University of Peloponnese

Thesis

CoveX: Quantum Circuit Simulator

Author:
Vyron Vasileiadis

Supervisor:
Dimitrios Vlachos

A thesis submitted in fulfilment of the requirements for the degree of Computer Science and Technology

in the

Department of Computer Science and Technology
Faculty of Science and Technology

March 2015

Abstract

CoveX: Quantum Circuit Simulator

by Vyron Vasileiadis

The unfamiliar nature and concepts of quantum computing make the creation of easy-to-use tools for experimenting with it a necessity. While many quantum programming languages as well as quantum circuit simulators already exist, none of them provides neither cross-platform support nor exchangeability of results. Still far from complete, CoveX drives to be the first online collaboration tool for quantum circuit simulation which runs in every platform. This thesis provides an extended introduction to Quantum Circuit Model, the implementation of some quantum algorithms within it, as well as a presentation of the first stage of CoveX's development process.

Acknowledgements

First and foremost, i would like to thank my teacher and supervisor, Dimitrios Vlachos, for his guidance and patience during the fulfilment of my thesis. Without his valuable help, it would not be feasible.

I would also like to give special thanks to my family, which supported me on my studies and life.

Last but not least, i would like to thank all the people, friends or foes, who i met the past years in Tripoli. Without them, i would definitely not be the person i am today.

Contents

List of Figures

I ain't no physicist, but i know what matters.

– Popeye the Sailor

Chapter 1

History and Introduction

Quantum computation and quantum information is the study of the information processing tasks that can be accomplished using quantum mechanical systems. Quantum mechanical systems are the systems that evolve under and obey the laws of *quantum mechanics*. Quantum mechanics is the science of the very small; the body of scientific principles that explains the behaviour of matter and its interactions with energy on the scale of atoms and subatomic particles.

Like many simple but profound ideas it was a long time before anybody thought of doing information processing using quantum mechanical systems. In this chapter, we will see the the main events of the history of quantum mechanics and computer science and how fate brought them together.

1.1 Towards Quantum Mechanics

A physicist living around 1890 would have been well pleased with the progress of physics, but perhaps frustrated at the seeming lack of open research problems. It seemed as though the Newtonian laws of mechanics, Maxwell's theory of electromagnetism, and Boltzmann's theory of statistical mechanics explained most natural phenomena. In fact, Max Planck, one of the founding fathers of the quantum theory, was searching for an area of study in 1874 and his advisor gave him the following guidance:

> "In this field [of physics], almost everything is already discovered, and all that remains is to fill a few holes."

Fortunately, Planck did not listen to this advice and instead began his physics studies. Other physicists also did not agree with Planck's former advisor. Lord Kelvin stated in

his famous April 1900 lecture that "two clouds" surrounded the beauty and clearness of current theory. The first cloud was the failure of Michelson and Morley to detect a change in the speed of light as predicted by the *ether theory*, and the second cloud was the ultraviolet catastrophe, the prediction of classical theory that a black-body emits radiation with an infinite intensity at high ultraviolet frequencies. In that same year of 1900, Planck started the quantum revolution that began to clear the second cloud. He assumed that light comes in discrete bundles of energy and used this idea to produce a formula that correctly predicts the spectrum of black-body radiation. A few years later, in 1905, Einstein contributed a paper that helped to further clear the second cloud (he also cleared the first cloud with his other 1905 paper on special relativity). He assumed that Planck was right and showed that the postulate that light arrives in "quanta" (now known as the photon theory) provides a simple explanation for the photoelectric effect, the phenomenon in which electromagnetic radiation beyond a certain threshold frequency hitting a metallic surface induces a current in that metal.

These two explanations of Planck and Einstein fuelled a theoretical revolution in physics that is now called the first quantum revolution. Some years later, in 1924, Louis de Broglie postulated that every individual element of matter, whether an atom, electron, or photon, has both particle-like behaviour and wave-like behaviour. Two years later, Erwin Schrodinger used the de Broglie idea to formulate a wave equation, now known as Schrodinger's equation, that governs the evolution of a closed quantum-mechanical system. His formalism later became known as wave mechanics and was popular among physicists because it appealed to notions with which they were already familiar. Meanwhile, in 1925, Werner Heisenberg formulated an alternate" quantum theory called matrix mechanics. His theory used matrices and theorems from linear algebra, mathematics with which many physicists at the time were not readily familiar. For this reason, Schrodinger's wave mechanics was more popular than Heisenberg's matrix mechanics. In 1930, Paul Dirac published a textbook that unified the formalisms of Schrodinger and Heisenberg, showing that they were actually equivalent. In a later edition, he introduced the now ubiquitous "Dirac notation" for quantum theory that we will employ in this thesis.

Quantum mechanics has been an indispensable part of science ever since, and has been applied with enormous success to everything under and inside the Sun, including the structure of the atom, nuclear fusion in stars, superconductors, the structure of DNA, and the elementary particles of Nature.

1.2 Computer Science Kicks In

People have been using mechanical devices to aid calculation for thousands of years. For example, the abacus probably existed in Babylonia about 3000 B.C.E. The ancient Greeks developed some very sophisticated analog computers. In 1901, an ancient Greek shipwreck was discovered off the island of Antikythera. Inside was a device (now called the Antikythera mechanism) that consisted of rusted metal gears and pointers. When this c. 80 B.C.E. device was reconstructed, it produced a mechanism for predicting the motions of the stars and planets.

Unfortunately, the extended abuse of humankind's worst invention -*religion*- led to a huge gap in advancement of science, generally now called *the Middle Ages*, which spans roughly from 500 C.E to 1500 C.E.

However, around 1570, John Napier, the Scottish inventor of logarithms, invented Napier's rods to simplify the task of multiplication. In 1641 the French mathematician and philosopher Blaise Pascal built a mechanical adding machine. Similar work was done by Gottfried Wilhelm Leibniz. Leibniz also advocated use of the binary system for doing calculations. Joseph-Marie Jacquard invented a loom that could weave complicated patterns described by holes in punched cards. Charles Babbage[1] worked on two mechanical devices: the Difference Engine and the far more ambitious Analytical Engine (a precursor of the modern digital computer), but neither worked satisfactorily. The Difference Engine can be viewed nowadays in the Science Museum in London, England. One of Babbage's friends, Ada Augusta Byron, Countess of Lovelace, wrote a report on Babbage's Analytical Engine theorizing that this machine could one day perform all sorts of complex equations and even create music. She also wrote a set of commands that would allow it to generate the Bernoulli numbers. She is now considered the first programmer and the programming language Ada was named after her.

A step toward automated computation was the introduction of punched cards, which were first successfully used in connection with computing in 1890 by Herman Hollerith working for the U.S. Census Bureau. He developed a device which could automatically read census information which had been punched onto card. Surprisingly, he did not get the idea from the work of Babbage, but rather from watching a train conductor punch tickets. As a result of his invention, reading errors were consequently greatly

[1] Babbage was a bit of an eccentric - one biographer calls him an "irascible genius" - and was probably the model for Daniel Doyce in Charles Dickens' novel, Little Dorrit. A little-known fact about Babbage is that he invented the science of dendrochronology - tree-ring dating - but never pursued his invention. In his later years, Babbage devoted much of his time to the persecution of street musicians (organ-grinders).

reduced, work flow was increased, and, more important, stacks of punched cards could be used as an accessible memory store of almost unlimited capacity; furthermore, different problems could be stored on different batches of cards and worked on as needed. Hollerith's tabulator became so successful that he started his own firm to market the device. This company eventually became International Business Machines (IBM).

In 1928, the German mathematician David Hilbert addressed the International Congress of Mathematicians. He posed three questions:

1. Is mathematics complete; i.e. can every mathematical statement be either proved or disproved?
2. Is mathematics consistent, that is, is it true that statements such as "0 = 1" cannot be proved by valid methods?
3. Is mathematics decidable, that is, is there a mechanical method that can be applied to any mathematical assertion and (at least in principle) will eventually tell whether that assertion is true or not?

This last question was called the *Entscheidungsproblem*.

In 1931, Kurt Gödel answered two of Hilbert's questions. He showed that every sufficiently powerful formal system is either inconsistent or incomplete. Also, if an axiom system is consistent, this consistency cannot be proved within itself. The third question remained open, with 'provable' substituted for 'true'.

In 1936, Alan Turing provided a solution to Hilbert's Entscheidungsproblem by constructing a formal model of a computer, the *Turing machine*, and showing that there were problems such a machine could not solve. One such problem is the so-called *halting problem*. The Turing machine was designed to perform logical operations and could read, write, or erase symbols written on squares of an infinite paper tape. This kind of machine came to be known as a finite state machine because at each step in a computation, the machine's next action was matched against a finite instruction list of possible states. Turing's purpose was not to invent a computer, but rather to describe problems which are logically possible to solve. His hypothetical machine, however, foreshadowed certain characteristics of modern computers that would follow. For example, the endless tape could be seen as a form of general purpose internal memory for the machine in that the machine was able to read, write, and erase it; just like modern day RAM.

By the late 1930s punched-card machine techniques had become so well established and reliable that Howard Aiken, in collaboration with engineers at IBM, undertook construction of a large automatic digital computer based on standard IBM electromechanical parts. Aiken's machine, called the Harvard Mark I, handled 23-decimal-place numbers (words) and could perform all four arithmetic operations; moreover, it had special built-in programs, or subroutines, to handle logarithms and trigonometric functions. The Mark I was originally controlled from pre-punched paper tape without provision for reversal, so that automatic "transfer of control" instructions could not be programmed. Output was by card punch and electric typewriter. Although the Mark I used IBM rotating counter wheels as key components in addition to electromagnetic relays, the machine was classified as a relay computer. It was slow, requiring 3 to 5 seconds for a multiplication, but it was fully automatic and could complete long computations without human intervention. The Harvard Mark I was the first of a series of computers designed and built under Aiken's direction.

With the success of Aiken's Harvard Mark-I as the first major American development in the computing race, work was proceeding on the next great breakthrough by the Americans. Their second contribution was the development of the giant ENIAC machine by John W. Mauchly and J. Presper Eckert at the University of Pennsylvania. ENIAC (Electrical Numerical Integrator and Computer) used a word of 10 decimal digits instead of binary ones like previous automated calculators/computers. ENIAC also was the first machine to use more than 2,000 vacuum tubes, using nearly 18,000 vacuum tubes. Storage of all those vacuum tubes and the machinery required to keep the cool took up over 167 square meters (1800 square feet) of floor space. Nonetheless, it had punched-card input and output and arithmetically had 1 multiplier, 1 divider-square rooter, and 20 adders employing decimal "ring counters", which served as adders and also as quick-access (0.0002 seconds) read-write register storage.

The executable instructions composing a program were embodied in the separate units of ENIAC, which were plugged together to form a route through the machine for the flow of computations. These connections had to be redone for each different problem, together with presetting function tables and switches. This "wire-your-own" instruction technique was inconvenient, and only with some license could ENIAC be considered programmable; it was, however, efficient in handling the particular programs for which it had been designed. ENIAC is generally acknowledged to be the first successful high-speed electronic digital computer (EDC) and was productively used from 1946 to 1955.

In 1945, mathematician John von Neumann undertook a study of computation that demonstrated that a computer could have a simple, fixed structure, yet be able to

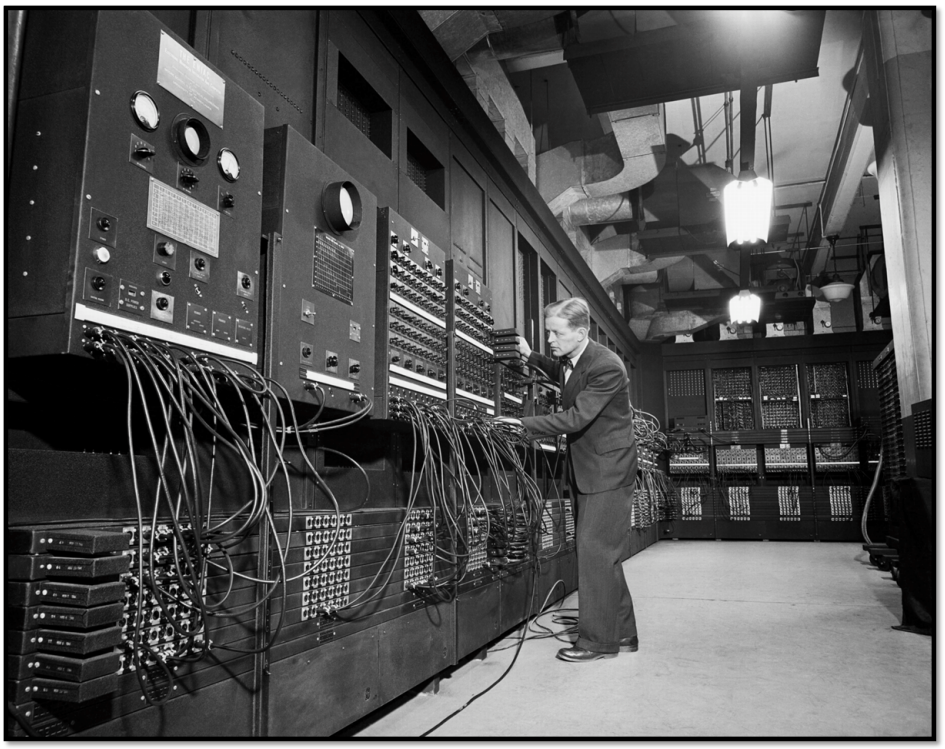


FIGURE 1.1: Picture of the ENIAC computer.

execute any kind of computation given properly programmed control without the need for hardware modification. Von Neumann contributed a new understanding of how practical fast computers should be organized and built; these ideas, often referred to as the stored-program technique, became fundamental for future generations of high-speed digital computers and were universally adopted. The primary advance was the provision of a special type of machine instruction called conditional control transfer, which permitted the program sequence to be interrupted and re-initiated at any point, similar to the system suggested by Babbage for his analytical engine, and by storing all instruction programs together with data in the same memory unit, so that, when desired, instructions could be arithmetically modified in the same way as data. Thus, data was the same as program.

As a result of these techniques and several others, computing and programming became faster, more flexible, and more efficient, with the instructions in subroutines performing far more computational work. Frequently used subroutines did not have to be reprogrammed for each new problem but could be kept intact in "libraries" and read into memory when needed. Thus, much of a given program could be assembled from the subroutine library. The all-purpose computer memory became the assembly place in which parts of a long computation were stored, worked on piecewise, and assembled to form the final results. The computer control served as an errand runner

for the overall process. As soon as the advantages of these techniques became clear, the techniques became standard practice. The first generation of modern programmed electronic computers to take advantage of these improvements appeared in 1947.

This group included computers using random access memory (RAM), which is a memory designed to give almost constant access to any particular piece of information. These machines had punched-card or punched-tape input and output devices and RAMs of 1,000-word. Physically, they were much more compact than ENIAC: some were about the size of a grand piano and required 2,500 small electron tubes, far fewer than required by the earlier machines. The first- generation stored-program computers required considerable maintenance, attained perhaps 70 to 80 reliable operation, and were used for 8 to 12 years. Typically, they were programmed directly in machine language, although by the mid-1950s progress had been made in several aspects of advanced programming. This group of machines included EDVAC and UNIVAC, the first commercially available computers.

In the 1950s,two devices would be invented which would improve the computer field and cause the beginning of the computer revolution. The first of these two devices was the transistor. Invented in 1947 by William Shockley, John Bardeen, and Walter Brattain of Bell Labs, the transistor was fated to replace vacuum tubes in computers, radios, and other electronics.

The vacuum tube, used up to this time in almost all the computers and calculating machines, had been invented by American physicist Lee De Forest in 1906. The vacuum tube worked by using large amounts of electricity to heat a filament inside the tube until it was cherry red. One result of heating this filament up was the release of electrons into the tube, which could be controlled by other elements within the tube. De Forest's original device was a triode, which could control the flow of electrons to a positively charged plate inside the tube. A zero could then be represented by the absence of an electron current to the plate; the presence of a small but detectable current to the plate represented a one.

Vacuum tubes were highly inefficient, required a great deal of space, and needed to be replaced often. Computers such as ENIAC had 18,000 tubes in them and housing all these tubes and cooling the rooms from the heat produced by 18,000 tubes was not cheap. The transistor promised to solve all of these problems and it did so. Transistors, however, had their problems too. The main problem was that transistors, like other electronic components, needed to be soldered together. As a result, the more complex the circuits became, the more complicated and numerous the connections between the individual transistors and the likelihood of faulty wiring increased.

In 1958, this problem too was solved by Jack St. Clair Kilby of Texas Instruments. He manufactured the first integrated circuit or chip. A chip is really a collection of tiny transistors which are connected together when the transistor is manufactured. Thus, the need for soldering together large numbers of transistors was practically nullified; now only connections were needed to other electronic components. In addition to saving space, the speed of the machine was now increased since there was a diminished distance that the electrons had to follow.

In 1971, Intel released the first microprocessor. The microprocessor was a specialized integrated circuit which was able to process four bits of data at a time. The chip included its own arithmetic logic unit, but a sizable portion of the chip was taken up by the control circuits for organizing the work, which left less room for the data-handling circuitry. Thousands of hackers could now aspire to own their own personal computer. Computers up to this point had been strictly the legion of the military, universities, and very large corporations simply because of their enormous cost for the machine and then maintenance. In 1975, the cover of Popular Electronics featured a story on the "World's first minicomputer kit to rival commercial models... Altair 8800.". The Altair, produced by a company called Micro Instrumentation and Telementry Systems (MITS) retailed for 397 dollars, which made it easily affordable for the small but growing hacker community.

The Altair was not designed for a computer novice. The kit required assembly by the owner and then it was necessary to write software for the machine since none was yet commercially available. The Altair had a 256 byte memory and needed to be coded in machine code; 0s and 1s. The programming was accomplished by manually flipping switches located on the front of the Altair.

Two young hackers were intrigued by the Altair, having seen the article in Popular Electronics. They decided on their own that the Altair needed software and took it upon themselves to contact MITS owner Ed Roberts and offer to provide him with a BASIC which would run on the Altair. BASIC (Beginners All-purpose Symbolic Instruction Code) had originally been developed in 1963 by Thomas Kurtz and John Kemeny, members of the Dartmouth mathematics department. BASIC was designed to provide an interactive, easy method for upcoming computer scientists to program computers. It allowed the usage of statements such as print "hello" or let b=10. It would be a great boost for the Altair if BASIC were available, so Robert's agreed to pay for it if it worked. The two young hackers worked feverishly and finished just in time to present it to Roberts. It was a success. The two young hackers were William Gates and Paul Allen. They later went on to form Microsoft and produce BASIC and operating systems for various machines.

FIGURE 1.2: Cover of Popular Electronics magazine in January 1975 featuring Altair 8800.

Following the introduction of the Altair, a veritable explosion of personal computers occurred, starting with Steve Jobs and Steve Wozniak exhibiting the first Apple II at the First West Coast Computer Faire in San Francisco. The Apple II boasted built-in BASIC, color graphics, and a 4100 character memory for only 1298$. Programs and data could be stored on an everyday audio-cassette recorder. Before the end of the fair, Wozniak and Jobs had secured 300 orders for the Apple II and from there Apple took off.

Also introduced in 1977 was the TRS-80. This was a home computer manufactured by Tandy Radio Shack. In its second version, the TRS-80 Model II, came complete with a 64,000 character memory and a disk drive to store programs and data on. At this time, only Apple and TRS had machines with disk drives. With the introduction of the disk drive, personal computer applications took off as a floppy disk was a most convenient publishing medium for distribution of software.

IBM, which up to this time had been producing mainframes and minicomputers for medium to large-sized businesses, decided that it had to get into the act and started working on the Acorn, which would later be called the IBM PC. The PC was the first computer designed for the home market which would feature modular design so that pieces could easily be added to the architecture. Most of the components, surprisingly, came from outside of IBM, since building it with IBM parts would have cost too much for the home computer market. When it was introduced, the PC came with a 16,000 character memory, keyboard from an IBM electric typewriter, and a connection for tape cassette player for 1265.

By 1984, Apple and IBM had come out with new models. Apple released the first generation Macintosh, which was the first computer to come with a graphical user interface(GUI) and a mouse. The GUI made the machine much more attractive to home computer users because it was easy to use. Sales of the Macintosh soared like nothing ever seen before. IBM was hot on Apple's tail and released the 286-AT, which with applications like Lotus 1-2-3, a spreadsheet, and Microsoft Word, quickly became the favourite of business concerns.

That brings us up to about twenty years ago. Now people have their own personal graphics workstations and powerful home computers. The average computer a person might have in their home is more powerful by several orders of magnitude than a machine like ENIAC. The computer revolution has been the fastest growing technology in man's history.

1.3 Quantum Computing Is Born

The concept of doing computations using quantum resources was first pointed out by one of the greatest physicists of the 20th century, Richard Feynman. Feynman did an incredible research work throughout his life. He involved deeply in the development of the first atomic bomb, proposed the significant theories of quantum electrodynamic, and also predicted that antiparticles, particles which possess a charge opposite to that of their mirror particle, are actually just normal particles which move backwards in time!

In June 1982, Feynman shows, in his famous paper "Simulating physics with computers", that *quantum mechanical systems cannot be efficiently simulated on conventional computing hardware.* He begins, among others, to investigate the generalization of conventional information science concepts to quantum physical processes, considering the representation of binary numbers in relation to the quantum states of two-state

quantum systems. In other words, simulating quantum systems not with conventional computers but with other quantum systems constructed for this purpose.

In 1985, David Deutsch publishes a theoretical paper describing a universal quantum computer, proving that if two-state system could be made to evolve by means of a set of simple operations, any such evolution could be produced, and made to simulate any quantum system. These operations came later to be called *quantum gates*, as they function similarly to binary logic gates in classical computers. He also demonstrates a simple algorithm using these gates, now known as *Deutsch's algorithm*, which solves a certain problem faster than any classical computer could.

A major breakthrough came in 1994, when Peter Shor, working for AT&T, proposes a method using entanglement of qubits and superposition to find the prime factors of an integer exponentially faster than any classical computer could. Prime factorization is a rather valuable process as many encryption systems exploit the difficulty of classical computers in finding factors of large numbers. Shor's discovery proves quite instrumental in provoking a storm of research both by physicists and computer scientists.

In 1995, The National Institute of Standards and Technology and the California Institute of Technology jointly look at the problem of shielding a quantum system from environmental influences and perform experiments with magnetic fields, which allow particles, specifically ions, to be trapped and cooled to a quantum state.

Next year, in 1996, a team composed of University of California at Berkeley, MIT, Harvard University, and IBM researchers pursue a somewhat similar technique, but using nuclear magnetic resonance(NMR), a technology which seems to manipulate quantum information in liquids. NMR acts on quantum particles in the atomic nuclei of the fluid by creating a certain *spin*. The alignment of a given particle's spin represent its value, 0 or 1. By varying the electromagnetic field used, certain oscillations are found which allow certain spins to flip between these states, allowing them to exist in both at once. The team develops a two-qubit quantum computer based on a liquid of chloroform; input consists of radio frequency pulses into the liquid containing, in essence, the compiled program to be executed.

In 1993, the feasibility of quantum teleportation is proposed by an international team of researchers, who based their conclusions on the EPR effect, which describes entanglement. The group theorizes that two entangled, *transporter* particles introduced to a third, *message* particle might transfer properties from one to the other. The idea is put into practice six years later by researchers at the University of Innsbruck in Austria. Two pairs of entangled photons were exposed to each other, and it is revealed

that the polarization state of one may be transferred to the other. An illustration of the experimental setup is shown in Figure 1.3

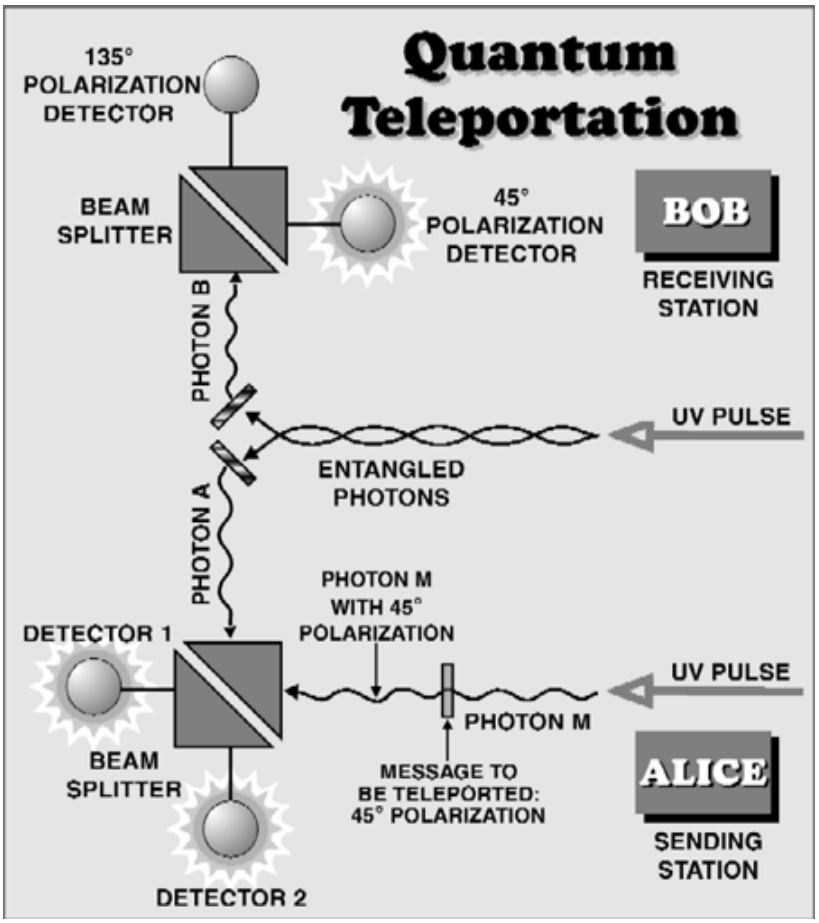


FIGURE 1.3: Experimental setup for quantum teleportation of University of Innsbruck.

