UNIVERSIDADE Đ
COIMBRA

André Paulo Pires Miranda Ferreira da Silva

Simulation of a Quantum Algorithm using Quantum Fourier Transforms

Thesis submitted to the
University of Coimbra for the degree of
Master in Engineering Physics

September 2018

FCTUC FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE DE COIMBRA

André Paulo Pires Miranda Ferreira da Silva

Simulation of a Quantum Algorithm using Quantum Fourier Transforms

Thesis submitted to the
University of Coimbra for the degree of
Master in Engineering Physics

Supervisor:
Prof. Dr. Maria Helena Vieira Alberto

Coimbra, 2018

Acknowledgments

O desenvolvimento desta tese foi uma aventura, no que toca a todas as áreas novas que tive que explorar. Sabendo que a minha formação na área de computação quântica era básica, não bastou gostar do tema para suceder, foi preciso um grande esforço que sozinho não seria possível.

Merece uma menção especial, a Prof. Dra. Maria Helena Vieira Alberto. Mesmo sem ter obrigações (nem ser a sua área), disponibilizou-se a ajudar-me, orientar-me e corrigir-me durante todo o esforço que foi a conclusão da tese. Nos temas mais difíceis nunca negou a disponibilidade para me ajudar. E teve sempre muita paciência para me explicar os mais fáceis. Por todo o esforço e dedicação, um muito obrigado. Não seria possível sem a sua ajuda, literalmente.

Aos meus pais, e família próxima, quero agradecer a paciência para me educarem e me instigarem o gosto de perseguir as coisas que mais gosto. Quando a duração da tese se estendeu, revelaram disponibilidade porque perceberam que era essencial. Obrigado Pai e Mãe.

À minha companheira tenho que fazer um agradecimento especial, porque foi ela que me incentivou durante este tempo todo. Nos momentos mais difíceis, foi ela quem me levou a continuar a perseguir os meus objectivos. E quando a vontade de trabalhar era pouca, foi ela que me repreendeu e me fez voltar para a frente dos livros. Ainda me ajudou, sempre disponível, quando eu precisei de ajuda para rever o texto e fazer alterações. A sua ajuda foi extramente valiosa para que o texto fosse perceptível. Obrigado por tudo, Adriana.

Resumo

Computação quântica é uma área de grande desenvolvimento hoje em dia e isso traz uma necessidade para desenvolver hardware e software. Uma vez que o hardware disponível é muito pouco e o software precisa de ser testado, existe uma necessidade de uma boa plataforma para testar as soluções de software. Simular um computador quântico num computador clássico não é ideal, contudo é uma resposta viável a esta necessidade, para um pequeno número de qubits. Além disso, os computadores quânticos necessitam dos computadores clássicos de forma a criar um sistema real prático, isto apresenta-se como uma excelente oportunidade para testar técnicas de programação clássica para melhorar a programação quântica. Esta tese propõe o uso de programação clássica para melhorar a implementação de um algoritmo de adição modular usando transformadas de Fourier quânticas. Os principais objectivos deste trabalho são: simular um circuito quântico; testar um algoritmo quântico e compará-lo com o seu equivalente clássico; implementar o algoritmo de adição modular, usando as capacidades de aceleração de computação das transformadas de Fourier quânticas e optimizá-lo; Testar e implementar códigos quânticos de correcção de erros; fazer um teste num computador quântico real.

De forma a testar a linguagem e comparar um algoritmo quântico com um clássico, fez-se um teste ao algoritmo de Shor que está disponível de raiz na plataforma LIQ$Ui\,|\rangle$. Apesar do algoritmo de Shor ter, teoricamente, uma dependência de tipo polinomial com a dimensão do problema, os testes realizados mostraram que o tempo de computação tem um comportamento exponencial. Simular um algoritmo quântico num computador clássico requer tempo de computação que cresce de forma exponencial, pois é necessário armazenar e processar todos os estados do sistema, que é um espaço de Hilbert de dimensão 2^n.

