Graphical and programming support for simulations of quantum computations

joanna patrzyk

Supervisor:

Dr inż. Katarzyna Rycerz

Kraków 2014
OŚWIADCZENIE AUTORA PRACY

Oświadczam, świadoma odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałam osobiście i samodzielnie, i nie korzystałam ze źródeł innych niż wymienione w pracy.

...................................

PODPIS
Praca Magisterska

Graficzne i programowe wsparcie dla symulacji obliczeń kwantowych

Joanna Patrzyk

Opiekun:

DR INŻ. KATARZYNA RYcerZ

Kraków 2014
Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr Katarzyna Rycerz, for the continuous support of my M.Sc. study, for her patience, motivation, enthusiasm, and immense knowledge. Her guidance helped me a lot during my research and writing of this thesis.

I would also like to thank Dr Marian Bubak, for his suggestions and valuable advices, and for provision of the materials used in this study.

I would also thank Dr Włodzimierz Funika and Dr Maciej Malawski for their support and constructive remarks concerning the QuIDE simulator.

My special thank goes to Bartłomiej Patrzyk for the encouragement, suggestions, ideas and a great support during this study.
Abstract

The field of Quantum Computing is recently rapidly developing. However before it transits from the theory into practical solutions, there is a need for simulating the quantum computations, in order to analyze them and investigate their possible applications.

Today, there are many software tools which simulate quantum computers. One of the part of this thesis is the detailed and comparative review of these simulators. Since it showed many limits of them, we proposed a new tool aimed at overcoming these limits.

In this thesis we proposed, designed and implemented a new quantum computer simulator — the Quantum Integrated Development Environment (QuIDE). Its novel interface, by integrating the features of different types of existing simulators, enables performing both simple, presentational simulations and the advanced algorithms in the same, usable way.

The QuIDE simulator was evaluated in terms of its functionality and performance. We compared it thoroughly with the other such simulators. Then, it was used as a main simulation tool on an academic course concerning quantum computations. During that, its usability was also verified.

This thesis is organized as follows: Chapter 1 explains the motivation and the discussed problems, and presents the goals of the thesis. Chapter 2 introduces the main concepts of Quantum Information and Computation Theory. In Chapter 3 we review the existing quantum computer simulators and classify them in terms of their external interfaces. In Chapter 4 we present an overview of the internal, algorithmic techniques for simulating quantum computations. Basing on these two reviews, in Chapter 5 we specify requirements for a new simulator and outline our proposal for this tool. In Chapter 6, we show how it was actually implemented — we describe the architecture of the QuIDE simulator. In Chapter 7 we describe algorithms and data structures used in the core simulation module. Chapter 8 demonstrates the first result of this work — the features and example applications of QuIDE. Chapter 9 show the results of the functionality, usability and performance evaluation of QuIDE and other tools being compared. In Chapter 10 we summarize the thesis and outline the future directions of the study.
1 Introduction 1
  1.1 Motivation ......................................................... 1
  1.2 Problem Outline ............................................... 3
  1.3 Related Work ............................................................... 4
  1.4 Goals of This Thesis .............................................. 5
  1.5 Contribution of the Other Authors ................................. 6
  1.6 Thesis Outline ..................................................... 6
2 Introduction to Quantum Computing 9
  2.1 Preface .................................................................. 9
  2.2 Main Concepts ............................................................... 10
    2.2.1 Quantum Bits .................................................. 10
    2.2.2 Operations on Quantum Bits .................................. 11
    2.2.3 Quantum-Mechanical Phenomena in Quantum Computation .. 16
  2.3 Summary ............................................................... 17
3 Overview of Quantum Computer Simulators 19
  3.1 State of the Art ............................................................. 19
    3.1.1 Simulation Libraries ........................................ 19
    3.1.2 Quantum Programming Languages ......................... 20
    3.1.3 Interpreters ..................................................... 21
    3.1.4 Graphical Simulators ......................................... 22
    3.1.5 Toolboxes for the Scientific Software ....................... 24
  3.2 Summary ............................................................... 24
4 Survey of Simulation Techniques 25
  4.1 Problems with Simulation of Quantum Processes ................. 25
  4.2 Survey of Simulation Algorithms and Data Structures .......... 26
    4.2.1 Numerical Linear Algebra Methods .......................... 26
    4.2.2 Qubit-wise Multiplication ................................... 27
    4.2.3 P-blocked State Representation ............................... 27
    4.2.4 Binary Decision Diagrams .................................... 28
    4.2.5 Hash Table State Representation .............................. 29
    4.2.6 Other Solutions .................................................. 30
  4.3 Summary ............................................................... 31
5 QuIDE — Motivation and Proposal 33
  5.1 Motivation ............................................................... 33
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1</td>
<td>Drawbacks of Existing Simulators</td>
<td>34</td>
</tr>
<tr>
<td>5.2</td>
<td>Users and Interfaces Specification</td>
<td>35</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Types of Users</td>
<td>35</td>
</tr>
<tr>
<td>5.2.2</td>
<td>User Interfaces</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>Requirements Specification</td>
<td>35</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Functional Requirements</td>
<td>36</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Nonfunctional Requirements</td>
<td>37</td>
</tr>
<tr>
<td>5.4</td>
<td>Solution Outline</td>
<td>38</td>
</tr>
<tr>
<td>5.5</td>
<td>QuIDE Components</td>
<td>39</td>
</tr>
<tr>
<td>5.5.1</td>
<td>QuIDE Simulation Library</td>
<td>39</td>
</tr>
<tr>
<td>5.5.2</td>
<td>QuIDE GUI</td>
<td>40</td>
</tr>
<tr>
<td>5.6</td>
<td>Summary</td>
<td>42</td>
</tr>
<tr>
<td>6</td>
<td>QuIDE Architecture</td>
<td>45</td>
</tr>
<tr>
<td>6.1</td>
<td>General Architectural and Technical Decisions</td>
<td>45</td>
</tr>
<tr>
<td>6.2</td>
<td>Application Layers — the MVVM Pattern</td>
<td>46</td>
</tr>
<tr>
<td>6.3</td>
<td>QuIDE Inner Components within the MVVM Layers</td>
<td>47</td>
</tr>
<tr>
<td>6.3.1</td>
<td>View and ViewModel Components</td>
<td>47</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Model Layer Components</td>
<td>49</td>
</tr>
<tr>
<td>6.4</td>
<td>Quantum Computation Model</td>
<td>51</td>
</tr>
<tr>
<td>6.5</td>
<td>Summary</td>
<td>53</td>
</tr>
<tr>
<td>7</td>
<td>Simulation Algorithms and Implementation Details</td>
<td>55</td>
</tr>
<tr>
<td>7.1</td>
<td>Data Structures</td>
<td>55</td>
</tr>
<tr>
<td>7.2</td>
<td>Implementation of the Quantum Operations</td>
<td>57</td>
</tr>
<tr>
<td>7.3</td>
<td>Complexity Analysis</td>
<td>62</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Space Complexity</td>
<td>62</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Time Complexity</td>
<td>63</td>
</tr>
<tr>
<td>7.4</td>
<td>Summary</td>
<td>63</td>
</tr>
<tr>
<td>8</td>
<td>QuIDE Features and Applications</td>
<td>65</td>
</tr>
<tr>
<td>8.1</td>
<td>Simulation API</td>
<td>65</td>
</tr>
<tr>
<td>8.2</td>
<td>Graphical User Interface</td>
<td>67</td>
</tr>
<tr>
<td>8.3</td>
<td>Implemented Quantum Algorithms and Subroutines</td>
<td>69</td>
</tr>
<tr>
<td>8.4</td>
<td>Example Application</td>
<td>70</td>
</tr>
<tr>
<td>8.5</td>
<td>Summary</td>
<td>71</td>
</tr>
<tr>
<td>9</td>
<td>Research Results</td>
<td>73</td>
</tr>
<tr>
<td>9.1</td>
<td>Functionality Evaluation</td>
<td>73</td>
</tr>
<tr>
<td>9.1.1</td>
<td>Comparison of the Quantum Computer Simulators</td>
<td>73</td>
</tr>
<tr>
<td>9.1.2</td>
<td>Results — A Comparison</td>
<td>74</td>
</tr>
<tr>
<td>9.2</td>
<td>System Usability Evaluation</td>
<td>81</td>
</tr>
<tr>
<td>9.2.1</td>
<td>QuIDE Implementation in the Quantum Computation Theory Course</td>
<td>81</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Evaluation Methodology</td>
<td>82</td>
</tr>
<tr>
<td>9.2.3</td>
<td>Results</td>
<td>82</td>
</tr>
<tr>
<td>9.3</td>
<td>Performance Tests</td>
<td>85</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Evaluation Methodology</td>
<td>85</td>
</tr>
</tbody>
</table>
## Contents

9.3.2 Results .......................................................... 86
9.4 Summary ......................................................... 90

10 Conclusion ......................................................... 91
  10.1 Goals Achieved .................................................. 91
  10.2 Results of the Study ............................................ 93
  10.3 Lessons Learned ................................................ 93
  10.4 Project Status .................................................. 94
  10.5 Future Directions .............................................. 95

A Further Comparison of Quantum Computer Simulators 97
B SUS Questionnaire for QuIDE 101
C Papers 103

List of Figures 105
List of Tables 107
Bibliography 109
Chapter 1

Introduction

This chapter describes the scope of this thesis and briefly summarizes the state of the art. It explains also the motivation and define the goals of the thesis. At the end, it outlines the content of the following chapters.

1.1 Motivation

In recent years a significant progress has been made in the field of constructing a quantum computer[1–4]. As it has been proven, this technology can offer an exponential speedup in comparison to classical computers[5]. At the same time, today’s possibilities of increasing computational power of standard hardware architectures become limited by physical and theoretical laws. They dictate for example minimal transistor size, maximal speed of electrical signal propagation or maximum speedup achievable by parallelizing the computations (the Amdahl’s law)[6].

What opportunities gives us then a quantum computer? In essence, it exploits the rules of quantum mechanics to perform computations. While conventional computers work on bits, the quantum ones use qubits. Qubit is the representation of a particle, e. g. photon, electron or ion. Classical bit can store at once only one value: 0 or 1. In contrast, a qubit stores linear combination of 0 and 1. These values represent certain property of the particle such as spin (1 - spin up, 0 - spin down) and the linear combination of them means that the particle has certain spin with some probability. To get one of possible values, we perform measurement. However, this operation does not give any information about the possibilities. What is more, after measurement the linear combination is destroyed and qubit stores only the result measured.

When multiple qubits are grouped into registers, the exponential growth of computational power shows up. For register containing $n$ qubits, all of possible values, from 0 to $2^n – 1$ are stored and can be processed simultaneously. Even when final measurement destroys such superposition of values, operating on quantum registers opens up an opportunity to solve today’s NP problems in polynomial time.
In computational complexity theory, NP, as well as P and PSPACE, are classes of
decision problems. P contains problems solvable in polynomial time. It means that with
increasing problem size \( n \) (\( n \) could be for example a number of elements in an array), the
amount of time needed to solve a problem grows polynomially — it is upper bounded by
\( Cn^k \), for constant real numbers \( C \) and \( k \). For NP problems the solution can be verified in polynomial
time, but the algorithm for finding a solution might require more than polynomial time. PSPACE describes all problems solvable in polynomial amount of space
(memory). All those classes refer to solving problems on classical computers. However,
recently the new classes for quantum computers have been proposed. One of them is
the BQP class — its name stands for "Bounded error Quantum Polynomial time". It is
a class of problems which can be solved by a quantum computer in polynomial time and
with bounded error probability. The figure 1.1 presents the suspected relations between
the BQP class and the previous, standard classes. As shown, the quantum computers
could solve many NP problems in polynomial time. Today, NP problems are common in
science and industry. The inability to solve them in reasonable time significantly slows
down the development of these fields. The quantum computers could bring a remedy
against many such issues.

The theory about quantum computations will be discussed in the next chapter. However,
some of the elements of quantum computation theory has already been implemented in
practice. Successfully performed experiments have confirmed theoretical results[2, 9, 10].
What is more, not only the basic operations have been executed, but also full algorithms.

While physicists are working on constructing quantum computer, it is now needed to
work out the algorithms that could be applied on it. They can be designed, analysed
and proven analytically. However, it will be worth to perform a simulation on a classical
computer to confirm results.
Chapter 1. Introduction

The main motivation of this thesis is the need for simulation of theoretical quantum computer. First of all, it is now necessary to prepare for this technology. We should learn about it and also develop algorithms that could exploit its capabilities. Secondly, the simulation will be very helpful even if we manage to construct the quantum computer. Today’s first prototypes are accessible only in few laboratories. The experiments are difficult and costly. As in other fields, widely available simulations greatly support research. Finally, quantum effects are very cumbersome in observing and analysing. Each measurement disturbs the state of the system. Processes run immediately and could not be paused. The attributes of elementary particles used as qubits make them difficult to localize and store. The method which enables observing, repeating and redesigning quantum processes in convenient and inexpensive way is highly desirable.

1.2 Problem Outline

Richard Feynman pointed out that quantum mechanical system cannot be effectively simulated by any classical system. He proposed that it could be effectively modeled only by another quantum mechanical system. As he showed, a classical Turing machine would experience an exponential slowdown when simulating quantum phenomena[5]. A quantum computer simulator running on a classical machine has to deal with this slowdown. It is however a nontrivial question, how to do it in the most efficient way.

A single quantum bit (qubit) exists in states 0 and 1 simultaneously. It is represented as a 2-element vector of complex numbers, which describes ‘to what extent’ the qubit is in each of these states. When considering the n-qubit register, $2^n$ complex numbers are needed to precisely describe its state, since it can hold values from 0 to $2^n - 1$. What is more, any operation on such register must operate on each of these values. As a result, a simulation of a quantum computation is exponential both in space and time.

This thesis focuses on the simulation of quantum computations, which are executed on conventional computers. Therefore, the term quantum computer simulator used in further parts of the thesis always refers to a computer software performing simulations of quantum computations. What is more, the simulation of quantum computations slightly differs from computer-aided simulation (or modeling) of the quantum processes. The latter does not concern the quantum computations and is not the subject of this thesis.

The variety of methods has been proposed in order to make quantum computer simulations doable in reasonable time and with efficient memory usage. The crucial difference between them is the data representation. The simplest approaches use typical matrices to represent operators, and vectors to describe states. The operations on quantum register are then a simple, but not optimal matrix multiplications. To simulate in this way an operation on n-qubit register, we need to multiply $2^n \times 2^n$ matrix by $2^n$-element vector, which gives $2^{2n} + 2^n$ complex numbers. For $n = 14$, and each complex number represented by two 64-bit real numbers, the memory usage exceeds 4GB. To overtake this problem, another data structures such as Bayesian nets, Binary Decision Trees or Hash Maps have been proposed [11–15].
Another important issue which must be considered when designing a quantum computer simulator is the interface that it should provide to outer world. It could be graphical or textual and can support batch or interactive mode. It may be a library for standard programming languages or an independent program. It could also work as interpreter or compiler. The format of an output has to be chosen as well - it could be a file, an interactive console output, a graphical chart or any combination of these. The choice must address the needs of expected users and could differ depending on which group is targeted: computer scientists, physicists, students or others interested in quantum computations.

1.3 Related Work

Till today, a number of quantum computer simulators has been developed [16–18]. Many of them are libraries extending standard programming languages. They provide functions simulating basic blocks of quantum computation process. By combining these elements together, users can build any algorithm. Such algorithms could divide into quantum part, including functions from the library, and the classical one, with mathematical operations, standard loops, and so forth. Such libraries are implemented in a number of popular programming languages, such as C, C++, Java, C# (.NET), Python, and even Haskell or Scheme [15, 19–27].

An alternative approach is represented by so-called quantum programming languages[28]. They are programming interfaces specially designed to best fit the expected quantum computer capabilities. Many of them are only theoretical, but some provides also interpreters which simulate computations [12, 29–33].

Next important group of simulators is formed by GUI-based applications [11, 34–45]. The great strength of this approach is the fact that graphical interface makes them simpler and more user-friendly. As a consequence, they could be used not only by computer scientist, but also by all interested in quantum computations. However, most of these simulators focus only on some simple part of this field — they do not provide methods for constructing complex algorithms. Moreover, they often suffer from badly designed GUI. Since they are mostly research projects, the GUI is probably not very important for their authors. However, providing such poor GUI, often worse than simple command line interface, seems unreasonable.

Quantum computations can be also simulated in programs such as MATLAB or Mathematica. There are many toolboxes which provide robust simulation methods [46, 47]. They are thus a kind of libraries for programming languages. However, their drawback is that they require an additional software — often costly.

Every simulator presents distinct solution to the problem of modeling quantum computations. They vary in performance, precision, capabilities, user interfaces and inner architecture. There is thus a strong need for comparative review of these simulators. Such work has been already done in 1999[17] and in 2004[18], however
since that time many new simulators have come out. One of the goals of this thesis is to review and summarize existing quantum computer simulators. On this basis a new solution is then proposed.

1.4 Goals of This Thesis

The main goal of this thesis is to propose a new quantum computer simulator. This goal results from an observation that there are important requirements for such software which are not fully accomplished by existing simulators.

What is more, the aim of this study is to implement proposed solution and evaluate it. To sum up, the work have to be split into several stages, realizing following sub-goals:

Requirements specification
First of all, it is needed to consider, what are the practical use cases of quantum computer simulator. Then it have to be determined, what main functions and features should such simulator provide.

Existing simulators review
Simulating quantum computation on classical computers is exponential both in space and time. Therefore, we need to investigate the present methods for dealing with these growing memory and time requirements. The existing quantum computer simulators should be reviewed and evaluated in relation to requirements specified in the previous stage.

Design and implementation of a quantum computer simulator
In this stage we describe a proposal of a new quantum computer simulator, which fulfill the specified requirements. It is expected to unify the advantages of existing simulators and to avoid their drawbacks. Moreover, we propose an innovative interface of the simulator.

The simulator have to provide a graphical user interface, which should allow to easily design, run and analyze quantum computations. It could be therefore used by people who do not know computer programming languages. Moreover, it have to be possible to translate the graphical model directly into a program code. This code could be then incorporated into a complex solution. This feature could be thus used by specialists, willing to simulate complex algorithms. The translation from the source code to a circuit model have to be supported as well. To sum up, the new simulator should provide both graphical and programming interfaces, which could be freely swapped at any time.

This stage includes also the implementation of the proposed simulator.

Functionality and performance evaluation
Our goal is to use the simulator to implement the most important quantum computation algorithms. Next, we have to verify its performance and capabilities. In these terms, the new simulator should also be compared to the other simulators.
Deployment and validation in an academic course

The goal of this stage is to validate the simulator on the academic course concerning quantum computation. It should be used to build, run and analyze quantum computation algorithms. The opinions of the users have be gathered and are expected to help in improving the simulator. It is needed to collect all useful ideas which can be implemented in the future. Additionally, this stage is expected to verify posed requirements. In particular, the usability of the simulator have to be evaluated.

1.5 Contribution of the Other Authors

This thesis covers the full development of the new quantum computer simulator — QuIDE. This study includes also the implementation of the example quantum circuits, which can be run on QuIDE. They include the most important quantum algorithms, such as Quantum Fourier Transform, Grover’s Database Search, Deutsch Problem or Quantum Teleportation [7, 48–51]. The implementation of the Shor’s Factorization Algorithm [52] is authored by Bartłomiej Patrzyk. In his M.Sc. thesis he presents the different optimization variants of this algorithm and implements two of them on the QuIDE simulator [53].

1.6 Thesis Outline

In Chapter 2 we introduce the key concepts of the Quantum Information and Computation Theory. We explain thereby the basic theoretical issues which the subsequent chapters refer to.

Chapter 3 contains a detailed review of the existing quantum computer simulation software. They are classified in terms of the types of external interfaces, which they provide to users. These types are compared and summarized.

In Chapter 4 we look deeper at the quantum computer simulation methodologies. We investigate the data structures used for representing quantum computations, and the algorithms operating on these data structures.

In Chapter 5 we explain the idea of the new simulator. We specify the functional and nonfunctional requirements. Then, we explain our proposal for this tool and describe its components and features.

Chapter 6 describes the details of the architectural design of QuIDE. We explain main design decisions and present the components of QuIDE and the interactions between them.

In Chapter 7 we describe the implementation of the core simulation library of QuIDE. We present the data structures used to represent quantum computations and explain the algorithms for calculating the simulation results. We also estimate the time and memory complexity of the chosen simulation technique.
Chapter 8 briefly demonstrates the capabilities of the new simulator and its user interface. We also list the quantum algorithms implemented on QuIDE. Also, an example application of this tool is demonstrated.

In Chapter 9 we describe the methodology of the functionality, usability and performance evaluation of the new simulator. Next, we present and discuss their results. All these evaluations concern also a set of other existing simulators, which are compared to QuIDE.

In Chapter 10 we summarize the thesis. We discuss the goals achieved and lessons learned. Finally, we show our proposals for the further work.

Appendix A contains a more detailed continuation of the functionality evaluation from Chapter 9. In Appendix B we enclose the usability survey questionnaire.

Appendix C contains the reference to the paper enclosed to this thesis.
Chapter 2

Introduction to Quantum Computing

This chapter introduces the most important terms of the Quantum Computation Theory. First of all, we explain what is a quantum bit (qubit), quantum gate, quantum measurement and quantum circuit. Next, we show what quantum phenomena are used in the Quantum Computation Theory — we explain what is quantum parallelism and entanglement, and show their most important implementations (the Bell’s states, the EPR paradox, the GHZ state). In the next section we present the most important algorithms and problems, which proves the great power of quantum computations. These are: Shor’s Factorization Algorithm, Grover’s Fast Database Search Algorithm, Quantum Teleportation and Deutsch’s Problem.