During the following years the interest in quantum computing has increased dramatically. Nowadays, around 200 research groups around the globe study the different aspects of this new field. There are many questions that still need answer. How to efficiently implement physically a quantum computer, a complete theory of quantum information and computation, and new algorithms for solving problems faster than with classical computers.

Chapter 2

Linear Algebra

2.1 Linear Algebra

Linear algebra is the study of vector spaces and of linear operations on those vector spaces. A good understanding of quantum mechanics is based upon a solid grasp of elementary linear algebra. The chief obstacle to assimilation of the postulates of quantum mechanics, presented in the next chapter, is not the postulates themselves, but rather the large body of linear algebraic notions required to understand them. Coupled with the unusual Dirac notation adopted by physicists for quantum mechanics, it can appear (falsely) quite fearsome.

2.2 Vector Space

The basic objects of linear algebra are vector spaces. Its definition may appear dry. However, vector spaces are quite easy to visualize as sets of geometric vectors. Like regular numbers, vectors can be added to and subtracted from each other to form new vectors. They can also be multiplied by numbers, which from now on will refer to as *scalars*. Unlike numbers, vectors cannot be multiplied or divided by one another (more exactly, we don't have to define multiplication in order to introduce the vector space). One important peculiarity of the linear algebra used in quantum mechanics is the so-called Dirac notation for vectors. To denote elements of the Hilbert space[1], instead of writing, for example, $\hat{v}$ we write $|v\rangle$.

Definition 2.1. A linear vector space V over a field F is a set in which the following operations are defined:

[1] Hilbert space is the vector space where quantum mechanics resides in.

1. **Addition of vectors.** $\forall |a\rangle, |b\rangle \in V, \quad \exists$ unique $|a\rangle + |b\rangle \in V$.
2. **Multiplication of vector by scalar.** $\forall |a\rangle \in V, \forall \lambda \in F, \quad \exists$ unique $\lambda |a\rangle \in V$.

This operations must obey the following *axioms*:

1. Commutativity of addition: $|a\rangle + |b\rangle = |b\rangle + |a\rangle$
2. Associativity of addition: $(|a\rangle + |b\rangle) + |c\rangle = |a\rangle + (|b\rangle + |c\rangle)$
3. Existence of unique zero element: $\exists |zero\rangle \in V$ such that $\forall |a\rangle \in V$ then $|a\rangle + |zero\rangle = |a\rangle$
   Note: As an alternative notation for $|zero\rangle$ we use 0 but not $|0\rangle$.
4. Existence of the additive inverse for each element: $\forall |a\rangle, \exists |a^{'}\rangle$ such that $|a\rangle + |a^{'}\rangle = 0$
   Notation: $|a^{'}\rangle = -|a\rangle$
5. Distributivity of vector sums: $\lambda(|a\rangle + |b\rangle) = \lambda|a\rangle + \lambda|b\rangle$
6. Distributivity of scalar sums: $(\lambda + \mu)|a\rangle = \lambda|a\rangle + \mu|b\rangle$
7. Associativity of scalar multiplication: $\lambda(\mu|a\rangle) = (\lambda\mu)|a\rangle$
8. Scalar multiplication identity: for $1 \in F$, $\forall |a\rangle \in V$, then $1 \cdot |a\rangle = |a\rangle$

Subtraction of vectors in a linear space is defined as follows:

$$|a\rangle - |b\rangle \equiv |a\rangle + (-|b\rangle)$$

2.3 Basis & Dimension

The basis is the smallest subset of a linear space such that all other vectors can be expressed as a linear combination of the basis elements. The term *basis* may suggest that each linear space has only one basis. Actually, in any non-trivial linear space, there are infinitely many bases.

Definition 2.2. A set of vectors $|v_i\rangle$ is called *linearly independent* if no non-trivial linear combination $\lambda_1|v_1\rangle + \ldots + \lambda_N|v_N\rangle$ equals $|zero\rangle$.

Definition 2.3. A subset $\{|v_i\rangle\}$ of a vector space V is called its *spanning set* if any vector in $\mathbb{V}$ can be expressed as a linear combination of $|v_i\rangle$'s.

Definition 2.4. A *basis* of $\mathbb{V}$ is any linearly independent spanning set. A *decomposition* of a vector into a basis is its expression as a linear combination of the basis elements.

Definition 2.5. The number of elements in a basis is called the *dimension* of V. We denote dimension of V as $dim\, V$.

Definition 2.6. For a decomposition

$$|a\rangle = \sum_i a_i |v_i\rangle, \tag{2.1}$$

we may use the notation

$$|a\rangle \equiv \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix} \tag{2.2}$$

This is called the *matrix form* of a vector. The quantities a_i are called the *coefficients* or *amplitudes* of the decomposition.

2.4 Inner Product

Although vectors cannot be multiplied by each other in the same way that numbers can, one can define a multiplication operation that maps any pair of vectors onto a number. This operation generalizes the scalar product that is familiar from geometry.

Definition 2.7. $\forall |a\rangle, |b\rangle \in \mathbb{V}$ we define an *inner (scalar) product* $\langle a|\, b\rangle \in \mathbb{C}$, such that:

1. $\langle a|(|b\rangle + |c\rangle) = \langle a|\, b\rangle + \langle a|\, c\rangle$
2. $\langle a|(\lambda|b\rangle) = \lambda\langle a|\, b\rangle$
3. $\langle a|\, b\rangle = \langle b|\, a\rangle^*$
4. $\langle a|\, a\rangle$ is a real number. $\forall |a\rangle$ then $\langle a|\, a\rangle \geq 0$. $\langle a|\, a\rangle = 0$ if and only if $|a\rangle = 0$.

According to the definition of the inner product, generally $\langle a|\, b\rangle \neq \langle b|\, a\rangle$.

Definition 2.8. $|a\rangle$ and $|b\rangle$ are called *orthogonal* if $\langle a|\, b\rangle = 0$. **Note:** A set of mutually orthogonal vectors is linearly independent.

Definition 2.9. $\| |a\rangle \| = \sqrt{\langle a|\, a\rangle}$ is called the *norm (length)* of a vector. Vectors of norm 1 are called *normalized*.

Definition 2.10. The linear space over field $\mathbb{C}$, in which the inner product is defined, is called the *Hilbert Space*.

2.5 Orthonormal Basis

Definition 2.11. An *orthonormal basis* $\{|v_i\rangle\}$ is a basis whose elements are mutually orthogonal and have norm 1. It is that

$$\langle v_i|\, v_j\rangle = \delta_{ij}, \text{ where } \delta_{ij} \begin{cases} = 0, & \text{if } i = j \\ \neq 0, & \text{if } i \neq j \end{cases} \tag{2.3}$$

Any orthonormal set of $N = dim\, \mathbb{V}$ vectors forms a basis. Also, if $\begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix}$ and $\begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix}$ are the decompositions of states $|a\rangle$ and $|b\rangle$ in an orthonormal basis, their inner product can be written as

$$\langle a|\, b\rangle = a_1^* b_1 + \cdots + a_N^* b_N. \tag{2.4}$$

Last equation can be expressed in the matrix form using the "row-times-column" rule:

$$\langle a|\, b\rangle = \begin{bmatrix} a_1^* & \cdots & a_N^* \end{bmatrix} \begin{bmatrix} b_1 \\ \vdots \\ b_N \end{bmatrix} \tag{2.5}$$

2.6 Useful Inequalities

There are two inequalities that are very useful when solving linear algebra problems. The first, called the *Gram-Schmidt inequality* tells us the relation between inner product and norm of two vectors:

$$|\, \langle a|\, b\rangle\, | \quad \leq \quad \|\, |a\rangle\, \| \cdot \|\, |b\rangle\, \| \tag{2.6}$$

The equality is reached if and only if the states $|a\rangle$ and $|b\rangle$ are collinear (i.e. $|a\rangle = \lambda|b\rangle$).

The second inequality, called the *triangle inequality* is about the relation between the sum of norms of two states and the norm of their sum:

$$\|\, (|a\rangle + |b\rangle)\, \| \quad \leq \quad \|\, |a\rangle\, \| + \|\, |b\rangle\, \| \tag{2.7}$$

2.7 Adjoint Space

It is sometimes convenient to think of the scalar product $\langle a|\, b\rangle$ as the product of two objects, $\langle a|$ and $|b\rangle$.

Definition 2.12. For the Hilbert space $\mathbb{V}$, we define the *adjoint space* $\mathbb{V}^\dagger$ (pronounced V-dagger) which has one-to-one correspondence to $\mathbb{V}$; for each vector $|a\rangle \in \mathbb{V}$ there is one and only one *adjoint* vector $\langle a| \in \mathbb{V}$ with the property

$$Adjoint(\lambda|a\rangle + \mu|b\rangle) = \lambda^*\langle a| + \mu^*\langle b| \tag{2.8}$$

Direct and adjoint vectors are more often called *ket-* and *bra-* vectors, respectively. This terminology, together with the symbols $\langle a|$ and $|b\rangle$, was introduced by Paul Dirac. The inner product given by the combination of those two symbols is called *braket*.

Although the adjoint space is a linear vector space too, $\mathbb{V}$ and $\mathbb{V}^\dagger$ are different linear spaces. We cannot add a bra-vector and a ket-vector.

If $\{|v_i\rangle\}$ is a basis in $\mathbb{V}$, $\{\langle v_i|\}$ is a basis in $\mathbb{V}^\dagger$ and if a ket-vector is decomposed in this basis as $|a\rangle = \sum a_i|v_i\rangle$, the decomposition of its adjoint is

$$\langle a| = \sum a_i^*\langle v_i| \tag{2.9}$$

If we want to write a bra-vector in a matrix form, it is convenient to write it as a row rather than column:

$$\langle a| = \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix}^\dagger = \begin{bmatrix} a_1^* & \cdots & a_N^* \end{bmatrix} \tag{2.10}$$

The superscript † (now in application to a matrix) means transposition and complex conjugation.

2.8 Linear Operators

Definition 2.13. A *linear operator* A is a *map*[2] of one linear space $\mathbb{V}$ onto another linear space $\mathbb{W}$ such that:

1. $A(|a\rangle + |b\rangle) = A|a\rangle + A|b\rangle$

[2] A map $f : \mathbb{A} \mapsto \mathbb{B}$ is a function f such that for every element a in $\mathbb{A}$, there is a unique "image" $f(a)$ in $\mathbb{B}$.

2. $A(\lambda|a\rangle) = \lambda A|a\rangle$

The linear operators that interests us are the operators that map vector spaces onto themselves ($\mathbb{V} \mapsto \mathbb{V}$).

For any operator A and any scalar λ, we can define their product, an operator λA which maps vectors according to

$$(\lambda A)|a\rangle \equiv \lambda(A|a\rangle) \tag{2.11}$$

For any two operators A and B, we can define their sum, an operator $A + B$ which maps vectors according to

$$(A + B)|a\rangle \equiv A|a\rangle + B|a\rangle \tag{2.12}$$

Definition 2.14. The operator $\mathrm{I} : \mathbb{V} \mapsto \mathbb{V}$ which maps every vector onto itself is called the *identity operator*.

Note: When writing products of a scalar with the identity operator, we can omit the symbol I when the context allows no ambiguity. For example, instead of writing $\mathrm{A} - \lambda\mathrm{I}$, we may simply write $\mathrm{A} - \lambda$.

Definition 2.15. *Operator product* AB is an operator that maps vector $|a\rangle$ onto $\mathrm{AB}|a\rangle \equiv \mathrm{A}(\mathrm{B}|a\rangle)$. That is, in order to find the action of operator AB onto a vector, we must first apply B to that vector, and then apply A to the result.

It matters in which order the two operators are multiplied. Generally, $\mathrm{AB} \neq \mathrm{BA}$. Such operators for which $\mathrm{AB} = \mathrm{BA}$ are said to *commute*. Commutation relations between operators play an important role in quantum mechanics.

Definition 2.16. The *matrix of an operator* in a basis $\{|v_i\rangle\}$ is an NxN square table.

If $|a\rangle \equiv \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix}$, then the state $\mathrm{A}|a\rangle$ is given by the matrix product

$$\mathrm{A}|a\rangle \equiv \begin{bmatrix} \mathrm{A}_{11} & \cdots & \mathrm{A}_{1N} \\ \vdots & & \vdots \\ \mathrm{A}_{N1} & \cdots & \mathrm{A}_{NN} \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_N \end{bmatrix} \tag{2.13}$$

In other words, the action of an operator on a vector is equivalent to the multiplication of the corresponding matrices. The matrix of an operator in a known basis fully defines

the operator, because using the matrix we can find out what the operator does to every vector. Operations with operators and vectors are identical to operations with matrices and columns.

2.9 Tensor Products

The *tensor product* is a way of putting vector spaces together to form larger vector spaces. This construction is crucial to understanding the quantum mechanics of multiparticle systems.

Suppose $\mathbb{V}$ and $\mathbb{W}$ are vector spaces of dimension m and n respectively. For convenience we also suppose that $\mathbb{V}$ and $\mathbb{W}$ are Hilbert spaces. Then $\mathbb{V} \otimes \mathbb{W}$ (pronounced "V tensor W") is an mn dimensional vector space. The elements of $\mathbb{V} \otimes \mathbb{W}$ are linear combinations of "tensor products" $|v\rangle \otimes |w\rangle$ of elements $|v\rangle$ of $\mathbb{V}$ and $|w\rangle$ of $\mathbb{W}$. In particular, if $|i\rangle$ and $|j\rangle$ are orthonormal bases for the spaces $\mathbb{V}$ and $\mathbb{W}$ then $|i\rangle \otimes |j\rangle$ is a basis for $\mathbb{V} \otimes \mathbb{W}$. We often use the notations $|v\rangle|w\rangle$, $|v, w\rangle$, or even $|vw\rangle$ for the tensor product $|v\rangle \otimes |w\rangle$.

By definition the tensor product satisfies the following properties:

1. For any arbitrary scalar λ and elements $|v\rangle \in \mathbb{V}$ and $|w\rangle \in \mathbb{W}$,

$$\lambda(|v\rangle \otimes \{ketw) = (\lambda|v\rangle) \otimes |w\rangle = |v\rangle \otimes (\lambda|w\rangle). \tag{2.14}$$

2. For arbitrary $|v_1\rangle$ and $|v_2\rangle$ in $\mathbb{V}$ and $|w\rangle$ in $\mathbb{W}$,

$$(|v_1\rangle + |v_2\rangle) \otimes |w\rangle = |v_1\rangle \otimes |w\rangle + |v_2\rangle \otimes |w\rangle. \tag{2.15}$$

3. For arbitrary $|v\rangle$ in $\mathbb{V}$ and $|w_1\rangle$ and $|w_2\rangle$ in $\mathbb{W}$,

$$|v\rangle \otimes (|w_1\rangle + |w_2\rangle) = |v\rangle \otimes |w_1\rangle + |v\rangle \otimes |w_2\rangle. \tag{2.16}$$

If A and B are linear operators on $\mathbb{V}$ and $\mathbb{W}$ respectively, then linear operator $\mathrm{A} \otimes \mathrm{B}$ on $\mathbb{V} \otimes \mathbb{W}$ is defined by the equation

$$\mathrm{A} \otimes \mathrm{B}(|v\rangle \otimes |w\rangle) \equiv \mathrm{A}|v\rangle \otimes \mathrm{B}|w\rangle. \tag{2.17}$$

There is a convenient matrix representation known as *Kronecker product*. Suppose A is an m by n matrix, and B is a p by q matrix. Then we have the matrix representation:

$$\mathrm{A} \otimes \mathrm{B} \equiv \begin{bmatrix} \mathrm{A}_{11}\mathrm{B} & \mathrm{A}_{12}\mathrm{B} & \cdots & \mathrm{A}_{1\mathrm{n}}\mathrm{B} \\ \mathrm{A}_{21}\mathrm{B} & \mathrm{A}_{22}\mathrm{B} & \cdots & \mathrm{A}_{2\mathrm{n}}\mathrm{B} \\ \mathrm{A}_{\mathrm{m}1}\mathrm{B} & \mathrm{A}_{\mathrm{m}2}\mathrm{B} & \cdots & \mathrm{A}_{\mathrm{mn}}\mathrm{B} \end{bmatrix} \tag{2.18}$$

In this representation terms like $\mathrm{A}_{11}\mathrm{B}$ denote p by q submatrices whose entries are proportional to B, with overall proportionality constant A_{11}. For example:

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 1 \times 2 \\ 1 \times 3 \\ 2 \times 2 \\ 2 \times 3 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 4 \\ 6 \end{bmatrix} \tag{2.19}$$

The tensor product of the Pauli matrices $\mathrm{X} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ and $Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$ is

$$\mathrm{X} \otimes \mathrm{Y} = \begin{bmatrix} 0 \cdot \mathrm{Y} & 1 \cdot \mathrm{Y} \\ 1 \cdot \mathrm{Y} & 0 \cdot \mathrm{Y} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \end{bmatrix} \tag{2.20}$$

Finally, we mention the useful notation $|\psi\rangle^{\otimes k}$, which means $|\psi\rangle$ tensored with itself k times. For example, $|\psi\rangle^{\otimes 3} = |\psi\rangle \otimes |\psi\rangle \otimes |\psi\rangle$. An analogous notation is also used for operators on tensor product spaces.

2.10 Operator Functions

Definition 2.17. *Trace* of an operator A is the sum of the diagonal elements of its matrix in an orthonormal basis. (**Note:** The trace of an operator is basis independent.)

$$tr(\mathrm{A}) \equiv \sum_i \mathrm{A}_{\mathrm{ii}} \tag{2.21}$$

The trace is easily seen to be *cyclic*, $tr(\mathrm{AB}) = tr(\mathrm{BA})$, and *linear*, $tr(\mathrm{A} + \mathrm{B}) = tr(\mathrm{A}) + tr(\mathrm{B})$ and $tr(\lambda \mathrm{A}) = \lambda tr(\mathrm{A})$.

Chapter 3

Quantum Mechanics

3.1 Postulates of Quantum Mechanics

What is quantum mechanics? Quantum mechanics is a mathematical framework or set of rules for the construction of physical theories. For example, there is a physical theory known as *quantum electrodynamics* which describes with fantastic accuracy the interaction of atoms and light. Quantum electrodynamics is built up within the framework of quantum mechanics, but it contains specific rules not determined by quantum mechanics. The relationship of quantum mechanics to specific physical theories like quantum electrodynamics is rather like the relationship of a computer's operating system to specific applications software – the operating system sets certain basic parameters and modes of operation, but leaves open how specific tasks are accomplished by the applications.

In the following sections a brief description of the basic postulates of quantum mechanics is provided. These postulates provide a connection between the physical world and the mathematical formalism of quantum mechanics.

3.1.1 State Space

> **Postulate 1.** Associated to any isolated quantum system is a complex vector space with inner product (that is, a Hilbert space) known as the *state space* of the system. The system is completely described by its *state vector*, which is a unit vector in the system's state space.

Quantum mechanics does not tell us, for a given system, what the state space of that system is, nor does it tell us what the state vector of the system is. Figuring that

out for a specific system can be a difficult problem. The simplest quantum system, and the system which we will be most concerned with, is the *qubit*. A qubit has a two-dimensional state space. Suppose $|0\rangle$ and $|1\rangle$ form an orthonormal basis for that space state. Then an arbitrary state vector in the state space can be written

$$|\psi\rangle = a|0\rangle + b|1\rangle \tag{3.1}$$

where a and b are complex numbers. The condition that $|\psi\rangle$ is unit vector provides that $\langle\psi|\,\psi\rangle = 1$ or equivalently $|a|^2 + |b|^2 = 1$. The condition $\langle\psi|\,\psi\rangle = 1$ is known as *normalization condition*.

We will take the qubit as our fundamental quantum mechanical system. Any discussion on qubits will always be referred to some orthonormal set of basis vectors, $|0\rangle$ and $|1\rangle$. The states $|0\rangle$ and $|1\rangle$ are analogous to the two values 0 and 1 which a bit may take. The way a qubit differs from a bit is that *superpositions* of these two states, of the form $a|0\rangle + b|1\rangle$, can also exist, in which it is not possible to say that the qubit is definitely in the state $|0\rangle$, or definitely in the state $|1\rangle$.

We say that any linear combination $\sum_i a_i|\psi_i\rangle$ is a superposition of the states $|\psi_i\rangle$ with *amplitude* a_i for the state $|\psi_i\rangle$. So for example, the state

$$\frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

is a superposition of the states $|0\rangle$ and $|1\rangle$ with amplitude $\dfrac{1}{\sqrt{2}}$ for the state $|0\rangle$, and amplitude $-\dfrac{1}{\sqrt{2}}$ for the state $|1\rangle$.

3.1.2 Evolution

> **Postulate 2.** The evolution of a *closed* quantum system is described by a unitary transformation. That is, the state $|\psi\rangle$ of the system at time t_1 is related to the state $|\psi^{'}\rangle$ of the system at time t_2 by a unitary operator U, which depends only on times t_1 and t_2,
>
> $$|\psi^{'}\rangle = U|\psi\rangle \tag{3.2}$$

Postulate 2 requires that the system being described be closed. That is, it is not interacting in any way with other systems. In reality, of course, all systems (except the Universe as a whole) interact at least somewhat with other systems. Nevertheless, there are interesting systems which can be described to a good approximation as being

closed, and which are described by unitary evolution to some good approximation. Being able to successfully isolating a quantum system, making it completely closed, is one of the main challenge for making quantum computers feasible. For the purpose of this thesis, we will assume that our system is ideally closed and there are no external interactions.

3.1.3 Quantum Measurement

The evolution of systems which don' t interact with the rest of the world is all very well, but there must be also times when w would like to observe the system to find out what is going inside it, an interaction which makes the system no longer closed, and thus not subject to unitary evolution. Postulate 3 provides a means for describing the effects of measurements on quantum systems.

> **Postulate 3.** Quantum measurements are described by a collection $\{M_m\}$ of *measurement operators*. These are operators acting on the state space of the system being measured. The index m refers to the measurement outcomes that may occur in the experiment. If the state of the quantum system is $|\psi\rangle$ immediately before the measurement then the probability that result m occurs is given by
>
> $$p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle, \tag{3.3}$$
>
> and the state of the system after the measurement is
>
> $$\frac{M_m|\psi\rangle}{\sqrt{\langle\psi|M_m^\dagger M_m|\psi\rangle}}. \tag{3.4}$$
>
> The measurement operators satisfy the *completeness equation*,
>
> $$\sum_m \mathrm{M}_m^\dagger \mathrm{M}_m = \mathrm{I}. \tag{3.5}$$

A simple but important example of a measurement is the *measurement of a qubit in the computational basis*. This is a measurement on a single qubit with two outcomes defined by the two measurement operators $M_0 = |0\rangle\langle 0|$ and $M_1 = |1\rangle\langle 1|$

Suppose the state $|\psi\rangle = a|0\rangle + b|1\rangle$ is being measured. The probability of obtaining measurement outcome 0 is

$$p(0) = \langle\psi|M_0^\dagger M_0|\psi\rangle = \langle\psi|M_0|\psi\rangle = |a|^2 \tag{3.6}$$

Similarly, the probability of obtaining the measurement outcome 1 is $p(1) = |b|^2$. The state after measurement in the two cases is therefore

$$\frac{\mathrm{M}_0|\psi\rangle}{|a|} = \frac{a}{|a|}|0\rangle \tag{3.7}$$

$$\frac{\mathrm{M}_1|\psi\rangle}{|b|} = \frac{b}{|b|}|1\rangle \tag{3.8}$$

Multipliers like $\frac{a}{|a|}$ can effectively be ignored, so the two post-measurement states are effectively $|0\rangle$ and $|1\rangle$.