O algoritmo de adição modular usando transformadas de Fourier quânticas foi implementado na plataforma LIQ$Ui\,|\rangle$. Seguidamente fizeram-se melhorias e optimizações. O número de operações teórico é $\mathbf{O}(n^2)$. Contudo, depois de fazer melhorias ao circuito, removendo gates desnecessárias para casos específicos, o número de operações passa a ter um valor mínimo de $\boldsymbol{\Omega}(n)$. Com este tipo de alterações o sistema pode apresentar melhorias no que respeito ao tempo de simulação até um factor de ordem 10. Isto mostra que programar um algoritmo quântico tendo em conta casos específicos pode melhorar imenso o desempenho da simulação.

Os códigos de correção de erros quânticos foram implementados num circuito simples, com as ferramentas disponíveis na plataforma LIQ$Ui\,|\rangle$, e obteve-se um sistema que consegue lidar com erros com probabilidade inferior ou igual a 2%, com uma taxa de sucesso na sua correção de 95%. Contudo são necessários circuitos muito grandes, pouco exequíveis.

A plataforma disponibilizada pela IBM para fazer testes num computador quântico real, foi utilizada para testar o algoritmo de adição modular e adição normal (com carry bit). Uma vez que esta plataforma tem um número limitado de gates, reforça a ideia de que se deve otimizar os algoritmos tendo em conta casos particulares. Os resultados destes testes mostram também que é importante utilizar correcção de erros quânticos, mas para que isso seja viável é necessário que haja mais qubits disponíveis ou outro tipo de códigos.

Palavras-Chave: Simulação de Circuitos Quânticas, Transformada de Fourier Quântica, Adição Modular, Algoritmo de Shor, Correcção de Erros Quânticos, LIQ$Ui\,|\rangle$, IBM Q

Abstract

Quantum computation is a great area of development nowadays, so there is a great need to produce hardware and implement and create new software. Since hardware is very limited, and the software needs to be tested, there is a real need for a platform to test software solutions. Simulating a quantum computer in a classical one is not ideal, but is a viable answer to test quantum software with a small number of qubits as input. Also, since quantum computers need classical computers in order to become a real system, this is an opportunity to test techniques using classical and quantum programming together. This master thesis uses classical computation to improve a quantum addition algorithm based on quantum Fourier transforms (QFT). The main objectives of this work are: to simulate a quantum circuit; to test a quantum algorithm and to compare it with a classical equivalent; to implement a modular addition algorithm using the speedup capabilities of QFT and to optimize it; to test Quantum Error Correction (QEC) implementation; to run a simplified version of the modular addition algorithm in a real quantum computer.

A test to the Shor's algorithm, available in LIQ$Ui\,|\rangle$, was made in order to test the language and compare the algorithm with a classical one. Even though Shor's algorithm has a polynomial dependence with the size of the problem, the tests showed that regarding computing time, it performed in an exponential way. Simulating a quantum algorithm in a classical computer will take exponential time because it needs to store and process all the states of the system, a Hilbert space of dimension 2^n.

Using QFT, the algorithm of modular addition was implemented in LIQ$Ui\,|\rangle$. It was then improved, and all versions were optimized. The theoretical values for the number of gates is $\mathbf{O}(n^2)$. However, improving the circuit by eliminating gates in cases where they are not necessary, can bring the gate number down to $\mathbf{\Omega}(n)$ in the best cases. This approach can reduce the simulation time by an order of magnitude. This shows that programming a quantum algorithm analysing specific cases can improve considerably the simulation efficiency.

QEC codes were implemented in a simple circuit using the tools available in LIQ$Ui\,|\rangle$, and we got a system that can handle a probability of errors of 0.02, maximum, with 95% success rate. However, the number of qubits of this circuit will be very significant.

The web service IBM Q Experience was used to make a test, in a real quantum computer, with the algorithm for modular addition, as well as the algorithm for normal addition (that is, including a carry bit). Since the IBM Q Experience had a limited set of available gates and entry qubits, it reinforced the idea that it is important to optimize algorithms considering specific cases for which the circuits can be simplified. The results also show the importance of QEC codes, they are fundamental to have a proper functional system. Its implementation, however, requires a much larger number of entry qubits or the development of optimized QEC codes.