2.1 Preface

The idea that the elementary particles can be used for computations was firstly introduced in 1980 [7]. In 1982, Richard Feynman showed, that a nature of quantum-mechanical systems makes them impossible to be effectively simulated on a classical computer [5]. At the same time, he proposed a basic model for quantum computer that would be capable of such simulations. That showed, that the laws of Quantum Mechanics could be used for performing very powerful computations.

This chapter aims to briefly introduce all the theoretical terms used in the next parts of this thesis. Our goal was to describe them in a short and straightforward way. Thus, they are presented very briefly and do not follow mathematical formalism rules. Further information can be found in the referenced bibliography [7, 54–57].
2.2 Main Concepts

2.2.1 Quantum Bits

A bit is the basic unit of information in the classical computation and information theory. In Quantum Computation and Information Theory, this role is played by a quantum bit — a qubit. A bit can be in the state 0 or 1. A qubit — in the state $|0\rangle$ or $|1\rangle$ or even in their superposition:

$$|\varphi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

where $\alpha, \beta \in \mathbb{C}$ (2.1)

The notation ‘$|\cdot\rangle$’ is called the **Dirac notation** (or bra–ket notation) and is a standard notation for describing quantum states. The states $|0\rangle$ and $|1\rangle$ are called the (computational) **basis states**, as they are orthogonal, unit vectors in a two-dimensional complex space ($\mathbb{C}^2$). Thus, in this space they can be expressed as:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The coefficients of the linear combination — $\alpha$ and $\beta$ — are called the **amplitudes**. They must follow the normalization rule:

$$|\alpha|^2 + |\beta|^2 = 1$$

Thus, a vector $\varphi$ from (2.1) is a unit vector in $\mathbb{C}^2$. It is called the **state vector**.

The qubit can be viewed as a representation of an elementary particle such as photon, electron, ion or a cold atom. From this point of view, the basis states represents the two opposite values of particle’s polarization (photons), spin (electrons etc.) or other such a attribute. For example, the state $|0\rangle$ denotes the spin down ($\downarrow$), and $|1\rangle$ — spin up ($\uparrow$).

Multiple Qubits

Let us consider $n$ qubits in the states $|\varphi_1\rangle$, $|\varphi_2\rangle$, ... to $|\varphi_n\rangle$. The state $|\Phi\rangle$ of this $n$-qubit system can be expressed as a **tensor product** of these 1-qubit states:

$$|\Phi\rangle = |\varphi_1\rangle \otimes |\varphi_2\rangle \otimes \cdots \otimes |\varphi_n\rangle$$

This tensor product is computed as the Kronecker product of the state vectors. For example, the tensor product of two 1-qubit state vectors $|\varphi_1\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle$ and
|φ₂⟩ = β₀ |0⟩ + β₁ |1⟩ is:

|φ₂⟩ ⊗ |φ₂⟩ = \begin{bmatrix} α₀ \\ α₁ \end{bmatrix} \otimes \begin{bmatrix} β₀ \\ β₁ \end{bmatrix} = \begin{bmatrix} α₀β₀ \\ α₀β₁ \\ α₁β₀ \\ α₁β₁ \end{bmatrix}

This result can also be obtained by simply multiplying the sums:

|φ₂⟩ ⊗ |φ₂⟩ = (α₀ |0⟩ + α₁ |1⟩) ⊗ (β₀ |0⟩ + β₁ |1⟩) =

α₀ |0⟩ ⊗ β₀ |0⟩ + α₀ |0⟩ ⊗ β₁ |1⟩ + α₁ |1⟩ ⊗ β₀ |0⟩ + α₁ |1⟩ ⊗ β₁ |1⟩ =

α₀β₀ |00⟩ + α₀β₁ |01⟩ + α₁β₀ |10⟩ + α₁β₁ |11⟩

The computational basis of a single-qubit quantum system consists of the states |0⟩ and |1⟩. For 2 qubits, it is formed by four states: |00⟩, |01⟩, |10⟩ and |11⟩, which are all possible variations of tensor products of the states |0⟩ and |1⟩. They can be also represent as |0⟩ ⊗ |0⟩, |0⟩ ⊗ |1⟩, |1⟩ ⊗ |0⟩ and |1⟩ ⊗ |1⟩, |0⟩ |0⟩, |0⟩ |1⟩, |1⟩ |0⟩ and |1⟩ |1⟩ or |0⟩₂, |1⟩₂, |2⟩₂ and |3⟩₂.

While the state of two classical bits can be either 00, 01, 10 or 11, the state of two qubits is a superposition of the basis states |00⟩, |01⟩, |10⟩ and |11⟩.

In general, the state of n-qubit system is represented by a unit vector in the \( \mathbb{C}^{2^n} \) space:

\[
|φ⟩ = \sum_{j=0}^{2^n-1} α_j |j⟩_n = α₀ |0⟩_n + α₁ |1⟩_n + \cdots + α_{2^n-1} |2^n-1⟩_n \quad \text{where } α_j ∈ \mathbb{C}
\]

The normalization rule must be kept as well:

\[
\sum_{j=0}^{2^n-1} |α_j|^2 = 1
\]

For example, 4 qubits can be in the state |φ₁⟩ = \( \frac{3}{5} |4⟩_4 + \frac{4}{5} |9⟩_4 \), which can be expressed also as |φ₁⟩ = \( \frac{2}{5} |0100⟩ + \frac{4}{5} |1001⟩ \). It means, that these four qubits are in the states |0100⟩ and |1001⟩ simultaneously (with amplitudes \( \frac{2}{5} \) and \( \frac{4}{5} \) respectively). In contrary, four classical bits could be only in one of these states.

An n-qubit quantum system is also called the n-qubit quantum register.

### 2.2.2 Operations on Quantum Bits

The evolution of a quantum state can be expressed in the language of Quantum Computation Theory. Analogously to the classical computer, which is built from logic circuits and logic gates, the quantum computer is built from quantum circuits consisted of elementary quantum gates.
Quantum Gates

As stated in the second postulate of Quantum Mechanics, the evolution of a quantum state can be represented by a unitary operator which acts in the space of state vectors. In the Quantum Computation Theory language, any operation on a quantum system can be represented by a unitary matrix $U$ which operates on its state vector. The unitarity is the only one requirement which has to be met by quantum gates.

The matrix $U$ is unitary, when $UU^\dagger = U^\dagger U = I$. $U^\dagger$ is a Hermitian conjugate of $U$ and $U^\dagger = (\overline{U})^T = \overline{U^T}$, where $\overline{U}$ denotes a complex conjugate a matrix $U$ and $U^T$ denotes the matrix transposition.

A quantum gate which operates on a single qubit is thus represented by a $2 \times 2$ unitary matrix. Any such matrix can define a quantum gate. However, in Quantum Computation Theory, there is a set of commonly used 1-qubit quantum gates which already have their names and definitions. They are presented in Table 2.1.

After the application of quantum gate, the qubit changes its state from $|\varphi\rangle$ to $|\varphi'\rangle$ as in the equation (2.9).

For $U = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ and $|\varphi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$, $|\varphi'\rangle = U|\varphi\rangle = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} a\alpha + b\beta \\ c\alpha + d\beta \end{bmatrix}$ (2.9)

Multi-qubit Quantum Gates

While the $2 \times 2$ unitary matrix represents a 1-qubit quantum gate, the $2^n \times 2^n$ unitary matrix represents an $n$-qubit quantum gate, which acts on $n$-qubit quantum system.

The most important such a gate is the controlled Not gate (C-Not). It is represented by a unitary matrix:

$$C_{10} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$ (2.10)

C-Not acts on two qubits. One is called the control bit, and the other — the target bit. The notation $C_{ct}$ specifies the indexes of control ($c$) and target ($t$) bits, where 0 means the least significant bit.

If the control bit is set (in the state $|1\rangle$), C-Not acts as a negation on the target bit: it changes its state from $|0\rangle$ to $|1\rangle$ or from $|1\rangle$ to $|0\rangle$. If the control bit is in the state $|0\rangle$, application of C-Not has no effect.

A variation of C-Not gate is Toffoli gate — a negation with two control bits. There is also a theoretical, so called unbound Toffoli gate — a negation with any number of control bits. The Swap gate acts on two qubits by interchanging their states. The Fredkin gate is a Swap gate with additional control bit. These gates are presented in Table 2.2
Table 2.1: A set of 1-qubit quantum gates commonly used in Quantum Computation Theory.

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hadamard</td>
<td>H</td>
<td>$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 &amp; 1 \ 1 &amp; -1 \end{bmatrix}$</td>
</tr>
<tr>
<td>Pauli-X (Not)</td>
<td>X</td>
<td>$\begin{bmatrix} 0 &amp; 1 \ 1 &amp; 0 \end{bmatrix}$</td>
</tr>
<tr>
<td>Pauli-Y</td>
<td>Y</td>
<td>$\begin{bmatrix} 0 &amp; -i \ i &amp; 0 \end{bmatrix}$</td>
</tr>
<tr>
<td>Pauli-Z</td>
<td>Z</td>
<td>$\begin{bmatrix} 1 &amp; 0 \ 0 &amp; -1 \end{bmatrix}$</td>
</tr>
<tr>
<td>Phase</td>
<td>S</td>
<td>$\begin{bmatrix} 1 &amp; 0 \ 0 &amp; i \end{bmatrix}$</td>
</tr>
<tr>
<td>$\pi/8$</td>
<td>T</td>
<td>$\begin{bmatrix} 1 &amp; 0 \ 0 &amp; e^{i\pi/4} \end{bmatrix}$</td>
</tr>
<tr>
<td>Rotation X (about $x$ axis of the Bloch sphere by angle $\gamma$)</td>
<td>$R_x$</td>
<td>$\begin{bmatrix} \cos \frac{\gamma}{2} &amp; -i \sin \frac{\gamma}{2} \ -i \sin \frac{\gamma}{2} &amp; \cos \frac{\gamma}{2} \end{bmatrix}$</td>
</tr>
<tr>
<td>Rotation Y (about $y$ axis of the Bloch sphere by angle $\gamma$)</td>
<td>$R_y$</td>
<td>$\begin{bmatrix} \cos \frac{\gamma}{2} &amp; -\sin \frac{\gamma}{2} \ \sin \frac{\gamma}{2} &amp; \cos \frac{\gamma}{2} \end{bmatrix}$</td>
</tr>
<tr>
<td>Rotation Z (about $z$ axis of the Bloch sphere by angle $\gamma$)</td>
<td>$R_z$</td>
<td>$\begin{bmatrix} e^{-i\frac{\gamma}{2}} &amp; 0 \ 0 &amp; e^{i\frac{\gamma}{2}} \end{bmatrix}$</td>
</tr>
<tr>
<td>Phase Scale (by angle $\gamma$)</td>
<td>$\theta$</td>
<td>$\begin{bmatrix} e^{i\gamma} &amp; 0 \ 0 &amp; e^{i\gamma} \end{bmatrix}$</td>
</tr>
<tr>
<td>Phase Shift (by angle $\gamma$)</td>
<td>$R$</td>
<td>$\begin{bmatrix} 1 &amp; 0 \ 0 &amp; e^{i\gamma} \end{bmatrix}$</td>
</tr>
<tr>
<td>$R_k$ (a special phase shift used in Quantum Fourier Transform)</td>
<td>$R_k$</td>
<td>$\begin{bmatrix} 1 &amp; 0 \ 0 &amp; e^{i\pi/2^k} \end{bmatrix}$</td>
</tr>
<tr>
<td>Square Root of Not</td>
<td>$\sqrt{X}$</td>
<td>$\frac{1}{2} \begin{bmatrix} 1 - i &amp; 1 + i \ 1 + i &amp; 1 - i \end{bmatrix}$</td>
</tr>
</tbody>
</table>
Table 2.2: Multi-qubit quantum gates.

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Controlled Not</td>
<td><img src="image" alt="C-Not" /></td>
<td>( C_{10} = \begin{bmatrix} 1 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 1 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 1 \ 0 &amp; 0 &amp; 1 &amp; 0 \end{bmatrix} )</td>
</tr>
<tr>
<td>(C-Not)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swap</td>
<td><img src="image" alt="Swap" /></td>
<td>( S_{10} = S_{01} = \begin{bmatrix} 1 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 1 &amp; 0 \ 0 &amp; 1 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 1 \end{bmatrix} )</td>
</tr>
<tr>
<td>Toffoli</td>
<td><img src="image" alt="Toffoli" /></td>
<td>( T_{210} = T_{120} = \begin{bmatrix} 1 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 1 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 1 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 1 &amp; 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 0 &amp; 1 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 1 \ 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 1 &amp; 0 \end{bmatrix} )</td>
</tr>
<tr>
<td>Fredkin</td>
<td><img src="image" alt="Fredkin" /></td>
<td>( F_{210} = F_{201} = \begin{bmatrix} 1 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 1 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 1 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 1 &amp; 0 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 0 &amp; 1 &amp; 0 &amp; 0 &amp; 0 \ 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 0 &amp; 1 \end{bmatrix} )</td>
</tr>
</tbody>
</table>
Chapter 2. *Introduction to Quantum Computing*

Measurement of Quantum State

In order to extract information stored in qubits, we need to measure them. However, the measurement of a quantum state is a non-unitary, irreversible operation.

The result of measuring a single qubit in the state $|\varphi\rangle = \alpha |0\rangle + \beta |1\rangle$ could be either 0 or 1. The result can be completely random — we only know the probabilities of getting 0 or 1. The probability of obtaining 0 is $|\alpha|^2$, and getting 1 has the probability $|\beta|^2$. After measurement, if we get the result 0, the qubit changes its state to $|0\rangle$, and if we get the result 1, the qubit changes its state to $|1\rangle$.

We can also measure the state of $n$-qubit system. After the measurement, the system changes its state to one of the basis states $|j\rangle_n$ (where $j \in \{0, 1, 2, \ldots, 2^n - 1\}$) with its corresponding probability $|\alpha_j|^2$.

**Figure 2.1:** The symbol of a quantum measurement.

Quantum Circuits

A quantum computation is built of quantum gates acting on quantum bits. At the beginning the qubits could be initialized to pure basis states. At the end, or at any other point, the qubits can be measured. A conventional method for describing such computations is the quantum circuit representation. The example quantum circuit is shown in Figure 2.2.

$$\begin{align*}
|0\rangle & \rightarrow X \rightarrow H \\
|1\rangle & \rightarrow X
\end{align*}$$

**Figure 2.2:** An example quantum circuit. The top ring is the most significant bit.

This form of representation is very readable. The same computation as in Figure 2.2 can be also expressed as $C_{10}(H \otimes I)(X \otimes X)(|0\rangle \otimes |1\rangle)$. In this textual notation we put the gates symbols from right to left, because of the rules of matrix multiplication. In this example the C-Not gate has to be applied at the end, which is much more clear when we look at the circuit representation.

The tensor product of the matrices representing quantum gates is computed as the Kronecker product of these matrices. For example:

$$
(X \otimes X)(|0\rangle \otimes |1\rangle) = (X \otimes X)
$$

$$
= \left( \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \right) \left( \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = |1\rangle \otimes |0\rangle
$$
2.2.3 Quantum-Mechanical Phenomena in Quantum Computation

Quantum Entanglement

In classical computations, we can always determine the state of any subset of $n$-element group of bits (e.g. the state of two least significant bits of 1101 is 01). However, it is not always possible for qubits. This phenomenon is called the quantum entanglement.

A multi-qubit state is called the entangled state, if it cannot be expressed as the tensor product of 1-qubit states of each its qubits.

The example entangled states are the Bell states:

$$
|\Phi^+\rangle = \frac{1}{\sqrt{2}} (|00\rangle + |11\rangle) \\
|\Phi^-\rangle = \frac{1}{\sqrt{2}} (|00\rangle - |11\rangle) \\
|\Psi^+\rangle = \frac{1}{\sqrt{2}} (|01\rangle - |10\rangle) \\
|\Psi^-\rangle = \frac{1}{\sqrt{2}} (|01\rangle - |10\rangle) 
$$

Another common example is the Greenberger–Horne–Zeilinger (GHZ) state:

$$
|GHZ\rangle = \frac{1}{\sqrt{2}} \left( |0\rangle^\otimes M + |1\rangle^\otimes M \right), \quad \text{where } M > 2
$$

The simplest GHZ state, for $M = 3$, is $|GHZ\rangle = \frac{1}{\sqrt{2}} (|000\rangle + |111\rangle)$.

The quantum entanglement is a unique phenomenon of Quantum Mechanics. Entangled qubits cannot be described independently — they have to be treated as a whole. Performing an action on a single qubit from the entangled pair (for example measuring it) causes that the second qubit from this pair also reacts on this action. This strange behavior was observed in real experiments. What is more, the entangled particles (photons or atoms) behaved in the same manner even if they were geographically separated. Albert Einstein called this phenomenon the ‘spukhafte Fernwirkung’ (‘spooky action at a distance’). He also is the coauthor of the Einstein-Podolsky-Rosen (EPR) paradox — a theoretical experiment based on properties of entangled states.

Quantum Parallelism

Quantum Parallelism is a fundamental aspect of quantum computing, which is the main reason of the enormous computational power of quantum computers.

In conventional computers, parallel computing is performed by having several processors linked together, so that each processor performs one computation. The number
of computations which can be done simultaneously is thus equal to the number of processors.

In quantum computer, a single processor is able to perform multiple simultaneous computations on its own. It utilities the fact, that the quantum bits can exist in multiple states simultaneously — when their state is a superposition of many basis states (as shown in section 2.2.1).

The state of $n$-qubit quantum register can be a superposition of even $2^n$ basis states. In this case, a single quantum processor can operate on $2^n$ values simultaneously. As a result, the quantum computers are exponentially faster than classical computers.

However, exploiting this computational power is not as easy as it seems to be. If a collection of qubits has a definite but unknown state, there is in general no way to find out what that state is. Measuring that state destroys the superposition and leaves qubits in one of computational basis states. Nevertheless, there are methods (e.g. Quantum Fourier Transform [7]) for extracting information from such an unknown state and exploiting the power of quantum computers. However, their strange nature causes that the quantum computational process is completely different from any classical method.

2.3 Summary

In this chapter we introduced the key aspects of quantum computing. We explained a notions of quantum bit, quantum gate, quantum measurement and quantum circuit. We show how they are represented mathematically and how this representation is used for model the quantum computational processes. Next, we outlined the unusual properties of quantum systems: the quantum entanglement and quantum parallelism phenomena.

This chapter is a brief theoretical introduction to the issues which are described in the next chapters. It should be sufficient for understanding all of them. Nevertheless, it presents only the most basic elements of the Quantum Information and Computation Theory, which is very rich and rapidly developing field.

To sum up, utilizing the laws of Quantum Mechanics for performing computations gives us access to computational power which outclasses the computational capabilities of any of today technologies. However, due to their strange nature, they have to be used in a way totally different from the classical methods. Thus, there is a need for investigating the capabilities of quantum computers. It can be done by simulating them. In the next two chapters we present, how it is realized in existing solutions. Then, we propose our solution for a new quantum computer simulator.
Chapter 3

Overview of Quantum Computer Simulators

In this chapter we show, what kind of software are the quantum computer simulators and we describe their main types. The goal of this part of thesis is to present the State-of-the-Art in the field of quantum computing simulations.

The chapter presents a review of the existing quantum computer simulators. They are classified in terms of the external interfaces that they provide. The study from this chapter eventuates in a detailed comparison of these simulators, which findings are presented in Chapter 9 and Appendix A.

3.1 State of the Art

There are a vast amount of a software programs which can simulate quantum computations. They can be classified in terms of the interface which they provide to users. The simulators from these classes has different advantages and disadvantages. Thus, they are suitable for different purposes. In this chapter we describe the benefits and drawbacks of each class of simulators. This review shows, what are the most important features of such tools and why they are necessary.

3.1.1 Simulation Libraries

This is a group of simulators, which are a libraries for one of the programming languages. They expose an Application Programming Interface (API) which enables performing the simulations. Usually, they include functions to allocate and deallocate quantum bits or registers, and to operate on them. In order to perform a simulation, the user have to write a source code, where the functions from the library’s API are called.

The advantages of such approach are:

- They enable to mix quantum and classical computations.
• All of the complex syntax statements of the programming language are available.

• User can take advantage of the code reuse by building subroutines and make a use of other convenient features offered by the programming language in which the simulation program is written.

• By writing a source code, the user gain full control over the simulation process.

• Users can take advantage of other libraries accessible for that programming language.

On the other hand, these simulators have following drawbacks:

• They are tools for programmers. They require the user to be familiar with the programming language in which the library is available.

• Most of them provide only the textual form of presenting the simulated quantum state. This representation is hard to read — it includes usually the long sequence of numbers. Thus, it is very hard to know exactly, what happens during the simulation.

• Even if the library provides a methods for visualizing the simulated quantum state, they have to be carefully programmed and placed in the right phases of the simulation.

• Every change in simulation (even additional printing the preview of the internal quantum state) needs editing the code and recompilation.

To sum up, these simulators offer the greatest flexibility, but also require the biggest knowledge: in both programming and the quantum computation theory fields.