3.1.4 Composite Systems

Suppose we are interested in a composite quantum system made up of two or more distinct physical systems. How should we describe states of the composite system? The fourth postulate describes how the state space of a composite system is built up from the state spaces of the component systems.

> **Postulate 4.** The state space of a composite physical system is the tensor product of the state spaces of the component physical systems. Moreover, if we have systems numbered 1 through n, and system number i is prepared in the state $|\psi_i\rangle$, then the joint system of the total system is $|\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle$.

For example, if $|\psi_1\rangle = a_1|0\rangle + b_1|1\rangle$ and $|\psi_2\rangle = a_2|0\rangle + b2|1\rangle$ then

$$\begin{aligned}|\psi_1\rangle \otimes |\psi_2\rangle = |\psi_1\rangle|\psi_2\rangle = |\psi_1\psi_2\rangle &= (a_1|0\rangle + b_1|1\rangle)(a_2|0\rangle + b2|1\rangle) \\ &= a_1a_2|0\rangle|0\rangle + a_1b_2|0\rangle|1\rangle + b_1a_2|1\rangle|0\rangle + b_1b_2|1\rangle|1\rangle \\ &= a_1a_2|00\rangle + a_1b_2|01\rangle + b_1a_2|10\rangle + b_1b_2|11\rangle\end{aligned}$$

3.2 Entanglement

Postulate 4 also enables us to define one of the most interesting and puzzling ideas associated with composite quantum systems - *entanglement*.

Entanglement is a uniquely quantum mechanical resource that plays a key role in many of the most interesting applications of quantum computation and quantum information; entanglement is iron to the classical world's bronze age. In recent years there has been a tremendous effort trying to better understand the properties of entanglement

considered as a fundamental resource of Nature, of comparable importance to energy, information, entropy, or any other fundamental resource. Although there is as yet no complete theory of entanglement, some progress has been made in understanding this strange property of quantum mechanics. It is hoped by many researchers that further study of the properties of entanglement will yield insights that facilitate the development of new applications in quantum computation and quantum information.

Consider the two qubit state

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{3.9}$$

This state has the remarkable property that there are no single qubit states $|\psi_0\rangle$ and $|\psi_1\rangle$ such that $|\psi\rangle = |\psi_0\rangle|\psi_1\rangle$. Assume that

$$(a_0|0\rangle + b_0|1\rangle) \otimes (a_1|0\rangle + b_1|1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle), \tag{3.10}$$

where $|a_0|^2 + |a_1|^2 = 1$ and $|b_0|^2 + |b_1|^2 = 1$. Then we obtain the system of equations

$$a_0 b_0 = \frac{1}{\sqrt{2}}, \quad a_0 b_1 = 0, \quad a_1 b_0 = 0, \quad a_1 b_1 = \frac{1}{\sqrt{2}}. \tag{3.11}$$

This set of equations admits no solution. Thus the state $|\psi\rangle$ cannot be written as a product state. We say that the sate $|\psi\rangle$ is *entangled*.

For reasons which nobody fully understands,such entangled states play a crucial role in quantum computation and quantum information, and arise repeatedly in the speeding that quantum algorithms succeed over their classical analogues. We will see some of these algorithms in chapter **??**.

3.3 Phase

Phase is a commonly used term in quantum mechanics, with two distinct meanings. Consider the state $e^{i\theta}|\psi\rangle$, where θ is a real number. We say that the state $e^{i\theta}|\psi\rangle$ is equal to $|\psi\rangle$, up to the *global phase factor* $e^{i\theta}$. That means that the *statistics of measurement* predicted for these two states are the same. Therefore, from an observational point of view these two states are identical. For this reason we may ignore global phase factors as being irrelevant to the observed properties of the quantum system.

There is another kind of phase known as *relative phase*. Consider the states

$$|\psi_1\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \text{and} |\psi_2\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{3.12}$$

In the first state the amplitude of $|1\rangle$ is $1/\sqrt{2}$. For the second state it is $-1/\sqrt{2}$. In each case the *magnitude* of the amplitudes is the same, but they differ in sign. More generally, we say that two amplitudes, a and b, *differ by a relative phase* if there is a real θ such that $a = e^{i\theta}b$. More generally still, two states are said to *differ by a relative phase* in some basis if each of the amplitudes in that basis is related by such a phase factor. For example, the two states displayed above are the same up to a relative phase shift because the $|0\rangle$ amplitudes are identical (a relative phase factor of 1), and the $|1\rangle$ amplitudes differ only by a relative phase factor of -1. The difference between relative phase factors and global phase factors is the fact that relative phase factors may vary from amplitude to amplitude. This makes the relative phase a basis-dependent concept unlike global phase. As a result, states which differ only by relative phases in some basis give rise to physically observable differences in measurement statistics, and it is not possible to regard these states as physically equivalent.

3.4 No-Cloning Theorem

The *No-Cloning Theorem* states that there is no operation that can produce a copy of an arbitrary quantum state[1]. This prevents one from making a copy of an arbitrary quantum system and observing that copy to get around the problem of observation collapsing the system. Thus observation not only produces an answer, but the only way to obtain another answer is to re-run the computation. Due to the No-Cloning Theorem there is no way to *"undo"* an observation. In essence, the No-Cloning Theorem implies that there is a limited amount of information we can obtain from an arbitrary quantum state. However, the above argument does not prohibit the copying of orthogonal states. This is because orthogonal states are distinguishable. So, states $|0\rangle$ and $|1\rangle$ *can* be copied. This is a strong intuition that *quantum computation is at least as powerful as classical.*

[1] Even though it is impossible to make perfect copies of an unknown quantum state, it is possible to produce imperfect copies. This can be done by coupling a larger auxiliary system to the system that is to be cloned, and applying a unitary transformation to the combined system. If the unitary transformation is chosen correctly, several components of the combined system will evolve into approximate copies of the original system. In 1996, V. Buzek and M. Hillery showed that a universal cloning machine can make a clone of an unknown state with the surprisingly high fidelity of 5/6.

Chapter 4

Quantum Circuit Model

Changes occurring to a quantum state can be described using the language of quantum computation. Analogous to the way a classical computer is built from an electrical circuit containing wires and logic gates, a quantum computer is built from a quantum circuit containing wires and elementary *quantum gates* to carry around and manipulate the *quantum information*.

4.1 Introduction

Let's look at the elements of a quantum circuit. We can see an example circuit in figure 4.1 The circuit is to be read from left-to-right. Each line in the circuit represents a wire in the quantum circuit. This wire does not necessarily correspond to a physical wire; it may correspond instead to the passage of time, or perhaps to a physical particle such as a photon – a particle of light – moving from one location to another through space. It is conventional to assume that the state input to the circuit is a computational basis state, usually the state consisting of all $|0\rangle$'s. This rule is broken frequently in the literature on quantum computation and quantum information, but it is considered polite to inform the reader when this is the case.

There are a few features allowed in classical circuits that are not usually present in quantum circuits. First of all, we don't allow "loops", that is, feedback from one part of

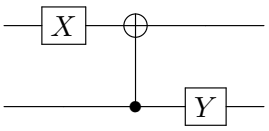


FIGURE 4.1: An example circuit

the quantum circuit to another; we say the circuit is acyclic. Second, classical circuits allow wires to be "joined" together, an operation known as `FANIN`, with the resulting single wire containing the bitwise `OR` of the inputs. Obviously this operation is not reversible and therefore not unitary, so we don't allow `FANIN` in our quantum circuits. Third, the inverse operation, `FANOUT`, whereby several copies of a bit are produced is also not allowed in quantum circuits. This is a direct consequence of the no-cloning theorem!

The squares and symbols that wires pass through are quantum gates. A quantum gate perform an operation on the qubits of the wires it acts on. What are the conditions for a quantum gate to be valid? It turns out that the appropriate condition on the matrix U representing the gate is that it must be *unitary*, that is $U^\dagger U = I$. Amazingly, this *unitarity* constraint is the *only* constraint in quantum gates. *Any unitary matrix specifies a valid quantum gate!*

Finally, a common used term is the *quantum register*. This is simply a group of qubits, that we group together for convenience of analysing algorithms. For example, we may have a circuit of 8 qubits, and say register A consists of the first five qubits and register B of the last three.

4.2 Single Qubit Operations

In classical computing, the only non-trivial gate operating in a single bit is the NOT gate, whose operation is defined by its *truth table*, in which $0 \mapsto 1$ and $1 \mapsto 0$, that is the value of the bit is flipped.

Can an analogous quantum NOT gate for qubits be defined? Imagine that we have a process which takes the state $|0\rangle$ to the state $|1\rangle$, and vice versa. Such a process would obviously be a good candidate for a quantum analogue to NOT gate. However, specifying the action of the gate on the states $|0\rangle$ and $|1\rangle$ does not tell us what happens to their superpositions, without the knowledge of another property of quantum gates; *linearity*. In fact, quantum NOT gate, as all quantum gates, acts on qubits *linearly*[1]. That is, it takes the state

$$\alpha|0\rangle + \beta|1\rangle$$

to the corresponding state in which the role of $|0\rangle$ and $|1\rangle$ are interchanged,

$$\alpha|1\rangle + \beta|0\rangle$$

[1] Why quantum gates act linearly and not in some non-linear fashion is a very interesting question and way beyond the scope of this thesis to answer. In short, it turns out this linear behaviour is a general property of quantum mechanics.

In quantum computing, as has been stated before, a single qubit is a vector $|\psi\rangle = a|0\rangle + b|1\rangle$, where a and b are complex numbers satisfying $|a|^2 + |b|^2 = 1$. Operations on a qubit must preserve this norm, and thus are described by $2x2$ unitary matrices. Quantum NOT gate is defined by such a matrix and denoted by symbol X:

$$X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

If the quantum state $|\psi\rangle = a|0\rangle + b|1\rangle$ is written in vector notation as

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix},$$

then the corresponding output from the quantum NOT gate is

$$X \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}$$

Quantum NOT gate is often called the Pauli X gate. There are four Pauli gates in total:

$$X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad Y \equiv \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad Z \equiv \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} I \equiv \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

The last one is the identity gate. It maps any state onto itself. That is, $I|\psi\rangle = |\psi\rangle$.

Three other single qubit quantum gates that play a major part in quantum computing are the Hadamard gate (denoted H), phase gate (denoted S) and the $\pi/8$ gate (denoted T):

$$H \equiv \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad S \equiv \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \quad T \equiv \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$$

From these, the Hadamard gate is specially important. This gate is sometimes described as being like a "square-root of NOT" gate, in that it turns a $|0\rangle$ into $\frac{(|0\rangle + |1\rangle)}{\sqrt{2}}$, which is "halfway" between $|0\rangle$ and $|1\rangle$, and turns $|1\rangle$ into $\frac{(|0\rangle - |1\rangle)}{\sqrt{2}}$, which is also "halfway" between $|0\rangle$ and $|1\rangle$. Note, however that H^2 is not a NOT gate, as it can be easily shown that $H^2 = I$, and thus applying H twice to a state does nothing to it.

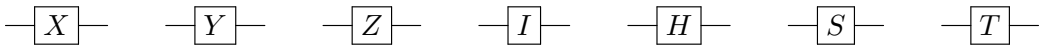


FIGURE 4.2: Circuit representation of most used single-qubit quantum gates. From left to right, Pauli X, Pauli Y, Pauli Z, Identity, Hadamard, Phase and $\pi/4$ gate.

4.3 Multiple Qubit Operations

"If A is true, then B". This type of controlled operation is one of the most useful in computing, both classical and quantum.

Suppose U is any unitary matrix acting on some number n of qubits, so U can be regarded as a quantum gate on those qubits. Then we can define a *controlled-U* gate. Such a gate has a single control qubit, indicated by the line with the black dot, and n target qubits, indicated by the boxed U. If the control qubit is set to 0 then nothing happens to the target qubits. If the control qubit is set to 1 then the gate U is applied to the target qubits.

The prototypical example of the controlled-U gate is the controlled-NOT gate, which is a controlled-U gate with $U = X$. We'll often refer to it as CNOT. In terms of the computational basis, the action of the CNOT is given by $|c\rangle|t\rangle \mapsto |c\rangle|t \oplus c\rangle$. The matrix representation of CNOT is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.1}$$

and its circuit representation is shown in Figure **??**.

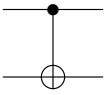


FIGURE 4.3: Circuit representation of the controlled-NOT gate. The top line represents the control qubit, the bottom line the target qubit.

Any complex controlled gate can be constructed from the controlled-NOT gate and one-qubit gates. For example, if we would like to construct an "inverse" controlled-NOT gate, in the sense that NOT is applied on the target qubit if the control qubit is 0 instead of 1, we can easily do so as show in Figure **??**.

Another example is shown in Figure 4.5. On the left is a gate, similar to controlled-NOT gate, which is however controlled by 2 qubits. This gate is known as the *Toffoli*

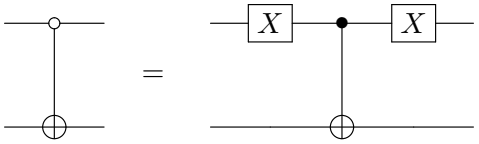


FIGURE 4.4: Circuit of "inverse" CNOT gate.

gate. On the right is a simple implementation using Hadamard, phase, controlled-NOT and gates.

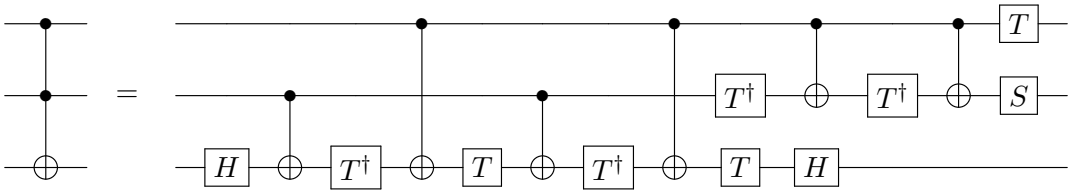


FIGURE 4.5: Implementation of the Toffoli gate.

4.4 Measurement

A final element used in quantum circuits, almost implicitly sometimes, is measurement. In our circuits, we shall denote a projective measurement in the computational basis using a "meter" symbol, illustrated in Figure 4.6.

FIGURE 4.6: Symbol for projective measurement on a single qubit. In this circuit nothing further is done with the measurement result, but in more general quantum circuits it is possible to change later parts of the quantum circuit, conditional on measurement outcomes in earlier parts of the circuit. Such a usage of classical information is depicted using wires drawn with double lines.

4.5 Example Circuits

4.5.1 Swap Gate

Swapping a bit is a simple but very useful task. The same can be said for swapping a qubit. The swap operation is accomplished by the circuit in Figure 4.7. To see why this is the case, note that the sequence of gates has the following sequence of effects

on a computational basis state $|a, b\rangle$,

$$\begin{aligned} |a, b\rangle &\mapsto |a, a \oplus b\rangle \\ &\mapsto |a \oplus (a \oplus b), a \oplus b\rangle = |b, a \oplus b\rangle \\ &\mapsto |b, (a \oplus b) \oplus b\rangle = |b, a\rangle \end{aligned}$$

where all additions are done modulo 2. The effect of the circuit, therefore, is to interchange the state of the two qubits.

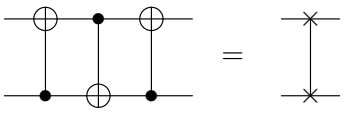


FIGURE 4.7: Circuit swapping two qubits, and an equivalent schematic symbol notation for this common and useful circuit.

4.5.2 Bell States

Let's take a look at a slightly more complicated circuit, shown in Figure 4.8, which has a Hadamard gate followed by a CNOT. If the circuit takes the input $|00\rangle$, then the Hadamard gate transforms it to $(|0\rangle + |1\rangle)|0\rangle/\sqrt{2}$, and then the CNOT gives the output $(|00\rangle + |11\rangle)/\sqrt{2}$. We saw this state when we talked about entanglement. Now we know how to produce it with a quantum circuit. This state is one of the four possible outputs the circuit produces depending on the input. The output states

$$|\beta_{00}\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{4.2}$$

$$|\beta_{01}\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}} \tag{4.3}$$

$$|\beta_{10}\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}} \tag{4.4}$$

$$|\beta_{11}\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}} \tag{4.5}$$

are known as the *Bell states*, or sometimes the *EPR states* or *EPR pairs*, after the people - Bell, and Einstein, Podolsky, Rosen - who first pointed out the strange properties of states like these. The mnemonic notation $|\beta_{00}\rangle$, $|\beta_{01}\rangle$, $|\beta_{10}\rangle$, $|\beta_{11}\rangle$ may be understood via the equation

$$|\beta_{xy}\rangle = \frac{|0, y\rangle + (-1)^x |1, \bar{y}\rangle}{\sqrt{2}} \tag{4.6}$$

where $\bar{y}$ is the negation of y.

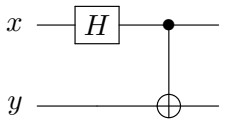


FIGURE 4.8: Quantum circuit to create Bell states

4.5.3 Quantum Teleportation

Another famous example of quantum computation capabilities is the, otherwise very simple, *quantum teleportation* protocol. Quantum teleportation is a technique for moving quantum states around, even in the absence of a quantum communication channel linking the sender of the quantum state to the recipient.

Here's how quantum teleportation works. Alice and Bob met long ago, but now live very far apart. While together they generated an EPR pair, each taking one qubit of the EPR pair when they seperated. Many years later, Alice wants to send a qubit $|\psi\rangle$ to Bob. Of course, she does not know the state of the qubit, and moreover she can only send *classical* information to Bob. Can she achieve this? Fortunately, for Alice, quantum teleportation is a way of utilizing the entangled EPR pair in order to send $|\psi\rangle$ to Bob, with only a small overhead of classical communication.

The steps of the protocol are as follows: Alice interacts her qubit $|\psi\rangle$ with her half of the EPR pair, and then measures the two qubits, obtaining one of the four possible classical results, $00, 01, 10, 11$. She sends her information to Bob. Depending on Alice's classical message, Bob performs one of four operations on his half of EPR pair. Surprisingly, by doing this he can recover the original state $|\psi\rangle$

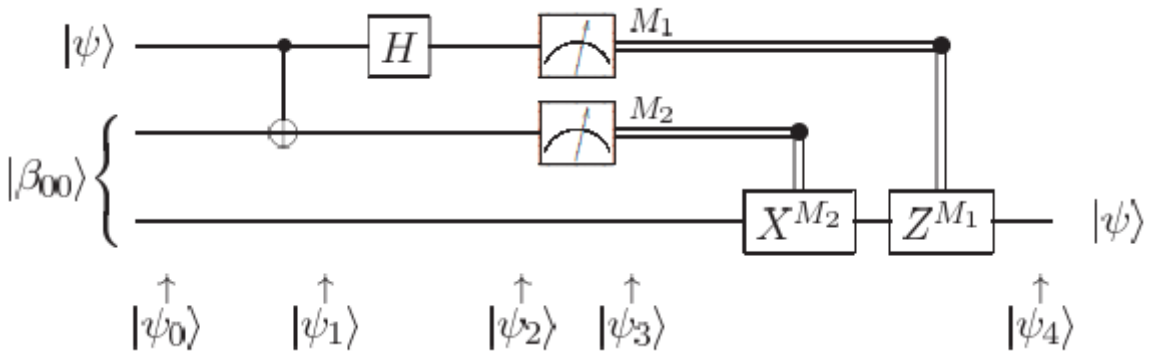


FIGURE 4.9: Quantum circuit for teleporting a qubit. The meters represent measurement, and the double lines coming out of them carry classical bits.

The quantum circuit shown in Figure 4.9 gives a more precise description of quantum teleportation. The state to be teleported is $|\psi\rangle = a|0\rangle + b|1\rangle$, where a and b are unknown amplitudes. The state input $|\psi_0\rangle$ into the circuit is

$$|\psi_0\rangle = |\psi\rangle|\beta_{00}\rangle \tag{4.7}$$

$$= \frac{1}{\sqrt{2}}\left[a|0\rangle(|00\rangle + |11\rangle) + b|1\rangle(|00\rangle + |11\rangle)\right] \tag{4.8}$$

where by convention the first two qubits belong to Alice, and the third qubit to Bob. Alice sends here qubits through a CNOT gate, obtaining

$$|\psi_1\rangle = \frac{1}{\sqrt{2}}\left[a|0\rangle(|00\rangle + |11\rangle) + b|1\rangle(|10\rangle + |01\rangle)\right] \tag{4.9}$$

She then send the first qubit through a Hadamard gate, obtaining

$$|\psi_2\rangle = \frac{1}{2}\left[a(|0\rangle + |1\rangle)(|00\rangle + |11\rangle) + b(|0\rangle - |1\rangle)(|10\rangle + |01\rangle)\right] \tag{4.10}$$

This state may be re-written in the following way, simply by regrouping terms:

$$\begin{aligned}|\psi_2\rangle = \frac{1}{2}[&|00\rangle(a|0\rangle + b|1\rangle) + |01\rangle(a|1\rangle + b|0\rangle) \\ &+|10\rangle(a|0\rangle - b|1\rangle) + |11\rangle(a|1\rangle - b|0\rangle)]\end{aligned} \tag{4.11}$$

This expression naturally breaks down into four terms. The first term has Alice's qubits in state $|00\rangle$, and Bob's qubit in the state $a|0\rangle + b|1\rangle$. That means, if Alice performs a measurement and obtains the result 00 then Bob's system will be in the state $|\psi\rangle$. Similarly, from the previous expression we can read off Bob's post-measurement state, given the result of Alice's measurement:

$$00 \longmapsto |\psi_3(00)\rangle \equiv [a|0\rangle + b|1\rangle] \tag{4.12}$$

$$01 \longmapsto |\psi_3(01)\rangle \equiv [a|1\rangle + b|0\rangle] \tag{4.13}$$

$$10 \longmapsto |\psi_3(10)\rangle \equiv [a|0\rangle - b|1\rangle] \tag{4.14}$$

$$11 \longmapsto |\psi_3(11)\rangle \equiv [a|1\rangle - b|0\rangle] \tag{4.15}$$

Depending on Alice's measurement outcome, Bob's qubit will end up in one of these four possible states. Once Bob has learned the measurement outcome, Bob can "fix up" his state, recovering $|\psi\rangle$, by applying the appropriate quantum gate. For example, in the case where the measurement yields 00, Bob doesn't need to do anything. If the measurement is 01 then Bob can fix up his state by applying the X gate. If the measurement is 10 then Bob can fix up his state by applying the Z gate. If the

measurement is 11 then Bob needs to apply first an X and then a Z gate. Summing up, Bob need to apply the transformation $Z^{M_1}X^{M_2}$[2].

Quantum teleportation has a lot to teach us, and the results can be very deep. Quantum teleportation emphasizes the interchangeability of *different* resources in quantum mechanics, showing that one shared EPR pair together with two classical bits of communication is a resource at least the equal of one qubit of communication. Many more methods of interchanging resources are built upon quantum teleportation. Moreover, a plethora of the properties of quantum error-correcting codes are intimately connected with quantum teleportation.

4.6 Quantum Parallelism

Quantum parallelism is a fundamental feature of many quantum algorithms. In short, and maybe with a bit of over-simplification, quantum parallelism allows quantum computers to evaluate a function $f(x)$ for many *different* values of x simultaneously.

Suppose $f(x)\{0,1\} \mapsto \{0,1\}$ is a function with a one-bit domain and range. We consider a two qubit quantum circuit which starts in the state $|x,y\rangle$. With an appropriate sequence of logic gates it is possible to transform this state into $|x, y \oplus f(x)\rangle$, where $\oplus$ indicates addition modulo 2. We give the transformation defined by the map $|x,y\rangle \mapsto |x, y \oplus f(x)\rangle$ a name U_f, that is unitary[3].

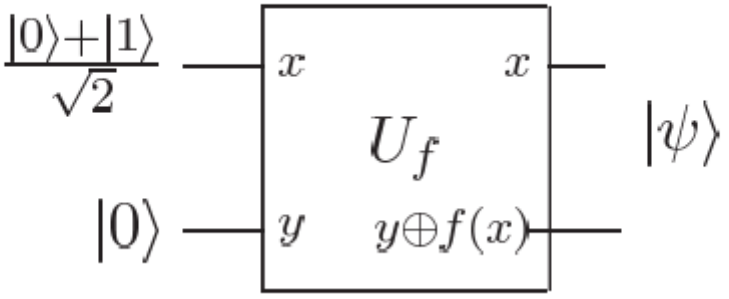


FIGURE 4.10: Quantum circuit for evaluating $f(0)$ and $f(1)$ simultaneously. U_f is the quantum circuit which takes inputs like $|x,y\rangle$ to $|x, y \oplus f(x)\rangle$.