Keywords: Quantum Circuit Simulation, Quantum Fourier Transform, Modular Addition, Shor's Algorithm, Quantum Error Correction, LIQ$Ui\,|\rangle$, IBM Q

List of Figures

List of Tables

Contents

1

Introduction

Quantum computers are having great improvements on both hardware and software fronts. However, the physical computers available today are still few, and the ones that exist are very limited because they only have a small number of qubits to compute (of the order of ten or less than ten). Meanwhile, software for quantum machines needs to be improved and tested for the near future, when quantum computers are expected to have a number of qubits exceeding 50. So, software development needs to be tested using another physical platform: quantum simulation in a classic computer is the answer. Simulating a quantum algorithm in a classic computer allows testing and the development of optimization methods. In addition, since quantum computers need classical computers to become a real system, this is an opportunity to test techniques using classical and quantum programming together. The main objectives of this work are: to simulate a quantum circuit; to test a quantum algorithm and to compare it with a classical equivalent; to implement a modular addition algorithm using the speedup capabilities of quantum Fourier Transforms and to optimize it; to test Quantum Error Correction (QEC) implementation; to run a simplified version of the modular addition algorithm in a real quantum computer.

Chapter 2 will present the basic concepts to understand quantum computation and the algorithm that will be implemented. Those concepts are important to understand how quantum computation can have outstanding performances to solve some problems, namely using the Quantum Fourier Transforms (QFT).

In chapter 3 the simulation algorithm will be discussed, as well as the environment that was chosen for this work, LIQ$Ui\,|\rangle$. A small test for one of the LIQ$Ui\,|\rangle$ available algorithms, Shor's algorithm, will be performed and compared with a similar test for a classical factoring algorithm.

Chapter 4 is central to the objectives of this work. It concerns the programming and testing of modular addition using QFT, which is the objective of this master thesis. This means writing all the functions from scratch, including QFT, inverse QFT (IQFT) and all the other operations needed to implement the algorithm. The operations of the algorithm will use the basic gate set available in LIQ$Ui\,|\rangle$, although some of the gates need to be adapted from other gates and functions available. An effort was made to optimize the implementation of the algorithm and understand how classical computers can be used in the process.

Chapter 5 is devoted to QEC codes, which are essential to assure that the quantum system is reliable enough to produce good results. This codes will be explored, namely, the way they work and how they are implemented in LIQ$Ui\,|\rangle$. There will be tests to see how the stabilizer environment works, and which are the benefits of using it.

Lastly, after implementing a modular addition algorithm, testing it, and understanding the QEC role in quantum computation, it's time to make a quick test in a real quantum computer, and verify how reliable the results are. This tests will be performed in the IBM Q Experience, which is the subject of chapter 6.

1.1 Motivation

The theme of this master thesis was self-proposed because I have a big interest in this area of research. It's a fast developing area and now is the right time to start learning its basic principles, in order to aspire to make some valuable work in the future.

As this is a first approach to the area of quantum computation, this thesis may not follow a classic outline. Instead, this thesis will have more introductory concepts than it's usual in an Engineering Physics thesis, since one of its main goals is to provide the understanding of essential concepts in order to work in this area in the future. Therefore, it will explore areas of quantum computation that are not essential to perform a quantum algorithm simulation. The most important and relevant details are explored in chapter 4, however, Quantum Error Correction codes are explored as well, but only to get accustomed to them.

The theme itself is very interesting. Quantum computation has some marvellous characteristics, such as quantum parallelism which allows some computing problems to be solved faster. However, in order to be able to use these characteristics, there is a need for algorithms to be optimized for efficiency. Algorithm developers have always found answers to the problems using the most elegant and smart solutions to design them. The use of QFT in addition algorithms is one example, as is the use of quantum parallelism to improve algorithm efficiency.