This is presumably the biggest group of the quantum computer simulators. The examples for C/C++ are libquantum [15], Eqcs [19], QDD [20] and Q++ [21], for Java: m@th IT Java Mathematics Library [22], for .NET: Cove [23], for Python: QuTiP [24], qclib [25], and even Haskell Simulator of Quantum Computer [26] for Haskell and qlambda [27] for Scheme.

3.1.2 Quantum Programing Languages

This is a group of simulators, which expose an unique languages to program quantum computations. Such languages provide similar functions as those included in API of the simulation libraries, described in the previous section.

The main benefits of Quantum Programing Languages (QPLs) are:

• They are specially designed to perform quantum computations, which are different than classical. The existing, classical programming languages can be unsuitable
for the programming quantum computers. The QPLs better fit to the architecture and capabilities of the quantum computers.

- They have relatively small set of commands and are designed for only the one purpose — programming the quantum computations. Thus, they can be much easier learned by non-programmers than the simulators from the previous section.

Using the QPLs has however following disadvantages:

- The small set of instructions lowers their flexibility. The classical computations needed in pre- or post-processing parts in some quantum algorithms sometimes cannot be programmed. Mixing classical and quantum computations is impossible or highly difficult.

- They do not provide many useful constructions of the standard programming languages, such as loops, subroutines, et cetera.

All in all, the QPLs can be more relevant, when a quantum computer will be built and it will need a special programming language to expressing the computations. Now, QPLs are only experimental and thus they often offer only a small set of functions, which can be expressed also in standard programming languages. QPLs are less flexible and less robust than Simulation Libraries, however, they are generally simpler to use.

There are variety of Quantum Programming Languages. Some of them are similar to Assembler and enable only absolutely basic operations on quantum bits and registers. The examples are qMIPS [29] and CHP [30]. There are also the more robust languages, similar to the C programming language. These are QCL [31], LanQ [32], and kulka [33]. The next example is QuIDDPro [12], which can be accessed via language similar to MATLAB.

### 3.1.3 Interpreters

These are the simulators, which can be accessed via the Command Line Interface (CLI). They can interpret a file with the source code or read command typed interactively by the user. This type of interface is provided by most of the Quantum Programing Languages.

The advantages of this kind of interface are:

- Performing a quantum simulation is simpler than in the libraries for the standard programming languages.

- The interactive mode enables to execute simulation step by step.

- It gives an opportunity to observe the internal quantum state during the simulation. The user gain a better control of the simulation process during its execution.
- The changes in the quantum circuits can be made instantly — without recompilation.

The main drawback of these simulators are:

- Performing a batch of simulations is problematic. For scientific purposes (for example to perform intensive tests of some quantum algorithm) this type of interface can be unsuitable.
- The opportunity for code reusing is limited.
- Interpreters do not support as many syntax constructions as the programming languages, or using them is very cumbersome.

To conclude, the big merit of Interpreters is their interactivity. Since the quantum computer simulators are mostly used for educational and presentational purposes, the interactivity is very important. In this field, they are miles ahead the Simulation Libraries and QPLs. However, for more sophisticated application the Simulation Libraries are more suitable.

This type of interface is provided by CHP [30], QCL [31], LanQ [32], kulka [33] and QuIDDPro [12].

### 3.1.4 Graphical Simulators

This group includes all of the quantum computer simulators which are accessible via Graphical User Interface (GUI). They can be divided into the three main groups:

**Quantum Circuit Simulators**

They enable to build and simulate a quantum circuit, which are the most popular representation of the quantum computations. Thus, the theoretical circuits which can be found in the literature can be directly designed in these simulators and run.

**Graph-based Quantum Computation Simulators**

These tools represent the quantum bits and gates in a form of a graph. They attempt to imitate the physical construction of a quantum computation — the elementary particles which pass quantum gates. The big merit of these tools is that they show alternative methods of representing quantum computation, which would better fit the future architecture of a quantum computer. However, this makes them much harder to use. To implement a quantum algorithm from the literature, the quantum circuit have to be translated into that graph-based representation. This needs a good understanding of the Quantum Computation Theory. Moreover, such a translation is not always possible.
Single-Subject Simulation Tools

There are a vast amount of simulators which presents only a single quantum effect or algorithm. These are presentational tools, and not a complete quantum computer simulators.

The main advantages of the GUI-based quantum computer simulators are:

- Visual representation make it easier to understand the simulation.
- Some of them provide a run-time preview of the internal quantum state.
- They are accessible for non-programmers.
- To start using some of these simulators, users are not required to know much about quantum computations. Thus, these tools can be used for educational purposes.
- The Quantum Circuit Simulators enables to build and simulate quantum circuits presented in the literature. Thus, studying these circuits can be easier.

The common drawbacks of these simulators are:

- Some of the advantageous features presented above are supported by only a small subset of GUI-based simulators.
- Many such simulators are not intuitive. The GUI is cumbersome to use. In many cases it costs much more time to build a circuit graphically, than in a one of the Interpreters or Simulation Libraries.
- Many of these simulators are not completely finished. Some of their functions do not work properly.
- They generally do not support mixing classical and quantum computations. They enable only to simulate the quantum part of an algorithm. The user has to perform the classical pre- or post-processing independently in some other tool.
- Many of these simulators highly limit the number of qubits, quantum gates or simulation steps.
- Most of them do not provide methods for preparing reusable components. Thus, building even middle-sized quantum circuit is usually very time-consuming.
- They are unsuitable for performing a batch of tests.

Generally, the GUI-based simulators can be a great choice if one looks for a simple, presentational or educational tool. However, their limited functionality prevents them from performing more complex simulations. In that matter they often lag behind all other groups.
The example GUI-based simulators are:

**Quantum Circuit Simulators**
- QCAD [34], Quantum Computer Emulator [35], SimQubit [36], jQuantum [37], Qubit101 [38], Zeno [39], Online Quantum Computer Simulator [40] and Javascript Quantum Circuit Simulator [41]

**Graph-based Quantum Computation Simulators**

**Single-Subject Simulation Tools**
- Java Quantique Simulator [43], Grover’s Quantum Search Simulation Applet [44] or Bloch Sphere Simulation [45].

### 3.1.5 Toolboxes for the Scientific Software

This group includes packages for MATLAB, Octave, Mathematica, Maxima and Maple. There is a large number of such simulators, since the Quantum Computation Theory is mostly based on the linear algebra. The comprehensive list of these tools is maintained by the Quantiki Portal for Quantum Information community [16].

These tools are in fact a special kind of the Simulation Libraries for standard programming languages. Thus, they have similar advantages and drawbacks. However, they can be used only within the scientific software mentioned before. That fact makes them less accessible, especially in terms of costs.

### 3.2 Summary

This chapter outlined the current State-of-the-Art in the field of quantum computer simulation software. It presented a review of the existing tools of this type. They were grouped in several classes. Each class was described and compared with each other. We also listed the main advantages and drawbacks of every class.

This overview gave us a motivation for proposing a new quantum computer simulator. We saw the many drawbacks of existing simulators and realized, that many of them could be eliminated by a new tool which would provide a novel set of features. The proposal of such a tool is described in Chapter 5.

The work described in this chapter is concluded by a detailed comparison of the quantum computer simulators. In this comparison, several such tools are evaluated in terms of exposed interfaces, offered functions and other capabilities. The tested tools include also the new simulator, which is a result of this study. The outcomes of the comparison are presented in Chapter 9 and further expanded in Appendix A. They can be helpful for those which look for such a software.
Chapter 4

Survey of Simulation Techniques

The previous chapter reviewed the external properties of the quantum computer simulators, especially the types of their interfaces. This chapter focuses on the internal methods for simulating quantum processes. First of all, it explains, why simulation of a quantum computer is a challenging task. Next, it presents a variety of methods currently used to simulate quantum computations. We describe what are the internal algorithms and data structures used in the simulators. Finally we explain, which of them we choose for the new simulator.

4.1 Problems with Simulation of Quantum Processes

As Feynman showed, the quantum mechanical system can be efficiently simulated only by another quantum mechanical system [5]. Furthermore, conventional computers, when simulating quantum processes, would experience exponential slowdown. A quantum computer can operate directly on exponentially more data than a classical computer with a similar number of operations and information units.

Not surprisingly, that observation fuels the huge research field, which main goal is to build a quantum computer. On the other hand, many methods for simulating quantum processes on conventional computers have been proposed. Each of them tries to deal with exponential slowdown mentioned before. They however just skirt around that obstacle, but not overcome it. Nevertheless, this chapter shows that the choice of simulation method can imply the maximum size and complexity of modeled quantum system.

Let us consider a quantum computer with \( n \) qubits. Its internal state is represented by a unit vector in a \( 2^n \) dimensional space (see Section 2.2.1). From Schrödinger’s equation solved for discrete time, the evolution of this quantum system can be represented by a unitary matrix, which operates on this vector [59]. Thus the matrix size is \( 2^n \times 2^n \). In addition, each element of operation matrix and state vector is a complex number.

In most trivial approach, to perform an operation on \( n \)-qubit quantum system, we need to multiply the \( 2^n \times 2^n \) matrix (representing operation) by \( 2^n \)-element vector (which
represents system state). Denoting $D = 2^n$, such matrix-vector multiplication requires $O(D)$ operations for very sparse matrices, but even $O(D^2)$ for others. Then, if we want to perform a sequence of such operations, there are two possibilities. One is to multiply operation matrices and then the resulting matrix multiply by state vector. Second – to consecutively multiply operation matrix by state vector. The complexity of $D \times D$ matrices multiplication varies from $O(D^2)$ to $O(D^3)$, depending on algorithm and matrices features.

Nevertheless, the main weak point of this method is enormous memory consumption. The table below presents example usage for several small quantum system. (It is assumed, that the complex number is represented as two 8-bytes double precision floating-point numbers.) As shown in last two columns, adding one more qubit doubles the vector size and quadruples the matrix size. It seems thus clear, that such big matrices have to be avoided.

<table>
<thead>
<tr>
<th>Qubits Number</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Usage (state vector)</td>
<td>512 B</td>
<td>16 kB</td>
<td>16 MB</td>
<td>32 MB</td>
</tr>
<tr>
<td>Memory Usage (operation matrix)</td>
<td>16 kB</td>
<td>16 MB</td>
<td>16 TB</td>
<td>64 TB</td>
</tr>
</tbody>
</table>

### 4.2 Survey of Simulation Algorithms and Data Structures

This section presents various methods of internal representation of the quantum system. It includes the algorithms and data structures used for computing the results of the simulations. They directly imply the simulation time and memory consumption.

#### 4.2.1 Numerical Linear Algebra Methods

The most general technique is to simulate quantum computation by solving the Schrödinger’s equation. In that method, the approximated unitary matrix $U(t)$ is computed, where $U(t) = e^{-itH}$ and $H$ is a Hamiltonian representing the time evolution of the quantum system [59].

Computations are based on numerical linear algebra methods. The most straightforward method is to use standard linear algebra algorithms to diagonalize the matrix $H$. For $H$ of size $D \times D$, where $D = 2^n$ for $n$-qubit quantum system, the diagonalization can be done in $O(D^3)$ CPU time and $O(D^2)$ memory [60]. Another idea is to make a use of numerically exact polynomial approximation to the matrix exponential $U(t) = e^{-itH}$. It is referred to as Chebyshev Polynomial Algorithm [18]. For large time steps and reduced precision, that method requires $O(D)$ memory and CPU time to find $U(t)$. The next approaches are Short-Iterative Lanczos Algorithm [61] and Suzuki-Trotter Product-Formula Algorithms [62]. The latter, with memory and computational complexity of $O(D)$, is used by Quantum Computer Emulator [35].
The techniques mentioned in this section can be chosen, when there is a need for a general simulation of the quantum system evolution in time. However, the quantum computations can be expressed in much simpler way, without loss of generality.

### 4.2.2 Qubit-wise Multiplication

The most popular method which relevantly optimizes the simulation memory usage is to split the operation on \(2^n\)-element vector into smaller matrices, without explicitly storing \(2^n \times 2^n\) operation matrix.

It has been shown, that any unitary operation on \(n\)-qubit system can be expressed as composition of simple, one- or two-qubit elementary gates [63]. What is more, many such gates have been already physically built. It is presumed, that any arbitrary unitary multi-qubit operation will not be directly implementable, but will require splitting into elementary gates. That assumption is used by many quantum computer simulators. They do not allow to perform directly an arbitrary multi-qubit operation. Rather than, they enable to apply a sequence of elementary gates.

Let us consider an example: we need to apply a 1-qubit Hadamard gate on the third qubit of the register \(|00000\rangle\). As per naive method, firstly we should compute the \(2^5 \times 2^5\) operation matrix \(A = I \otimes I \otimes H \otimes I \otimes I\) and then multiply it by \(2^5\)-element state vector \(|0\rangle \otimes |0\rangle \otimes |0\rangle \otimes |0\rangle \otimes |0\rangle\), to get an output state vector. In this approach, the output vector is computed directly as \(|0\rangle \otimes |0\rangle \otimes H |0\rangle \otimes |0\rangle \otimes |0\rangle\), which only requires multiplying \(2 \times 2\) matrix \(H\) by 2-element vector \(|0\rangle\) (see Section 2.2.2). Analogous rules are applicable in more complicated cases, for example when input state vector is a superposition of multiple states. These rules will be further described in Section 7.2.

This method has many advantages: it can be parallelized [64], it reduces memory usage from \(O(2^{2n})\) to \(O(2^n)\) and can also decrease the simulation time [65], thanks to avoiding operations on big matrices. The main drawback is that memory consumption of \(O(2^n)\) can be still insufficient. However, this method is widely used in quantum computer simulators [22, 25, 64–66].

### 4.2.3 P-blocked State Representation

In both of the previously described methods, the state of the \(n\)-qubit quantum system required \(O(2^n)\) memory for storing state vector. However, the p-blocked state representation enables to significantly reduce this memory complexity – even to \(O(n)\) in best case.

Simulating quantum system via p-blocked states was proposed by Jozsa and Linden [67]. The state \(\rho\) is p-blocked if \(\rho\) can be expressed as a tensor product of states \(\rho_i\)

\[
\rho = \rho_1 \otimes \rho_2 \otimes \cdots \otimes \rho_k
\]
where $\rho_i$ is the state of at most $p$ qubits. All of the states are represented as density matrices $2^p \times 2^p$ (density matrices are further described in [7] or [55]). While in previous method, where $n$-qubit system required $O(2^n)$ memory, here each $\rho_i$ state require at most $O(2^p)$. Therefore, such state representation uses $O(k2^p)$ memory. For best case, when $p = 1$, the $n$-qubit quantum system can be represented by using only $O(4n)$ of memory.

However, the state can be represented as $p$-blocked if no $p + 1$ qubits are entangled. Hence, such representation needs an algorithm which keeps track of entangled qubits. Any operation affecting a block of multiple qubits can remove entanglement and leads to splitting that block into not-entangled qubits, which enables to decrease $p$. The detection of such cases could be inefficient for larger $p$. On the other hand, any operation affecting more than one block can cause entanglement of their qubits, which ends up with increasing $p$. In the worst case, when $p = n$, this method requires $O(2^n)$ memory – which is more than in previous method. At the same it is time much more complicated and time consuming, due to entanglement-tracing algorithm.

Unfortunately, there are many widely used states which cannot be efficiently represented via this method. These are, for example, Bell states (see Section 2.2.3 and equation (2.12)) or any $m$-qubit GHZ state [58]

$$|GHZ\rangle = \frac{|0\rangle^\otimes m + 1\rangle^\otimes m}{\sqrt{2}}$$

which are maximally entangled and cannot be split into smaller blocks.

### 4.2.4 Binary Decision Diagrams

The state of multi-qubit quantum system, as well as an operation on this system, can be effectively represented by Binary Decision Diagrams (BDD). This idea was introduced in QDD quantum computer simulator [20]. Similar technique was used to develop a data structure called QuIDD, which then was implemented in QuIDDPro simulator [12–14].

The BDD method is based on the fact, that many elements in state vector and operation matrices repeat themselves. The figure 4.1 shows example BDD-based representations of the 2-qubit states. The best, the worst and the middle-sized cases are presented. On the diagram, the node $I_0$ represents first qubit, and the node $I_1$ – second qubit. Each qubit node has two output edges: then (solid) and else (dashed). Going from node $I_i$ along the then edge chooses all the states where qubit $i$ has value 1. The else edges means values 0, respectively. The terminal value is an amplitude (a complex number) of chosen state.

This state representation can relevantly decrease memory usage. When all elements in state vector are the same, its BDD (QuIDD) requires only $O(1)$ memory, regardless of the vector size. In that case for $n$-qubit system, the memory usage is reduced from $O(2^n)$ to $O(1)$. However, in the worst case BDD-based representation still uses $O(2^n)$ memory.
Chapter 4. Survey of Simulation Techniques

29

a) State vector: \[ \begin{bmatrix} 1/2 & 1/2 & 1/2 & 1/2 \end{bmatrix} \]
BDD:  
\[ \begin{array}{c} \downarrow \\ 0.5 \end{array} \]

b) State vector: \[ \begin{bmatrix} 0.32 & 0.45 & 0.55 & 0.63 \end{bmatrix} \]
BDD:  
\[ \begin{array}{c} \downarrow \\ 0.32 \, 0.45 \, 0.55 \, 0.63 \end{array} \]

\[ \begin{array}{c} \downarrow \\ 1/\sqrt{2} \end{array} \]

\[ \begin{array}{c} \downarrow \\ 0 \end{array} \]

c) State vector: \[ \begin{bmatrix} 1/\sqrt{2} & 0 & 0 & 1/\sqrt{2} \end{bmatrix} \]
BDD:  
\[ \begin{array}{c} \downarrow \\ 1/\sqrt{2} \end{array} \]

\[ \begin{array}{c} \downarrow \\ 0 \end{array} \]

**Figure 4.1:** The example BDD (QuIDD) representing 2-qubit states for: a) best, b) worst, c) middle-sized case. The represented states are: a) \( \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) \), b) 0.32 \( |00\rangle + 0.45 |01\rangle + 0.55 |10\rangle + 0.63 |11\rangle \), c) \( \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \)

The operation matrices are represented similarly, using slightly extended BDDs. Each quantum computer operation, such as tensor product, applying a gate or measurement can be expressed as the operations on Binary Decision Diagrams. It is proven [12], that in many cases such operations have very promising complexity (for example \( O(n) \), \( O(n^4) \)), in comparison to other simulation techniques. Of course, there are always the worst cases, when the computational complexity of BDD-based simulations is not better as in other methods.

The main drawback of this method is the complexity of every operation on quantum state. The tensor product of two states, the measurement (especially affecting only small part of a larger system), the application of a quantum gate – all of them are nontrivial graph-based operations.

### 4.2.5 Hash Table State Representation

Storing the state of \( n \)-qubit quantum system is an important improvement of the Qubit-wise Multiplication method. Such state representation is a backbone of libquantum [15], a powerful quantum computer simulation library for C language.

This method exploits an observation, that a state vector is often very sparse. Thus, only its non-zero values are stored. Let us consider a GHZ state, which was very problematic for p-blocked-based method. Here, the \( n \)-qubit \( |GHZ\rangle = \frac{|0\rangle^\otimes n + |1\rangle^\otimes n}{\sqrt{2}} \) is stored as two-element hash table, which is shown on figure 4.2.

The properties of hash tables ensures that reading and changing the state value require \( O(1) \) operations in the best case and \( O(2^n) \) in the worst case, for \( n \)-qubit quantum
system. The complexity depends on the hash function. Here, the hash function is based on the integer representation of possible pure state, for which the amplitude must be stored (in given example, they are 0 for $|0000\rangle$ and 15 for $|1111\rangle$). These values are unique and easily comparable, thus ideal as a argument for hash function.

As the two previous methods, the hash-table-based simulation technique offers variable memory usage: from $O(1)$ to $O(2^n)$ in the worst case. These estimations are the same as for Binary Decision Diagram representation. Nonetheless, the worst cases for hash tables are more common than the worst cases for BDDs. For example, widely used $n$-qubit state $|\psi\rangle = (H^\otimes n) |0\rangle^\otimes n$ require $O(2^n)$ memory for hash table, while only $O(1)$ for BDD (QuIDD).

The main advantage of this method is its simplicity. Every operation, such as applying a quantum gate, measurement or tensor product, can be simply expressed as a corresponding part of matrix-vector operation. What is more, the majority of them have computational complexity depending on the size of the hash table. Thus, computations can be performed efficiently, as long as the hash table’s size remains relatively small.

### 4.2.6 Other Solutions

An interesting idea is to use Bayesian Networks to model and simulate quantum computations. This method was implemented in Quantum Fog [11], a graph-based, graphical quantum computer simulator. In this technique, the operations such as preparing qubits, applying quantum gates or measurements, are represented by network diagrams called quantum Bayesian nets. As a result, the sequence of quantum computations can be very easily built by users. The simulation results are calculated by exploiting Monte Carlo methods.

Another simulation method uses a structure called tensor network. The essence of this technique is to use graphs of tensors, which are a multi-dimensional generalization of matrices [68]. Tensors represent quantum states (its density matrices) as well as the operation matrices. An important operation on such graph of tensors is called tensor contraction and consists in merging connected (tensors) into a single tensor. In this method, the computation time and memory usage grows exponentially with $d$ – the maximum dimension of any tensor created by contractions. A worthwhile fact is, that extended version of tensor networks was used to demonstrate that the quantum Fourier
transform (QFT), a key part of Shor’s factorization algorithm [52], can be simulated efficiently on classical computer [69].