If $y = 0$, then the final state is just the value $f(x)$. Now consider that we first apply a Hadamard gate on first qubit preparing it to the superposition $hfrac(|0\rangle + |1\rangle)\sqrt{2}$. If

[2]Note how time goes from left to right in circuit diagrams, but in matrix products terms on the *right* happens *first*

[3]It is proven that given a classical circuit for computing f there is a quantum circuit of comparable efficiency which computes the transformation U_f on a quantum computer. For our purposes it can be considered to be a *black box*.

we now apply U_f the resulting state is

$$\frac{|0, f(0)\rangle + |1, f(1)\rangle}{\sqrt{2}} \tag{4.16}$$

This is a remarkable state! The different terms contain information about both $f(0)$ and $f(1)$. It is almost as if we have evaluated $f(x)$ for two values of x simultaneously. This feature is called *quantum parallelism*. Unlike classical parallelism, where multiple circuits each build to compute $f(x)$ are executed simultaneously, here a *single* $f(x)$ circuit is employed to evaluate the function for multiple values of x simultaneously, by exploiting the ability of a quantum state to be in superposition.

This procedure can easily be generalized to functions of an arbitrary number of qubits, by using a general operation known as the *Hadamard transform*, or sometimes the *Walsh-Hadamard transform*. This operation is just n Hadamard gates acting in parallel on n qubits. For example, shown in Figure 4.11 is the case $n = 2$ with qubits initially prepared as $|0\rangle$, which gives

$$\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right)\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) = \frac{|00\rangle + |01\rangle + |10\rangle + |11\rangle}{\sqrt{2}} \tag{4.17}$$

as output. We write $H^{\otimes 2}$ to denote the parallel action of two Hadamard gates, and read $\otimes$ as "tensor". More generally, the result of performing the Hadamard transform of n qubits initially in the $|0\rangle$ state is

$$\frac{1}{\sqrt{2}} \sum_x |x\rangle, \tag{4.18}$$

where the sum is over all possible values of x, and we write $H^{\otimes n}$ to denote this action. That is, the Hadamard transform produces an equal superposition of all computational basis states. Moreover, it does this extremely efficiently, producing a superposition of 2^n states using just n gates.

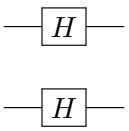


FIGURE 4.11: The Hadamard transform $H^{\otimes 2}$ on two qubits.

Chapter 5

Quantum Algorithms

5.1 Quantum Complexity Theory

It is impossible to imagine today's technological world without algorithms: sorting, searching, calculating, and simulating are being used everywhere to make our everyday lives better.

One natural question is what kinds of problem are susceptible to attack by a quantum computer. Unfortunately, even the classical analog of this question: *What kind of problems can be solved in polynomial time by a digital computer?* does not have a satisfactory answer. Computer scientists have a plethora of techniques they can try to apply to a problem: linear programming, divide-and-conquer, dynamic programming, Monte Carlo methods, semidefinite programming, and so forth. However, deciding which of these methods is likely to work for a given problem, and how to apply it, remains more of an art than a science, and there is no good way known to characterize the class of problems having polynomial-time algorithms. Characterizing the class of problems having polynomial-time quantum algorithms appears equally, if not more, difficult, one of the main additional difficulties being that we have so far discovered very few algorithmic techniques.

Computational complexity theory is the subject of classifying the difficulty of various computational problems, both classical and quantum, and to understand the power of quantum computers we will first examine some general ideas from computational complexity. The most basic idea is that of a *complexity class*. A complexity class can be thought of as a collection of computational problems, all of which share some common feature with respect to the computational resources needed to solve those problems.

The complexity class **P** consists of those problems which can be solved using algorithms running in time bounded by a polynomial in the length of the input. The class of problems with probabistic polynomial-time algorithms up to bounded probability of error (up to 1/4 is considered acceptable) is called **BPP**. Polynomial running times are considered to be efficient[1] by theoretical computer scientists. The class **NP** consists of those problems for which a solution can be verified in polynomial time. This class contains **P** and whether **P**≠**NP** is probably the most important unsolved problem in theoretical computer science.

Most researchers believe that **NP** contains problems that are not in **P**. In particular, there is an important subclass of the **NP** problems, the **NP-complete** problems, that are of especial importance for two reasons. First, there are thousands of problems, many highly important, that are known to be **NP-complete**. Second, any given **NP-complete** problem is proven to be *at least as hard* as all other problems in **NP**. That is, an algorithm to solve a specific **NP-complete** problem can be adapted to solve any other problem in **NP**. In particular, if **P**≠**NP**, then it follows that no **NP-complete** problem can be efficiently solved on a classical computer, and thus makes the study of quantum computing even more intriguing!

Another important complexity class is **PSPACE**. Roughly speaking, **PSPACE** consists of those problems which can be solved using resources which are few in spatial size, but not necessarily in time. That means, there exists algorithms that can solve problems in **PSPACE** using polynomially bounded bits, but we don't mind if they need non-polynomial computation time. **PSPACE** is believed to be strictly larger than both **P** and **NP** although, again, this has never been proved.

What of quantum complexity classes? We can define **BQP**[2] to be the class of all computational problems which can be solved efficiently by a quantum probabilistic polynomial-time algorithm up to a bounded probability of error. Exactly where **BQP** fits to respect to **P**, **NP** and **PSPACE** is unknown. What is known is that quantum computers can solve all problems in **P** efficiently, but there are no problems outside of **PSPACE** which they can solve efficiently. Therefore, **BQP** lies somewhere between **P** and **PSPACE** as illustrated in Figure **??**. There are many more classes and subclasses, both classical and quantum, but their complexity and relations are far beyond the scope of this thesis.

[1] This isn't strictly true; nobody would call an algorithm that runs in n^{100} steps efficient in practice, but this definition has proven to be a good compromise between theory and practice. It appears to be the case that most natural problems in **P** have algorithms with running time a relatively small power of n.

[2] Quantum algorithms are in general inherently probabilistic, and so the class **BQP** should most fairly be compared with the class **BPP** rather than **P**.

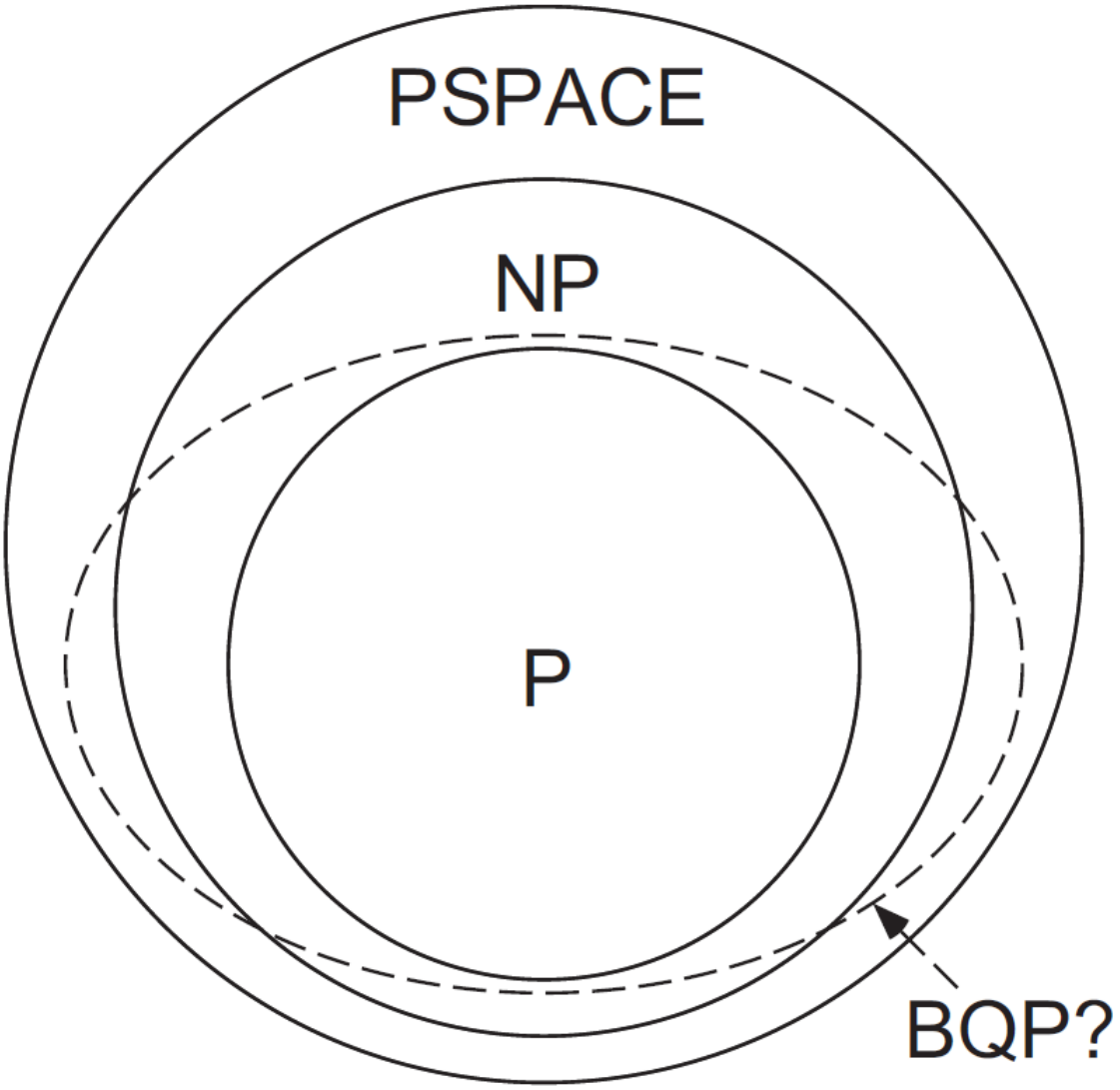


FIGURE 5.1: The relationship between classical and quantum complexity classes. Quantum computers can quickly solve any problem in **P**, and it is known that they can't solve problems outside of **PSPACE** quickly. Where quantum computers fit between **P** and **PSPACE** is not yet known.

In the following sections some interesting quantum algorithms will be presented. These algorithms solve important computational problems and are proven to have some degree of speedup over their best known classical analogue.

5.2 Deutsch's Algorithm

Deutsch's algorithm is very simple and makes use of quantum parallelism to demonstrate how quantum circuits can outperform classical ones. As in circuit in Figure

4.10 before, we use the Hadamard gate to prepare the first qubit to the superposition $(|0\rangle + |1\rangle)/\sqrt{2}$, but now we also prepare the second qubit y to the superposition $(|0\rangle - |1\rangle)/\sqrt{2}$, using a Hadamard gate applied to state $|1\rangle$. Let us follow the states to see what happens in this circuit, shown in Figure 5.2.

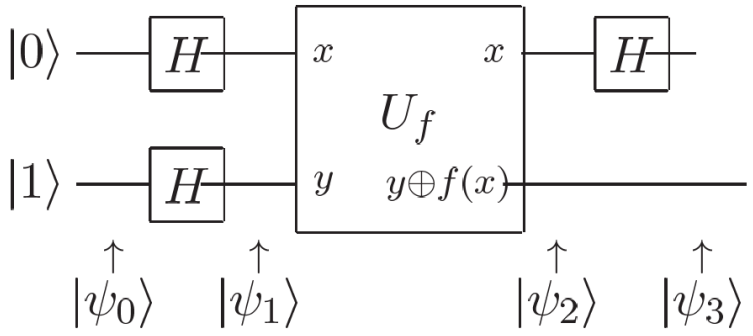


FIGURE 5.2: Quantum circuit implementing Deutch's algorithm.

The input state

$$|\psi_0\rangle = |01\rangle \tag{5.1}$$

is sent through a Hadamard transform to give

$$|\psi_1\rangle = \left[\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right]\left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right] \tag{5.2}$$

If we apply U_f to the state $|x\rangle(|0\rangle - |1\rangle)/\sqrt{2}$ then we obtain the state $(-1)^{f(x)}|x\rangle(|0\rangle - |1\rangle)/\sqrt{2}$. Therefore, applying U_f to $|\psi_1\rangle$ leaves us with one of two possibilities:

$$|\psi_2\rangle = \begin{cases} \pm\left[\frac{|0\rangle+|1\rangle}{\sqrt{2}}\right]\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right] & \text{if} f(0) = f(1) \\ \\ \pm\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right]\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right] & \text{if} f(0) \neq f(1) \end{cases} \tag{5.3}$$

The final Hadamard gate on the first qubit gives us

$$|\psi_3\rangle = \begin{cases} \pm|0\rangle\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right] & \text{if} f(0) = f(1) \\ \\ \pm|1\rangle\left[\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right] & \text{if} f(0) \neq f(1) \end{cases} \tag{5.4}$$

Realizing that $f(0) \oplus f(1)$ is 0 if $f(0) = f(1)$ and 1 otherwise, we can rewrite this result as

$$|\psi_3\rangle = \pm|f(0) \oplus f(1)\rangle\left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right], \tag{5.5}$$

so by measuring the first qubit we can determine $f(0) \oplus f(1)$. This is very interesting indeed; the quantum circuit has give us the ability to determine a *global property of* $f(x)$, namely $f(0) \oplus f(1)$, using only *one* evaluation of $f(x)$! This is faster than is possible with a classical computer, which would require at least two evaluations.

5.3 Grover's Search Algorithm

Suppose we are given an unsorted list of integers and we are asked to find the integer with the minimum value. On a classical computer, if there are N integers in total, it obviously takes $O(N)$ to find the minimum. Remarkably, there is a *quantum search algorithm*, also known as *Grover's algorithm*, which enables this search to be sped up, requiring only $O(\sqrt{N})$ operations.

Schematically, the search algorithm operates as shown in Figure 5.3. The algorithm makes use of a single n qubit register. The internal workings of the oracle, including the possibility of it needing extra qubits, are not important to the description of the algorithm. The goal of the algorithm is to find a solution to the search problem, using the smallest possible number of applications of the oracle.

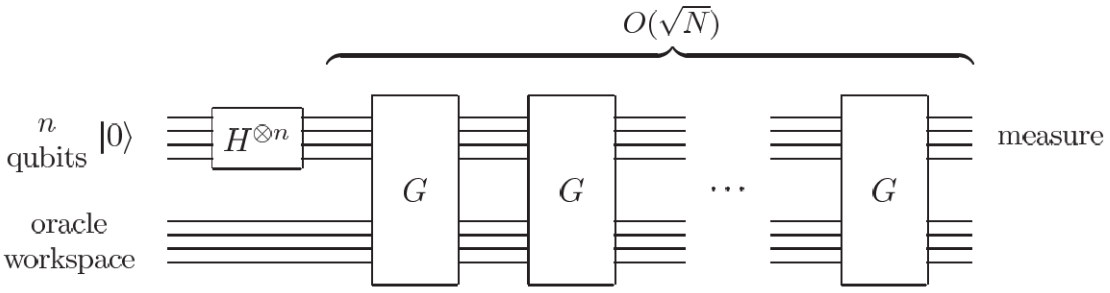


FIGURE 5.3: Schematic circuit for the quantum search algorithm. The oracle may employ work qubits for its implementation, but the analysis of the quantum search algorithm involves only the n qubit register.

The algorithm begins with the register in the state $|0\rangle^{\otimes n}$. The Hadamard transform is used to put it in the equal superposition state,

$$|\psi\rangle = \frac{1}{N^{1/2}} \sum_{x=0}^{N-1} |x\rangle \tag{5.6}$$

The quantum search algorithm then consists of repeated applications of a quantum subroutine, known as the *Grover iteration* or *Grover operator*, which we denote G. The Grover iteration, whose quantum circuit is illustrated in Figure **??**, can be broken up in four steps:

1. Apply the oracle O.
2. Apply the Hadamard transform $H^{\otimes n}$.
3. Perform a conditional phase shift on the computer, with every computational basis except $|0\rangle$ receiving a phase shift of -1,

$$|x\rangle \mapsto -(-1)^{\delta_{x_0}}|x\rangle \tag{5.7}$$

   The unitary operator corresponding to such phase shift is $2|0\rangle\langle 0| - I$.
4. Apply the Hadamard transform $H^{\otimes n}$.

Each of the operations in the Grover iteration may be efficiently implemented on a quantum computer. Steps 2 and 4, the Hadamard transforms, require $n = \log(N)$ operations each. The conditional phase shift may be implemented using $O(n)$ gates. The cost of the oracle depends upon the specific application; for now, we merely need not that the Grover iteration requires only a single oracle call. It is useful to note that the combined effect of steps 2, 3, and 4, is

$$H^{\otimes n}(2|0\rangle\langle 0| - I)H^{\otimes n} = 2|\psi\rangle\langle\psi| - I, \tag{5.8}$$

where $|\psi\rangle$ is the equally weighted superposition of states, (5.6). Thus the Grover iteration, G, may be written $G = (2|\psi\rangle\langle\psi| - I)O$.

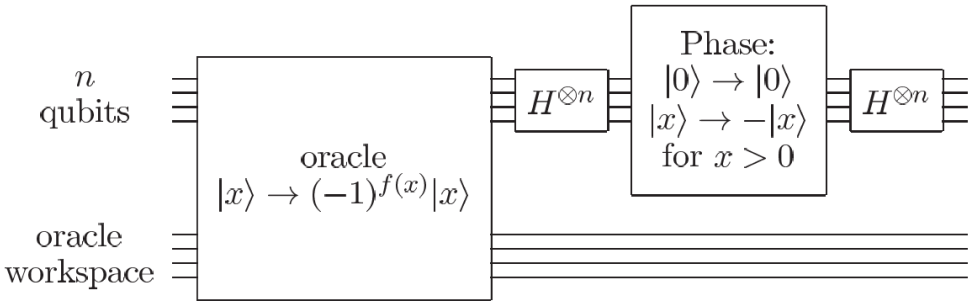


FIGURE 5.4: Schematic circuit for the Grover iteration, G.

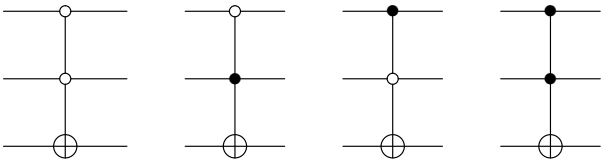
How many iterations of Grover's operator are required? It is shown that for an N item search problem with M solutions the upper bound is

$$R \leq \left\lceil \frac{\pi}{4}\sqrt{\frac{N}{M}} \right\rceil \tag{5.9}$$

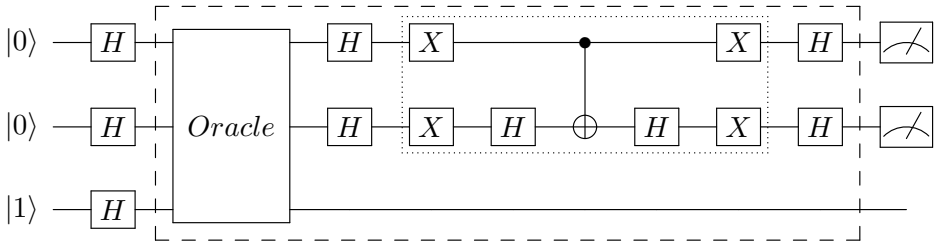
iterations. That is, $R = O(\sqrt{N/M})$ Grover iterations (and thus oracle calls) must be performed in order to obtain a solution to the search problem with high probability, a quadratic improvement over $O(N/M)$ oracle calls required classically.

5.3.1 A two-qubit example

Here is an explicit example illustrating how the quantum search algorithm works on a search space of size $N = 4$. The oracle, for which $f(x) = 0$ for all x except $x = x_0$, in which case $f(x_0) = 1$, can be taken to be one of the four circuits



corresponding to $x_0 = 0$, 1, 2, or 3 from left to right, where the top two qubits carry the query x, and the bottom qubit carries the oracle's response. The quantum circuit which perform the Grover's algorithm is



Initially, the top two qubits are prepared in the state $|0\rangle$, and the bottom one as $|1\rangle$. The gates in the dotted box perform the conditional phase shift operation $2|00\rangle\langle 00| - I$, while the gates within the dashed box perform the complete Grover iteration.

5.4 Quantum Prime Factorization

Probably, the most spectacular discovery in quantum computing to date is that quantum computers can efficiently solve prime factorization. On a classical computer, finding the prime factorization of a n-bit integer is thought to require $e^{\Theta(n^{1/3}\log^{2/3} n)}$ operations using the best known algorithm, the so-called *number field sieve*. This is exponential in the size of the number being factored, so factoring is generally considered to be an intractable problem on a classical computer, as it quickly becomes impossible to factor even modest numbers. In contrast, a quantum algorithm can accomplish the

same task using $O(n^2 \log n \log\log n)$ operations. That is, a quantum computer can factor a number *exponentially* faster than the best known classical algorithms.

The key ingredient for quantum factoring (as well as many other interesting quantum algorithms) is the *quantum Fourier transform*. The quantum Fourier transform, is an efficient quantum algorithm for performing a Fourier transform of quantum mechanical amplitudes. It does *not* speed up the classical task of computing Fourier transform of classical data. However, it enables one important task which is *phase estimation*, the approximation of the eigenvalues of a unitary operator.

5.4.1 Quantum Fourier Transform

The classical *discrete Fourier transform* takes as input a vector of complex numbers, $x_0, \ldots, x_{N-1}$, where the length N of the vector is a fixed parameter. It outputs the transformed data, a vector of complex numbers $y_0, \ldots, y_{N-1}$, defined by

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{2\pi i jk/N} \tag{5.10}$$

The *quantum Fourier transform* is exactly the same transformation, although the conventional notation is different. The quantum Fourier transform on an orthonormal basis $|0\rangle, \cdots, |N-1\rangle$ is defined to be a linear operator with the following action on the basis states:

$$|j\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi i jk/N} |k\rangle \tag{5.11}$$

Equivalently, the action on an arbitrary state may be written

$$\sum_{j=0}^{N-1} x_j |j\rangle \mapsto \sum_{k=0}^{N-1} y_k |k\rangle, \tag{5.12}$$

where the amplitudes y_k are the discrete Fourier transform of the amplitudes x_j. This transformation can be shown to be a unitary transformation, and thus can be implemented by a quantum computer.

In the following, we take $N = 2^n$, where n is some integer, and the basis $|0\rangle, \ldots, |2^n - 1\rangle$ is the computational basis for a n qubit quantum computer. It is useful to write the state $|j\rangle$ using the binary representation $j = j_1 j_2 \cdots j_n$. It is also convenient to adopt the notation $0.j_l j_{l+1} \ldots j_m$ to represent the *binary fraction* $j_l/2 + j_{l+1}/4 + \ldots + j_m/2^{m-l+1}$.

Now, with a little algebra, equation (5.11) can be written in the following useful *product representation*:

$$|j_1, \cdots, j_n\rangle \mapsto \frac{\left(|0\rangle + e^{2\pi i 0.j_n}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.j_{n-1}j_n}|1\rangle\right)\cdots\left(|0\rangle + e^{2\pi i 0.j_1 j_2 \cdots j_n}|1\rangle\right)}{2^{n/2}} \tag{5.13}$$

This product representation is so useful that some consider this to be the *definition* of the quantum Fourier transform. This representation provides insight into algorithms based upon the quantum Fourier transform and allows us to construct an efficient quantum circuit computing the Fourier transform. Such circuit is shown in Figure 5.5. The gate R_k denotes the unitary transformation

$$R_k \equiv \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix} \tag{5.14}$$

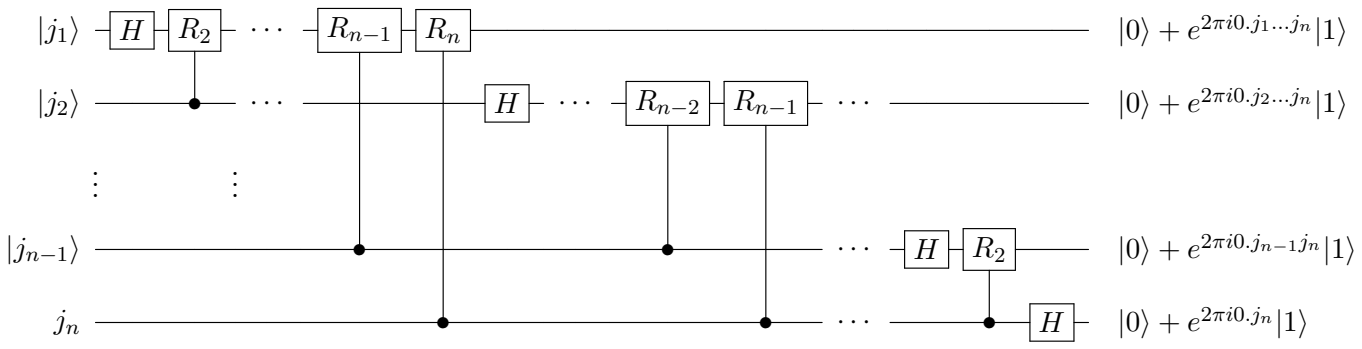


FIGURE 5.5: Circuit implementing quantum Fourier transform.