It may seem unusual at first sight to perform a basic operation, such as the addition, using a complex tool, like the QFT, since classical Fourier transforms need addition operations themselves. However, due to the nature of the Quantum Fourier transform, only Hadamard and Rotations gates are needed to implement the QFT, making it in an ideal method to perform an addition in the transform domain. In this domain, the addition is simply achieved with the use of controlled rotations without the need of any carry bits.

Furthermore, the available simulation platforms for quantum software are now a reality that works, giving good results. There are even options to implement quantum circuits into real quantum computers using a cloud service.

Without any doubt, quantum computation will become a very important area of physics in a not so distant future, and this is the ideal time to start working on it. It's an opportunity to be part of the group of people that contribute to developing a new computation area.

1.2 State of the Art

Quantum computation is an area with an amazing pace of development. On one side we have the hardware part, having great investment from the private and public sector, and is showing some impressive results. There are different approaches for implementing a physical quantum computer. Qubits can be based on particle spins, or on photons and even on superconductors systems for example [1]. In a near future, we might have a reliable quantum computer with enough qubits (more than 50) to implement algorithms for different applications. We are already seeing some examples of its use [3, 4] and plans to solve problems in the future [5, 6].

On the other side, we have the software, with an even more impressive pace of development. On the late $90's$ algorithm design was progressing rapidly but quantum languages were missing. When the first appeared, they were an important contribution [7, 8]. The software development of quantum computation, and also the whole quantum computation itself, were in a low pace of developing for the first decade of 2000. However, in the last few years, it's having a speedup that suggests it's going in the right direction and getting good results. So the number of available languages started to increase, in a matter of a few years, the software landscape changed its form. The comparison between an article from 2011 reviewing the scene and talking about a few languages [9], with a review article from 2017 [2], shows that software languages can become outdated very quickly in this area. Actually, since the beginning of this thesis work, there are some new additions to the language list. Even within the same company, namely Microsoft, a new language is being developed to replace LIQ$Ui\,|\rangle$ [10], IBM Q environment is also causing a loss of interest in LIQ$Ui\,|\rangle$. Algorithms that are already published are being tested, optimized, and sometimes modified to improve their implementation [11, 12]. Simulation is an important part of this process.

The available tools to simulate a quantum circuit are various and at present there is even the possibility to work with real quantum computers: in the IBM Q experience, a quantum computer is accessible from the cloud [13]. But it's no longer unique in the market, the Chinese Centre of Excellence in Quantum Computation has announced a cloud service to test their quantum computer of 11 qubits [14]. This center is from the same group that has already implemented 712 km of a quantum communication secure line that uses quantum encryptation [15]. Intel is also interested in this market and has announced the production of a 49 qubit quantum chip[16].

Quantum computation landscape is changing and shaping itself rapidly. Since the private sector started to invest on it, a race has begun to ensure a supremacy of one company over the others. At present, some of the big names of classical computation are on the race. In many ways, quantum computation is following the steps of classical computation: the preliminary algorithm design and the companies race to present the best initial products are some examples. However, quantum computation has its own concepts and challenges, namely the difficulties to implement a real system. Nevertheless, in the long term, positive results will outweigh the difficulties.

Regarding the quantum algorithm design, in the 90's there was a huge development triggered by the Shor's algorithm, published in 1994 [17]. It increased the interest in quantum computation since it is devoted to a problem with important applications in cryptography and its theoretical efficiency exceeds its classical equivalent.

The addition operation is one of the basic and most important operations used in computation and in our lives. We are exposed to the addition algorithm since the very early stages of our learning process. However, implementing this algorithm using a machine may not be as simple as we learn as infants. Nevertheless, there are ancient implementations of the addition algorithm such as the Abacus instrument. The first quantum addition algorithm appeared only in 1996 proposed by Vedral Barenco and Ekert [18], two years after Shor's algorithm. In 1998 a new variant for a quantum addition algorithm was proposed by Gosset [19]. Both used the concept of a classic well-known addition algorithm but adapted for reversible computation. Each one of them used its own variation for the manipulation of the carry bit. In the year 2000 Drapper publishes the article "Addition on a Quantum Computer"[20] presenting a completely new idea, being the first algorithm to perform a modular addition using a Quantum Fourier Transform. Just like the software, the algorithm design had a slow pace of development during the following decade. However, in recent years new interesting ideas showed up. In 2017, Perez and Garcia-Escartin published an article [11] suggesting some modifications to the Drapper addition algorithm in order to implement non-modular addition with only one carry bit, disregarding the number of entry qubits. Furthermore, they suggested to generalize this algorithm to perform more arithmetic operations, such as multiplications and averages.