When there are a small number of entangled qubits in $n$-qubit quantum system, it can be efficiently simulated with the method, which utilizes the Schmidt decomposition [70] of the quantum state [71]. It has been shown, that in this method the simulation time and memory usages grow only linearly with $n$, but exponentially with the number of entangled qubits. On the other hand, this technique gives very poor results, when simulating maximally entangled states. In the worst case, it require $O(n^{2^n})$ memory to store $n$-qubit state, while previous methods needed at most $O(2^n)$.

### 4.3 Summary

After the review of existing quantum computer simulators (presented in Chapter 3) we decided to build a new such simulator. Thus, the next step was to investigate, what are the internal algorithms and data structures used to simulate quantum computations. Knowing them, we could build a new tool of this type.

The methods presented in this chapter varies in terms of its computational complexity and possible purposes. Some of them, such as the $p$-blocked state representation, Bayesian networks, tensor networks of Binary Decision Diagrams are rather experimental and theoretical. Usually, they are great tools for specified purposes, but they are not a universal simulation methods. Often they omit some needed issues or the methods for solving these issues are ineffective.

There is also a group of simulation methods which are in fact inappropriate for needed purposes. This group includes mainly the complex numerical linear algebra methods. The computational complexity of these simulators is much worse than the other methods. They seem to be a good tool for the simulation of quantum processes in continuous time. However, such a feature is not needed in quantum computer simulators. They require simpler but faster method.

To sum up, the most promising simulation technique is the *Hash Table State Representation*. It is robust and has sufficient complexity — it is $O(2^n)$ both in space and time, which in the field of simulating quantum processes is a good result. At the same time, it covers a complete set of algorithms needed to perform all the necessary simulations. It is also relatively simple to implement.

As a result, in the new simulator it was decided to implement a variant of the *Hash Table State Representation* — by utilizing a dictionary data structure. The detailed implementation of this method is described in Chapter 7.
Chapter 5

QuIDE — Motivation and Proposal

In Chapter 3 we presented existing quantum computer simulators and concluded, that there is a need for a new tool of this type. In this chapter we describe this idea in details. First of all, we present the motivation for this project: why such a new simulator is needed and what problems should it solve. Next, the software functional and nonfunctional requirements are specified. After that, we show a solution which fulfills these requirements. It is a new quantum computer simulator — Quantum Integrated Development Environment (QuIDE). We describe its capabilities and key parts. The more detailed information about their actual implementation can be found in Chapters 6 and 7.

5.1 Motivation

The idea of creating a new quantum computer simulator emerged while author was attending a academic course about the Quantum Information and Computation Theory. During that course, students had to implement presented quantum algorithms on a one of the quantum computer simulators, namely libquantum[15]. The goal of those assignments was to let students explore the capabilities of quantum computer and to support understanding and analyzing quantum algorithms. However, the libquantum library seemed to be not a right tool for these purposes. One of its main drawbacks was the lack of visualization methods (it provided only a simple textual functions for showing the quantum state). What is more, it did not support the graphical, circuit-based representation of quantum computations, which is used in the literature. The transition from this representation into a source code with functions from libquantum was difficult.

Author noticed, that a suitable tool for designing, analyzing and understanding quantum computations would have to be totally different. Thus, the next step was to make a detailed review of existing quantum computer simulation software. It is important part
of this thesis, presented in Chapter 4. The review showed, that none of those simulators actually met all the expectations.

5.1.1 Drawbacks of Existing Simulators

The main reasons, why existing simulation software was insufficient, can be described as follows:

- **Focusing only on some subfield of the Quantum Information Theory**
  The existing simulators could be divided into two groups. The first was formed by multi-purpose simulators, which provided basic building blocks to construct any arbitrary quantum algorithm. The second group consisted of software allowing to simulate only a concrete quantum algorithm or effect, such as Quantum Key Distribution[72] or Quantum Walks[73]. Since it was needed to simulate any arbitrary quantum computation, the latter group had to be rejected.

- **Difficult textual interfaces**
  The simulators, as shown in Chapter 3, could be classified in terms of their user interfaces (UI). There were applications with graphical UI, command line interpreters and simulation libraries for programming languages. In fact, the latter ones could be used only by programmers. But such software should be suitable also for physicists, mathematicians or any others interested in quantum computations. The Command-Line Interface (CLI) interpreters might be a slightly better in that matter. However if truth be told, all of the simulators exposing only textual interfaces were hard to understand and to operate with.

- **Limited features of graphical interfaces**
  In contrast to simulators with textual interfaces, the applications with Graphical User Interface (GUI) seemed to be easily accessible even for non-programmers or non-scientists. However, they provided only a subset of functions offered by more robust textual simulators. Thus, they were totally not suitable for simulating more complex quantum algorithms.

- **Problems with combining classical and quantum computations**
  Many quantum algorithms include some classical computations, which should be performed on conventional computers. None of GUI-based quantum circuit simulators offered a possibility to incorporate such classical computations. Also the textual interpreters provided only basic arithmetic or matrix-vector operations.

All of those observations confirmed, that there is a need for a new, different simulation software. Furthermore, the noticed drawbacks of existing simulators were very enlightening and enforced looking for better solutions. At that time, detailed requirements could be defined.
5.2 Users and Interfaces Specification

This section describes the targeted users of the software. It is also specified, what types of interfaces should it provide.

5.2.1 Types of Users

The software should provide an appropriate interface for:

- users with basic knowledge of quantum computations (knowing what is qubit, quantum gate and quantum state should be enough)
- programmers, by allowing them to construct complex algorithms via a well-documented Application Programming Interface
- non-programmers, by offering them a functional Graphical User Interface
- students or any other people willing to learn about the Quantum Computation and Information Theory
- teachers, which want to present possible capabilities of quantum computers
- scientists or others, willing to design, simulate or analyze quantum computations.

5.2.2 User Interfaces

It is required to execute simulations via following interfaces:

Application Programing Interface

Simulator’s Application Programing Interface (API) has to provide maximal flexibility, allowing to build complex algorithms, mix classical and quantum computations and take advantage of programming languages’ features such as loops, conditions and so forth.

Graphical User Interface

The GUI should meet usability requirements. They are specified in Section 5.3.2. It has to allow to represent quantum computations in the same way as presented in books and lectures — namely, in the form of graphical circuit (see Figure 5.1).

5.3 Requirements Specification

The software is expected to be a tool for designing, building, correcting, executing and analyzing the possible applications of quantum computers. This section describes the required functions of the software and also the nonfunctional requirements.
5.3.1 Functional Requirements

The quantum computer simulator is required to provide following features:

1. Management of quantum registers
   (a) Allocation and deallocation of quantum registers (consisting of qubits)
   (b) Concatenating the quantum registers via tensor product
   (c) Operating on pure and mixed (entangled) quantum states

2. Performing computations
   (a) Applying 1-qubit or multi-qubit quantum gates onto quantum registers
   (b) Providing a built-in pool of most commonly used gates, such as Hadamard gate, Controlled NOT gate, Toffoli gate, Pauli gates or rotation gates
   (c) Applying any arbitrary 1-qubit unitary transformation onto quantum registers
   (d) Applying any custom multi-qubit operation (such operations are described in the next point)
   (e) Performing a measurement of any qubit, a quantum register or its any part

3. Building custom computation blocks
   (a) Enabling users to compose many gates into bigger blocks
   (b) Allowing to reuse these custom components
   (c) Providing methods for encapsulation — The goal is to facilitate building complex, high-level computations from lower-level blocks, which also could be composed from basic operations. It should be possible for users to define custom computation blocks on each level. Due to encapsulation, the bigger block should be handled in the same way as basic, built-in operations.
   (d) Providing a wide pool of reusable computation blocks from low-level operations (such as computing carry bit in addition) to high-level modules (such as Modular Exponentiation or Quantum Fourier Transform – two key parts of Shor’s factoring algorithm[52])
4. **Combining quantum computations with classical computations**
   It has to be possible to mix quantum and classical parts of the algorithm. Simulator must allow to add any classical operation or algorithm to the quantum computation part.

5. **Preview of the internal state of a quantum system**
   Preview has to be allowed in each stage of the computation. This is impossible on real quantum systems, because each observation interfere with its state. Thus, such simulation possibility would be highly desired to plan and analyze computations, even if quantum computer was actually constructed.

6. **Step-by-step execution mode**
   Simulator has to allow performing computations step by step, with possibility to easily preview the internal quantum state between steps.

### 5.3.2 Nonfunctional Requirements

Below are listed the nonfunctional requirements of the simulator. The methods of evaluating whether they are met are also presented.

1. **Operating environment**
   It should be easy to obtain, install and run the simulator, also for non-programmers.

2. **Performance requirements**
   Each of the most powerful quantum algorithms, including Shor’s Factoring algorithm [52] and Grover’s Fast Database Search [49], has to be executable by the simulator in reasonable time. This implies subsequent requirements:
   
   (a) The simulator has to be able to simulate sufficient number of qubits. The Shor’s algorithm, which is the most demanding, requires $2n+3$ or even $7n+3$ qubits to factor an $n$-bit number, depending on its inner implementation [75, 76].

   (b) The simulator has to be interactive. Thus, the maximum acceptable time for the simulation of above algorithm is few seconds on a standard PC.

   (c) Two above requirements has to be applicable for a simulation of a small-sized, but nontrivial problem (such as factoring a 6-bit number). For more complex problems, the above time constraints are not required.

3. **Usability requirements**
   (a) The simulator should be understandable and accessible for any person with a basic knowledge of the Quantum Information and Computation Theory.

   (b) It should help users in exploring and understanding the capabilities of quantum computers.

   (c) The above requirements have to be verified by surveys for users.
4. **Requirement of a learning base**

(a) There is a strong need for a set of example applications, including the implementations of the most important quantum algorithms such as Grover’s Fast Database Search [49], Shor’s Factorization [52] or Quantum Teleportation [74].

(b) Each function of the simulator has to be concisely documented in a user guide or accessible via an interactive help instructions.

5.4 **Solution Outline**

The analysis of the previous quantum computer simulators yielded the vision of the new simulator. It showed, that all the previous simulators exposed only a one type of user interface. In other words, when classified in terms of exposed user interface, they formed three exclusive groups: libraries for programming languages, interactive command line interpreters and graphical tools. Each of these groups offered different simulation capabilities. Often choosing a software from one group was insufficient. For example, we needed to build a program which would use both classical and quantum operations, and then simulate it graphically, step by step. To realize that goal, we had to use at least two different simulators. What is more, we were enforced to build the same simulation twice: separately for every simulator.

The idea of QuIDE is to integrate various simulation methods into a single application. Thus, the name of the proposed simulator — QuIDE, Quantum Integrated Development Environment — intends to express this idea. As shown in Figure 5.2, QuIDE joins the ideas from three different classes of quantum computer simulators. However, it do not simply aggregate several existing approaches. It provides a methods which connect them together — to make them form a single, consistent unit.

![Quantum Computer Simulators](image)

**Figure 5.2:** The previous quantum computer simulators formed three exclusive groups of software: libraries for programming languages, interactive command line interpreters and graphical tools. The proposed simulator, QuIDE, integrates these three approaches and enables users to take an advantage of each of them at the same time.
5.5 QuIDE Components

The proposed simulator is based on the three key parts:

1. **Simulation Library** — QuIDE exposes the API to manage the quantum registers and to perform operations on them. It is a backbone of a whole simulator, as it includes the methods for simulating quantum processes. As shown in Chapter 3, such simulations are very problematic for conventional computers. The choice of the simulation algorithms as well as the necessary data structures is essential for the simulator performance.

2. **Graphical Environment** — The main goal of the proposed simulator is to facilitate learning, analyzing and building quantum algorithms. It is realized by enabling users to build quantum circuits graphically, to analyze computations step by step or to write programs which consist of both quantum and classical operations. All these functions are integrated and made available by Graphical User Interface.

3. **Run-Time Compiler and Translator** — This element allows to join the advantages of the Simulation Library and the Graphical Environment. It enables users to switch between the graphical circuit representation and the source code. It support run-time code compilation and generating the source code from a graphically designed circuit. Since it allows to interactively execute instructions typed in textual editor, this element can be viewed as an equivalent of an Interpreter.

In the next paragraphs we describe in details these three parts. However, the Run-Time Compiler and Translator is associated with the Graphical Environment. Thus, it will be further described as a part of the GUI.

5.5.1 QuIDE Simulation Library

The idea of the simulation library is based on following elements:

1. **Inner data representation and simulation algorithms** — Quantum state vectors are represented by a dictionary data structure. This implementation method is further described in Chapter 7.

2. **Application Program Interface** — It was decided, that QuIDE simulation functions should be available via API for one of the popular programming languages. An alternative was to design a new programming language to perform quantum computations. However, the first option was chosen, which entailed following advantages:
   - it is not necessary to learn syntax of a new programming language
any function or library which is available for chosen programming language can be combined with quantum computations from QuIDE simulation library.

We chose the C# programming language. It is widely used and very similar to other popular languages, such as Java or C/C++. The simulation API is object oriented, which is a standard nowadays. What is important, the API is designed to be easily extensible. Each user is allowed to define their custom quantum gate or complex, reusable operation block. One of the C# features, namely the extension methods, is a perfect tool to achieve this goal.

5.5.2 QuIDE GUI

The main features offered by the simulator’s GUI can be described as follows:

1. **Graphical quantum circuit representation** — The quantum computations can be represented in the same way as in the referenced bibliography. These computations can be also expressed via a program source code. The graphical representation is however a step towards non-programmers and every user willing to directly rewrite quantum algorithms from the literature into the simulator.

2. **Dynamic switching between code and graphical circuit representation** — QuIDE enables users to build quantum algorithms by designing quantum circuits graphically or by write program source code with functions from QuIDE simulation library. At any time user can switch between these two representations. (as on the Figure 5.3). No other tool is required. Such transitions are performed dynamically at runtime. What is more, both representations can be edited and executed – also at runtime.

3. **Performing computations step by step** — To better analyze and learn what exactly happens in each part of quantum computation, QuIDE provides an
opportunity to execute such computations step by step. The option of executing all the computations at once is given as well.

4. **Possibility to step back** — Every quantum operation (except measurement) can be expressed as an unitary transformation of a quantum state vector\(^{(59)}\). Unitary transformations are always reversible. In other words, if we know the resulting vector and the transformation, we can determine the input vector. This property is utilized by QuIDE. It allows to undo any computation step, as long as it does not include measurement.

5. **Up-to-date preview of the internal quantum state** — At any moment during computations, the internal state of the quantum system is visible (as illustrated in Figure 5.4). By internal quantum state we mean the vector of complex numbers, which fully characterizes the state of the whole quantum register (as described in Section 2.2.1). This vector describes also, what would be the result if we measured the quantum register, and gives the probabilities of obtaining these results.

![Figure 5.4: QuIDE allows to execute the quantum computations step by step. Since they are reversible unitary transformations\(^{(59)}\), stepping back is allowed as well. During the execution, QuIDE is displaying a live preview of the internal state of the simulated quantum system.](image)

6. **Mixing classical and quantum operations** — Since quantum operations are available via QuIDE simulation library API, user can unrestrictedly combine them with classical operations such as loops, conditions or functions from standard or mathematic libraries (as on the Figure 5.5). Even such combination can be interpreted by QuIDE to build a valid quantum circuit, which is fully interactive and ready to run step by step. On the other hand, there is also an option to view the results of the classical operations, such as the post-processing computations after the quantum part of the algorithm – in the simulator’s console output window, which displays the standard output of the program.

7. **Custom high-level computation blocks** — QuIDE allows to build custom computational blocks from basic quantum gates. These blocks can be further
Chapter 5. QuIDE — Motivation and Proposal

Figure 5.5: QuIDE enables users to build algorithms consisting of both quantum and classical operations. It can be done by writing a program code, which uses standard instructions and the functions from simulator’s API. QuIDE is able to identify the quantum computations and to represent them as a graphical circuit. A way of executing and showing the results of the remaining classical computations are also provided.

reused, especially to build another higher-level computation unit. This idea is illustrated in Figure 5.6. QuIDE realizes this feature by:

- providing a method to define custom blocks in source code
- correctly interpreting such source code and building a graphical quantum circuit which uses those custom blocks
- providing an ability to graphically preview the inner operations within a complex custom block
- enabling users to build such blocks graphically, by using only the interactive GUI
- correct source code generation from the custom blocks built interactively in GUI.

5.6 Summary

In this chapter we described all the requirements which the new simulator is expected to met. What is more, we proposed also the vision, how to meet them. In other words, we specified the concepts of the new simulator — QuIDE. The next step was to implement all of these concepts.

In the next two chapters we describe in details the design of the QuIDE simulator. In Chapter 6 we present its architecture. What is important, the specification of the architecture contains the elements from the solution outline presented in this chapter — the Source Code Editor, Interactive Circuit Designer, Console Output Window and Run-Time Quantum State Preview.

In Chapter 7 the internal data structures and simulation algorithms are presented. They were chosen and specified on the basis of the review of such internal simulation techniques described in Chapter 4.
Figure 5.6: Supporting a higher-level computation blocks is the feature, which distinguishes QuIDE from the previous simulators. All of them have common drawback: they provide only basic blocks to build a full computation algorithm. The proposed simulator provides not only basic, Assembler-like operations, but also a higher level functional blocks. Furthermore, it allows to combine them into even bigger blocks. Such approach is nowadays fundamental in software engineering. It should be thus extended also for the Quantum Computer Science.
Chapter 6

QuIDE Architecture

This chapter describes in details the inner structure of the QuIDE simulator, which results from the specification described in Chapter 5. First of all, we describe the general design decisions made at the beginning of the design process. Then, the main application layers are presented. These are the Model, the View and the ViewModel from the MVVM architectural pattern. Next, we look closer at each of these layers. We describe their inner components and interactions between them. Last of all, we describe an object model used by QuIDE to represent quantum computations and simulation results.

6.1 General Architectural and Technical Decisions

At the beginning of the design process, it was needed to decide about some technical aspects of the new simulator. The chosen options were mainly influenced by the software requirements, specified in Chapter 5.

First of all, it was needed to choose whether the simulator has to be a standalone application or rather an on-line solution. Secondly, it was decided that the core simulation module has to be available also as a standalone library which could be used without the QuIDE’s graphical interface. Next, the pure technological aspects such as programming language and architectural frameworks have to be decided.

Standalone Application Type

It was decided, that the simulator had to be a client-side, standalone application. This choice was dictated by the fact, that the simulations are computationally intensive — their complexity is exponential both in space and time (see Chapters 4 and 7). If we decided to perform these computations on the common server, we would require a server with a great computational power — in order to support many users at the same time. Moreover, the standard personal computers have nowadays enough resources to perform such simulations (for a single user). We wanted to use these resources. Finally, the client-side, standalone application can be easier distributed. We did not need to maintain (and even pay for) a server.
Independent Core Module — the Simulation Library

As shown in the next sections, QuIDE consists of many components of different roles. The basic component is the Simulation Library, which is responsible for performing the simulations and computing their results. It was decided, that the Simulation Library had to be accessible also as an independent component — a standalone library which could be used directly in source code, without the GUI layer.

Technology Choice

At the beginning of this study, we specified a demanding set of requirements — both for the GUI and the core simulation module (see Chapter 5). Thus, we decided to choose one of the most robust technologies which support both rich GUI design and complex numerical computations — the Microsoft .NET Framework. The QuIDE core simulation library, QuIDE.dll, is thus a standalone C# library. The GUI-based QuIDE simulator is a WPF desktop application (WPF — Windows Presentation Foundation, the Microsoft’s framework for designing and rendering graphical user interfaces [77]).

6.2 Application Layers — the MVVM Pattern

QuIDE is based on the Model View ViewModel (MVVM) architectural design pattern [78], shown in Figure 6.1. MVVM, as well as its predecessor — the widely used Model-View-Controller pattern [79], defines the rules for implementing user interface and communicating it with the business logic of the application. MVVM was proposed by Microsoft as a variation of MVC, which allows to totally separate the view layer from the business logic. The MVVM pattern is a base of the Windows Presentation Foundation (WPF) framework.

![Figure 6.1: The Model View ViewModel (MVVM) architectural pattern][78], which explains how to organize user interface and business logic into separate layers. In MVVM, the View layer is concerned only on the graphical user interface, while the Model layer — only on the business logic. The whole communication between them is realized by the ViewModel layer.

View

In the MVVM pattern, the View layer contains no business logic. As in MVC, it refers only to GUI components, such as buttons or labels. It is only responsible for capturing
Chapter 6. QuIDE Architecture

the UI events and displaying data. This is realized by two event-driven mechanisms: the Command Binding and the Data Binding. The GUI components are bound to the elements from the ViewModel layer. Whenever an UI event occurs, the bounded ViewModel element is notified. Similarly, when the data in the ViewModel changes, the notification to the corresponding GUI component is sent.

**ViewModel**

This layer is responsible for translating the data from the Model into a form which could be displayed by the View — the ViewModel is thus called a ‘model of the view’. Moreover, the ViewModel is a mediator between the Model and the View. It handles UI events from the View, as well as notifications from the Model. The ViewModel can respond to them by notifying the View, changing the Model or also performing an additional action such as data validation.

**Model**

This is the inner layer of the application. All the actions connected with the UI are performed by the View and the ViewModel. Thus, this layer is concerned only on the application logic. It should be noted, that the Model could be very complicated — it could be composed of several sub-layers or sub-components. However, it is not necessary to expose these inner elements. Instead, the Model should communicate with the ViewModel via a certain interface.