To see how the circuit computes the quantum Fourier transform, let's consider what happens to the input state $|j_1 \ldots j_n\rangle$. Applying the Hadamard gate to the first qubit produces the state

$$\frac{1}{2^{1/2}}\left(|0\rangle + e^{2\pi i 0.j_1}|1\rangle\right)|j_2 \ldots j_n\rangle \tag{5.15}$$

Applying the controlled-R_2 gate produces the state

$$\frac{1}{2^{1/2}}\left(|0\rangle + e^{2\pi i 0.j_1 j_2}|1\rangle\right)|j_2 \ldots j_n\rangle \tag{5.16}$$

We continue applying the controlled-R_3, R_4 through R_n gates, each of which adds an extra bit to the phase of the co-efficient of the first $|1\rangle$. At the end of this procedure we have the state

$$\frac{1}{2^{1/2}}\left(|0\rangle + e^{2\pi i 0.j_1 j_2 \ldots j_n}|1\rangle\right)|j_2 \ldots j_n\rangle \tag{5.17}$$

Next, we perform a similar procedure on the second qubit. First we apply the Hadamard gate and then the controlled-R_2 through R_{n-1} gates and we obtain the state

$$\frac{1}{2^{2/2}}\left(|0\rangle + e^{2\pi i 0.j_1 j_2 \ldots j_n}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.j_2 \ldots j_n}|1\rangle\right)|j_3 \ldots j_n\rangle \tag{5.18}$$

We continue in this fashion for each qubit, giving the final state

$$\frac{1}{2^{2/2}}\left(|0\rangle + e^{2\pi i 0.j_1 j_2 \ldots j_n}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.j_2 \ldots j_n}|1\rangle\right)\cdots\left(|0\rangle + e^{2\pi i 0.j_n}|1\rangle\right) \tag{5.19}$$

As a final step we apply some swap operations, which were omitted from Figure 5.5 for clarity and are used to reverse the order of the qubits. After the swap operations the state of the qubits is

$$\frac{1}{2^{n/2}}\left(|0\rangle + e^{2\pi i 0.j_n}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.j_{n-1} j_n}|1\rangle\right)\cdots\left(|0\rangle + e^{2\pi i 0.j_1 j_2 \cdots j_n}|1\rangle\right) \tag{5.20}$$

which is our desired state. This circuit also proves that the quantum Fourier transform is unitary, since each gate in the circuit is unitary.

How many gates does this circuit use? We start by applying a Hadamard gate and $n-1$ conditional rotations on the first qubit - a total of n gates. This is followed by a Hadamard gate and $n-2$ rotations on the second qubit, for a total of $n+(n-1)$ gates. Continuing in this way, we see that $n + (n-1) + (n-2) + \ldots + 1 = n(n+1)/2$ gates are required plus the gates involved in the swaps. At most $n/2$ swaps are required, and each swap can be accomplished using three controlled-NOT gates. Therefore, this circuit provides a $\Theta(n^2)$ algorithm for performing the quantum Fourier transform.

In contrast, the best classical algorithms for computing the discrete Fourier transform on 2^n elements are algorithms such as the *Fast Fourier Transform (FFT)*, which use $\Theta(n2^n)$ gates. That is, it requires exponentially more operations to compute the Fourier transform on a classical computer than it does to implement the quantum Fourier transform on a quantum computer.

5.4.2 Phase Estimation

The quantum Fourier transform is the key to a procedure known as *phase estimation*. Suppose a unitary operator U has an eigenvector $|u\rangle$ with eigenvalue $e^{2\pi i\phi}$, where the value of ϕ is unknown. The goal of the phase estimation algorithm is to estimate ϕ. To perform this estimation we assume we have available *black boxes*, sometimes known as *oracles* capable of preparing the state $|u\rangle$ and performing the controlled-U^{2^j} operation, for suitable non-negative integers j. The use of black boxes indicates that the phase estimation procedure is not a complete quantum algorithm in its own right. Rather,

we should think of phase estimation as a kind of "subroutine", that when combined with other subroutines, can be used to perform interesting computational tasks.

The quantum phase estimation procedure uses two registers. The first register contains t qubits initially in the state $|0\rangle$. The value of t depends on two things: the number of digits of accuracy we wish to have in the estimate for ϕ, and with what probability we wish the phase estimation procedure to be successful. The second register begins in the state $|u\rangle$, and contains as many qubits as necessary to store $|u\rangle$.

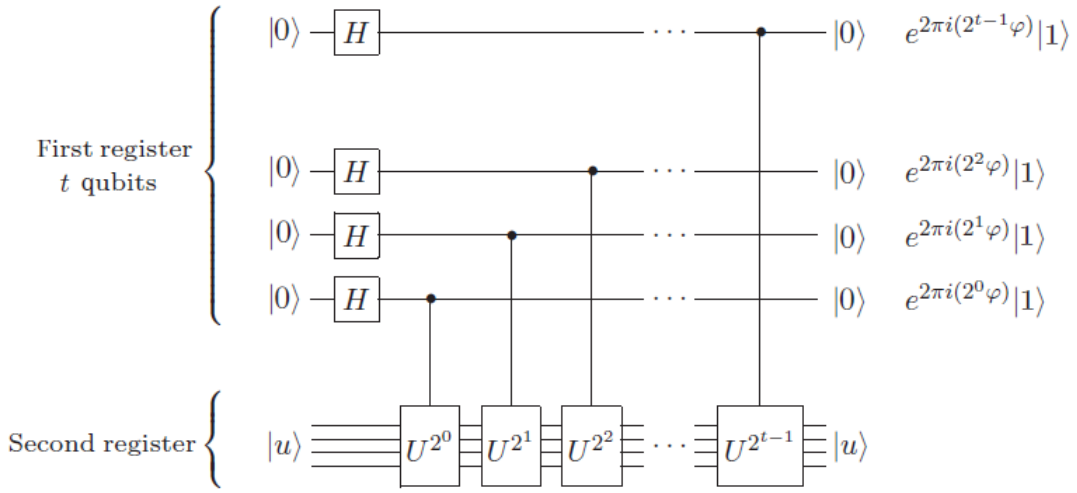


FIGURE 5.6: The first state of the phase estimation procedure. Normalization factors of $1/\sqrt{2}$ have been omitted.

The procedure is performed in two stages. First, we apply the circuit shown in Figure **??**. The circuit begins by applying the Hadamard transform to the first register, followed by application of controlled-U on the second register, with U raised to successive powers of two. The final state of the first register is:

$$\frac{1}{2^{t/2}}\left(|0\rangle + e^{2\pi i 2^{t-1}}\phi|1\rangle\right)\left(|0\rangle + e^{2\pi i 2^{t-2}}\phi|1\rangle\right)\cdots\left(|0\rangle + e^{2\pi i 2^{0}}\phi|1\rangle\right) = \frac{1}{2^{t/2}}\sum_{k=0}^{2^t-1} e^{2\pi i\phi k}|k\rangle \tag{5.21}$$

The second stage of phase estimation is to apply the *inverse* quantum Fourier transform on the first register. This is obtained by reversing the circuit for the quantum Fourier transform, and can be done in $\Theta(t^2)$ steps. Supposing ϕ can be expressed exactly in t bits, as $\phi = 0.\phi_1 \ldots \phi_t$, then the state of Equation (5.21) may be rewritten

$$\frac{1}{2^{t/2}}\left(|0\rangle + e^{2\pi i 0.\phi_t}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.\phi_{t-1}\phi_t}|1\rangle\right)\cdots\left(|0\rangle + e^{2\pi i 0.\phi_1\phi_2\ldots\phi_t}|1\rangle\right) \tag{5.22}$$

Comparing this equation with the product form for the Fourier transform, Equation (5.13), we see that the output state from the second stage is the product state $|\phi_1 \ldots \phi_t\rangle$.

The third and final stage is to read out the state of the first register by doing a measurement in the computational basis and thus obtaining ϕ.

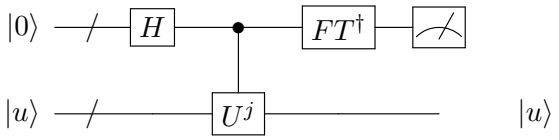


FIGURE 5.7: Schematic of the overall phase estimation procedure. The top t qubits are the first register, and the bottom qubits are the second register, numbering as many as required to perform U. $|u\rangle$ is an eigenstate of U with eigenvalue $e^{2\pi i\phi}$. To obtain ϕ accurate to n bits with probability of success at least $1-\epsilon$ we choose $t = n + \lceil \log\left(2 + \frac{1}{2\epsilon}\right) \rceil$.

The phase estimation procedure can be used to solve a variety of interesting problems.We now describe two of the most interesting of these problems: the order-finding problem, and the factoring problem. These two problems are, in fact, equivalent to one another, so in Section 5.4.3 we explain a quantum algorithm for solving the order-finding problem, and in Section 5.4.4 we explain how the order-finding problem implies the ability to factor as well.

5.4.3 Order-finding

For positive integers x and N, $x < N$, with no common factors, the *order* of x modulo N is defined to be the least positive integer, r, such that $x^r = 1(\mathrm{mod}N)$. The order-finding problem is to determine the order for some specified x and N. Order-finding is believed to be a hard problem on a classical computer.

The quantum algorithm for order-finding is just the phase estimation algorithm applied to the unitary operator

$$U|y\rangle \equiv |xy(\mathrm{mod}N)\rangle, \tag{5.23}$$

with $y \in 0, 1^L$. It is easily shown that the states defined by

$$|u_s\rangle \equiv \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \exp\left[\frac{-2\pi isk}{r}\right] \left|x^k \mathrm{mod}N\right\rangle, \tag{5.24}$$

for integer $0 \leq s \leq r-1$ are eigenstates of U, since

$$U|u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \exp\left[\frac{-2\pi isk}{r}\right] \left|x^{k+1} \mathrm{mod} N\right\rangle \qquad = \exp\left[\frac{2\pi is}{r}\right] |u_s\rangle \tag{5.25}$$

Using the phase estimation procedure allows us to obtain, with high accuracy, the corresponding eigenvalues $\exp(2\pi is/r)$, from which we can obtain the order r with a little bit more work.

There is one tricky part in this algorithm. Preparing state $|u_s\rangle$ requires that we know r, which is of course out of question. Fortunately, there is a clever observation which allows us to prepare u_s, which is that

$$\frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |u_s\rangle = |1\rangle \tag{5.26}$$

In performing the phase estimation procedure, if we use $t = 2L + 1 + \lceil \log(2 + \frac{1}{2\epsilon}) \rceil$ qubits in the first register and prepare the second register in the state $|1\rangle$, it follows that for each s in the range 0 through $r-1$, we will obtain an estimate of the phase $\phi \approx s/r$ accurate to $2L+1$ bits, with probability at least $(1-\epsilon)/r$. The order-finding algorithm is schematically shown in Figure 5.8.

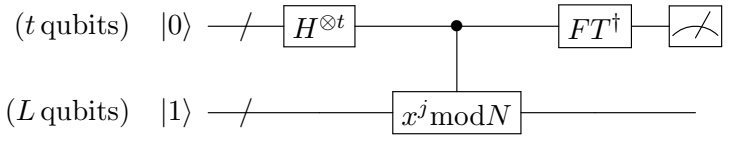


FIGURE 5.8: Schematic of the order-finding algorithm.

The final step of order-finding algorithm is to obtain the desired r, from the result of the phase estimation $\phi \approx s/r$. We only know ϕ to $2L+1$ bits, but we also know that it is a *rational* number, and if we could compute the nearest such fraction to ϕ we might obtain r. Remarkably, there is an algorithm which accomplishes this task efficiently, known as the *continued fractions algorithm*. The reason this algorithm satisfies our needs is the following theorem:

Theorem 5.1. Suppose s/r is a rational number such that

$$\left|\frac{s}{r} - \phi\right| \leq \frac{1}{2r^2} \tag{5.27}$$

Then s/r is a convergent of the continued fraction for ϕ, and thus can be computed in $O(L^3)$ operations using the continued fractions algorithm.

Since ϕ is an approximation of s/r accurate to $2L+1$ bits, it follows that $\mid s/r - \phi \mid \leq 2^{-2L-1} \leq 1/2r^2$, since $r \leq N \leq 2^L$. Thus, the theorem applies.

Summarizing, given ϕ the continued fractions algorithm *efficiently* produces numbers $\acute{s}$ and $\acute{r}$ with no common factor, such that $\acute{s}/\acute{r} = s/r$. The number $\acute{r}$ is our candidate for the order. We can check to see whether it is the order by calculating $x^{\acute{r}} \mathrm{mod} N$ and seeing if the result is 1. If so, then $\acute{r}$ is the order of $x \mathrm{modulo} N$, and we are done!

Performance

How can the order-finding algorithm fail? There are two possibilities. First, the phase estimation procedure might produce a bad estimate to s/r. This occurs with probability at most ϵ, and can be made small with a negligible increase in the size of the circuit. More seriously, it might be that s and r have a common factor, in which case the number $\acute{r}$ returned by the continued fractions algorithm be a factor of r, and not r itself. In this case, we repeat the phase estimation and continued fractions procedures twice, obtaining $\acute{r}_1$,$\acute{s}_1$ the first time, and $\acute{r}_2$,$\acute{s}_2$ the second time. Provided $\acute{s}_1$ and $\acute{s}_2$ have no common factors, r may be extracted by taking the least common multiple of r_1 and r_2. The probability that $\acute{s}_1$ and $\acute{s}_2$ have no common factors is given by

$$1 - \sum_q p(q \mid \acute{s}_1) p(q \mid \acute{s}_2) \tag{5.28}$$

where the sum is over all prime numbers q, and $p(x \mid y)$ means the probability of x dividing y. If q divides $\acute{s}_1$ then it must also divide the true value of s, s_1, on the first iteration, so the upper bound $p(q \mid \acute{s}_1)$ it suffices to upper bound $p(q \mid s_1)$, where s_1 is chosen randomly from 0 through $r-1$. It is easy to see that $p(q \mid s_1 \leq 1/q$, and thus $p(q \mid \acute{s}_1 \leq 1/q$. Similarly, $p(q \mid \acute{s}_2) \leq 1/q$, and thus the probability that $\acute{s}_1$ and $\acute{s}_2$ have no common factors satisfies

$$1 - \sum_q p(q \mid \acute{s}_1) p(q \mid \acute{s}_2) \geq 1 - \sum_q \frac{1}{q^2} \tag{5.29}$$

The right hand side can be upper bounded, which gives

$$1 - \sum_q p(q \mid \acute{s}_1) p(q \mid \acute{s}_2) \geq \frac{1}{4} \tag{5.30}$$

and thus the probability of obtaining the correct r is at least $1/4$.

What resource requirements does this algorithm has? The Hadamard transform requires $O(L)$ gates, and the inverse Fourier transform requires $O(L^2)$ gates. The major

cost in the quantum circuit actually comes from the modular exponentiation, which uses $O(L^3)$, for a total of $O(L^3)$ gates. The continued fractions algorithm adds $O(L^3)$ gates, for a total of $O(L^3)$ gates. The method for obtaining r from $\acute{r}$ needs repeating the procedures a constant number of times, so the total cost of algorithm is $O(L^3)$.

5.4.4 Prime Factorization

Given a positive integer N, what prime numbers when multiplied together equal it? This *factoring problem* turns out to be equivalent to the order-finding problem in the sense that a fast algorithm for order-finding can easily be turned into a fast algorithm for factoring.

The reduction of factoring to order-finding proceeds in two basic steps. The first step is to show that we can compute a factor of N if we can find a non-trivial solution $x \neq \pm 1(\mathrm{mod}N)$ to the equation $x^2 = 1(\mathrm{mod}N)$. The second step is to show that a randomly chosen y co-prime to N is quite likely to have an order r which is even, and such that $y^{r/2} \neq 1(\mathrm{mod}N)$, and thus $x \equiv y^{r/2}(\mathrm{mod}N)$ is a solution to $x^2 = 1(\mathrm{mod}N)$. These two steps are covered by the following theorems.

Theorem 5.2. Suppose N is an L bit composite number, and x is a non-trivial solution to the equation $x^2 = 1(\mathrm{mod}N)$ in the range $1 \leq x \leq N$, that is, neither $x = 1(\mathrm{mod}N)$ nor $x = N - 1 = -1(\mathrm{mod}N)$. Then at least one of $\gcd(x-1, N)$ and $\gcd(x+1, N)$ is a non-trivial factor of N that can be computed using $O(L^3)$ operations.

Theorem 5.3. Suppose $N = p_1^{a_1} \ldots p_m^{a_m}$ is the prime factorization of an odd composite positive integer. Let x be an integer chosen at random, subject to the requirements that $1 \leq x \leq N-1$ and x is co-prime to N. Let r be the order of x modulo N. Then

$$p(\ r \text{ is even and } x^{r/2} \neq -1(\mathrm{mod}N)) \geq 1 - \frac{1}{2^m} \tag{5.31}$$

These two theorems can be combined to give an algorithm which, with high probability, returns a non-trivial factor of any composite N. All the steps in the algorithm can be performed efficiently on a classical computer except the order-finding subroutine. The overall cost is $O((\log N)^3)$. The steps of the algorithm are the following:

1. If N is even, return the factor 2.
2. Determine whether $N = a^b$ for integers $a \geq 1$ and $b \geq 2$, and if so return the factor a.
3. Randomly choose x in the range 1 to $N-1$. If $\gcd(x, N) > 1$ then return the factor $\gcd(x, N)$.

4. Use the order-finding subroutine to find the order r of x modulo N.

5. If r is even and $x^{r^2} \neq -1(\mathrm{mod}N)$ then compute $\gcd(x^{r/2}-1, N)$ and $\gcd(x^{r/2}+1, N)$, and test to see if one of these is a non-trivial factor, returning the factor if so. Otherwise, the algorithm fails and we repeat it.

Chapter 6

CoveX

The first practical steps toward formulating a quantum programming language were made by Knill in 1996 in his proposal for conventions for a quantum pseudocode, and his description of the quantum random access machine (QRAM) model of a quantum computer. The QRAM model is built upon the (probably accurate) assumption that an practical quantum computer will in fact be a classical machine with access to quantum computing components, such as qubit registers. QRAM defines a set of specific operations to be performed on computer hardware including preparation of quantum states (from classical states), certain unitary operations, and measurement. Knill's quantum pseudocode provides a syntax for describing qubits, qubit registers, and operations involving both classical and quantum information. While extremely useful, Knill's proposal falls short of possessing all of the needed characteristics of a real quantum programming language due to its informal structure, lack of strong typing, and representation of only some of the quantum mechanical properties needed. A variety of tools have been created for simulating quantum circuits and modest quantum algorithms on classical computers using well-known languages such as C, C++, Java, and rapid prototyping languages such as Maple, Mathematica, and Matlab. A good on-line reference for these simulators is http://www.quantiki.org/wiki/index.php/List_of_QC_simulators. While simulators may provide an excellent means for quickly learning some of the basics concepts of quantum computing, they are not substitutes for actual quantum programming languages since they are designed to run only on classical computer architectures, and will not realize any of the computational advantages of quantum computing.

6.1 Cove Framework

Cove Framework was developed by Matthew Daniel Purkeypile in 2009 for his dissertation defence. Cove is framework for programming quantum computers written in C#. Cove is considered a framework and not a library since the framework can be extended by users. A framework is also easier to use in a practical sense since a user only has to search through several methods on a class in a framework once they find the appropriate class as opposed several thousand methods in a more traditional API which may only provide a collection of method calls. In light of this, a simulation of a quantum computer could be a part of the present implementation, but this could be switched out with an implementation that runs on actual quantum computers once they become viable.

There are several assumptions that this framework is based on. The encompassing theme of these assumptions is optimistic: the framework will run on an ideal quantum computer. While there are different physical challenges for building quantum computers, part of the reason for assuming the quantum computer will be ideal is to avoid building in limitations based on physical implementation problems that may be solved in the future. The assumptions taken are the following:

- **Cove is hardware independent.** Cove has been designed to be independent of the quantum hardware used to carry out quantum computation. It is not yet certain how quantum computers will be implemented physically, thus the assumption is that nothing specifically needs to be done within the programming environment for a particular physical implementation. This philosophy is much like that of existing high level languages where the particular processor does not matter, and in some cases such as Python not even the operating system.
- **Users are not concerned with error correction.** Currently, physically maintaining the state of a qubit is difficult due to decoherence. Essentially the qubits must be isolated from the outside environment so that unintended measurements do not alter the state. To help alleviate this problem, quantum error correction techniques have been developed by Shor and others. It is assumed that error correction takes place at a lower level of abstraction than the framework targets.
- **No time limit for execution.** There is no time limit that quantum states can be preserved for in a generic implementation of Cove, nor is there a limit on the number of operations that can be performed. Currently, there are limits in the current physical realizations of quantum computers for both time until decoherence and number of operations that can be performed on a register. Cove takes the optimistic approach that these problems will eventually be resolved.

- **Only quantum operations are supplied.** The framework is only concerned with carrying out quantum computation. Classical operations, including any needed for quantum algorithms, are provided by the classical language which the framework is built upon. The one exception to this is the result of measurement of a register in Cove. Any measurement of a quantum system produces a classical result, which can be considered an array of bits. In Cove this array of bits which is wrapped in a class to provide some common conversions, such as conversion to an unsigned 32 bit integer.

- **No delayed execution of operations.** Some other proposals, such as QCL, make use of a delayed execution stack. This stack builds up all the operations and sends them off to the quantum device. All operations are applied immediately in Cove. The primary motivation is that qubits may be shared between registers, and thus different computations. Thus some difficult to debug situations could arise if there was a delay in execution. The second motivation is to avoid differences between implementations, since implementations should be largely interchangeable. Thus once a call to apply an operation has returned, it can be considered done.

- **Maximum number of qubits in a register is 62.** There is a restriction on the maximum number of qubits within a register for the local simulation implementation, not the interfaces in the base library. This limitation has to do with the maximum array length within the language, which is $2^{63} - 1$ elements if a signed 64 bit integer is used for addresses. The maximum number of qubits in a register is thus set to 62, as any more would not be addressable.A register of this size would be represented by a matrix with 2^{62} elements. Assuming the complex numbers within each element only occupy 16 bytes, which is an optimistic estimation, then $16(2^{62})$ bytes or nearly 74 zettabytes would be required. So even though there is a constraint of the number of qubits in a register, the user will run into memory constraints long befor that limitation. Nonetheless, in order not to place arbitrary constraints on the simulation the maximum number of qubits is not set lower as in some other quantum programming techniques.

6.2 Cove Extended

Cove Extended, or CoveX in short, is based on Cove Framework extending its functionality and providing a graphical user interface (GUI). It is developed on Unity3D game engine. It is aspired to become the *first online collaboration tool for creating and simulating quantum circuits.*

6.2.1 Unity3D

Unity3D is a powerful cross-platform 3D engine and a user friendly development environment. Easy enough for the beginner and powerful enough for the expert, Unity should interest anybody who wants to easily create 3D games and interactive applications for mobile, desktop, the web, and consoles. Here we will cover the basics of the structure of a Unity application.

Unity's interface is quite simple and straightforward and is shown in Figure **??**. One thing to not is that every application in Unity is a *project*, and each project can contain multiple *scene*. Scenes represent different parts of a game, for example two different levels, and contain multiple *GameObjects*.