2

Basic Principles for Quantum Computation

2.1 Concept of Quantum Computer

2.1.1 Turing Machines

The concept of Turing Machines was introduced in 1936 by Alan Turing, and it's the starting point for the other computational models. So, it's very important that we understand this model in order to learn how algorithms work in a general way.

2.1.1.1 Classic model

The simplest model for a Turing Machine (figure (2.1)), only has the following components [1, 21]:

1. a program (set of sequential instructions);
2. a finite state control ($q_1 ... q_m$);
3. a tape, which works like a traditional memory;
4. a read-write tape-head (reads/writes in the tape).

The Turing machine follows the sequential instructions, varies the position in the tape and alters the state, that's the basic principle. The state control has a finite number of states of the system ($q_1 ... q_m$), plus the special states q_s (starting state), that represents the initial state, and q_h (halting state) that points out when the system has stopped, that is, halted. The state control processes the information that comes in and goes out of the system by changing the states in the tape. Basically, the

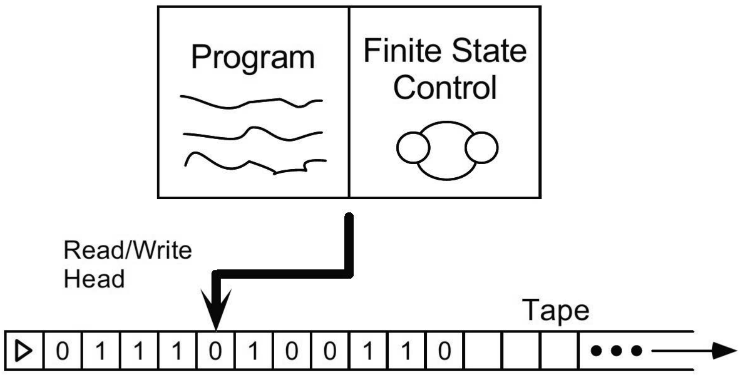


Figure 2.1: Schematics of a Turing Machine with the basic components present. Source:[1]

state control runs the execution of the program, in a similar way as the information is processed in a computer.

In the tape, there is an infinite number of squares (this is a perfect, theoretical machine), which are small places to store the states. There are a finite number of symbols, namely, in this case, 0,1,b, $\triangleright$. The symbol $\triangleright$ represents the beginning of the tape, and is always located in the first square, 0 and 1 are bits and b represents the empty squares.

The program for this machine has a specific formatting type for commands. The code lines will have the format: $< q,x,q',x',s >$, which are sequential instructions that the program will then execute. The symbols "<" and ">" represent the beginning and the end of the line. The first letter, q, corresponds to the current state of the internal system. The x represents the latter state read from the tape; q' and x' are the new internal states and tape symbols respectively. The s will tell the read-write tape-head the direction to go next.

We saw how the code lines work, now let's see how the system operates. The program executes the commands in sequential order. It will search the code for the exact line with the current states, the letter q being the internal state and the letter x being the state written in the tape, at that moment. When the program finds the line that starts by q,x, it executes the orders of the second half of the line, changes the internal state to q' and the symbol in the tape to x'. If s is $+1, -1$ or 0 it changes the tape-head to the right, left or does not change the position, respectively. For example, the system searches sequentially for the line $< q_s, \triangleright, q_m, 1, +1 >$ when it starts, since upon start the initial state will be q_s and the tape will have $\triangleright$. Then it changes the state to q_m and the symbol to 1. Lastly, it changes the tape-head to the right square and searches for the next line which will start with $< q_m$,1,... $>$.