### 6.3 QuIDE Inner Components within the MVVM Layers

As shown in Figure 6.2, QuIDE is built from several communicating parts of different responsibilities. Each of them accepts certain types of input and produces a defined output. Therefore, any feature of QuIDE is realized as a workflow between these components. In this section we describe each of these components. We show also, what input they need, what output they produce and in what situations.

#### 6.3.1 View and ViewModel Components

Here we present a presentational layer of QuIDE. The following four components are directly exposed to the user. They are integrated in a common, one-window GUI.

**Console Output Window**

In QuIDE graphical environment, this sub-window shows the standard output of the program. When user build a quantum circuit only in the *Interactive Circuit Designer*, this window is not used. However, its role become more important, when user writes a source code and wants to print some information to the program output.
Figure 6.2: The figure presents the architecture of QuIDE. Its base is formed by the Simulation Library, which can be used independently to simulate quantum computations. However, to meet all the requirements, especially to enable switching between a source code and a graphical circuit, QuIDE had to be extended by additional components.

Source Code Editor

Since one of the goal of QuIDE is to be a fully functional Integrated Development Environment, it provides a source code editor with syntax coloring. As a consequence, no other software is needed to build programs which use the simulation library API to simulate quantum computations. The source code written in the Source Code Editor can be directly executed within QuIDE, saved or restored.

As mentioned before, QuIDE can generate source code from the model of a quantum circuit. The generated code is displayed in the Source Code Editor. User can then modify it, save or run. Using this possibility can help in learning the API of the simulation library.

Interactive Circuit Designer

The Source Code Editor allow to build quantum computations by writing the simulation source code. This component provides an alternative, much easier method. In the
Interactive Circuit Designer, the quantum computations are represented in a way, which can be found in every publication concerning quantum computations. The example quantum circuits are presented in Chapter 2 or in Figure 5.1. The Interactive Circuit Designer provides easy methods for adding qubits or quantum gates, as well as editing, copying or deleting them. What is more, user is enabled to build custom, complex blocks by composing simpler blocks or built-in quantum gates.

The Interactive Circuit Designer is also used when QuIDE generates quantum circuit from the source code written in the Source Code Editor. The generated circuit can be modified or simulated in exactly the same way as a circuit built from scratch by user.

The Interactive Circuit Designer interacts with other components of QuIDE by exchanging with them the Circuit Model. It is an object model of the quantum circuit and is further described in Section 6.4.

During simulation, the Circuit Model is used to calculate the current stage of the computation, especially when it is executed step by step (see Figure 5.4). To perform a single step, the Interactive Circuit Designer passes the Circuit Model to the Step Evaluator. The Step Evaluator uses it to process the step to be made. Then, the output is passed to the Quantum State Preview, which shows it.

Quantum State Preview

While the Console Output Window shows the results of the classical printing instructions written in the Source Code Editor, the Quantum State Preview component shows the result of the quantum computations simulated as a quantum circuit in the Interactive Circuit Designer.

As described in Chapter 2, the \(n\)-qubit quantum register can store \(2^n\) different values at the same time. Each of this values has its corresponding amplitude (a complex number) and a probability — indicating that the value would be the result if the register was measured. The Quantum State Preview window shows the state of a quantum register which is currently simulated in the Interactive Circuit Designer. In particular, it shows the list of all the possible register values, their amplitudes and probabilities. When the simulation is executed step by step, the preview dynamically changes.

The information about the current quantum state is passed to the Quantum State Preview in the form of the Output State Model, which is further described in Section 6.4.

6.3.2 Model Layer Components

Simulation Library

This is a core library responsible for computing the results of the simulations. It can be used independently, beyond the QuIDE graphical environment. The idea of this library is presented in Section 5.5.1. The details of its inner implementation are described in Section 7.1.
Chapter 6. QuIDE Architecture

Code Compiler

This component is used whenever the source code written in the Source Code Editor has to be processed. However, the code is not parsed. Instead, it is dynamically compiled at runtime. Such compilation is performed in two situations: to build a quantum circuit model from the source code and to execute the code. In both cases the dynamic assembly is built and then straightaway executed.

To generate a model of a quantum circuit, the code is compiled with the proxy library, instead of the core simulation library. Then, the compiled assembly is executed. Since the Proxy Classes are used, no simulations are performed. Instead, the proxy methods build the model of a quantum circuit. It is then passed to the Interactive Circuit Designer, which show it to the user.

The source code can be also executed without generating a circuit model. It is dynamically compiled with the core Simulation Library and immediately executed. The output of the program is displayed in the Console Output Window.

Proxy Classes

The Proxy Classes are used when QuIDE generates the Circuit Model from the source code. To accomplish this task, the calls to the simulation library used in the code are replaced by calls to the proxy library. This proxy exposes exactly the same API as the core simulation library. Whenever a call to the simulation library occurs, the corresponding proxy method is called. These proxy methods build a model of a quantum circuit. For example, if a function applying the Hadamard gate is called, the model of that gate is added to the Circuit Model. Additionally, the proxy classes encapsulate the classes from the Simulation Library and call their methods when needed.

Code Generator

The input of the Code Generator is the Circuit Model — a model of a quantum circuit, which is got from the Interactive Circuit Designer. The structure of this model is shown in Figure 6.3. The Code Generator analyze this model, step by step. For each operation in every step, it builds the text string expressing the appropriate method call. Next, all these strings are grouped and used to build a valid program source code. It is an exact equivalent of a quantum circuit from the Interactive Circuit Designer.

Step Evaluator

The Step Evaluator processes the Circuit Model from the Interactive Circuit Designer. As shown in Figure 6.3, this model contains information about the quantum registers and the subsequent computation steps consisting of quantum gates, which operate on those registers. At the beginning of the simulation, the Step Evaluator creates quantum registers by calling methods from the Simulation Library. Then, it gets the next step
which has to be evaluated, and calls the Simulation Library methods for each quantum gate included in this step. Afterwards, it use the Simulation Library to read the output state of the quantum registers. Having this output, it constructs the Output State Model — a model of the internal quantum state (further described in Section 6.4). This model is passed to the Quantum State Preview window, which renders it to the user.

6.4 Quantum Computation Model

As we can see in Figure 6.2, the QuIDE components often exchange the information about computations in the form of models: the Circuit Model and the Output State Model. The structure of these models is presented in Figure 6.3.

![Class Diagram](image)

**Figure 6.3**: This class diagram shows the model of the quantum computations. This model is shared between the components of QuIDE. It holds all the information needed to describe quantum computations: the data storage (quantum registers), the operations (quantum gates grouped into subsequent computation steps) and the result (a set of all possible resulting quantum states).

First of all, the model describes the registers used for computations — the number of qubits included and their initial values. The second part of the model represents all the operations within the whole quantum computation. They are grouped into subsequent steps, which allows user to simulate them in step mode and to perform stepping back. The last modeled element is an output of the computation. It contains the description of the state of the quantum registers in the current computation step. Since this state is represented by a superposition of pure states, the model contains the all elements of this superposition and their parameters.

The most important element of the model is the Operation. It is used to describe every action which can by applied on the quantum bits. Thus, this element is an abstract class — a generalization of many specific quantum operations such as quantum gates, measurement or complex computation blocks. The Figure 6.4 shows the subclasses of the Operation class and the dependencies between them.

The elementary quantum operation is the SingleQubitGate. It is a class of the unitary operations which can act on a single quantum bit. All such operations can be represented
by a $2 \times 2$ unitary matrix of complex numbers — which is modeled by the UnitaryGate class. However, the most important operations such as rotations or Pauli’s gates are modeled as separate classes. This facilitates rendering their graphical symbols and improves performance, since some of them can be computed very fast (see Section 7.1).

The CNotGate and the ToffoliGate are examples of quantum gates which operate on two or more qubits. One qubit is a target, as in the single-qubit gates, while others are control bits (as explained in Chapter 2). QuIDE supports a set of such controlled multi-qubit operations, for example controlled-Z or controlled phase shift gates. They are represented as a appropriate model class as well.

The Measurement is also a type of quantum operation, however neither unitary nor reversible. It represents performing a measurement on quantum bit or on a group of them at once. If the step contains measurements, it cannot be performed backward. Instead, user has to restart the simulation.

The CompositeOperation represents the complex, higher-level operation blocks which can be built interactively by user. It can be composed of basic quantum gates, but also of more complex operations, including other such composites.

In QuIDE, users can define custom quantum operations in a source code. These operations can include both quantum and classical instructions. When the circuit model is generated from the source code, such operations are represented by the ParametricOperation class. Its field ExecutionMethod indicates the user’s definition, which is called during simulation. Therefore, this class of operation offers the greatest flexibility, allowing to build computation blocks unavailable in other quantum computer simulators (as shown in Chapter 9, in the results of a Functionality Evaluation).
6.5 Summary

This chapter described the architectural design of the QuIDE simulator. Its form resulted from the requirements presented in Chapter 5. However, while that chapter specified what the simulator need to do, this chapter presented how the required functions were actually realized.

We started with describing the key architectural and technical decisions which had to be made at the beginning of the designing of QuIDE. Then, we presented the three main layers of QuIDE: the Model, the View and the ViewModel. Next, we looked deeper and described the inner components forming these three layers. We explained their roles and interactions. Also, we described the models used for representing the information exchanged by these components.

In this chapter we do not focused on the internal methods used for simulating quantum computations. All of them are simply encapsulated in the Simulation Library component. However, they are extremely important in terms of the simulation performance (the reason for it was presented in Chapter 4). Thus, we look closer at the inner implementation of the Simulation Library in the next chapter.
Chapter 7

Simulation Algorithms and Implementation Details

The previous chapters described all the important functions of QuIDE simulator. However, the simulator as a whole would not be usable at all, if the core simulation module was inappropriately designed. Therefore, it was needed to carefully choose the data structures and algorithm used for simulating quantum computations. The survey of the simulation techniques, presented in Chapter 4, greatly helped in this task. One of the methods presented there, a Hash Table State Representation, became a basis for the implementation which we present in this chapter.

First of all, we show, what data structures we have chosen to represent the quantum computations. Then, we present the algorithms that are used to operate on these data structures. We describe in details a representative group of them — namely, the most important algorithms which the other ones are based on. Finally, we estimate the time and memory complexity of the presented methods

7.1 Data Structures

The state of the $n$-qubit quantum register is represented by a linear combination of basis (pure) states, as shown in (7.1).

$$|\varphi\rangle = \sum_{j=0}^{2^n-1} \alpha_j |j\rangle = \alpha_0 |0\rangle + \alpha_1 |1\rangle + \cdots + \alpha_{2^n-1} |2^n-1\rangle \quad \text{where} \quad \alpha_j \in \mathbb{C} \quad (7.1)$$

In the simulation library, quantum registers are represented by a Register class. It has an attribute Width representing the number of qubits within the register (thus, for the register $|\varphi\rangle$ from (7.1), the Width would equal $n$). The Register class and all its members are described in Section 8.1 and further in the API Reference available on the QuIDE’s website.$^1$

In order to perform quantum computations, user can operate on many quantum registers. They can be created and deleted as needed. These operations are executed by a \texttt{QuantumComputer} class. The \texttt{QuantumComputer} is a singleton which manages the lifetime of the registers and is responsible for performing any cross-register operation. The dependencies between these classes are shown in Figure 7.1.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure7.1}
\caption{In QuIDE simulation library, the quantum register is represented by the \texttt{Register} class. Many registers can be used at the same time. They are allocated and deallocated by a singleton \texttt{QuantumComputer} class. The data structure used for storing the information about the internal quantum state is the dictionary.}
\end{figure}

In terms of performance, the most important is the inner data representation of the \texttt{Register}. In QuIDE simulation library we use a dictionary. This data structure, known also as associative array, map or symbol table, is an collection of \texttt{(key, value)} pairs, such that each possible key appears at most once in the collection \cite{80}. As shown in Figure 7.1, the dictionary’s keys are unsigned long integers representing the basis states (see (7.1)). The value for the given key is an amplitude of that basis state which is a complex number.

The use of the dictionary as a main data structure is a variant of the simulation technique based on a hash table state representation, described in Section 4.2.5. In our solution, the role of a hash table is fulfilled by a built-in dictionary structure. This approach was chosen for several reasons:

- The dictionary allow to optimize the memory consumption in exactly the same way as the hash table. Namely, only the non-zero amplitudes are kept in memory. There is no need to store whole \(2^n\)-element state vector of complex numbers. The operations are also performed in similar way as in the hash table technique. They are described further in the next section.

- The inner implementation of the built-in dictionary could be very fast and optimized.

- Using the dictionary is very simple.

- The dictionary is one of the most widely used data structure in modern programming languages and technologies such as JavaScript or Python. The
author decided to check whether the computations on this data structure could compete with the specialized hash table — in terms of performance results. If they could, the presented here implementation could be easily used in these modern technologies, for example to build an in-browser quantum computer simulator.

Since the chosen technology for this project is the Microsoft .NET framework, we use the generic Dictionary class from the System.Collections.Generic namespace. Also, the complex numbers are represented by a Complex structure from the System.Numerics namespace.

7.2 Implementation of the Quantum Operations

As shown in Chapter 2, for given input state $|\varphi\rangle$ and an operation matrix $M$, the resulting state is $|\varphi'\rangle = M|\varphi\rangle$. If $|\varphi\rangle$ is a state of an $n$-qubit register, such operation would require to multiply a $2^n \times 2^n$ matrix by a $2^n$-element vector. As shown in Chapter 4, this is extremely ineffective.

The dictionary-based state representation method used in the QuIDE simulation library allows to compute the same result, but without using any extra data to store operation matrix. In order to achieve it, we apply the following tactics:

1. We do not compute the $2^n \times 2^n$ operation matrix, neither for the whole computation process nor for the single computation step. Instead, we apply elementary quantum gates, one by one.

2. When applying each of these elementary quantum operations, we use techniques which operate only on the state vector. The most important of them are described in the next paragraphs. Namely, there are presented the methods for computing the Not gate, the controlled gates, the matrix-defined unitary gates and the measurement operation. Every other quantum operation supported by QuIDE is implemented on the basis of these algorithms.

Not (Sigma X) Gate

Let us consider the simple not gate applied to one qubit:

$$
X(\alpha |0\rangle + \beta |1\rangle) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix} = \alpha |1\rangle + \beta |0\rangle
$$

(7.2)

As we can see, the not operation swaps the amplitudes of the basis states. The same rule applies to the multi-qubit quantum registers. The equation (7.3) shows the application the not gate to the qubit on the position 1 (where 0 is the least significant bit) to the 2-qubit register.
\[(X \otimes I)(\alpha_0 |00\rangle + \alpha_1 |01\rangle + \alpha_2 |10\rangle + \alpha_3 |11\rangle) = \]
\[= \alpha_0 |10\rangle + \alpha_1 |11\rangle + \alpha_2 |00\rangle + \alpha_3 |01\rangle \quad (7.3)\]

We do not need to perform any computation to obtain a result. We only have to swap
the amplitudes for these basis states, which have opposite values of the acting bit. In
QuIDE dictionary state representation, it can be done by interchanging the values for
the corresponding keys.

### Controlled Gates

The simplest controlled quantum operation is the C-Not gate. The equation (7.4) shows
the C-Not applied to the 2-qubit register, where the 0th qubit is the control and the 1st
is the target.

\[C_{01}(\alpha_0 |00\rangle + \alpha_1 |01\rangle + \alpha_2 |10\rangle + \alpha_3 |11\rangle) = \]
\[= \alpha_0 |00\rangle + \alpha_1 |11\rangle + \alpha_2 |10\rangle + \alpha_3 |01\rangle \quad (7.4)\]

Similar to the Not gate, the C-Not operation swaps the amplitudes of the basis states.
However, they are interchanged if and only if the control bit is set. Such action can be
easily implemented on the dictionary state representation, as shown in algorithm 7.1.

Any controlled gate can be implemented similarly – by iterating through a dictionary
keys and performing any actions only if the control bit is set to 1 (as in line 4). QuIDE
enables users to add a control bit to any single-qubit gate (for example, to build a
controlled-Hadamard gate). Such gates are computed by exploiting the same method.
Likewise, the Toffoli gate, with two or more control bits, is implemented similarly.

### Matrix-Defined 1-Qubit Gate

QuIDE allows to define and apply any 1-qubit quantum operation which can be
represented by an arbitrary unitary matrix. This paragraph shows, how it could be
computed without any additional space. This method was proposed and implemented

As mentioned before, we compute only single gate at once. Thus, in this case, we need to
apply an unitary \(U\) to a single qubit from the register \(|\psi\rangle\): \((I \otimes \cdots \otimes I \otimes U \otimes I \otimes \cdots \otimes I)|\psi\rangle\). Similarly to Not and C-Not operations, we need to find pairs of basis states, which differs
only in the target bit. Let \(\alpha\) be an amplitude of the basis state \(|\cdots \cdots 0 \cdots \cdots \rangle\) and \(\beta\) an
amplitude of \(|\cdots \cdots 1 \cdots \cdots \rangle\) (the rest of bits – noted as ‘*’ – are exactly the same for
Algorithm 7.1 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 \( D[\text{key}] \).

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

1: procedure C-Not
2: \( S \leftarrow \emptyset \)
3: for all \( \text{state} \in \text{keys}(D) \) do
4:   if \( \text{state} \notin S \) then
5:     if ControlBitIsSet(state) then
6:       amplitude \( \leftarrow D[\text{state}] \)
7:       reversedTargetState \( \leftarrow \text{ReverseTargetBit}(\text{state}) \)
8:       if reversedTargetState \( \in \text{keys}(D) \) then
9:         \( D[\text{state}] \leftarrow D[\text{reversedTargetState}] \)
10:        \( D[\text{reversedTargetState}] \leftarrow \text{amplitude} \)
11:       \( S \leftarrow S \cup \{\text{reversedTargetState}\} \)
12:     else
13:       \( D[\text{reversedTargetState}] \leftarrow \text{amplitude} \)
14:      \( \text{Delete}(D[\text{state}]) \)
15:   end if
16: end if
17: end for
18: end procedure

Comment: 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 the applying C-Not from equation (7.4). 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 that 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 else 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).
both states). The resulting amplitudes for these basis states, \( \alpha' \) and \( \beta' \) can be computed as shown in (7.5). If we perform these computations for each such pair of basis states, we will obtain a final result.

\[
\begin{bmatrix} \alpha' \\ \beta' \end{bmatrix} = U \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} a\alpha + b\beta \\ c\alpha + d\beta \end{bmatrix}
\] (7.5)

This method can be straightforwardly implemented on a dictionary-based state representation. The implementation is described in algorithm 7.2.

**Measurement**

The quantum register stores a superposition of the basis states. As described in Chapter 2, the measurement of the quantum register cause this superposition to collapse to only one basis state – which is the result of the measurement. A certain result can be obtained with a certain probability – the squared magnitude of the amplitude corresponding to this basis state.

QuIDE provides two methods for measuring quantum bits. It is possible to measure the state of the whole quantum register, or only the state of the single qubit within it. The dictionary-based implementation of the measuring the whole quantum register is presented in algorithm 7.3.

In the second case, it is possible to measure only one qubit from \( n \)-qubit quantum register. After that operation, we have to determine the state of the remaining qubits. For this, we make use of the generalized Born rule, which was introduced in [54] and is one of the implication of the Born rule enunciated by Max Born. According to the generalized Born rule, the state of \( n \)-qubit quantum system (7.1) can be represented in the form (7.6).

\[
|\varphi\rangle = \gamma_0 |0\rangle |\Phi_0\rangle + \gamma_1 |1\rangle |\Phi_1\rangle, \quad |\gamma_0|^2 + |\gamma_1|^2 = 1
\] (7.6)

The qubit to be measured is written on the left, and the states \( |\Phi_0\rangle \) and \( |\Phi_1\rangle \) represent remaining \( n - 1 \) qubits. To preserve consistency with (7.1), the states \( |\Phi_0\rangle \) and \( |\Phi_1\rangle \) can be expressed as in (7.7), up to normalization constants.

\[
|\Phi_0\rangle \propto \sum_{j=0}^{2^{n-1}-1} \alpha_j |j\rangle_{n-1}, \quad |\Phi_1\rangle \propto \sum_{j=0}^{2^{n-1}-1} \alpha_{j+2^{n-1}} |j\rangle_{n-1}
\] (7.7)

The form (7.7) allows to determine the state of the \( n \)-qubit quantum register after measuring a single qubit within it. If the resulting value equals \( x \) (0 or 1), the whole register will remain in the state \( |x\rangle |\Phi_x\rangle \). QuIDE exploits this rule to perform a measurement only on single qubit within the quantum register.
Algorithm 7.2 Matrix-defined 1-qubit unitary gate application

Require: A dictionary $D$ representing the state vector (as in algorithm 7.1); A $2 \times 2$ unitary matrix $U$, consists of numbers $a, b, c$ and $d$, as in equation (7.5).