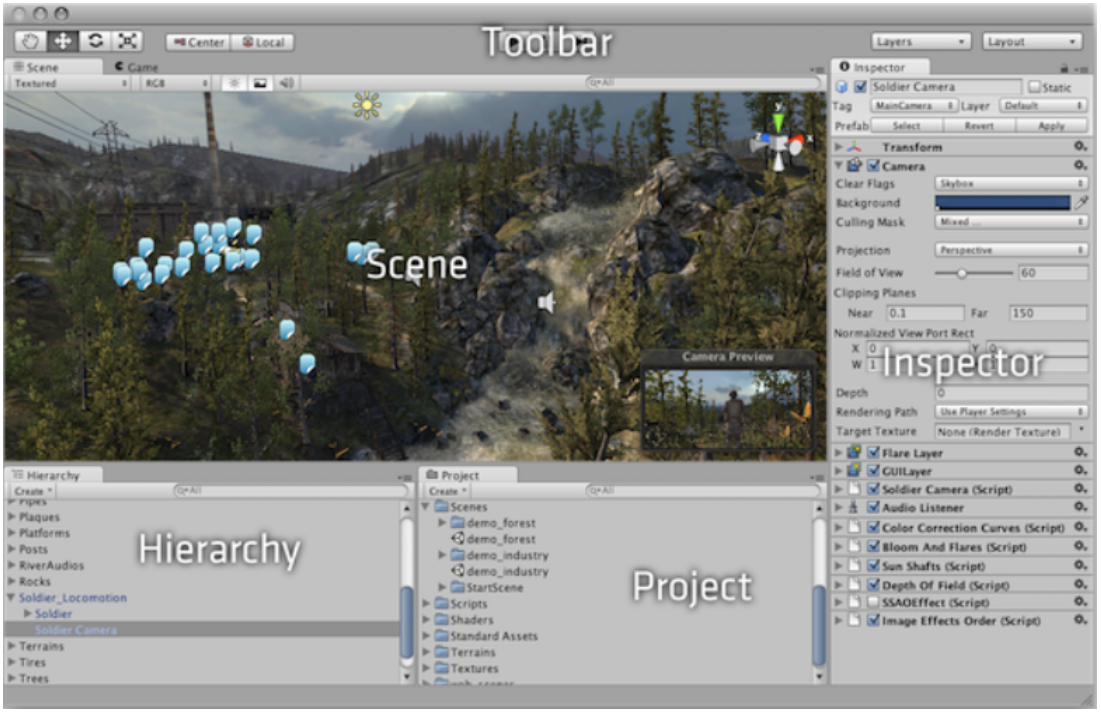


FIGURE 6.1: Unity's interface. In Project browser all the assets that belong to the project can be accessed. Hierarchy contains every GameObject in the current scene. The Scene View is the interactive sandbox, where the placement and orientation of GameObjects takes place. The Game View is rendered from the Camera in the game. It is representative of the final, published game. The Inspector displays detailed information about currently selected GameObject in the Hierarchy, including all attached Components and their properties.Any property that is displayed in the Inspector can be directly modified. Finall, Toolbar provides controls to manipulate gameobjects and scenes. For example, in the far left are Transform Tools, used in the Scene View to change the position, rotation and scale of a GameObject, while in the center are the Play and Pause buttons.

GameObjects are the fundamental concept in Unity. In short, every object in the game is a GameObject. However, they do not accomplish much in themselves but they act as *containers* for *Components*, which implement the real functionality.A GameObject

is a container for many different Components. By default, all GameObjects automatically have a *Transform Component*. This is because the Transform dictates where the GameObject is located, and how it is rotated and scaled. Without a Transform Component, the GameObject wouldn't have a location in the world.

When you create a *script* and attach it to a GameObject, the script appears in the GameObject's Inspector just like a Component. This is because scripts become Components when they are saved - a script is just a specific type of Component. In technical terms, a script compiles as a type of Component, and is treated like any other Component by the Unity engine. So basically, a script is a Component that you are creating yourself. You will define its members to be exposed in the Inspector, and it will execute whatever functionality you've written. By default, all public variables of the script are exposed.

A script makes its connection with the internal workings of Unity by implementing a class which derives from the built-in class called *MonoBehaviour*. You can think of a class as a kind of blueprint for creating a new Component type that can be attached to GameObjects. Each time you attach a script component to a GameObject, it creates a new instance of the object defined by the blueprint. The name of the class is taken from the name you supplied when the file was created. The class name and file name must be the same to enable the script component to be attached to a GameObject.

This class can be considered a usual C# however there are two major differences:

1. Initialization of an object is not done using a constructor function. This is because the construction of objects is handled by the editor and does not take place at the start of gameplay as you might expect. If you attempt to define a constructor for a script component, it will interfere with the normal operation of Unity and can cause major problems with the project.

2. There is a certain life cycle of scripts in Unity tightly bound to the game's frames. This lifecycle is exposed to developers by certain functions. These functions are executed at certain points of the script's life cycle. Three of the most important are the *Awake()*, *Start()*, and *Update()*. Awake is executed once after the object has been initialized. Start is executed before the first frame update and after every other active objects have been initialized. Update is called once per frame. It is the main function for object manipulation during gameplay. The complete life cycle of a script is shown in Figure **??**.

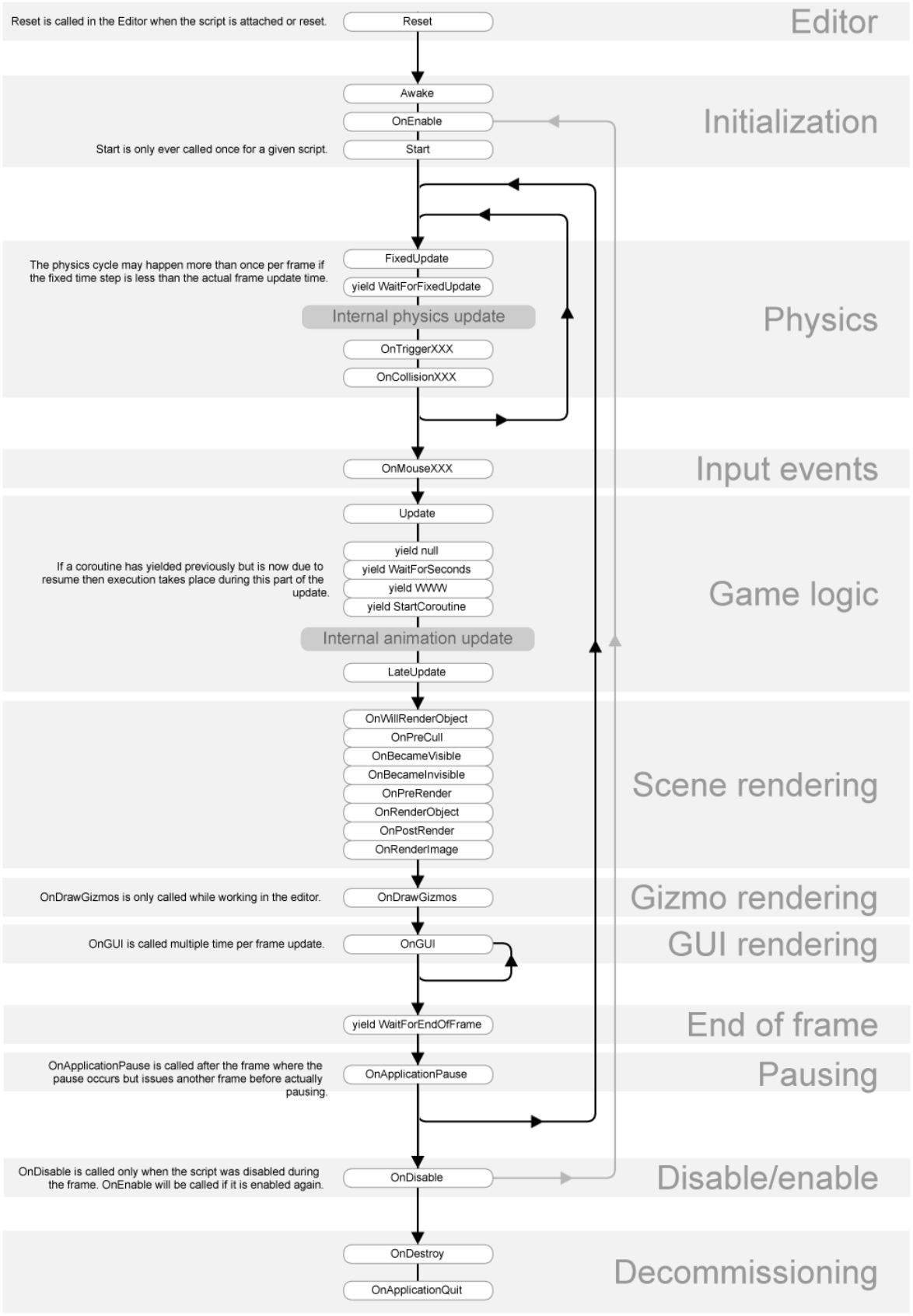


FIGURE 6.2: The lifecycle of a script in Unity.

6.2.2 Development

CoveX comes with a simple and straightforward UI, which is shown in Figure 6.3. User interacts with only the mouse; left- and right-clicking and scrolling up and down. The UI is developed with the new UI system Unity presented in version 4.6.

In respect to programming architecture, CoveX is based heavily on the Singleton pattern. Singleton pattern is a design pattern that restricts the instantiation of a class to one object. It also provides global, application wide, access to that instantiation. It is especially useful for *manager* classes that control global properties of an application, like sound, databases, application logic etc. Implementation of Singleton can be sometimes tricky, since we need to decide at which point of the application should we instantiate the class and ensure this happens only one. However, in Unity this task is trivial, since Awake() is called only once for all classes. Let's see the code for creating a Database manager:

```
public class CX_Database : Monobehaviour
{

    static CX_Database instance;
    public static CX_Database Instance { get { return instance; } }

    public void Awake()
    {
        instance = this;
    }
}
```

We can access public variables, properties, and functions of `CX_Database` through Instance from anywhere:

```
CX_Database.Instance.GetSpriteOfLine();
```

The are six Singleton classes in CoveX managing their appropriate section:

- `CX_Circuit`
  This class contains all information about the current circuit. It is concerned only about the visual aspects of the circuit and interaction with user's input, and not about the internal state of the Cove Framework `QuantumRegister`.
- `CX_Gatebar`
  This class controls the panel of available quantum gates. It keeps a record of the current active gate. Ultimately, this class will add gates to panel dynamically when the user creates a new one or loads from file.

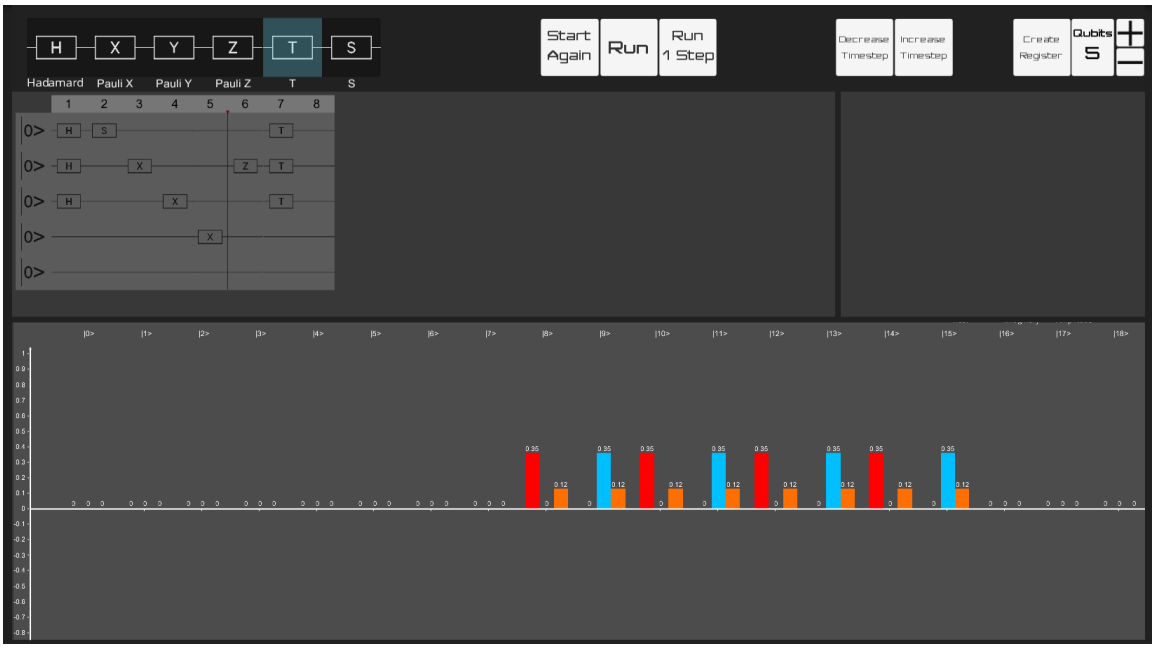


FIGURE 6.3: CoveX UI. Top panel contains the `Gatebar`, where user chooses the gate to insert to circuit, and buttons for creating a new circuit, evaluating it and decreasing/increasing the time steps of the circuit. The rest space of the UI is dividing into three panels. `Circuit Panel` contains the circuit, `Output Panel` visualizes the state of the circuit at the time shown by the `Time Pointer`, and `Info Panel` shows information about the active gate.

- `CX_Circuit_Parser`

  This class parses the information stored in `CX_Circuit` to Cove Framework classes. It creates the `QuantumRegister`, applies the gates of the circuit, and controls the time pointer, extracting a result state.

- `CX_Output_Graph_Manager`

  This class receives information about the state of the register on the current time step and updates the graphical visualization of it. This class concerns only about the bar graph visualization. Other class will of the form `CX_Output_xxx_Manager` will be added as different visualization options are added.

- `CX_Database`

  This class contains information about available elements of the CoveX such as gate elements, inputs etc. In the current version, information is hard-coded from Unity editor. Ultimately, it will fill up dynamically during initiation.

- `CX_UI_Manager`

  As the name suggests, this class provides functions for manipulating UI elements' visibility, ability to receive mouse events, the cursor texture and such relative tasks.

These six Singleton classes are all Components of a single GameObject called `Managers`, which can be put anywhere in the Hierarchy. Also note that only `CX_Circuit` and `CX_Circuit_Parser` contain *application logic*. The rest just expose functions, which those two classes call. The prefix `CX`, short abbreviation for CoveX, helps identifying classes made specifically for CoveX and avoiding naming conflicts with other classes. Essentially, it is equivalent with using a *namespace*.

Next two classes represent the elements of the circuit. `CX_Circuit_Element` is the basic block of the circuit. By default it is a "line", and with user interaction it can become any available gate. When left-clicking a `CX_Circuit_Element` it changes its value to the current active gate of `CX_Gatebar`, while right-clicking returns its value to "line". `CX_Circuit_Input` represents the value of an input qubit. For now only pure states are available, $|0\rangle$ and $|1\rangle$.

Finally, there are three classes, which provide some functionality to UI. `HorizontalDivider` and `VerticalDivider` are used to make two panels horizontal and vertical aligned respectively, with divider in between controlling each panels width or height respectively. `ZoomWithScroll` grants the ability to zoom in and out the contents of a panel.

6.2.3 Future Improvements

The current state of CoveX is far from complete. There are many things and features left to be done. Future improvements contain but not limited to:

- New, modern UI.
- Full support for multiple qubit gates.
- Creation of arbitrary gate on the fly.
- Serialization / deserialization methods for dynamic saving / loading of gates and circuits.
- New method for visualizing quantum states through label signed binary trees. This method is proposed by Mate Galambos and Sandor Imre. This method enables beautiful and easy to read visualization of large states.
- Implementation of back-end server. User can use application anonymously or logged in with an account. If logged in, he is able to download gates and circuits made by other users, commend them, edit them and upload another version.

6.3 Other Implementations

There are a couple other implementations of quantum circuit simulator with a GUI. Three of them are distinguished for their unique properties and presented here.

6.3.1 jQuantum

jQuantum is a quantum computer simulator developed by Andreas de Vries. It is written in Java and its source code is available under the GNU General Public Licence. It simulates the implementation of quantum circuits on a small quantum register up to about 15 qubits. It can run both as an applet in browser and as standalone application. It provides an easy to use GUI shown in Figure 6.4.

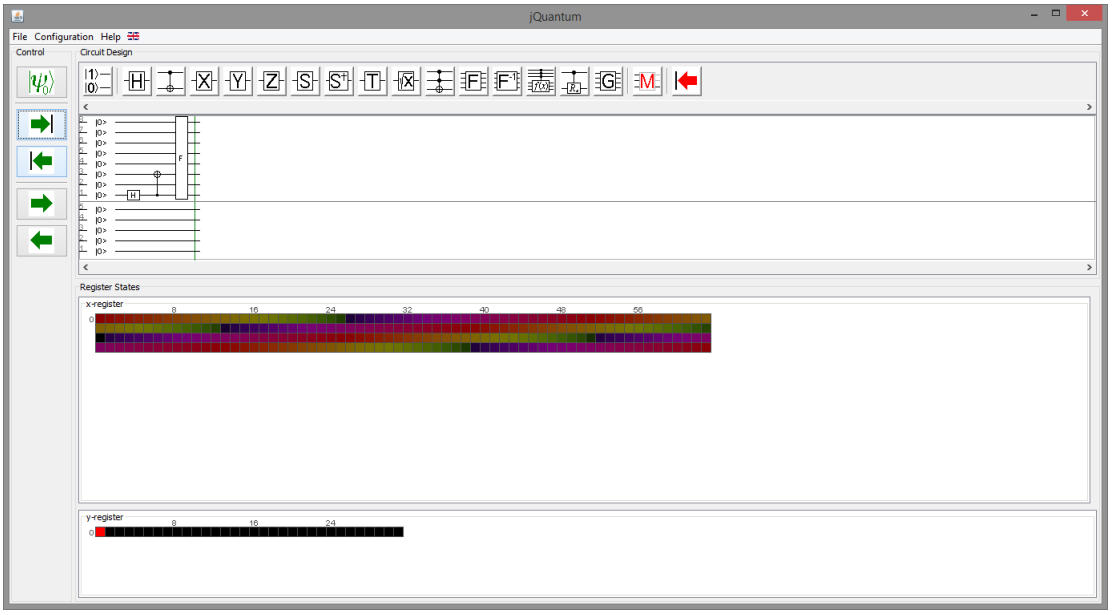


FIGURE 6.4: The graphical user interface (GUI) of jQuantum. On the left part you find the control panel, in which there are buttons to initialize the design and to control the run of the quantum program. The upper horizontal panel consists of buttons with which you can add quantum gates to the circuit design canvas directly below the panel. At the bottom you find a representation of the quantum register. It is divided in a part which receives the input and yields the output, called the x-register, and into a part called y-register, which serves mainly as a temporary qubit storage.

The key elements of jQuantum are:

- It provides a saving and loading functionality. The user can save the designed circuit and load it at a later time. jQuantum saves the circuit in a `".qc"` file.
- It provides a unique representation of the register state using colors. As shown in Figure 6.4, in the register panel of the work place window, each basis state is

represented as a single square whose amplitude z is given by its color according to the color map shown in Figure 6.5. This representation may not provide an accurate value, but helps to quickly distinguish different states.

jQuantum executables, its full source code, as well as its documentation can be found on the project's website: http://jquantum.sourceforge.net/

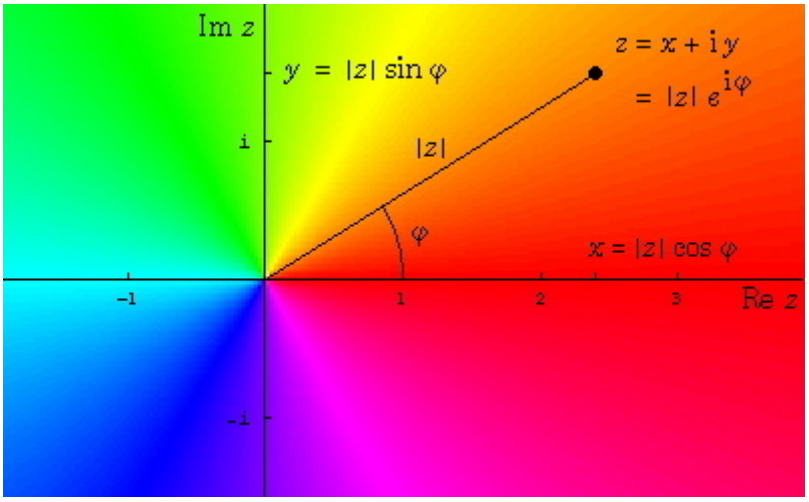


FIGURE 6.5: The color map of the complex plane used for state representation in jQuantum. This representation is based on the polar decomposition of complex number $z = re^{i\phi}$. The hue, as a compination of color and brightness, is given by the complex number's phase ϕ and its absolute value r. The color of the origin $z = 0$ remains undetermined, however since its absolute value is 0, its hue is black.

6.3.2 Zeno

Zeno quantum circuit simulator is the master thesis work of Gustavo Eulalio Cabral. It is developed in Java language.

Zeno is a quite advanced simulator in the sense that provides features not seen in other implementations:

- It provides input of both pure ($|0\rangle$ and $|1\rangle$) and mixed states. This way, the user can define as input to the circuit any value, as long as, its a valid value for a qubit. More specifically, the mixed state is entered by its density matrix.
- It gives the ability to create your own gate by providing its matrix with a limitation to one, two, and three qubit gates.
- Zeno provides measurement gates for both projective measurement and partial trace, as well as ancilla bit gates for simulating quantum channel algorithms.

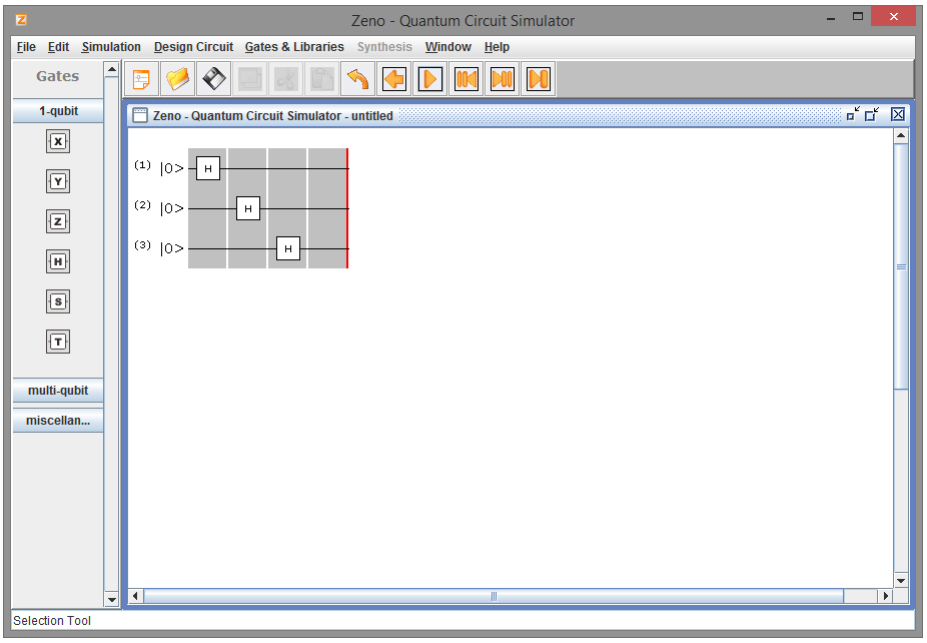


FIGURE 6.6: The GUI of Zeno simulator. The upper horizontal menu provides all the functions; creation of new circuit, saving and loading, creation of new gates and simulation options. The left panel provides access to the gates, while the window on the right shows the current circuit.

- Zeno also provides saving and loading functionality using `".zeno"` files.
- It provides three different representations of the quantum register's state. A mathematical representation in Dirac's notation, a matrix representation, as well as a histogram. These different representations can be seen in Figure 6.7.

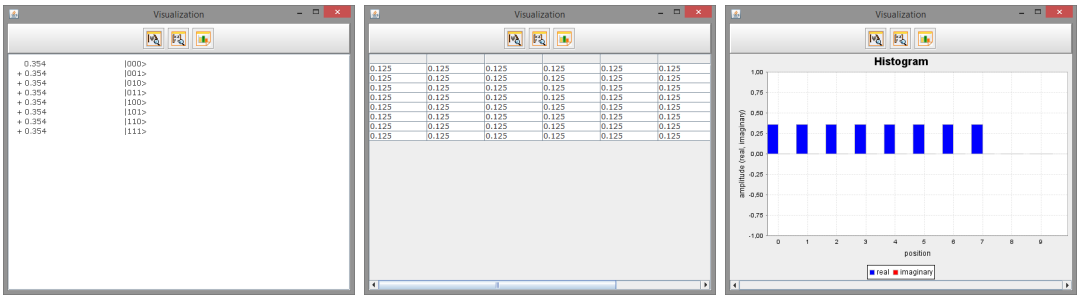


FIGURE 6.7: Different representations of the same state in Zeno.