If the machine can't find a line with the right state and symbol, the program will halt. This may not be the most efficient way to execute the program, since it has to search line by line, sequentially, until finding the right one, every cycle. So, the code has to be written with special care. For example, to execute the last line of written code it needs to search all the others first, which consumes time and may execute the wrong line if there is an error. However, if the programmer writes the code carefully it will be executed without any errors. This simple process allows us to write all kinds of functions, which gives the Church-Turing thesis:

The class of functions computable by a Turing Machine corresponds exactly to the same class computable by an algorithm.

Besides, a Turing Machine is perfectly capable of reproducing any other machine, this is the origin of system simulation. In order to simulate any system with a Turing Machine that can compute any algorithm, we need a global Turing machine that can reproduce a different machine and all its states, which is a Universal Turing Machine [22].

Let us consider the case of a probabilistic machine, which brings a certain random factor to the system. The first discussed model reads code lines in sequence so the states of the system will be in succession. This is called a **deterministic machine model**. In a **non-deterministic** model every possible state of the system has several possible "paths" of action. This means that if every state has 2 possibilities (paths), then after only two cycles we will have 4 paths and so on. This is an enormous increase in possibilities, it's an exponential growth, which means it takes a lot of computational power and it will be impossible to know all the possible paths at the same time. But if we introduce a random factor, like a coin toss, it can make the path decision faster. In short, it is introduced a certain level of chaos (randomness) in the system that can make computation run faster. A common example of this model is described as the coin toss model. To execute it in a deterministic Turing machine, it will need to compute every possible path of the system. With the probabilistic model, it just has a certain probability of choosing one specific path, hence being faster. From this, we go to the strong Church-Turing thesis:

Any computational model can be simulated on a Probabilistic Turing Machine with at most a polynomial increase in the number of elementary operations required [1].

2.1.1.2 Quantum model

A reversible machine is a device for which we can recover the information of the initial state from the recorded information of the final state, that is, the process is reversible. An isolated quantum system it's a dynamic system where it is guaranteed the reversible principle without any ambiguity [21]. Bennioff created a scheme for the reversible Turing machine and this particular model contributed to the understanding and evolution of quantum computing. Nevertheless, the model used today is the circuit model, which is equivalent to the Turing machine model but simpler. It doesn't need an exhaustive representation which makes it a better option to work with.

The Quantum Turing Machine (QTM) has some differences when compared with the classical model [22]. The operations of reading, writing and position changing are made by processes based on quantum mechanics principles. Besides, the tape has the ability to register non-classical states. This means that the machine can have superposition states, a mixed state between a zero and one classical states, which is unknown until a measurement is made. This particular feature allows a Quantum Turing Machine to make a lot of internal operations at the same time, which translates into faster information processing. However, this quantum system has a problem of information loss after measurement. In order to use the model in a real situation and since the state will no longer be in a superposition, there is a need to make measurements to the tape states. These states will lose information as they will settle as one or zero. When we talked about a probabilistic machine we mentioned the probability of the machine choosing one particular path. In the QTM model, the system will compute all the different paths at the same time, and then after measurement, we, the users, will know the final state. It is possible to describe the state of the system as $c_0|0> +c_1|1>$, where the probability of finding the system in a certain state after measurement is given by the square of the corresponding amplitude, c_n, with n = 0,1. For example, the probability of obtaining a state $|0\rangle$ after measurement is $|c_0|^2$. Also, since this is a quantum system, the bits, which in a quantum system are called qubits, can be entangled. In simple words, this means that qubits can have a special correlation between them which will influence the measurement output. In a system with more than one qubit, measuring one qubit will determine its state and the state of the other or other qubits to which the first one is entangled, without measuring the latter.

The after effects of making a measurement are far more serious in a quantum than in a classical machine, where the measurement has no consequence at all. So, the decision to measure and when is of great importance. Besides, we should also have into account other important details, such as the fact that the probability factors c are complex numbers. The addition of some of the factors can indeed cancel or amplify the final states probability. This is quantum interference, similar to what is described by a classical model to study waves.