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

1: procedure Unitary($U$)
2: $S \leftarrow \emptyset$
3: for all state $\in \text{keys}(D)$ do
4:   if state $\notin S$ then
5:     $\alpha \leftarrow D[state]$
6:     $\beta \leftarrow 0$
7:     reversedTargetState $\leftarrow \text{ReverseTargetBit}(state)$
8:     if reversedTargetState $\in \text{keys}(D)$ then
9:        $\beta \leftarrow D[\text{reversedTargetState}]$
10:       $S \leftarrow S \cup \{\text{reversedTargetState}\}$
11:     end if
12:   if $\text{TargetBitIsZero}(state)$ then
13:      $\alpha' \leftarrow a\alpha + b\beta$
14:      $\beta' \leftarrow c\alpha + d\beta$
15:   else
16:      $\beta' \leftarrow a\beta + b\alpha$
17:      $\alpha' \leftarrow c\beta + d\alpha$
18:   end if
19:   if $|\alpha'|^2 < \epsilon$ then
20:      $\text{Delete}(D[state])$
21:   else
22:      $D[state] \leftarrow \alpha'$
23:   end if
24:   if $|\beta'|^2 < \epsilon$ then
25:      $\text{Delete}(D[\text{reversedTargetState}])$
26:   else
27:      $D[\text{reversedTargetState}] \leftarrow \beta'$
28:   end if
29: end for
30: end procedure

Comment: As in algorithm 7.1, 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 $\alpha$ and $\beta$. After line 11 they hold amplitudes of the currently processed state and the corresponding state with reversed target bit — as in equation (7.5). In (7.5) 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 (7.5). For the opposite case we need to swap the computations. The $\alpha'$ and $\beta'$ 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 firstly 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 $D$. 
Algorithm 7.3 MEASUREMENT implementation for the dictionary-based state representation

**Require:** A dictionary \( D \) representing the state vector (as in algorithm 7.1).

**Ensure:** The result of the measurement – one of the possible basis states; the actualized \( D \), after the measurement operation.

```plaintext
1: function Measure
2:     random ← N\textit{extRandomReal}(0.0, 1.0)
3:     sum ← 0
4:     result ← 0
5:     for all state ∈ keys(\( D \)) do
6:         result ← state
7:         α ← \( D[\text{state}] \)
8:         sum ← sum + |α|^2
9:         if sum ≥ random then
10:             break
11:     end if
12:     \text{DeleteAll}(\( D \))
13:     \( D[\text{result}] \) ← 1
14: end for
15: return result
16: end function
```

**Comment:** Firstly 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 become the result of the measurement. In this way, we ensure that the measurement of the quantum state return 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, and after that it contains only the measured state with the 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).

### 7.3 Complexity Analysis

#### 7.3.1 Space Complexity

During the simulation of an \( n \)-qubit quantum system, QuIDE stores its state vector in the dictionary. Its keys are the basis states, and its values are the amplitudes corresponding to these states. What is important, no other data structures are used for representing quantum computations. The simulator does not store neither compute any operation matrix or other structure, which size is determined by \( n \).

If the amplitude for any basis state equals zero, this key-value pair is not stored. For example, the state \( \ket{\varphi_1} = \frac{3}{5} \ket{1000100} + \frac{4}{5} \ket{1100101} \) is stored in only two-element dictionary. However, the same 7-qubit quantum system at some other point of simulation
can be in the fully mixed state $|\psi_2\rangle = \alpha_0 |100000\rangle + \alpha_1 |000001\rangle + \ldots + \alpha_{2^7-1} |111111\rangle$, with each of the amplitudes $\alpha_i \neq 0$ and need a $2^7$-element dictionary.

Thus, the best case is when the $n$-qubit quantum system is one of the pure basis states. Even for big values of $n$, the amount of used memory is relatively small and does not depend of $n$. 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, the space complexity in the worst case equals $O(2^n)$.

### 7.3.2 Time Complexity

The operations on the state vector are implemented as in the algorithms 7.1, 7.2 and 7.3. Each of them contains the for loop, which iterates all the entries in the dictionary representing the state vector. This loop is common for each quantum operation implemented in QuIDE. Besides this loop, no other control statements, which iteration number depends on $n$, are used. Therefore, the complexity of any operation (even the simplest elementary gate) depends on the size of the dictionary representing the state vector.

In the best case, the dictionary stores only a single key-value pair. Any operation on an $n$-qubit state needs only one iteration of the loop described before. Therefore, in the best case, the time complexity of a single operation on $n$-qubit quantum system is $O(1)$.

In the worst case, the size of the dictionary equals $2^n$, so the loop perform this number of iterations. As a result, the time complexity of a single operation on an $n$-qubit quantum system equals in the worst case $O(2^n)$.

### 7.4 Summary

In this chapter we described the key aspects of the internal simulation module implementation. We showed the data structures used for representing the quantum computations. We discussed their advantages and drawbacks and explained, why they were chosen.

Next, we presented the algorithms used for operating on the chosen data structures. In this chapter we focused on the most important of them. The implementations of the other operations were similar, so we needed to present only the representative group of them.

Finally, we estimated the time and space complexity of the simulation method presented in this chapter. As we shown, both of these estimations depends on the size of the superposition, which represents the simulated quantum system. The best case is when there is actually no superposition — the qubits are in pure basis states. In this case the time and space complexity equal $O(1)$. In the worst case, the full superposition of basis states is simulated. It needs $O(2^n)$ of both time and memory.
We expected such complexity estimations. The best case, with no superposition, is de facto the same as operating on a conventional computer, so it needs no further resources. In the worst case, the simulated quantum system fully exploits its unique computational power — the quantum parallelism. As explained in Section 2.2.3, this is the feature, which puts the quantum computers miles ahead the conventional computers — by increasing their computational power exponentially. Therefore, simulating this effect cannot be effective and in fact has to have exponential complexity. We should rather be satisfied, that the exponent value is only $n$ — as shown in Chapter 4, there are many methods which simulate $n$-qubit quantum system with the complexity $O(2^{2n})$ or similar.
Chapter 8

QuIDE Features and Applications

The previous chapters presented how QuIDE was designed and implemented. In this chapter we show the result — we present the features of QuIDE from the user’s point of view. First of all, we describe the API of the core simulation library QuIDE.dll, which can be also used without the QuIDE graphical environment. Then, we demonstrate the GUI and its capabilities. Next, we list the quantum algorithms and examples from the bibliography, which we implemented on QuIDE. Finally, we show an example application of this simulator.

8.1 Simulation API

Provision of an Application Programing Interface (API), allowing to program quantum computer simulations, was one of the main requirements for QuIDE. To realize this goal, the simulation library QuIDE.dll was implemented. The QuIDE.dll is a C# library, which provides all the methods needed to perform any quantum computation compatible with the theory described in Chapter 2. It provides an object model for quantum computations. The Figure 8.1 shows the UML class diagram of this model.

The full API reference of QuIDE.dll is available on the QuIDE’s website. In short, the library allows to allocate and deallocate quantum registers and to operate on them: by applying quantum gates or measuring them.

QuantumComputer Class

This is a main class needed for performing computations. At the beginning of any simulation, the reference to QuantumComputer (which is a singleton) should be obtained by calling its static method GetInstance. Then, QuantumComputer is needed whenever a quantum register has to be allocated (by calling the method NewRegister) or deallocated (via the DeleteRegister method). QuantumComputer is also able to combine one or more quantum registers into a single, wide register — by performing the tensor product of them in the method TensorProduct.

Register Class

The Register class represents a quantum register. It can be created and deleted only by QuantumComputer.

The Register is a set of one or more quantum bits, which number is represented by the Width property. A nonempty subset of subsequent qubits of any Register is referenced also by a Register class.

The state vector of a quantum register can be obtained as a pure array by calling the method GetVector. This operation is however not always possible. If a given Register is a reference to a part of any wider Register, its state vector could be undefinable due to the quantum entanglement phenomenon (described in Section 2.2.3). In that case, the method returns null.

The operations on the qubits within a Register can be performed via its methods. A whole register or a subset of its qubits can be measured — by calling the Measure method. An application of a quantum gate is realized by the methods of corresponding names: CNot for C-Not gate, Hadamard for Hadamard gate, SigmaX for Pauli-X gate (since it is noted in the literature also as $\sigma_x$ — ‘sigma’), SigmaY, SigmaZ for Pauli-Y and Pauli-Z gates, RotateX and consecutive for rotation gates, and many more. The API documentation available on QuIDE project website describes all these methods in detail.
Extension Methods

The extension methods adds new functions to the QuantumComputer class. In this way many of complex quantum subroutines are implemented (including the Quantum Fourier Transform (QFT), Modular Addition and many more). This is also a way, in which the users can add their own functions. This action does not require the source code of QuIDE.dll.

8.2 Graphical User Interface

The GUI of the QuIDE simulator is shown in Figure 8.2. It has a form of a one-window Integrated Development Environment. It consists of four main panels: Source Code Editor, Interactive Circuit Designer, Run-Time Quantum State Preview and Console Output Window. They match the four main UI elements, which was described in Chapters 5 and 6.

![Figure 8.2: The Graphical User Interface of the QuIDE simulator.](image)

The elements of the GUI, which are numbered in the Figure 8.2, give the user access to the most important features. They are:

1. **‘Build Circuit’ button** — the source code from the Source Code Editor is translated into a graphical quantum circuit in the Interactive Circuit Designer.

2. **‘Run In Console’ button** — the source code from the Source Code Editor is executed and its output is printed in the Console Output Window.
3. ‘Restart’, ‘Step Back’, ‘Step Forth’ and ‘Run to End’ buttons — these buttons enable to execute the whole circuit, restart the simulation from the beginning and to execute it step by step. If the previously made step consists of reversible operations, it can be undone by clicking the ‘Step Back’ button.

4. ‘To Code’ button — the graphically designed quantum circuit from the Interactive Circuit Designer is translated into the source code in the Source Code Editor.

5. ‘Ungroup’ toggle button — after switching it on, when the user clicks on a compound gate, it is replaced by its inner sub-gates.

6. ‘Group’ button — the gates selected in the Interactive Circuit Designer become a one, compound gate, with a chosen name. After that, the newly created gate encapsulates its components (on the circuit, its symbol replaces their symbols). It can be also chosen from the toolbar and repeated at any point of the circuit. After clicking the ‘To Code’ button, the implementation of this compound gate is written in a form of a new extension method.

7. ‘Select Composite Gate’ drop-down list — it contains all of the compound gates (described in the previous point). Also, it contains the extension methods for the QuantumComputer class, written in the Source Code Editor. Additionally, it contains all of the complex quantum subroutines (such as QFT) which are implemented in the QuIDE.dll library.

In addition, the GUI of QuIDE offers also the following features:

- Creating, opening, and saving the source code files.

- Syntax coloring in the Source Code Editor.

- Support of a rich set of keyboard shortcuts, especially for code edition.

- Support of Cut / Copy / Paste / Delete operations in the Interactive Circuit Designer.

- A rich set of elementary quantum gates, accessible directly from a toolbar.

- Adding a control bit to any of these built-in gates.

- A possibility to preview the state of any subset of the quantum register. If it is in an entangled state, the amplitudes cannot be extracted (as shown in Section 2.2.3) — but the probabilities are displayed.

- Displaying the properties of the chosen basis state within the state of the whole register — its probability and amplitude (a complex number displayed as a vector in a complex plane).
• Displaying the properties of the chosen quantum gate: the definition of its unitary matrix and any needed parameters (for example the angle for rotation gates)

• Enabling to change the parameters of selected quantum gate (for example the angle for rotation gates) in the ‘Properties’ pane. Before applying, the new values are validated — for example, the unitarity of the new matrix of a gate is checked.

• A basic calculator.

The QuIDE simulator is available for downloading at the QuIDE project website. Users can find there also a user manual which describes all of the QuIDE’s graphical UI capabilities.

8.3 Implemented Quantum Algorithms and Subroutines

The QuIDE.dll library contains the implementation of many quantum subroutines. What is more, we implemented also the important quantum algorithms. They can be opened in the simulator and run in the console or translated into a circuit. Their source code files are enclosed to the QuIDE distribution package. The implemented subroutines are listed below. For each of them, the inversed operation was implemented as well:

• Quantum Fourier Transform [7]
• Carry and Sum operations (a classic parts of an adder)
• Adder and Modular Adder
• Controlled Modular Multiplier
• Modular Exponentiation
• Load Number (loading a specified integer value into the quantum register)
• Walsh-Hadamard Transform (Hadamard gate application on each qubit)
• Swap gate
• Fredkin gate (controlled Swap)

The implemented quantum algorithms are:

• Grover’s Fast Database Search Algorithm [49]
• Shor’s Prime Factorization Algorithm [52]
• Quantum Teleportation [7, 54]

\(^2\url{http://student.agh.edu.pl/~bjoanna/quide/} \text{ (Accessed Sep 26, 2014)}\)
• Quantum Dense Coding [7, 54]
• Deutsch Problem [7, 50, 54]
• Bernstein-Vazirani Problem [8, 54]

The Shor’s Factorization Algorithm has been implemented in two optimization variants by Bartłomiej Patrzyk, which he describe in his M.Sc. Thesis [53].

The subroutines and algorithms listed above were implemented and exploited during the university course ‘Matematyka w informatyce przyszłości’, where the QuIDE simulator was used. The information about that course is available at the Syllabus AGH website. The course assessments are described at the course’s website.

8.4 Example Application

In this section we present an example application of QuIDE. At first, we show a source code of example simulation. It is included in Listing 8.1. This code was put into the Source Code Editor. Then, we used QuIDE to build a graphical circuit from this source code. The generated circuit is shown in Figure 8.3.

```csharp
using Quantum;
using Quantum.Operations;

namespace QuantumConsole {
    public class QuantumTest {
        // This is a simple example of an entangled state.
        // Firstly, we put the qubits into that state.
        // After that, we measure only a single qubit.
        // It is entangled with the other qubits,
        // so this measurement affects also the other two qubits.
        public static void Main()
        {
            // get access to the instance of quantum computer
            QuantumComputer comp = QuantumComputer.GetInstance();

            // create register with initial value 0 and 3 qubits (|000>)
            Register x = comp.NewRegister(0, 3);

            // for the 0th to the 1st qubit
            // (0 - the least significant, at the bottom of the circuit)
            for (int i = 0; i < 2; i++)
            {
                // apply the Hadamard gate onto i-th qubit of the register x:
                comp.Hadamard(x[i]);
            }
        }
    }
}
```

Listing 8.1: A source code of a simple simulation, which we want to translate into a graphical quantum circuit.

```csharp
// apply the C-Not gate, for which the i-th qubit is a control bit
// and (i+1)-th qubit a target bit:
   comp.CNot(control: x[i], target: x[i + 1]);
}

// measure only the 2nd qubit:
x.Measure(2);
// this operation will affect also the other qubits
}}
```

Figure 8.3: The quantum circuit generated by QuIDE from the source code presented in Listing 8.1

8.5 Summary

In this chapter we presented one of the results of the study. It is a full simulation environment, with a rich set of supported functions.

During the study we implemented both the `QuIDE.dll` simulation library and the QuIDE simulator with rich Graphical User Interface, which uses `QuIDE.dll` for performing the simulations. In this chapter we presented the API of the `QuIDE.dll` library and demonstrated the GUI of QuIDE simulator application. We listed also the quantum algorithms and common subroutines implemented on QuIDE. Next, we demonstrate an example application run on QuIDE.

The QuIDE simulator is publicly available at the QuIDE project website.\(^5\) The QuIDE simulator, as well as the `QuIDE.dll` can be downloaded and are ready to use. Their documentation (API Reference for `QuIDE.dll` and user manual for GUI) is also available there.

All of the functional and nonfunctional requirements were successfully implemented (they were described in Chapter 5). Therefore, the next step was to evaluate this newly

created software. We decided to carry out the functionality, usability and performance evaluation. In the next chapter we presents the results from these evaluations.
Chapter 9

Research Results

The goal of this thesis was to construct a quantum computer simulator which meet specific criteria. In this chapter we present the tests that were performed in order to check them, and the results achieved by the constructed simulator. The subsequent sections describe the functionality, usability and performance tests. In each of them, we first present evaluation methodology and criteria. Then, we show the results achieved by QuIDE. In all respects, we try to compare QuIDE with other, previously existed quantum computer simulators.

9.1 Functionality Evaluation

When we proposed a vision of a new quantum computer simulator in Chapter 5, we specified a novel set of features that it should provide for users. This section shows explicitly, what features are provided by QuIDE. Next, it presents, which of them are supported also in other quantum computer simulators. The result of this evaluation is a detailed comparison between QuIDE and other quantum simulation software.

9.1.1 Comparison of the Quantum Computer Simulators

QuIDE provide a rich set of features that are partially inspired by the functions of previously existed quantum computer simulators. Thus, it could seem unclear, why it has been built — if these features existed already in other software.

This section presents a comparative review of many quantum computer simulators. The review focuses on the features provided by this software. It shows, that the set of functions offered by QuIDE is truly rich and none of previous quantum computer simulators is endowed with all of them.

In addition, this comparison can be helpful, if one want to use a simulator other than QuIDE. This section presents also the important information about the supported operating systems, capabilities of the simulators or the documentation quality. Since
the number of such simulators is still growing and since some of them become outdated, it is important to prepare such comparison.

We had chose a set of quantum computer simulators. We decided to choose a software which meet following criteria:

- **It is a quantum computer simulator**, not a simulation software which is able only to illustrate single quantum phenomenon. For example, there are many simple software capable of simulating BB84 protocol [72]. They are undoubtedly a quantum simulation software, but not a quantum computer simulator.

- It is broadly available and free of charge. For this reason this comparison does not cover MATLAB® toolboxes and similar software.

- Author was capable of downloading and installing the software on Windows or Linux platform. Unfortunately, there are many project which are now obsolete, improperly built and distributed, unfinished or poorly documented. Those projects had to be omitted, too.

There are a vast amount of quantum computer simulators. Even after filtering them in terms of presented criteria, the number of such programs is still to big for the purpose of this comparison. Thus, the set of presented software is not complete. We attempted to choose the most powerful and usable tools. We wanted to present various tools and methods for simulating quantum computation and to locate our simulator (QuIDE) amongst them.

### 9.1.2 Results — A Comparison

We defined a set of qualitative and quantitative measures, which then were used to rate all the software. They were divided thematically into several groups.

1. **Technical information**
   The first group of information concerns the technology and supported platforms. The comparison is presented in Table 9.1.

2. **Interfaces**
   The second group describe the interfaces provided by the simulators. The comparison of the interfaces is shown in Table 9.2.

3. **Features included in the Requirements Specification**
   The third feature group concerns the features which formed the key requirements for the new simulator at the beginning of this study. The first four includes the ability to easy switching between graphical circuit representation and the source code. The points 3.5 and 3.6 are very important for analyzing quantum algorithms — the run-time preview of the quantum state and the step execution mode. The
point 3.7 is an important improvement of usability and user experience — by defining reusable blocks the time for building quantum circuits can be greatly saved. The last feature — mixing quantum and classical computation — is extremely important, because the most of currently known quantum algorithms perform also classical pre- or post-computations. The comparison covering these features is presented in Table 9.3.

4 Quantum computer capabilities
The forth batch of features is strictly related to quantum computation. They are based on Quantum Computation Theory, which is broadly described in Chapter 2. The first criterion verify the completeness of the quantum computer simulator. It check whether the simulator provides a set of elementary gates which is sufficient to construct every theoretically possible quantum computation [63]. Next, we check whether any gate can have added a control bit (an how many of them). Then we ask if it is possible to build any unitary multi-qubit gate. The next criterion shows what is the maximum number of quantum registers which can be used in a single simulation. The last issue is the presence of the Shor’s Factorization algorithm implementation. The positive answer in this point demonstrates the robustness of the simulator — since this is a complicated algorithm which proves the strengths of quantum computers. The comparison is presented in Table 9.4.

5 More detailed information
This study involves also a comparison of more specific attributes of the quantum computer simulators. We check which quantum gates are supported as built-in, ready to use blocks. Next, the quality of software documentation is evaluated. We check whether the example circuits are provided. We list, which quantum algorithms are implemented and can be directly run. Some other, more detailed usability issues are also checked. All this information is presented in Appendix A.

QuIDE Results
Concerning the group of Technical Information, QuIDE can be viewed as worse than some other simulators — because of the support of only the Microsoft Windows platform. This observation was an important hint for the future directions for QuIDE — it is planned to be changed into a web solution.

The unique set of exposed interfaces is a great advantage which puts QuIDE ahead of the other simulators. QuIDE provides both Graphical User Interface and the Application Programming Interface. Furthermore, they can be used simultaneously at run-time and exchanged at any time. It is unique feature, not supported by any of the remaining simulators.

In the field of requirements which were specified for a new quantum computer simulator, QuIDE shows the best results again. It meets all of these requirements, unlike any other existing simulator. This comparison explains, why QuIDE was created — the features
listed in this group were the functions missing in existing tools when they were reviewed at the beginning of this study.

The group of quantum computer capabilities is a set of usable features, which were observed during the review of existing simulators at the beginning of this study. Therefore, they are well supported by some of the existing simulators. Nevertheless, none of those simulators supports all of them. The most problematic issue here is the implementation of the Shor’s Factorization Algorithm [52]. Since this algorithm is very complex, only the most advanced simulators provide it. Thanks to the work done by Bartłomiej Patrzyk [53], QuIDE has this algorithm implemented and therefore it fulfills all of these requirements.

The more detailed features whose comparison is presented in Appendix A, are well supported by QuIDE and also by some other more advanced simulators. There are however some simulators, which lack of very important elements in this group — for example, they do not provide English documentation.
Table 9.1: Technical information about various quantum computer simulators and the QuIDE simulator.