The executables of Zeno, as well as information about its development can be found at http://dsc.ufcg.edu.br/~iquanta/zeno/index_en.html

6.3.3 Quantum Circuit Simulator

Quantum Circuit Simulator is -in the author's opinion- the best quantum simulator to date. It is developed by Davy Wybiral in Javascript using `NumericJS`, `processing.js`

and `jQuery` libraries. As you can guess, it runs on a browser and no installation is required.

Quantum Circuit Simulator has some innovative features:

- It is designed to support modular circuit design. Any circuit you make can be compiled into a gate for use in other circuits.
- Any gate can be made into a controlled version of itself. This is achieved by selecting the control gate (the black dot) and dragging from the control qubit to the target gate.
- You can export all of the gates you have created into the friendly JSON format. The exported JSON text can be reimported at a later time.
- Circuit diagram can be exported as an image.
- Circuit can be exported as a complex matrix in CSV format.

You can run Quantum Circuit Simulator at developer's website, http://www.davyw.com/quantum/. It's interface as can be seen in Figure **??** is minimal and easy to use.

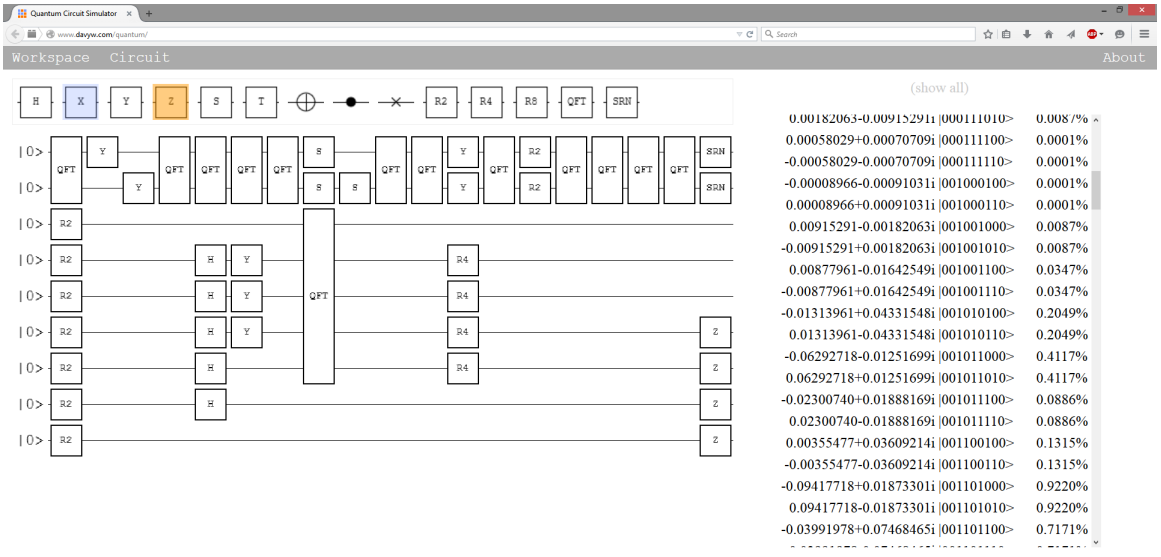


FIGURE 6.8: GUI of Quantum Circuit Simulator. All functions about importing/exporting, circuit size and evaluation can be accessed from the top menu. Below is the gate panel, which contains the default plus the user-created gates. Circuit takes the rest of the GUI's space, while, once evaluated, result is shown on the right.

Appendix A

Additions and Bug Fixes on Cove Framework

A.1 Bugs Fixed

OperationSGate.cs

- Fixed operation matrix of the S gate. New matrix is:

```
this.OperationMatrix = new ComplexMatrix(new Complex[,] {
    { new Complex(1), new Complex(0) },
    { new Complex(0), new Complex(0, 1) } });
```

OperationTGate.cs

- Fixed operation matrix of the T gate. New matrix is:

```
this.OperationMatrix = new ComplexMatrix(new Complex[,] {
    { new Complex(1), new Complex(0) },
    { new Complex(0), new Complex(1.0 / Constants.SQUARE_ROOT_OF_2, 1.0 /
   ↪ Constants.SQUARE_ROOT_OF_2) } });
```

Complex.cs

- Fixed function `Complex SquareRoot()`. New implementation is based on another formula and gives correct result for any complex number.

```
public Complex SquareRoot()
{
        double A = this.Real;
        double B = this.Imaginary;
```

```
        double p;
        double q;
        double ABSquaredRooted = 0;

        //if there is no imaginary part, just return the square root of the
    ↪  real
        if (B == 0)
        {
                if (A > 0) {
                        return new Complex (Math.Sqrt(A), 0);
                }
                if (A < 0) {
                        return new Complex (0, Math.Sqrt (Math.Abs(A)));
                } else {
                        return new Complex (0, 0);
                }
        }

        ABSquaredRooted = Math.Sqrt (Math.Pow(A, 2.0) + Math.Pow(B, 2.0));
        p = (1.0 / Math.Sqrt(2)) * Math.Sqrt(ABSquaredRooted + A);
        q = ((double) Math.Sign (B) / Math.Sqrt (2)) * Math.Sqrt (
    ↪ ABSquaredRooted - A);
        return new Complex (p, q);
}
```

A.2 Additions

ComplexMath.cs

- Added function `ComplexMatrix RREF()`. Function transforms the matrix into its reduced row echelon form and return the transformed matrix.

```
public ComplexMatrix RREF()
{
    int lead = 0;
    int rowCount = this.GetNumberOfRows();
    int columnCount = this.GetNumberOfColumns();
    for (int r = 0; r < rowCount; r++)
    {
        if (columnCount <= lead) break;
        int i = r;
        while (caCells[i, lead] == Complex.Zero)
        {
            i++;
            if (i == rowCount)
            {
                i = r;
                lead++;
                if (columnCount == lead)
                {
                    lead--;
```

```
                    break;
                }
            }
        }
        for (int j = 0; j < columnCount; j++)
        {
            Complex temp = caCells[r, j];
            caCells[r, j] = caCells[i, j];
            caCells[i, j] = temp;
        }
        Complex div = caCells[r, lead];
        if (div != Complex.Zero)
        {
            for (int j = 0; j < columnCount; j++)
            {
                caCells[r, j] /= div;
            }
        }
        for (int j = 0; j < rowCount; j++)
        {
            if (j != r)
            {
                Complex sub = caCells[j, lead];
                for (int k = 0; k < columnCount; k++)
                {
                    caCells[j, k] -= (sub*caCells[r, k]);
                }
            }
        }
        lead++;
    }
    return this;
}
```

- Implemented function `ComplexMatrix Inverse()`. Function transforms the matrix to its inverse and return the inverted matrix.

```
public ComplexMatrix Inverse()
{
    if (this.IsSquare() == false)
    {
        throw new MatrixSizeMismatchException("This matrix is not square.
  ↪ Only square matrices can be inverted.");
    }
    int dimension = this.GetNumberOfRows();
    ComplexMatrix matrix = new ComplexMatrix(dimension, dimension * 2);
    ComplexMatrix identity = new ComplexMatrix(dimension);
    for (int i = 0; i < dimension; i++)
    {
        for (int j = 0; j < dimension; j++)
        {
            matrix.SetValue(i, j, this.caCells[i, j]);
            matrix.SetValue(i, dimension + j, identity.caCells[i, j]);
        }
    }
```

```
    matrix.RREF();
    for (int i = 0; i < dimension; i++)
    {
        for (int j = 0; j < dimension; j++)
        {
            this.SetValue(i, j, matrix.caCells[i, dimension + j]);
        }
    }
    return this;
}
```

- Added function `static Complex Determinant(ComplexMatrix matrix)`. Static function takes as parameter a square matrix and returns its determinant. Throws exception if matrix is not square. Determinant makes use of private function `ComplexMatrix reduce(ComplexMatrix matrix, int index)`.

```
public static Complex Determinant(ComplexMatrix matrix)
{
    if (matrix.IsSquare() == false)
    {
        throw new MatrixSizeMismatchException("Matrix is not square. Only
  ↪ squared matrices can be determined by this function.");
    }
    Complex det = 0;
    int dimension = matrix.GetNumberOfRows();
    if (dimension == 2)
    {
        Complex p = matrix.caCells[0, 0]*matrix.caCells[1, 1];
        Complex n = matrix.caCells[0, 1]*matrix.caCells[1, 0];
        det = p - n;
        return det;
    }
    for (int h = 0; h < dimension; h++)
    {
        if (matrix.caCells[h, 0] == Complex.Zero)
        {
            continue;
        }
        ComplexMatrix reduced = ComplexMatrix.reduce(matrix, h);
        if (h%2 == 0) det += ComplexMatrix.Determinant(reduced)*matrix.
  ↪ GetValue(h, 0);
        if (h%2 == 1) det -= ComplexMatrix.Determinant(reduced)*matrix.
  ↪ GetValue(h, 0);
    }
    return det;
}

private static ComplexMatrix reduce(ComplexMatrix matrix, int index)
{
    int rowCount = matrix.GetNumberOfRows();
    ComplexMatrix reduced = new ComplexMatrix(rowCount - 1, rowCount - 1);
    for (int h = 0, j = 0; h < rowCount; h++)
    {
        if (h == index)
```

```
        {
            continue;
        }
        for (int i = 0, k = 0; i < rowCount; i++)
        {
            if (i == 0)
            {
                continue;
            }
            reduced.SetValue(j, k, matrix.caCells[h, i]);
            k++;
        }
        j++;
    }
    return reduced;
}
```

- Implemented function `bool IsInvertible()`. Function returns true if matrix can be inverted, false otherwise.

```
public bool IsInvertible()
{
    //implementation based on page 36 of
    //S. Lipschutz and M. Lipson, Theory and Problems of Linear Algebra, 3
   ↪ ed. New York: McGraw-Hill, 2001.
    if (this.IsSquare() == false)
    {
        return false;
    }
    if (ComplexMatrix.Determinant(this) == Complex.Zero)
    {
        return false;
    }
    return true;
}
```

ComplexMath.cs

- Added helper static class `ComplexMath`, which contains functions for manipulating complex numbers.

```
public static class ComplexMath
{
        private static readonly Complex sNi = new Complex(0, -1);

        public static double Absolute(Complex value)
        {
                if (Complex.IsNaN(value))
                {
                        return double.NaN;
                }
                if (Complex.IsInfinity(value))
                {
```

```
                return double.PositiveInfinity;
        }

        double real = System.Math.Abs(value.Real);
        double imag = System.Math.Abs(value.Imaginary);
        if (value.Real == 0)
        {
                return imag;
        }
        if (value.Imaginary == 0)
        {
                return real;
        }
        if (real > imag)
        {
                double temp = imag / real;
                return real * System.Math.Sqrt(1.0 + temp * temp);
        }
        else
        {
                double temp = real / imag;
                return imag * System.Math.Sqrt(1.0 + temp * temp);
        }
}

public static double AbsoluteSquared(Complex value)
{
        if (Complex.IsNaN(value))
        {
                return double.NaN;
        }
        if (Complex.IsInfinity(value))
        {
                return double.PositiveInfinity;
        }
        return System.Math.Sqrt(Absolute(value));
}

public static double Argument(Complex value)
{
        return System.Math.Atan2(value.Imaginary, value.Real);
}

public static Complex Conjugate(Complex value)
{
        return new Complex(value.Real, -value.Imaginary);
}

public static Complex Max(Complex v1, Complex v2)
{
        return Norm(v1) >= Norm(v2) ? v1 : v2;
}

public static Complex Min(Complex v1, Complex v2)
{
```

```
                return Norm(v1) <= Norm(v2) ? v1 : v2;
        }

        public static double Norm(Complex value)
        {
                return Absolute(value);
        }

        public static Complex Polar(Complex value)
        {
                return new Complex(Absolute(value), System.Math.Atan2(value
↪ .Imaginary, value.Real));
        }

        public static Complex Cos(Complex value)
        {
                if (Complex.IsNaN(value))
                {
                        return Complex.NaN;
                }
                return Cosh(new Complex(-value.Imaginary, value.Real));
        }

        public static Complex Cosh(Complex value)
        {
                if (Complex.IsNaN(value))
                {
                        return Complex.NaN;
                }
                if (Complex.IsInfinity(value))
                {
                        return Complex.Infinity;
                }

                return new Complex(System.Math.Cos(value.Imaginary)*System.
↪ Math.Cosh(value.Real), System.Math.Sin(value.Imaginary)*System.Math.
↪ Sinh(value.Real));
        }

        public static Complex Sin(Complex value)
        {
                if (Complex.IsNaN(value))
                {
                        return Complex.NaN;
                }
                return new Complex(System.Math.Sin(value.Real)*System.Math.
↪ Cosh(value.Imaginary), System.Math.Cos(value.Real)*System.Math.Sinh(
↪ value.Imaginary));
        }

        public static Complex Sinh(Complex value)
        {
                if (Complex.IsNaN(value))
                {
                        return Complex.NaN;
```

```
            }
            if (value.Imaginary == 0)
            {
                    return new Complex(System.Math.Sinh(value.Real));
            }
            if (value.Real == 0)
            {
                    return new Complex(0, System.Math.Sin(value.
↪ Imaginary));
            }
            return new Complex(System.Math.Sinh(value.Real)*System.Math
↪ .Cos(value.Imaginary), System.Math.Cosh(value.Real)*System.Math.Sin(
↪ value.Imaginary));
    }

    public static Complex Exp(Complex value)
    {
            if (Complex.IsNaN(value))
            {
                    return Complex.NaN;
            }

            double exp = System.Math.Exp(value.Real);
            if (value.Imaginary == 0)
            {
                    return new Complex(exp);
            }
            return new Complex(exp*System.Math.Cos(value.Imaginary),
↪ exp*System.Math.Sin(value.Imaginary));
    }

    public static Complex Pow(Complex leftSide, Complex rightSide)
    {
            return Exp(rightSide*Log(leftSide));
    }

    public static Complex Log(Complex value)
    {
            if (Complex.IsNaN(value))
            {
                    return Complex.NaN;
            }
            double r = Absolute(value);
            double i = System.Math.Atan2(value.Imaginary, value.Real);
            if (i > System.Math.PI)
            {
                    i -= (2.0*System.Math.PI);
            }
            return new Complex(System.Math.Log(r), i);
    }

    public static Complex Sqrt(Complex value)
    {
            double A = value.Real;
            double B = value.Imaginary;
```

```
            double p = 0;
            double q = 0;
            double ABSquaredRooted = 0;

            //if there is no imaginary part, just return the square
↪ root of the real
            if (B == 0) {
                    if (A > 0) {
                            return new Complex (Math.Sqrt(A), 0);
                    } else if (A < 0) {
                            return new Complex (0, Math.Sqrt (Math.Abs(
↪ A)));
                    } else {
                            return new Complex (0, 0);
                    }
            }

            ABSquaredRooted = Math.Sqrt (Math.Pow(A, 2.0) + Math.Pow(B,
↪  2.0));
            p = (1.0 / Math.Sqrt(2)) * Math.Sqrt(ABSquaredRooted + A);
            q = ((double) Math.Sign (B) / Math.Sqrt (2)) * Math.Sqrt (
↪ ABSquaredRooted - A);
            return new Complex (p, q);
    }

    public static Complex Tan(Complex value)
    {
            if (Complex.IsNaN(value))
            {
                    return Complex.NaN;
            }

            double real = 2.0*value.Real;
            double imag = 2.0*value.Imaginary;
            double denom = System.Math.Cos(real) + System.Math.Cosh(
↪ imag);

            return new Complex(System.Math.Sin(real)/denom, System.Math
↪ .Sinh(imag)/denom);
    }

    public static Complex Tanh(Complex value)
    {
            if (Complex.IsNaN(value))
            {
                    return Complex.NaN;
            }

            double real = 2.0*value.Real;
            double imag = 2.0*value.Imaginary;
            double denom = System.Math.Cosh(real) + System.Math.Cos(
↪ imag);

            return new Complex(System.Math.Sinh(real)/denom, System.
↪ Math.Sin(imag)/denom);
```

```
	}

	public static Complex Asin(Complex value)
	{
		if (Complex.IsNaN(value))
		{
			return Complex.NaN;
		}

		return sNi*Log(Complex.I*value + Sqrt(1 - value*value));
	}

	public static Complex Acos(Complex value)
	{
		if (Complex.IsNaN(value))
		{
			return Complex.NaN;
		}
		return sNi*Log(value + Complex.I*Sqrt(1 - value*value));
	}

	public static Complex Atan(Complex value)
	{
		if (Complex.IsNaN(value))
		{
			return Complex.NaN;
		}
		return (1/(Complex.I*2))*(Log((1 + Complex.I*value)) - Log
↪ ((1 - Complex.I*value)));
	}

	public static Complex Asinh(Complex value)
	{
		if (Complex.IsNaN(value))
		{
			return Complex.NaN;
		}
		return -Log(Sqrt(1 + value*value) - value);
	}

	public static Complex Acosh(Complex value)
	{
		if (Complex.IsNaN(value))
		{
			return Complex.NaN;
		}
		return 2*Log((Sqrt(((value + 1)/2)) + Sqrt((value - 1)/2)))
↪ ;
	}

	public static Complex Atanh(Complex value)
	{
		if (Complex.IsNaN(value))
		{
			return Complex.NaN;
```

```
                }
                return .5*(Log(1 + value) - Log(1 - value));
        }
}
```

Appendix B

CoveX Source Code

In this Appendix the main classes of CoveX are presented. Some of minor importance and especially UI-related classes have been omitted.

B.1 CX_Circuit

```
public class CX_Circuit : MonoBehaviour
{
    static CX_Circuit instance;
    public static CX_Circuit Instance { get { return instance; } }

    /// <summary>
    /// The circuit matrix. matrix[0][4] is the gate affecing the first qubit on
   ↪  5th time step.
    /// </summary>
    public List<List<GameObject>> matrix;
    public List<GameObject> qubits;
    public List<GameObject> timesteps;
    public GameObject timePointer;

    /// <summary>
    /// Number of qubits in circuit.
    /// </summary>
    public int c_qubits;

    /// <summary>
    /// Maximum qubits ever allowed.
    /// </summary>
    public int Max_Qubits = 10;
    public int Min_Qubits = 1;

    /// <summary>
    /// Number of current maximum time step.
    /// </summary>
    public int c_time;
```

```
    /// <summary>
    /// Maximum time step ever allowed.
    /// </summary>
    private int max_time = 100;

    [Header("Sizes")]
    public RectTransform circuitRoot;
    public Vector2 minCircuitSize;
    public Vector2 maxCircuitSize;
    public Vector2 circuitOffset;
    public Vector2 gateSize;
    public Vector2 timeStepSize;
    public Vector2 qubitSize;

    public ScrollRect scrollRectCircuit;
    public ToggleGroup gatesGroup;
    public Transform gatesRoot;

    public GameObject inputPrefab;
    public GameObject elementPrefab;
    public GameObject timeStepPrefab;
    public GameObject timePointerPrefab;

    //public Dictionary<string, Button> gateHash;

    private PointerEventData lastPointerData;
    private string inputingGate = null;


    protected void Awake()
    {
        base.Awake();
        instance = this;

        c_qubits = 0;
        c_time = 0;
    }

    // Use this for initialization
    protected void Start()
    {

    }

    // Update is called once per frame
    protected void Update()
    {

    }

    public void CheckForRectViewResize()
    {
        //Debug.Log(scrollRectCircuit.horizontalScrollbar.value + " | " +
↪ scrollRectCircuit.horizontalScrollbar.size);
```

```
        Vector2 oldCircuitSize = circuitRoot.sizeDelta;

        Vector2 circuitSize;
        circuitSize.x = Mathf.Abs(circuitOffset.x) * 2 + qubitSize.x + gateSize.
↪ x * c_time;
        circuitSize.y = Mathf.Abs(circuitOffset.y) * 2 + timeStepSize.y +
↪ gateSize.y * c_qubits;

        if (circuitSize.x < minCircuitSize.x) circuitSize.x = minCircuitSize.x;
        if (circuitSize.y < minCircuitSize.y) circuitSize.y = minCircuitSize.y;

        circuitRoot.sizeDelta = circuitSize;

        //Debug.Log(scrollRectCircuit.horizontalScrollbar.value + " | " +
↪ scrollRectCircuit.horizontalScrollbar.size);
        if (circuitRoot.sizeDelta.x > oldCircuitSize.x)
        {
            scrollRectCircuit.horizontalNormalizedPosition = 1f;
        }
        if (circuitRoot.sizeDelta.y > oldCircuitSize.y)
        {
            scrollRectCircuit.verticalNormalizedPosition = 0f;
        }
    }

    public void DestroyCircuit()
    {
        foreach (GameObject qubit in qubits)
        {
            Destroy(qubit);
        }
        foreach (GameObject timestep in timesteps)
        {
            Destroy(timestep);
        }
        if (matrix != null)
        {
            foreach (List<GameObject> list in matrix)
            {
                foreach (GameObject element in list)
                {
                    Destroy(element);
                }
            }
        }
        Destroy(timePointer);
        CX_UI_Manager.Instance.DisableAllCircuitControlElements();
    }

    public void NewCircuit(int q)
    {
        DestroyCircuit();
        if (q < Min_Qubits || q > Max_Qubits)
        {
            return;
```

```
        }
        matrix = new List<List<GameObject>>();
        qubits = new List<GameObject>();
        timesteps = new List<GameObject>();
        GameObject prefab;

        //Add Qubit Inputs
        for (int i = 0; i < q; i++)
        {
            prefab = (GameObject) Instantiate(inputPrefab);
            prefab.transform.SetParent(circuitRoot, false);
            prefab.transform.localPosition = new Vector3(circuitOffset.x,
↪ circuitOffset.y - timeStepSize.y + (-gateSize.y)*i, 0);

            qubits.Add(prefab);
        }
        //Add Time Pointer
        prefab = (GameObject) Instantiate(timePointerPrefab);
        prefab.transform.SetParent(circuitRoot, false);
        prefab.transform.localPosition = new Vector3(circuitOffset.x + qubitSize
↪ .x, circuitOffset.y, 0);
        prefab.transform.SetAsLastSibling();
        prefab.transform.FindChild("Line").GetComponent<RectTransform>().
↪ sizeDelta = new Vector2(2, q * qubitSize.y);
        timePointer = prefab;

        //Set number of qubits and increase time steps to 1
        c_qubits = q;
        c_time = 0;
        IncreaseTimeStep();
        CX_Circuit_Parser.Instance.CreateRegister();
        CX_UI_Manager.Instance.EnableAllCircuitControlElements();
    }

    public void ElementClicked(CX_Circuit_Element element, PointerEventData
↪ pointerData)
    {
        string aGate = CX_Gatebar.Instance.ActiveGate;
        string[] indexes = element.name.Split('_');
        int eQubit = int.Parse(indexes[0]);
        int eTime = int.Parse(indexes[1]);
        CX_Database.Gate gateInfo;
        CX_Database.Gate line = CX_Database.Instance.GetGateInfo("Line");

        switch (pointerData.button)
        {
            case PointerEventData.InputButton.Left:
                if (aGate == null)
                {
                    return;
                }

                element.SetGate(line.Name, line.MainSprite, CX_Circuit_Element.
↪ ElementState.Other);
                foreach (int index in element.sameGateIndexes)
```

```
            {
                matrix[eTime][index].GetComponent<CX_Circuit_Element>().
↪ SetGate(line.Name, line.MainSprite, CX_Circuit_Element.ElementState.Other)
↪ ;
                matrix[eTime][index].GetComponent<CX_Circuit_Element>().
↪ sameGateIndexes.Clear();
            }
            element.sameGateIndexes.Clear();

            gateInfo = CX_Database.Instance.GetGateInfo(aGate);
            if (gateInfo.NmberOfQuibts == 1)
            {
                element.SetGate(gateInfo.Name, gateInfo.MainSprite,
↪ CX_Circuit_Element.ElementState.Single);
            }
            if (gateInfo.NmberOfQuibts == 2)
            {
                if (gateInfo.Name == "CNOT_DOWN")
                {
                    if (eQubit <= c_qubits - 2)
                    {
                        element.SetGate(gateInfo.Name, gateInfo.MainSprite,
↪ CX_Circuit_Element.ElementState.Control);
                        element.sameGateIndexes.Add(eQubit + 1);
                        CX_Circuit_Element nextElement = matrix[eTime][
↪ eQubit + 1].GetComponent<CX_Circuit_Element>();
                        foreach (int index in nextElement.sameGateIndexes)
                        {
                            matrix[eTime][index].GetComponent<
↪ CX_Circuit_Element>().SetGate(line.Name, line.MainSprite,
↪ CX_Circuit_Element.ElementState.Other);
                            matrix[eTime][index].GetComponent<
↪ CX_Circuit_Element>().sameGateIndexes.Clear();
                        }
                        nextElement.sameGateIndexes.Clear();
                        nextElement.SetGate(gateInfo.Name, gateInfo.
↪ SecondarySprite, CX_Circuit_Element.ElementState.Target);
                        nextElement.sameGateIndexes.Add(eQubit);
                    }
                }
                if (gateInfo.Name == "CNOT_UP")
                {
                    if (eQubit <= c_qubits - 2)
                    {
                        element.SetGate(gateInfo.Name, gateInfo.MainSprite,
↪ CX_Circuit_Element.ElementState.Target);
                        element.sameGateIndexes.Add(eQubit + 1);
                        CX_Circuit_Element nextElement = matrix[eTime][
↪ eQubit + 1].GetComponent<CX_Circuit_Element>();
                        foreach (int index in nextElement.sameGateIndexes)
                        {
                            matrix[eTime][index].GetComponent<
↪ CX_Circuit_Element>().SetGate(line.Name, line.MainSprite,
↪ CX_Circuit_Element.ElementState.Other);
```

```
                            matrix[eTime][index].GetComponent<
↪ CX_Circuit_Element>().sameGateIndexes.Clear();
                        }
                        nextElement.sameGateIndexes.Clear();
                        nextElement.SetGate(gateInfo.Name, gateInfo.
↪ SecondarySprite, CX_Circuit_Element.ElementState.Control);
                        nextElement.sameGateIndexes.Add(eQubit);
                    }
                }
            }
            if (eTime == c_time - 1)
            {
                IncreaseTimeStep();
            }
            break;
        case PointerEventData.InputButton.Right:
            element.SetGate(line.Name, line.MainSprite, CX_Circuit_Element.
↪ ElementState.Other);
            foreach (int index in element.sameGateIndexes)
            {
                matrix[eTime][index].GetComponent<CX_Circuit_Element>().
↪ SetGate(line.Name, line.MainSprite, CX_Circuit_Element.ElementState.Other)
↪ ;
                matrix[eTime][index].GetComponent<CX_Circuit_Element>().
↪ sameGateIndexes.Clear();
            }
            element.sameGateIndexes.Clear();
            break;
    }
    if (eTime < CX_Circuit_Parser.Instance.currentTimeStep)
    {
        InputChanged();
    }
}

public void ResetRelativeElements(int time, int qubit)
{
    CX_Circuit_Element element = matrix[time][qubit].GetComponent<
↪ CX_Circuit_Element>();
    CX_Database.Gate line = CX_Database.Instance.GetGateInfo("Line");
    foreach (int index in element.sameGateIndexes)
    {
        matrix[time][index].GetComponent<CX_Circuit_Element>().SetGate(line.
↪ Name, line.MainSprite, CX_Circuit_Element.ElementState.Other);
        matrix[time][index].GetComponent<CX_Circuit_Element>().
↪ sameGateIndexes.Clear();
    }

}

public void GateAdded(GameObject gate)
{
    string[] indexes = gate.name.Split('_');
    int i = int.Parse(indexes[0]);
    int y = int.Parse(indexes[1]);
```

```
        if (y == c_time - 1)
        {
            IncreaseTimeStep();
        }
        if (y < CX_Circuit_Parser.Instance.currentTimeStep)
        {
            InputChanged();
        }
    }

    public void GateRemoved(GameObject gate)
    {
        string[] indexes = gate.name.Split('_');
        int i = int.Parse(indexes[0]);
        int y = int.Parse(indexes[1]);
        if (y < CX_Circuit_Parser.Instance.currentTimeStep)
        {
            InputChanged();
        }
    }

    public void InputChanged()
    {
        CX_Circuit_Parser.Instance.UpdateInput();
    }

    public void IncreaseTimeStep()
    {
        if (c_time == max_time)
        {
            return;
        }

        //Add Time Step
        GameObject prefab;
        prefab = (GameObject) Instantiate(timeStepPrefab);
        prefab.transform.SetParent(circuitRoot, false);
        prefab.transform.localPosition = new Vector3(circuitOffset.x + qubitSize
↪ .x + gateSize.x * c_time, circuitOffset.y, 0);
        prefab.transform.FindChild("label").GetComponent<Text>().text = (c_time
↪ + 1).ToString();
        timesteps.Add(prefab);

        //Add Circuit Elements
        matrix.Add(new List<GameObject>());
        for (int i = 0; i < c_qubits; i++)
        {
            prefab = (GameObject) Instantiate(elementPrefab);
            prefab.transform.SetParent(circuitRoot, false);
            prefab.transform.localPosition = new Vector3(circuitOffset.x +
↪ qubitSize.x + gateSize.x * c_time, circuitOffset.y - timeStepSize.y + (-
↪ gateSize.y) * i, 0);

            prefab.name = i + "_" + c_time;
            matrix[c_time].Add(prefab);
```

```
        }

        c_time += 1;
        UpdateTimePointerDepth();
        CheckForRectViewResize();
    }

    public void DecreaseTimeStep()
    {
        if (c_time == 0)
        {
            return;
        }
        Destroy(timesteps[c_time - 1]);
        timesteps.RemoveAt(c_time -1);
        foreach (GameObject obj in matrix[c_time - 1])
        {
            Destroy(obj);
        }
        matrix.RemoveAt(c_time - 1);
        c_time -= 1;
        CheckForRectViewResize();
    }

    public void UpdateTimePointerDepth()
    {
        timePointer.transform.SetAsLastSibling();
    }

    public void MoveTimePointerAtStart()
    {
        timePointer.transform.localPosition = new Vector3(circuitOffset.x +
↪ qubitSize.x, circuitOffset.y, 0);
    }

    public IEnumerator CoMoveTimePointerAtStart()
    {
        timePointer.transform.localPosition = new Vector3(circuitOffset.x +
↪ qubitSize.x, circuitOffset.y, 0);
        yield return 0;
    }

    public void MoveTimePointerAfterTimeStep(int index)
    {
        timePointer.transform.localPosition = new Vector3(circuitOffset.x +
↪ qubitSize.x + timeStepSize.x * (index + 1), circuitOffset.y, 0);
    }

    public IEnumerator CoMoveTimePointerAfterTimeStep(int index)
    {
        timePointer.transform.localPosition = new Vector3(circuitOffset.x +
↪ qubitSize.x + timeStepSize.x * (index + 1), circuitOffset.y, 0);
        yield return 0;
    }
```

```
    public void ElementClicked(GameObject gameObject)
    {

    }

}
```