2.2 Basic Concepts

2.2.1 Qubit

The most basic concept of classical computation is the bit, which is the representation of information using the binary base. Bits only have two different and unique states, $|0\rangle$ and $|1\rangle$ [1]. That is enough to represent all kinds of information, and it's easy to use to make simple operations such as read, write and delete data without changing the state in question. So the basic mathematical operations are possible to implement with some ease.

The quantum domain has adopted the binary base from its classical predecessor, although in some quantum systems a different basis could have been adopted. Since the quantum states obey quantum mechanic laws, **the state of the system is indeed a superposition of the states $|0\rangle$ and $|1\rangle$**. This is the quantum state called qubit:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2.1}$$

where α and β are amplitudes given by complex numbers such that

$$|\alpha|^2 + |\beta|^2 = 1 \tag{2.2}$$

The equation (2.2) just states that the probability of obtaining the system either in state $|0\rangle$ or in state $|1\rangle$, after a measurement, is one. The superposition of the states comes from the wave-like behaviour of the particles. In the classical domain, the information described in binary is either a $|0\rangle$ or a $|1\rangle$. The qubit can be in any combination of states $|0\rangle$ and $|1\rangle$, since α and β can have any complex value.

[1] The Dirac notation is used here despite discussing bits, because it's the notation used in quantum computation even when the qubits are only bits. Note that in classical computation bits don't require this notation, are only represented by 0 and 1.

Once a measurement is made the qubit is destroyed, becoming a bit, and reducing the number of possibilities available. This means that measurements are an extremely important operation to work within the quantum domain. They impose a strong limitation to quantum computation, unlike the classical measurements which don't affect the bit whatsoever.

Another consequence of the measuring process, is that it makes it impossible to know the initial state from the knowledge of the final state (after measurement). For example, obtaining the state $|1\rangle$ after measurement means, in classical bits, that the initial state was $|1\rangle$, but in quantum bits we can only conclude there was a non zero probability of getting $|1\rangle$ after measurement.

Gates are used in order to perform basic operations to qubits, similarly to what is done in classical computation. In contrast to measurements, the gates have the characteristic of not destroying the qubit, this will be discussed in the subsection (2.2.3), devoted to quantum gates.

The physical implementation of a qubit needs a quantum system of two possible states in order to have a binary base. The reference model are particles with spin 1/2, which have as possible spin states the so called up and down states, that respectively means positive and negative spin projection in a given direction. The commonly adopted notation is to associate the spin up to the state $|0\rangle$ and the spin down to the state $|1\rangle$, both spins in the z direction. There are other choices of physical systems to implement the states $|0\rangle$ and $|1\rangle$. The vertical and horizontal polarization of a photon or two electronic states of an atom are common examples. The choice of the physical system to implement qubits is a subject of intense research work. However, the rules to manipulate the information are the same as long as it's a quantum system.

Looking back to the equation (2.1), α e β are complex numbers, and therefore we can rewrite it as:

$$|\psi\rangle = A\,\mathrm{e}^{i\gamma}\,|0\rangle + B\,\mathrm{e}^{i\delta}\,|1\rangle$$

where A,B, γ and δ are real numbers. If we consider the angle $\phi = \delta - \gamma$, we can rearrange the equation into:

$$|\psi\rangle = \mathrm{e}^{i\gamma}(A\,|0\rangle + B\,\mathrm{e}^{i\phi}\,|1\rangle)$$

Now, since the term $\mathrm{e}^{i\gamma}$ is a global phase with no physical observable effect on the system, we can disregard it. Using the condition of normalization from equation (2.2), $A^2 + B^2 = 1$. Writing $A = \cos\frac{\theta}{2}$ and $B = \sin\frac{\theta}{2}$ we can have the final form:

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle \tag{2.3}$$

This will simplify the expression, converting a four real variable expression into a two real variable expression. This allows representing a qubit as shown in the figure (2.2), which is much more practical to visualize. The angles θ and ϕ are the spherical coordinates of a specific point in the unitary radius sphere, called **Bloch sphere**. Representing quantum states as points in the Bloch sphere is very useful to see the effects that operations have on the one-qubit states. We can see that the basis states $|0\rangle$ and $|1\rangle$ are in the top and bottom of the $z - axis$, respectively. This is known as the z basis or standard basis.