<table>
<thead>
<tr>
<th>Simulator name</th>
<th>1.1. Technology of design</th>
<th>1.2. Supported operating systems</th>
<th>1.3. Last actualization date</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuIDE</td>
<td>C#, .NET WPF</td>
<td>Windows</td>
<td>2014</td>
</tr>
<tr>
<td>QCAD200 [34]</td>
<td>C++, .NET</td>
<td>Windows</td>
<td>2011</td>
</tr>
<tr>
<td>jQuantum [37]</td>
<td>Java</td>
<td>All with Java</td>
<td>2010</td>
</tr>
<tr>
<td>SimQubit [36]</td>
<td>C++, .NET</td>
<td>Windows</td>
<td>2005</td>
</tr>
<tr>
<td>qMIPS [29]</td>
<td>Java</td>
<td>All with Java</td>
<td>2013</td>
</tr>
<tr>
<td>Qubit101 [38]</td>
<td>Java</td>
<td>All with Java</td>
<td>2013</td>
</tr>
<tr>
<td>Zeno [39]</td>
<td>Java</td>
<td>All with Java</td>
<td>2006</td>
</tr>
<tr>
<td>Cove [23]</td>
<td>C#, .NET WPF</td>
<td>Windows</td>
<td>2009</td>
</tr>
<tr>
<td>libquantum [15]</td>
<td>C</td>
<td>All</td>
<td>2013</td>
</tr>
<tr>
<td>CHP [30]</td>
<td>C</td>
<td>All</td>
<td>2005</td>
</tr>
<tr>
<td>LanQ [32]</td>
<td>Java</td>
<td>All with Java</td>
<td>2007</td>
</tr>
<tr>
<td>Q++ [21]</td>
<td>C++, .NET</td>
<td>Windows (probably also other)</td>
<td>2005</td>
</tr>
<tr>
<td>QCL [31]</td>
<td>C++</td>
<td>Linux</td>
<td>2014</td>
</tr>
<tr>
<td>QuIDDPro [12]</td>
<td>C++</td>
<td>Linux</td>
<td>2011</td>
</tr>
<tr>
<td>Javascript QC Simulator [41]</td>
<td>Javascript, HTML, CSS</td>
<td>All</td>
<td>2013</td>
</tr>
<tr>
<td>Online QC Simulator [40]</td>
<td>Python</td>
<td>All</td>
<td>2010</td>
</tr>
</tbody>
</table>
Table 9.2: Results of the functionality evaluation. A comparison of the interfaces exposed by various quantum computer simulators and the QuIDE simulators. The possible interfaces are Graphical User Interface (GUI), Application Programming Interface (API) and Command Line Interpreter (CLI).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1. GUI</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2.2. API</td>
<td>Yes(^1)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes(^2)</td>
<td>No</td>
<td>Yes(^3)</td>
<td>Yes(^4)</td>
<td>Yes(^5)</td>
<td>Yes(^6)</td>
<td>Yes(^7)</td>
<td>Yes(^8)</td>
<td>Yes(^9)</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>2.3. CLI</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

\(^1\) C#  
\(^2\) proprietary, similar to Assembler  
\(^3\) C#  
\(^4\) proprietary, similar to MATLAB  
\(^5\) QCL — proprietary, similar to C  
\(^6\) proprietary, similar to MATLAB  
\(^7\) QCL — proprietary, similar to C  
\(^8\) proprietary, similar to MATLAB
Table 9.3: Results of the functionality evaluation. A comparison covers the features listed in the Requirements Specification (see Chapter 5). The points 3.1. and 3.2. show, what are the methods for designing a quantum circuit. The criteria 3.3. and 3.4. check whether it is possible to switch between graphical circuit representation and a source code. The point 3.5. stands for the possibility to preview an internal quantum state during the simulation. The feature 3.6. enable users to execute the simulation in a stepping mode. The criterion 3.7. 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>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1. Source code edition</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Implementable — the user could insert the printing command into the source code.</td>
<td></td>
</tr>
<tr>
<td>3.2. Graphical circuit designing</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Implementable — the user could write a program able to perform computations interactively, step by step.</td>
<td></td>
</tr>
<tr>
<td>3.3. Source code ⇒ Graphical circuit</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Implementable — the user could insert the printing command into the source code.</td>
<td></td>
</tr>
<tr>
<td>3.4. Graphical circuit ⇒ Source code</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Implementable — the user could write a program able to perform computations interactively, step by step.</td>
<td></td>
</tr>
<tr>
<td>3.5. Run-time preview</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Implementable — the user could insert the printing command into the source code.</td>
<td></td>
</tr>
<tr>
<td>3.6. Stepping mode</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Implementable — the user could write a program able to perform computations interactively, step by step.</td>
<td></td>
</tr>
<tr>
<td>3.7. Reusable subroutines</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Implementable — the user could insert the printing command into the source code.</td>
<td></td>
</tr>
<tr>
<td>3.8. Mixing classical &amp; quantum</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Implementable — the user could write a program able to perform computations interactively, step by step.</td>
<td></td>
</tr>
</tbody>
</table>
Table 9.4: Results of the functionality evaluation. A comparison shows the supported quantum computer capabilities. In the point 4.1, we ask, whether the simulator provides a complete set of elementary quantum gates [63], which allows to build any theoretically possible quantum circuit. The criterion 4.2. checks whether any quantum gate can have added a control bit(s). The point 4.3. stands for the possibility to construct any arbitrarily defined multi-qubit gate. In the point 4.4. we show, how many quantum registers can be used in a single simulation. In the last point we ask whether the Shor’s Factorization Algorithm [52] is implemented. Since it is considered the most powerful and complicated quantum algorithm, this criterion verify the robustness of the simulators.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1. Complete set of gates</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>4.2. Control bit to any gate</td>
<td>Yes(^{12})</td>
<td>No</td>
<td>No</td>
<td>Yes(^{13})</td>
<td>Yes(^{14})</td>
<td>Yes(^{14})</td>
<td>Yes(^{14})</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes(^{13})</td>
<td>No</td>
<td>Yes(^{13})</td>
<td>Yes(^{13})</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>4.3. Custom multi-qubit gates</td>
<td>Yes(^{15})</td>
<td>No</td>
<td>No</td>
<td>Yes(^{16})</td>
<td>No</td>
<td>Yes(^{15})</td>
<td>No</td>
<td>Yes(^{15})</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes(^{15})</td>
<td>Yes(^{15})</td>
<td>Yes(^{15})</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>4.4. Quantum registers number</td>
<td>Many</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Many</td>
<td>Many</td>
<td>1</td>
<td>1</td>
<td>Many</td>
<td>Many(^{17})</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4.5. Shor’s Factorization</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No(^{18})</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

\(^{12}\)typically one, but in some cases many (for example in the unbounded Toffoli gate)
\(^{13}\)many
\(^{14}\)one
\(^{15}\)User can build any multi-qubit gate by grouping smaller gates.
\(^{16}\)User can define any \(n\)-qubit gate by defining its \(2^n \times 2^n\) unitary matrix.
\(^{17}\)User can operate on many quantum registers, but they altogether can have no more than 64 qubits.
\(^{18}\)The implementation for \(N = 15\) is broken. There is no other implementation.
9.2 System Usability Evaluation

As described in Chapter 5, the simulator is required to provide simple and intuitive user interface. These values seems unclear and hard to measure. However, the problem of measuring the usability of a software has today several proven and reliable solutions. One of them is the System Usability Scale (SUS) [81]. This section presents the results of the SUS surveys concerned QuIDE.

9.2.1 QuIDE Implementation in the Quantum Computation Theory Course

The usability evaluation was not only a short experiment. QuIDE was chosen to be the main simulation software in the course of Quantum Computation Theory for the 1st-year students of the second-cycle studies of Computer Science at AGH University of Science and Technology. In past years, students were working on the libquantum library [15]. In 2014, each lesson and exercise was based on QuIDE. The course was named as ‘Matematyka w informatyce przyszłości’. The information about it is available at the Syllabus AGH website.\(^\text{19}\) The course exercises and assessments are described at the course’s website.\(^\text{20}\)

The course was attended by over 100 students. They were asked to fill in the SUS forms. Moreover, during this half-year course they were asked for opinions about the new simulator. They proposed many features and some of them was immediately implemented. The rest of these functions are considered to be added in the future. This feedback was gained unofficially and through following four questions, which were enclosed to the SUS forms.

1. What functions do you find the most useful?
2. What in the simulator do you find badly designed?
3. What should be changed?
4. What features are missing?

The students were a perfect group of demanding users, which gave a great feedback about the simulator. What is more, this group was able to detect software bugs during the half-year working with the simulator. The errors was corrected as soon as it was possible. After that, the simulator, which is now available to download from the Internet, seems well-tested and reliable.


9.2.2 Evaluation Methodology

The SUS is a set of 10 simple questions addressed to the users of the system. They have to give an answer in Likert scale [82] — a 5-point scale of agreement, from 'Strongly disagree' to 'Strongly agree'.

In the evaluation, the original set of questions was used. However, two more questions were added. They are strictly related to the main goals which the simulator was required to fulfil. The full set of 12 question statements is presented in Appendix B.

These questions were asked at the beginning and at the end of the course. Moreover, in the initial SUS survey the same questions were asked about the libquantum library. The students were able to rate libquantum because one of the first lab exercises included working on libquantum. Then, all of the lessons and exercises were based on QuIDE. At the end of the course the students were surveyed again, however only about QuIDE.

9.2.3 Results

The Figure 9.1 presents the results of the initial SUS survey. It was carried out after the first lab exercises, which include getting started with the simulation software. Students were obliged to prepare a very simple quantum circuit and explain how it operates. They could use the libquantum library or the QuIDE simulator. Then, the students which tries both of them completed the SUS questionnaire.

As shown in Figure 9.1, QuIDE gained better rates at every aspect of the SUS evaluation. For every question it achieved averagely a 1-point-higher rate than libquantum, in 5-point scale. It is very satisfying result, since the libquantum library is very powerful simulation tool, with rich and simple API and greatly documented. libquantum was the main simulation tool used in this course in past years, because it was ahead of the remaining quantum computer simulators with their capabilities, flexibility, performance and documentation support. However, libquantum provides neither graphical interface nor command line interpreter. We assume that it is the main reason why it was distanced by QuIDE.

It could seem unclear, why we compared GUI-based QuIDE and a programming library libquantum. It would be better to compare two GUI-based simulators. The answer is given in the previous section, which describes the functionality comparison of the quantum computer simulators. In fact, the set of functions offered by the GUI-based simulators was insufficient for the course purposes. Those simulators are mostly presentational tools with poor flexibility and limited functionality. The course lessons could be not based on them as simulation software or it would be very complicated. The QuIDE simulator is not only a simple presentational tool. By supporting source code edition it can be used in professional academic courses about Quantum Computation Theory.

The Figure 9.2 shows the results from the second SUS survey, compared to the previous results. The second survey was carried out at the end of the half-year course. Since in
Figure 9.1: Results of the SUS survey which was carried out at the beginning of the Quantum Computation Theory course for the 4th-year Computer Science students. They were asked to rate usability of QuIDE and its predecessor, \texttt{libquantum}. The figure shows the average score obtained for each question, with their standard deviation (the questions are listed in Section 9.2.2). The scores ranges from 0 (the worst rate) to 4 (the best). The ratings are received from a group of 31 students.

all lessons the simulations were performed on QuIDE, only the QuIDE was evaluated. The set of questions remained the same. The answers were collected from the group of 57 students.

We can see, that in the second evaluation QuIDE gained slightly worse rates. The biggest decrease can be noticed for the first question (‘I think that I would like to use this system frequently’). However, this result was expected, since the course was ending and students apparently did not plan to work with quantum computer simulators in the nearest future. Nevertheless, all remaining aspects were also rated worse. The most probably reason for it is that the students got used to the comfortable QuIDE interface and became more demanding. What is more, they no longer had to deal with \texttt{libquantum}, which was much harder to use. If they had to work with these two tools simultaneously during the whole course, the QuIDE rates could even be higher than in the first survey. On the other hand, the collected results remained very good, and still were higher than the rates for \texttt{libquantum}.

In the Figure 9.3 we present a total SUS scores from the two SUS surveys. They are obtained by summing the rates for all questions and then by scaling the result to the range from 0 to 100. The figure shows the average scores and their standard deviations. The best score were obtained by QuIDE in the initial survey, when users had to compare it with the second tool, \texttt{libquantum}. The rate gained by QuIDE in the second survey is slightly worse, but still better than the result of the \texttt{libquantum} library.
Figure 9.2: Results of the second SUS survey, which was carried out at the end of the Quantum Computation Theory course. The results from the beginning of the course are shown in Figure 9.1. The students were asked to answer the same questions about the usability of QuIDE as in the initial SUS survey. The figure shows the average score obtained for each question, with their standard deviation. The scores ranges from 0 (the worst rate) to 4 (the best). The answers are collected from the group of 57 students.

Figure 9.3: The average total SUS score achieved by 	exttt{libquantum} and QuIDE in the two SUS surveys (presented in figures 9.1 and 9.2). The figure shows the average scores and their standard deviations. Results were obtained by summing the scores for all 12 questions and scaling the result to range from 0 (the worst) to 100 (the best).
9.3 Performance Tests

During this study we focused on the functionality and the usability of the new simulator. However, no tool with poor performance results can be considered to be usable. In other words, our goal was to prepare a simulator of good performance, which allow to use all of its features comfortably. This section shows the performance tests that were carried out, the results achieved by QuIDE and their relation to results of the some other simulators.

9.3.1 Evaluation Methodology

As described in Chapter 4, the quantum mechanical system cannot be effectively simulated on classical computer [5, 17, 18]. The main barrier is encountered when we want to simulate more than 20 qubits (see table 4.1). The simulation time is not so important, because the Achilles heel is the memory consumption. Thus, the performance tests focuses mainly on the capability of simulating a number of qubits.

The first evaluation is a comparison of the simulators from the Section 9.1. For each of them, the two following aspect were measured:

- What is the maximum number of qubits that can be simulated? We consider the best case.
- What is the maximum number of qubits that can be simulated in the worst case?

The worst case from the second question concerns the situation where the simulator has to use maximum RAM memory. Usually it is when qubits are in perfectly mixed state (when each basis state is equally probable).

The second performance test included measuring the amount of the memory used by the simulators in the worst case. For the test we chosen one GUI-base simulator (jQuantum) and one programming library (libquantum). They were compared to QuIDE and the QuIDE.dll — a programming library which was used in simple console program without GUI. QuIDE.dll is a core simulation library, used by the GUI-based QuIDE simulator. It can be used independently, as standalone .NET library and thus can be compared to libquantum. In each of this four simulators we prepared a fully mixed state of the given number of qubits, and then we measured the memory usage.

The next test measured the time of an example simulation performed by the QuIDE simulator. We chose Grover’s Fast Database Search Algorithm [7, 49] and measured the simulation time of its single iteration. However, we could not compare the results with the other simulators. Many of them do not provide an implementation of this algorithm. The others provide implementations which differ from each other and from the QuIDE’s implementation — for example, they use different quantum gates. The same situation was with implementations of other example algorithms. However, as we explained before, the simulation time is not a key performance issue. The exponential growth of memory usage limits the simulation opportunities much more strictly.
Each of the simulators was run on the same computer, a laptop with the 2-core 4-thread 2.50 GHz Intel Core i5-2520M processor and the 8 GB of RAM memory. They were run in one of the two operation systems: 64-bit Microsoft Windows 7 or 64-bit openSUSE Linux 13.1.

9.3.2 Results

The results from the first test are presented in Table 9.5. They describe the maximum number of qubits which can be simulated in the best and in the worst case.

<table>
<thead>
<tr>
<th>Simulator Name</th>
<th>Maximum number of qubits simulated in the best case</th>
<th>Maximum number of qubits simulated in the worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuIDE</td>
<td>no limits</td>
<td>23</td>
</tr>
<tr>
<td>QuIDE.dll</td>
<td>no limits</td>
<td>26</td>
</tr>
<tr>
<td>QCAD200</td>
<td>no limits</td>
<td>15</td>
</tr>
<tr>
<td>jQuantum</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>SimQubit</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>qMIPS</td>
<td>32</td>
<td>16</td>
</tr>
<tr>
<td>Qubit101</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Zeno</td>
<td>24</td>
<td>20</td>
</tr>
<tr>
<td>Cove</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>libquantum</td>
<td>no limits</td>
<td>27</td>
</tr>
<tr>
<td>CHP</td>
<td>over 10 000</td>
<td>over 10 000</td>
</tr>
<tr>
<td>LanQ</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Q++</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>QCL</td>
<td>64</td>
<td>25</td>
</tr>
<tr>
<td>QuIDDPro</td>
<td>no limits</td>
<td>n/d</td>
</tr>
<tr>
<td>Javascript QC Simulator [41]</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Online QC Simulator [40]</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

It is noticeable, that one of the simulator, CHP, can simulate a huge amount of quantum bits. This however results from the fact, that CHP is the simpler simulator in the comparison. It provides the smallest set of quantum gates (see Appendix A) and this set is not complete — so this tool cannot simulate any theoretically possible quantum circuit. This small set of quantum operations can be effectively simulated on a classical computer.
Besides CHP, the other simulators have similar capabilities. The maximum number of qubits which can be simulated in the worst case does not exceed 27 — the result of *libquantum*. This powerful library is designed in pure C and makes a use of MPICH functions, which puts it ahead of other simulators. However, QuIDE, when used as standalone library, is only slightly worse with the result 26. The full, GUI-based QuIDE simulator is yet a bit worse. It is caused by additional data structures used by GUI. This result is however actually good, since the majority of the other GUI-based simulators achieve much worse results.

In the best case, the good result is when the number of qubits is not limited. It allows to perform simulations of a great number of qubits, which are not in mixed state. This is a common case. Even in Shor’s algorithm there are many such a auxiliary qubits. Limiting the total number of qubits, even if they are in a single pure state, narrows down the range of simulations which can be performed.

All these results are caused by inner representation of quantum computations. It can be assumed, that the simulators with much worse scores uses a non-optimal method. This comparison confirms the fact, that the performance results are influenced by inner data representation as much as stated in Chapter 4.

### Memory usage

The Figure 9.4 shows the amount of memory used by four quantum computer simulators in relation to the number of simulated qubits. This test shows the results for the worst case, which means that the qubits were in a fully mixed state. We obtained such conditions by applying the Hadamard gate to every qubit. The tested simulators are: QuIDE (with GUI), QuIDE.dll (standalone C# library used with simple, command-line program), *libquantum* (C library used on the Linux system) and *jQuantum* (GUI-based simulator written in Java). The results are shown on a logarithmic scale.

As we can see in Figure 9.4, the memory consumption grows exponentially for all of the tested simulators. This result confirms that the space complexity of such simulations cannot be less than exponential, as shown in Chapters 4 and 7.

Interestingly, the lines on the chart for the number of qubits higher than 20 are geometrically parallel. It means, that the exponent $k$ of the complexity $O(2^k)$ is the same for all of these four simulators. In fact, all of them 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 $\alpha$ is a constant different for every simulator. Approximately, this function can be expressed in the form $y = \alpha 2^x$. This result proves, that the space complexity for each of these simulators equals exactly $O(2^n)$ where $n$ is a number of qubits. Thus, the complexity assessment shown in Chapter 4 and Chapter 7 is confirmed.

For small number of simulated qubits, the results are similar and do not grow exponentially. It is caused by the fact, that the auxiliary data structures used during
Chapter 9. Research Results

Figure 9.4: The amount of memory used by the simulators for the worst case — when the qubits are in a fully mixed state. The results are shown on a logarithmic scale. For the 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 \( \alpha \) 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 equal \( O(2^n) \).

the execution of a simulation need more memory than the representation of a \( n \)-qubit quantum state.

In the comparison, the best results are achieved by libquantum. It is a robust and efficient library written in pure C language. Thus, it was expected to win this comparison. However, the Quantum.dll is only a bit worse for the higher number of qubits. Moreover, it is better when it simulates a small amount of qubits. As expected, the GUI-based simulators lag behind the standalone simulation libraries. For smaller amounts of qubits the jQuantum gives the worst results. It is caused by the additional memory needed by the Java Virtual Machine. For higher number of qubits, the QuIDE simulator uses the most memory, which is its drawback. On the other hand, it is able to simulate greater number of qubits than jQuantum. Such a memory consumption results from a big set of features which are offered by QuIDE. Especially, the detailed run-time preview of the internal quantum state needs additional data structures which inflate the total memory usage.

Simulation Time

This test measured the time complexity of an example simulation performed by QuIDE. For the test we chose the Grover’s Fast Database Search Algorithm [49]. It consists of about \( \sqrt{n} \) repeated iterations. A single one of them has the complexity \( O(1) \) on the theoretical quantum computer. In this test we always tested the time of a single iteration.
By changing the search space, we tested this algorithm on different numbers of qubits (from 4 to 23). In each case we measured the time of a single iteration of the algorithm. The Figure 9.5 shows these times in relation to the number of simulated qubits.

![Simulation Time of a single iteration of Grover’s Fast Database Search Algorithm](image)

**Figure 9.5:** The simulation time of a single iteration of Grover’s Fast Database Search Algorithm [49]. 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 expected $O(2^n)$ time complexity is thus confirmed.

For the smaller numbers of qubits, the simulation times are close to zero. It is great for presentational purposes — for example, in the literature the Grover’s algorithm is presented often on plus/minus four qubits. Thus, one of the main goals of this study has been achieved — the proposed simulator can be an educational and presentational tool and is able to perform simple simulation quickly.