B.2 CX_Circuit_Element

```
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.UI;

public class CX_Circuit_Element : MonoBehaviour, IPointerClickHandler
{
    public enum ElementState
    {
        Control = 0,
        Target = 1,
        Fill = 2,
        Single = 3,
        Other = 4
    }
    public CX_Database.Gate ReferenceGate;

    public Image image;
    public string ActiveElement;
    public List<int> sameGateIndexes;
    public ElementState state;

    void Awake()
    {

    }

    // Use this for initialization
        void Start ()
        {
            ActiveElement = "Line";
            state = ElementState.Other;
        }

        // Update is called once per frame
        void Update () {

        }

    public void OnPointerClick(PointerEventData eventData)
    {
        CX_Circuit.Instance.ElementClicked(this, eventData);
    }

    public void SetGate(string name, Sprite sprite, ElementState state)
```

```
    {
        image.sprite = sprite;
        ActiveElement = name;
        this.state = state;
    }
}

class ElementEvent : EventTrigger
{
    void Start()
    {
        Debug.Log(delegates);
    }
}
```

B.3 CX_Circuit_Input

```
public class CX_Circuit_Input : MonoBehaviour, IPointerClickHandler
{

    public Image image;
    public bool ActiveInput;

        // Use this for initialization
        void Start ()
        {
            ActiveInput = false;
        }

        // Update is called once per frame
        void Update () {

        }

    public void OnPointerClick(PointerEventData eventData)
    {
        ActiveInput = !ActiveInput;
        image.sprite = CX_Database.Instance.GetSpriteOfInput(ActiveInput);
        CX_Circuit.Instance.InputChanged();
    }
}
```

B.4 CX_Circuit_Parser

```
public class CX_Circuit_Parser : MonoBehaviour {

    static CX_Circuit_Parser instance;
    public static CX_Circuit_Parser Instance { get { return instance; } }

    public QuantumRegister register;
    public int qubits;
    public bool[] input;
```

```
    public int currentTimeStep;
    public ComplexMatrix quantumState;

    void Awake()
    {
        instance = this;
        currentTimeStep = 0;
    }

        // Use this for initialization
        void Start () {

        }

        // Update is called once per frame
        void Update () {

        }

    public void CreateRegister()
    {
        qubits = CX_Circuit.Instance.c_qubits;
        List<GameObject> inputQubits = CX_Circuit.Instance.qubits;
        input = new bool[qubits];
        for (int q = 0; q < qubits; q++)
        {
            input[q] = (inputQubits[q].GetComponent<CX_Circuit_Input>().
↪ ActiveInput);
        }
        register = new QuantumRegister(input);
        currentTimeStep = 0;
        quantumState = register.PeekAtEntireState();
        CX_Output_Graph_Manager.Instance.NewGraph(quantumState.GetNumberOfRows()
↪ );
        CX_Output_Graph_Manager.Instance.UpdateResults(quantumState);
    }

    public void UpdateInput(bool rewind = true)
    {
        List<GameObject> inputQubits = CX_Circuit.Instance.qubits;
        input = new bool[qubits];
        for (int q = 0; q < qubits; q++)
        {
            input[q] = (inputQubits[q].GetComponent<CX_Circuit_Input>().
↪ ActiveInput);
        }
        register.SetQubits(input);
        if (rewind)
        {
            ParseTo(currentTimeStep, true);
        }
    }

    public void ParseTo(int endingTimestep, bool animateTimePointer)
    {
```

```
        StartCoroutine(CoParseFromTo(0, endingTimestep, animateTimePointer));
    }

    public void ParseFromTo(int startingTimestep, int endingTimestep, bool
↪ animateTimePointer)
    {

        StartCoroutine(CoParseFromTo(startingTimestep, endingTimestep,
↪ animateTimePointer));
    }

    private IEnumerator CoParseFromTo(int startingTimestep, int endingTimestep,
↪ bool animateTimePointer)
    {
        List<List<GameObject>> matrix = CX_Circuit.Instance.matrix;

        if (animateTimePointer)
        {
            yield return StartCoroutine(CX_Circuit.Instance.
↪ CoMoveTimePointerAfterTimeStep(startingTimestep - 1));
        }
        Debug.Log("Current: " + currentTimeStep + ", Starting: " +
↪ startingTimestep + ", Ending: " + endingTimestep);
        for (currentTimeStep = startingTimestep; currentTimeStep <
↪ endingTimestep; currentTimeStep++)
        {
            for (int qubit = 0; qubit < qubits; qubit++)
            {
                string activeElement = matrix[currentTimeStep][qubit].
↪ GetComponent<CX_Circuit_Element>().ActiveElement;
                switch (activeElement)
                {
                    case "Line":
                        break;
                    case "H":
                        (register.Slice(qubit, qubit)).OperationHadamard();
                        break;
                    case "X":
                        (register.Slice(qubit, qubit)).OperationNot();
                        break;
                    case "Y":
                        (register.Slice(qubit, qubit)).OperationYGate();
                        break;
                    case "Z":
                        (register.Slice(qubit, qubit)).OperationZGate();
                        break;
                    case "S":
                        (register.Slice(qubit, qubit)).OperationSGate();
                        break;
                    case "T":
                        (register.Slice(qubit, qubit)).OperationZGate();
                        break;
                    case "CNOT":
                        (register.Slice(qubit, qubit + 1)).OperationCNot();
                        qubit++;
```

```
                    break;
                case "CNOT_UP":
                    (register.Slice(qubit, qubit + 1)).OperationCNot(1, 0);
                    qubit++;
                    break;
            }
        }
        if (animateTimePointer)
        {
            yield return StartCoroutine(CX_Circuit.Instance.
↪ CoMoveTimePointerAfterTimeStep(currentTimeStep));
        }
    }
    quantumState = register.PeekAtEntireState();
    CX_Output_Graph_Manager.Instance.UpdateResults(quantumState);
}

public void ParseCircuit()
{
    StartCoroutine(CoParseCircuit());
}

public IEnumerator CoParseCircuit()
{
    int timesteps = CX_Circuit.Instance.c_time;
    List<List<GameObject>> matrix = CX_Circuit.Instance.matrix;
    currentTimeStep = 0;

    List<GameObject> inputQubits = CX_Circuit.Instance.qubits;
    input = new bool[qubits];
    for (int q = 0; q < qubits; q++)
    {
        input[q] = (inputQubits[q].GetComponent<CX_Circuit_Input>().
↪ ActiveInput);
    }
    register.SetQubits(input);

    yield return StartCoroutine(CX_Circuit.Instance.CoMoveTimePointerAtStart
↪ ());
    for (currentTimeStep = 0; currentTimeStep < timesteps; currentTimeStep
↪ ++)
    {
        for (int qubit = 0; qubit < qubits; qubit++)
        {
            string activeElement = matrix[currentTimeStep][qubit].
↪ GetComponent<CX_Circuit_Element>().ActiveElement;
            switch (activeElement)
            {
                case "Line":
                    break;
                case "H":
                    (register.Slice(qubit, qubit)).OperationHadamard();
                    break;
                case "X":
                    (register.Slice(qubit, qubit)).OperationNot();
```

```
                    break;
                case "Y":
                    (register.Slice(qubit, qubit)).OperationYGate();
                    break;
                case "Z":
                    (register.Slice(qubit, qubit)).OperationZGate();
                    break;
                case "S":
                    (register.Slice(qubit, qubit)).OperationSGate();
                    break;
                case "T":
                    (register.Slice(qubit, qubit)).OperationZGate();
                    break;
                case "CNOT":
                    (register.Slice(qubit, qubit + 1)).OperationCNot();
                    qubit++;
                    break;
                case "CNOT_UP":
                    (register.Slice(qubit, qubit + 1)).OperationCNot(1, 0);
                    qubit++;
                    break;
            }
        }
        yield return StartCoroutine(CX_Circuit.Instance.
↪ CoMoveTimePointerAfterTimeStep(currentTimeStep));
    }
    quantumState = register.PeekAtEntireState();
    CX_Output_Graph_Manager.Instance.UpdateResults(quantumState);
}

public void ParseNextTimeStep()
{
    if (currentTimeStep == CX_Circuit.Instance.c_time)
    {
        return;
    }
    for (int qubit = 0; qubit < qubits; qubit++)
    {
        string activeElement = CX_Circuit.Instance.matrix[currentTimeStep][
↪ qubit].GetComponent<CX_Circuit_Element>().ActiveElement;
        switch (activeElement)
        {
            case "Line":
                break;
            case "H":
                (register.Slice(qubit, qubit)).OperationHadamard();
                break;
            case "X":
                (register.Slice(qubit, qubit)).OperationNot();
                break;
            case "Y":
                (register.Slice(qubit, qubit)).OperationYGate();
                break;
            case "Z":
                (register.Slice(qubit, qubit)).OperationZGate();
```

```
                    break;
                case "S":
                    (register.Slice(qubit, qubit)).OperationSGate();
                    break;
                case "T":
                    (register.Slice(qubit, qubit)).OperationZGate();
                    break;
                case "CNOT_DOWN":
                    (register.Slice(qubit, qubit + 1)).OperationCNot();
                    qubit++;
                    break;
                case "CNOT_UP":
                    (register.Slice(qubit, qubit + 1)).OperationCNot(1, 0);
                    qubit++;
                    break;
            }
        }
        quantumState = register.PeekAtEntireState();
        CX_Output_Graph_Manager.Instance.UpdateResults(quantumState);
        CX_Circuit.Instance.MoveTimePointerAfterTimeStep(currentTimeStep);
        currentTimeStep += 1;
    }

    public void StartAgain()
    {
        currentTimeStep = 0;
        UpdateInput();
    }
}
```

B.5 CX_Database

```
public class CX_Database : MonoBehaviour
{
    public struct Gate
    {
        public string Name;
        public int NmberOfQuibts;
        public bool IsBlock;
        public Sprite MainSprite;
        public Sprite SecondarySprite;
        public Sprite FillSprite;
        public bool CustomGate;
        public ComplexMatrix Matrix;
    }
    public List<Gate> Gates;
    public Dictionary<bool, Sprite> Inputs;

    static CX_Database instance;
    public static CX_Database Instance { get { return instance; } }

    public void Awake()
    {
        base.Awake();
```

```
        instance = this;
    }

        // Use this for initialization
        public void Start ()
        {
            string json = Serializer.Serialize(typeof (List<CX_Gate>), Gates);
        Debug.Log(json);
        }

        // Update is called once per frame
        void Update () {

        }

    public Sprite GetSpriteOfInput(bool input)
    {
        return Inputs[input];
    }

    public Gate GetGateInfo(string gate)
    {
        return Gates.Find(x => x.Name == gate);
    }

}
```

B.6 CX_Gatebar

```
public class CX_Gatebar : MonoBehaviour
{
    private static CX_Gatebar instance;
    public static CX_Gatebar Instance { get { return instance; } }

    //public GameObject gatesRoot;
    public List<Toggle> gates;
    public string ActiveGate { get; private set; }

    void Awake()
    {
        base.Awake();
        instance = this;
    }

    void Start()
    {
        ActiveGate = null;
    }

    void Update()
    {

    }
```

```
    public void GateToggled(bool isOn)
    {
        Toggle t = gates.Find(x => x.isOn);
        if (t != null) ActiveGate = t.name;
        else ActiveGate = null;
    }
}
```

B.7 CX_Output_Graph_Manager

```
public class CX_Output_Graph_Manager : MonoBehaviour {

    static CX_Output_Graph_Manager instance;
    public static CX_Output_Graph_Manager Instance { get { return instance; } }

    public RectTransform graphRoot;
    public ScrollRect scrollRectGraph;

    public GameObject graphPrefab;

    public GameObject graphObject;
    public WMG_Axis_Graph graph;
    public WMG_Series realBars;
    public WMG_Series imaginaryBars;
    public WMG_Series absoluteBars;

    public float desiredBarWidth;
    private int numberOfBars;

    void Awake()
    {
        instance = this;
    }

        // Use this for initialization
        void Start () {

        }

        // Update is called once per frame
        void Update () {

        }

    public void CheckForRectViewResize()
    {

    }

    public void NewGraph(int numberOfStates)
    {
        if (graphObject != null)
        {
            Destroy(graphObject);
```

```
        }
        GameObject prefab = (GameObject)Instantiate(graphPrefab);
        prefab.transform.SetParent(graphRoot, false);
        prefab.transform.localPosition = new Vector3(10, -10, 0);
        graphObject = prefab;
        graph = graphObject.GetComponent<WMG_Axis_Graph>();
        realBars = graph.lineSeries[0].GetComponent<WMG_Series>();
        imaginaryBars = graph.lineSeries[1].GetComponent<WMG_Series>();
        absoluteBars = graph.lineSeries[2].GetComponent<WMG_Series>();

        realBars.pointValues.Clear();
        imaginaryBars.pointValues.Clear();
        absoluteBars.pointValues.Clear();
        for (int i = 0; i < numberOfStates; i++)
        {
            realBars.pointValues.Add(Vector2.zero);
            imaginaryBars.pointValues.Add(Vector2.zero);
            absoluteBars.pointValues.Add(Vector2.zero);
        }

        float barWidth = desiredBarWidth;

        graph.xAxisNumTicks = numberOfStates + 2;

        float xAxis = (4.0f*numberOfStates + 4)*barWidth;

        float distBetweenPoints = 4.0f * barWidth;
        float extraBarSpace = (5.0f/2.0f)*barWidth;
        realBars.extraXSpace = extraBarSpace;
        imaginaryBars.extraXSpace = extraBarSpace;
        absoluteBars.extraXSpace = extraBarSpace;
        realBars.xDistBetweenPoints = distBetweenPoints;
        imaginaryBars.xDistBetweenPoints = distBetweenPoints;
        absoluteBars.xDistBetweenPoints = distBetweenPoints;

        graph.barWidth = barWidth;
        graph.xAxisLength = xAxis;

        graph.yAxisLength = 1080f;

        Vector2 graphSize = graph.GetComponent<RectTransform>().sizeDelta;
        graphSize.x = graph.xAxisLength;
        graphSize.y = graph.yAxisLength;
        graph.GetComponent<RectTransform>().sizeDelta = graphSize;
        graph.transform.localPosition = new Vector3((-1) * graphSize.x / 2.0f +
↪ 140.0f + graph.backgroundPaddingLeft, (-1) * graphSize.y / 2.0f + 70f -
↪ graph.backgroundPaddingTop, 0);

        graphSize.x += graph.backgroundPaddingLeft + graph.
↪ backgroundPaddingRight;
        graphSize.y += graph.backgroundPaddingBottom + graph.
↪ backgroundPaddingTop;
        graphRoot.sizeDelta = graphSize;

        float minScale;
```

```
        if (graphRoot.sizeDelta.x > graphRoot.sizeDelta.y)
        {
            minScale = 1920.0f/graphRoot.sizeDelta.x;
        }
        else
        {
            minScale = 1080.0f/graphRoot.sizeDelta.y;
        }
        if (minScale > 1) minScale = 1;
        //float minScaleY = 1080.0f / graphRoot.sizeDelta.y;
        scrollRectGraph.GetComponent<ZoomWithScroll>().minScale = minScale;

        graph.xAxisLabelSpacingY = (-1)*(graph.yAxisLength + graph.
    ↪ backgroundPaddingTop)/2.0f;
        graph.xAxisLabels.Clear();
        for (int i = 0; i < graph.xAxisNumTicks; i++)
        {
            if (i == 0 || i == graph.xAxisNumTicks - 1)
            {
                graph.xAxisLabels.Add("");
            }
            else
            {
                graph.xAxisLabels.Add("|" + (i - 1) + ">");
            }
        }
        graph.legendParentOffsetY = graph.backgroundPaddingTop*(3.0f/4.0f);
    }

    public void UpdateResults(ComplexMatrix results)
    {
        for (int i = 0; i < results.GetNumberOfRows(); i++)
        {
            realBars.pointValues[i] = new Vector2(0, (float) results.GetValue(i,
    ↪  0).Real);
            imaginaryBars.pointValues[i] = new Vector2(0, (float)results.
    ↪ GetValue(i, 0).Imaginary);
            absoluteBars.pointValues[i] = new Vector2(0, (float) results.
    ↪ GetValue(i, 0).AbsoluteValueSquared());
        }
    }
}
```

Bibliography

[1] Cathryn Carson. The origins of quantum theory, 2000.

[2] Mark M. Wilde. *Quantum Information Theory*. Cambridge University Press, jun 2013.

[3] Phillip Kaye, Raymond Laflamme, and Michele Mosca. *An Introduction to Quantum Computing*. Oxford University Press, jan 2007.

[4] Gennady P. Berman, Gary D. Doolen, Ronnie Mainieri, and Vadimir I. Tsifrinovich. *Introduction to Quantum Computers*. World Scientific, 1998.

[5] Prof Dr. Helmut G. Kart. Quantum computing. Proseminar in Theoretical Physics, 2008.

[6] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2001.

[7] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete lograithms on a quantum computer. *SIAM Journal on Computing*, pages 1484–1509, 1997.

[8] Artur Ekert and Richard Jozsa. Quantum computation and shor's factoring algorithm. *Review of Modern Physics, 68*, pages 733–753, 1996.

[9] Stanley Rabinowitz. *How to Find the Square Root of a Complex Number*. Reprinted from Mathematics and Informatics Quarterly, 3 (1993) 54-56.

[10] A. Mostowski and M. Stark. *Introduction to Higher Algebra*. Pegamon Press, 1964.

[11] Steven T. Flammia Bryan Eastin. *Q-circuit Tutorial*. Department of Physics and Astronomy, University of New Mexico.

[12] Seymour Lipschutz and Marc Lars Lipson. *Schaum's Outline of Theory and Problems of Linear Algebra*. The McGraw-Hill Companies, 2004.

[13] Ritajit Majumdar. No cloning theorem. Calcutta University, feb 2014.

[14] Mate Galambos and Sandor Imre. New method for representation of multi-qubit systems using fractals. *ICQNM 2011 : The Fifth International Conference on Quantum, Nano and Micro Technologies*, 2011.

[15] S. Imre and B. Ferenc. *Quantum Computing and Communications: An Engineering Approach*. Wiley, 2005.

[16] Matthew Daniel Purkeypile. *COVE: A PRACTICAL QUANTUM COMPUTER PROGRAMMING FRAMEWORK*. PhD thesis, Colorado Technical University, 2009.

[17] Spiros Mais. *Orthogonality in Hilbert spaces*. National Technical University (NTUA).