] Vries de A.: math IT – Mathematics and Information Technology. http://www.math-it.org/. Accessed May 10, 2014.

[57] Vries de A.: *jQuantum: A Quantum Computer Simulator*, 2010. http://jquantum.sourceforge.net/jQuantum.pdf. Accessed May 10, 2014.

[58] Watanabe H., Suzuki M., Yamazaki J.: QCAD, GUI environment for Quantum Computer Simulator, 2011. http://qcad.sourceforge.jp/. Accessed Sep 18, 2014.

[59] Wecker D., Svore K. M.: LIQUid: A Software Design Architecture and Domain-Specific Language for Quantum Computing, 2014. http://research.microsoft.com/apps/pubs/default.aspx?id=209634.

[60] Wybiral D.: Quantum Circuit Simulator, 2012. http://www.davyw.com/quantum/. Accessed Sep 18, 2014.

Affiliations

Joanna Patrzyk
AGH University of Science and Technology, Department of Computer Science, Krakow, Poland, jpatrzyk@quide.eu

Bartłomiej Patrzyk
AGH University of Science and Technology, Department of Computer Science, Krakow, Poland, bpatrzyk@quide.eu

Katarzyna Rycerz
AGH University of Science and Technology, Department of Computer Science, Krakow, Poland, ACC Cyfronet AGH, Krakow, Poland, kzajac@agh.edu.pl

Marian Bubak
AGH University of Science and Technology, Department of Computer Science, Krakow, Poland, ACC Cyfronet AGH, Krakow, Poland, University of Amsterdam, Institute for Informatics, Faculty of Science, Science Park 904, 1098XH Amsterdam, The Netherlands, bubak@agh.edu.pl

Received: 28.11.2014
Revised: 13.02.2015
Accepted: 15.02.2015