The times for the number of qubits more than 8 form a straight line, which means that the time complexity is exponential. This line can be approximated by a function $y = \alpha e^{0.70x}$, where $y$ is the simulation time and $x$ is the number of qubits. This function is approximately equal to the function $y = \alpha 2^x$. Thus, the time complexity of a single iteration of the Grover’s algorithm is $O(2^n)$ for $n$ qubits. In a theoretical quantum computer, the time complexity of the single Grover’s algorithm iteration depends on the quantum gates used to build it [7, 49, 54]. If they were the same as in the implementation run by QuIDE, the complexity would equal $O(1)$ — so, if one used the same quantum gates as in QuIDE, the number of steps of single Grover’s iteration would not depend on $n$. The result $O(2^n)$, achieved in this test by classical computer, confirms the complexity analysis from Chapter 7. As it was shown, in the worst case (fully mixed quantum state) a single operation performed by a quantum computer can be simulated with the $O(2^n)$ time complexity. Since in the Grover’s algorithm we work with such mixed states, this estimation is confirmed.
9.4 Summary

In this chapter we presented the evaluation of the QuIDE simulator. First of all, we thoroughly evaluated its functionality and compared it with a relatively big set of other simulators.

Then, the System Usability Scale (SUS) surveys were utilized to verify the usability of this tool. This evaluation was performed on a group of students attending an academic course of quantum computations, where QuIDE was used. The students rated also the previously used libquantum simulation library. The opinions about QuIDE and libquantum were gathered and compared.

Next, the performance tests were carried out. First of all, we compared the capabilities of QuIDE and other simulators in terms of maximum possible number of simulated qubits. Then we measured the memory consumption and the time needed to simulate different amounts of qubits.

The functionality evaluation showed, that all of the required features were properly implemented. What is more, it revealed, that QuIDE provides an unique and very rich set of features, which is not supported by other simulators.

The SUS surveys proved the good usability of the QuIDE simulator. Its resulting rate was distinctly higher, than the rate obtained by the previously used simulation tool.

Also the performance tests verified QuIDE positively. The QuIDE simulator turned out to be more memory-consuming than other two tested tools, but its was able to simulate more qubits than any other GUI-based simulator. What is more, the performance tests gave expected results, which proved the complexity analysis from Chapter 7.
Chapter 10

Conclusion

In this chapter we summarize the thesis. First of all, we describe, how its goals was realized. Then we summarize its results and explain its contribution. Next, the lessons learned are described. Finally, we outline the ideas for the further work.

10.1 Goals Achieved

The aim of this thesis was to propose, implement and evaluate a novel tool for performing the simulations of a Quantum Computer. In the Introduction of this thesis we listed all the goals chosen to be achieved in order to make this study complete. In this section we review them and show, how they were fulfilled.

Requirements specification

We had to consider the practical use cases of a quantum computer simulator and specify the functional and nonfunctional requirements for it.

The vision of requirements was daftly developed while the author was attending the course of Quantum Computation and Information Theory. Then they were specified in details, by cooperating with the teachers of this course and other attendants. This part of study is presented in Chapter 5.

Existing simulators review

The next step was to review existing software of this type — in order to find out, if such a new tool was actually needed. The review was expected to investigate existing methods for simulation quantum computations. Moreover, it was required to form a comparison of existing simulators and evaluation of their capabilities.

First of all, the quantum computer simulators were classified in terms of the types of external interfaces which they provided. We discussed the advantages and drawbacks of each of these classes, which is shown in Chapter 3. Next, we presented the core simulation techniques used by these tools — we described the inner data structures and algorithms used for simulating the evaluation of a quantum state.
These methods are presented in Chapter 4. Finally, we prepared a comparison and evaluation of capabilities offered by these simulators. The results are shown in Chapter 9 and Appendix A.

**Design and implementation of a quantum computer simulator**

*We had to build a new simulator, which fulfill the specified requirements. It was expected to provide an innovative interface — an Integrated Development Environment (IDE) for designing, executing and analyzing quantum computations. The most important and desired feature of this IDE was the support of both source code edition and graphical quantum circuit designing, with the ability to effortlessly switch these two representations at run-time.*

We successfully created a rich simulation tool — QuIDE (for Quantum IDE). We built both a core simulation library, QuIDE.dll, and the GUI-based simulation environment — the QuIDE simulator. Its functions, inner architecture, core implementation details and example applications are presented in Chapters 5, 6, 7 and 8, respectively. As shown in Chapter 9, QuIDE proved that it met all the functional and nonfunctional requirements.

**Functionality and performance evaluation**

*The goal of that stage was to verify, whether the QuIDE simulator met the functional and nonfunctional requirements. It had to be also compared with other tools of this type.*

At first, we evaluated the functionality of a group of simulators, including QuIDE — we prepared a list of features and checked, whether they were supported. That work resulted in a detailed comparison of a big group of simulators — it is presented in Chapter 9 and Appendix A.

During the performance tests, the simulation time and memory usage were measured. We also compared QuIDE with other simulators. The results are presented in Chapter 9.

**Deployment and validation in an academic course**

*We planned to use QuIDE in an academic course concerning quantum computations, and to gather the opinions about this tool. We also had to validate its usability.*

The QuIDE simulator was the main simulation tool in the Quantum Computation Theory course, attended by over 100 students. It was able to simulate all the most important quantum algorithms. During that course, we gathered feedback from the users. It resulted in adding new useful features and fixing all noticed bugs. Users opinions showed also many improvements that could be added in the future. We also evaluated the usability of QuIDE, by carrying out a survey amongst the course attendants. They rated QuIDE and the previously used simulator. The results of those surveys are presented in Chapter 9.
10.2 Results of the Study

By realizing the goals of the thesis, we achieve several important results. The main of them are a detailed, comparative review of existing quantum computer simulators and a new simulator which provides a novel, usable environment for simulating quantum computations. The review, the new simulator, the ideas which it realizes and the evaluation of the simulator and these ideas — all these results forms an important contribution of this thesis.

Review of Existing Simulation Tools

This review outlined the main problems with such simulations — especially their exponential space and time complexity. Moreover, it helped in specifying more detailed requirements for the new simulator and sketched a vision for fulfilling them. What is the most important, it resulted in a detailed comparison of the existing software of this type. Their functionalities, interfaces and performance capabilities were reviewed and evaluated. These findings can be helpful for those who need to choose a one of them.

Novel Simulation Environment

In this thesis we proposed and implemented a new tool for simulating quantum computations. It owes its innovative character to the following features:

- support of both code edition and graphically building quantum circuit diagrams
- a possibility to translate the source code into the interactive graphical circuit, and vice versa
- a great support for building reusable computational blocks and extending the initial set of quantum operations.

These new features, and the simulator as a whole, turned out to be very useful. The simulator proved its usability during an academic course, where all of its features were widely exploited. This evaluation of the proposed new features is one of the biggest contribution of this study. It gives important hints for future projects of this type. Moreover, some of these features can be also implemented in the software other than another quantum computer simulator — for example in other simulation tool.

10.3 Lessons Learned

The study showed a great value of the idea of transition between the graphical model and the source code which it represents. Nowadays, this approach is becoming more and more popular, and this study proved its usability once again.
The simulator is supported only on the Microsoft Windows system. Although it enables it to be run on nearly 90% desktop computers [83], in today world of Web solutions such a software seems out of date. We do not state, that we have chosen a bad technology — our goal was to test a rich set of advanced features and the chosen technology allowed us to implement all of them effectively. However, after evaluating all these features, now there is a time for transfer them into a Web-based solution.

We learned also, how important is the inner representation of quantum computations for the performance of the simulation software. When the complexity of a problem is so big (exponential in this case), even small optimization could lead to big performance improvement.

10.4 Project Status

The simulator was chosen to be a basic simulation tool in the course ‘Matematyka w informatyce przyszłości’ — a half-year course concerning the Quantum Computation and Information Theory for first-year students of the second-cycle Computer Science studies at the AGH University of Science and Technology.1 The course was attended by over 100 students.

QuIDE was capable of simulating all of the quantum algorithms presented in the course. The group of users was big enough to detect most of the errors and misbehaviors of this software. They were fixed straightaway after being detected. Also, the users proposed many ideas of possible improvements. Some of them (mostly the simpler ones) were also implemented. The other of these ideas are planned to be implemented in the future versions of this project.

To sum up, QuIDE is a complete software tool, able to simulate various quantum algorithms, including the most demanding ones. It is tested and void of a vast majority of faults. Its usability evaluation gave really good results. Its application on an academic course showed, that it is a good tool for such purposes.

The QuIDE simulator is publicly available at the QuIDE project website.2

10.5 Future Directions

The most desirable improvement of QuIDE would be to make it more portable. Currently the simulator can be run only on the Microsoft Windows, what is its main drawback. The optimal solution would be to change it to a Web application, which would need only an Internet browser to run.

Another interesting solution which could be considered, is to port the implementation of the core simulation library (QuIDE.dll) into a concurrent, high-performance architecture. As a result, the more complex simulations would be possible. If the simulator was also equipped with a Web interface, the simulations could be managed remotely and executed on some powerful infrastructure.

The Web interface could give the user the choice: whether he want to perform a simple, presentational simulation and use for it only the client-side resources, or does he need to reserve a remote, high-performance infrastructure and perform on it some complex simulations.
Appendix A

Further Comparison of Quantum Computer Simulators

In this appendix we continue a comparison which was begun in Chapter 9. It presents the more detailed information about a set of existing simulators. The QuIDE simulator is also included in this comparison.

Comparison Criteria and Results

The criteria used for this further comparison concerns more detailed attributes of the simulators. These are:

1 List of built-in quantum gates
   List of quantum gates that are directly available, without the need for manual specifying their behavior.

2 Possibility to load and save constructed circuits
   This feature is only the matter of the usability and user experience. Nevertheless, without ability to save and load the built quantum algorithm, the simulator become hardly usable.

3 Documentation quality
   We check whether all of the functions are documented and whether the documentation presents examples which demonstrate them.

4 List of implemented example algorithms
   We list the example algorithms that are implemented and available to simulation. The presence of such examples is immeasurably important, especially for non-programmers or people wanted to learn about quantum computations. The listed quantum algorithms have only short names. They are briefly described in Chapter 2.
### Table A.1: List of built-in quantum gates.

<table>
<thead>
<tr>
<th>Simulator name</th>
<th>1. List of built-in quantum gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuIDE</td>
<td>H, X, Y, Z, C-Not, Toffoli, $R_x$, $R_y$, $R_z$, M, Sqrt(X), Phase Shift, Phase Scale, $R_k$, $R_k^{-1}$, any 1-qubit matrix-defined unitary gate, Fredkin</td>
</tr>
<tr>
<td>QCAD200 [34]</td>
<td>H, X, Y, Z, C-Not, Toffoli, Fredkin, $R_z$, C-$R_z$, M</td>
</tr>
<tr>
<td>jQuantum [37]</td>
<td>X, Y, Z, H, C-Not, M, $S$, $S^{-1}$, T, Sqrt(X), Toffoli, C-$R_z$, C-$R_y$, C-$R_z$ (rotation only in few possible angles, e.g. $\frac{\pi}{2}$), irreversible 2-register gate $y=f(x)$ (not unitary)</td>
</tr>
<tr>
<td>SimQubit [36]</td>
<td>H, X, Y, Z, S, T, -1, any n-qubit matrix defined unitary gate</td>
</tr>
<tr>
<td>qMIPS [29]</td>
<td>H, X, Y, Z, Phase Shift, M, Reset</td>
</tr>
<tr>
<td>Qubit101 [38]</td>
<td>H, X, Y, Z, Phase Shift, M, Reset</td>
</tr>
<tr>
<td>Cove [23]</td>
<td>H, X, Y, Z, C-Not, C-U (controlled matrix-defined 1-qubit unitary gate), Fredkin, Swap, Phase Shift, $R_x$, $R_y$, $R_z$, S, T, $R_k$, Reset, M</td>
</tr>
<tr>
<td>libquantum [15]</td>
<td>H, X, Y, Z, C-Not, n-qubit Toffoli, $R_x$, $R_y$, $R_z$, Phase Scale, Controlled Phase Shift, $R_k$, C-$R_k$, $R_k^{-1}$, any 1- and 2-qubit matrix-defined unitary gate, M</td>
</tr>
<tr>
<td>CHP [30]</td>
<td>H, C-Not, P (phase shift by $\frac{\pi}{2}$), M</td>
</tr>
<tr>
<td>LanQ [32]</td>
<td>H, X, Y, Z, C-Not, M</td>
</tr>
<tr>
<td>Q++ [21]</td>
<td>H, X, Y, Z, S, T, C-Not, Toffoli, Swap, $R_k$, any n-qubit matrix-defined unitary gate, M</td>
</tr>
<tr>
<td>QCL [31]</td>
<td>any matrix-defined 1-, 2- or 3-qubit unitary gate, C-Not, Swap, Controlled Phase Shift, $R_y$, X, n-qubit</td>
</tr>
<tr>
<td>QuIDDPro [12]</td>
<td>C-Not, Fredkin, H, X, Y, Z, $R_x$, $R_y$, $R_z$, Swap, Toffoli, Controlled Phase Shift, C-U (any n-qubit controlled gate)</td>
</tr>
<tr>
<td>Javascript QC Simulator [41]</td>
<td>H, X, Y, Z, S, T, Phase Shift, C-Not, Toffoli, Sqrt(X)</td>
</tr>
<tr>
<td>Online QC Simulator [40]</td>
<td>H, X, C-Not, Toffoli, C-Phase Shift</td>
</tr>
</tbody>
</table>
### Table A.2: Possibility to load and save constructed circuits, Documentation quality.

<table>
<thead>
<tr>
<th>Simulator name</th>
<th>2. Load/Save Support</th>
<th>3. Documentation Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuIDE</td>
<td>Yes</td>
<td>Very good: full GUI and API documentation with examples</td>
</tr>
<tr>
<td>QCAD200 [34]</td>
<td>Yes</td>
<td>Good: full GUI documentation, but lack of examples</td>
</tr>
<tr>
<td>jQuantum [37]</td>
<td>Yes</td>
<td>Very good: full GUI documentation with examples</td>
</tr>
<tr>
<td>SimQubit [36]</td>
<td>No (not working)</td>
<td>Very good: full GUI documentation with examples</td>
</tr>
<tr>
<td>qMIPS [29]</td>
<td>Yes</td>
<td>Moderate: features are explained very laconically</td>
</tr>
<tr>
<td>Qubit101 [38]</td>
<td>Yes</td>
<td>Moderate: features are explained very laconically</td>
</tr>
<tr>
<td>Zeno [39]</td>
<td>Yes</td>
<td>No english version</td>
</tr>
<tr>
<td>Cove [23]</td>
<td>Yes</td>
<td>Very good: full API documentation with examples</td>
</tr>
<tr>
<td>libquantum [15]</td>
<td>Yes</td>
<td>Very good: full API documentation with examples</td>
</tr>
<tr>
<td>CHP [30]</td>
<td>Yes</td>
<td>Very good: full API documentation with examples</td>
</tr>
<tr>
<td>LanQ [32]</td>
<td>Yes</td>
<td>Very good: full API documentation with examples</td>
</tr>
<tr>
<td>Q++ [21]</td>
<td>Yes</td>
<td>Good: full API documentation with examples</td>
</tr>
<tr>
<td>QCL [31]</td>
<td>Yes</td>
<td>Very good: full API documentation with examples</td>
</tr>
<tr>
<td>QuIDDPro [12]</td>
<td>Yes</td>
<td>Very good: full API documentation with examples</td>
</tr>
<tr>
<td>Javascript QC Simulator [41]</td>
<td>Yes</td>
<td>Moderate: explanation in blog post</td>
</tr>
<tr>
<td>Online QC Simulator [40]</td>
<td>No</td>
<td>No english version</td>
</tr>
</tbody>
</table>
Table A.3: List of implemented example algorithms.

<table>
<thead>
<tr>
<th>Simulator name</th>
<th>4. List of implemented algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuIDE</td>
<td>QFT, Adder, Modular Adder and Multiplier, Modular Exponentiation, Grover’s Search, Shor’s Factorization, Quantum Teleportation, Quantum Dense Coding, Deutsch Problem, Bernstein-Vazirani Problem</td>
</tr>
<tr>
<td>QCAD200 [34]</td>
<td>No example algorithm implemented</td>
</tr>
<tr>
<td>jQuantum [37]</td>
<td>QFT, Grover’s Search, Shor’s Factorization, Deutsch-Jozsa Problem, Adder, AND, OR, Bell states, GHZ state, W State</td>
</tr>
<tr>
<td>SimQubit [36]</td>
<td>Grover’s Search</td>
</tr>
<tr>
<td>qMIPS [29]</td>
<td>Grover’s Search, Deutsch-Jozsa Problem</td>
</tr>
<tr>
<td>Qubit101 [38]</td>
<td>Grover’s Search, OR, AND, XOR, Negation, Swap</td>
</tr>
<tr>
<td>Zeno [39]</td>
<td>Deutsch-Jozsa Problem</td>
</tr>
<tr>
<td>Cove [23]</td>
<td>QFT, Swap, Adder, Modular Adder, Shor’s Factorization (but only factoring $N = 15$)</td>
</tr>
<tr>
<td>libquantum [15]</td>
<td>QFT, Adder, Modular Exponentiation, Shor’s Factorization, Grover’s Search</td>
</tr>
<tr>
<td>CHP [30]</td>
<td>EPR, GHZ, Quantum Teleportation, Simon’s Problem, Quantum Dense Coding, Quantum Error Correction</td>
</tr>
<tr>
<td>LanQ [32]</td>
<td>BB84, Quantum Teleportation, Deutsch-Jozsa Problem, random number generator, simple communication</td>
</tr>
<tr>
<td>Q++ [21]</td>
<td>QFT, Grover’s Search</td>
</tr>
<tr>
<td>QCL [31]</td>
<td>Adder, Modular Adder, Modular Multiplier, Modular Exponentiation, QFT, Shor’s Factorization</td>
</tr>
<tr>
<td>QuIDDPro [12]</td>
<td>Grover’s Search, W State, Cat State, Shor’s Factorization</td>
</tr>
<tr>
<td>Javascript QC Simulator [41]</td>
<td>Bell state, 2- and 4-qubit QFT, Grover’s Search, Quantum Teleportation</td>
</tr>
<tr>
<td>Online QC Simulator [40]</td>
<td>No example algorithms implemented</td>
</tr>
</tbody>
</table>
Appendix B

SUS Questionnaire for QuIDE

System Usability Scale


Please rate these statements with the numbers 1 to 5, where 1 means ‘Strongly agree’ and 5 means ‘Strongly disagree’.

1. I think that I would like to use this system frequently
2. I found the system unnecessarily complex
3. I thought the system was easy to use
4. I think that I would need the support of a technical person to be able to use this system
5. I found the various functions in this system were well integrated
6. I thought there was too much inconsistency in this system
7. I would imagine that most people would learn to use this system very quickly
8. I found the system very cumbersome to use
9. I felt very confident using the system
10. I needed to learn a lot of things before I could get going with this system
11. I think that the system make it easier to understand the quantum computations
12. I think that the system is a good tool to design and analyse quantum algorithms
Appendix C

Papers

The results obtained in this thesis were published in the following paper. The author of this thesis is also co-author of the presented paper.

## List of Figures

1.1 BQP class in Quantum Complexity Theory .................................. 2

2.1 Quantum measurement graphical symbol ................................. 15

2.2 An example quantum circuit .................................................. 15

4.1 QuIDD Binary Decision Diagram ......................................... 29

4.2 Hash-Table representation of an example quantum state ........... 30

5.1 An example quantum circuit .................................................. 36

5.2 QuIDE and the other quantum computer simulators .................. 38

5.3 QuIDE proposal: source code and model representations switching . 40

5.4 QuIDE proposal: stepping mode and run-time preview ............... 41

5.5 QuIDE proposal: mixing classical and quantum operations .......... 42

5.6 QuIDE proposal: custom high-level computation blocks ............. 43

6.1 Model-View-ViewModel pattern ............................................. 46

6.2 QuIDE architecture components and their relations ................. 48

6.3 QuIDE object model for representing quantum computations ........ 51

6.4 QuIDE model of quantum operations types ............................. 52

7.1 QuIDE data structures in the core simulation module ............... 56

8.1 UML class diagram for the QuIDE.dll simulation library .......... 66

8.2 A view of the GUI of QuIDE ............................................... 67

8.3 An example quantum circuit generated by QuIDE ..................... 71
9.1 Results of the initial SUS survey ........................................ 83
9.2 Results of the second SUS survey ....................................... 84
9.3 Total results of the SUS surveys ......................................... 84
9.4 Performance results — The memory usage ......................... 88
9.5 Performance results — Simulation time .............................. 89
List of Tables

2.1 Common 1-qubit quantum gates ........................................... 13
2.2 Multi-qubit quantum gates ................................................. 14

4.1 Memory usage in naive matrix-vector representation ................. 26

9.1 Functionality evaluation — Technical information ..................... 77
9.2 Functionality evaluation — Interfaces .................................. 78
9.3 Functionality evaluation — Requirements specified .................. 79
9.4 Functionality evaluation — Quantum computer capabilities ........ 80
9.5 Performance evaluation — Maximum available number of qubits .... 86

A.1 Functionality evaluation — List of built-in quantum gates ........ 98
A.2 Functionality evaluation — Other questions .......................... 99
A.3 Functionality evaluation — Example algorithms implemented .... 100
Bibliography