When the system has more than a single qubit, the Bloch sphere cannot be used to represent the system.

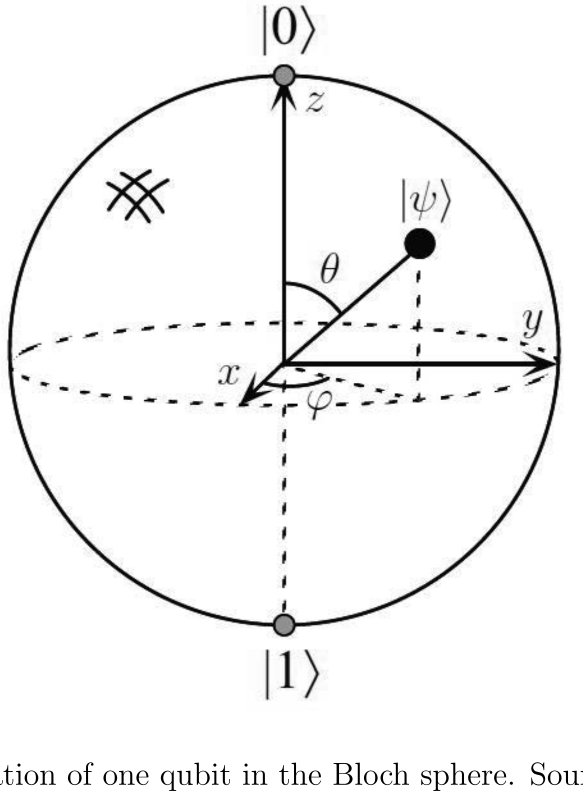


Figure 2.2: Representation of one qubit in the Bloch sphere. Source: [1]

2.2.2 System of N Qubits

Having more than one qubit will make the system more complex, since the superposition will still hold for the complete system. As the number of qubits gets larger, the number of combinations of superposition states grows with 2^n, n being the number of qubits. As will be discussed during chapter 3, this behaviour has very important effects on the simulation of a quantum system.

For systems with more than one qubit, the **qubits can be entangled**. In other words, the qubit states can have a non-classical correlation. For a two-qubit system this means that when someone measures one of the qubits of the entangled pair, those who know the result of the measurement will automatically know the state of the other one, even if they are far apart. This is the basic principle behind quantum teleportation, one of the most important applications of quantum computation.

2.2.2.1 Tensor Product

In order to study systems with multiple qubits, we need to introduce the concept of tensor product first. The tensor product will make vector spaces larger, joining multiple vector spaces together. This is an important concept in quantum mechanics, when we work with systems with multiple particles, in this case qubits. The tensor product multiplies the dimension of the vector spaces. In a case with two vector spaces with dimension 2, the final tensor product, represented by the symbol $\otimes$, will have dimension $2 \times 2 = 4$. The final vector space will be computed as follows:

$$\begin{pmatrix} a \\ b \end{pmatrix} \otimes \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} ac \\ ad \\ bc \\ bd \end{pmatrix}$$

We multiply the first element from the first matrix with the second matrix elements line by line, and then repeat with the second element of the first matrix. For example, for the tensor product with 3 vectors of dimension 2, we have:

$$\begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0\\1\\0\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0\\0\\0\\1\\0\\0\\0\\0\end{pmatrix}$$

This is the basic concept of the tensor product. Its properties will not be discussed here, for more information see [1, 23].

Now that we understand the basic principle of the tensor product, we can explore its use in quantum computers. When there is more than one qubit, the tensor product will be used to construct the vector space.

2.2.2.2 Example of a 2 Qubit System

First, let's consider the vectors for the basis states, represented as:

$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \ ; \ |1\rangle = \begin{pmatrix}0\\1\end{pmatrix} \tag{2.4}$$

As in classical computation, a 2 qubit system will have four possible state combinations. The four combinations are known in quantum computation as the computational basis states:

$$|0\rangle|0\rangle = |00\rangle \ ; \ |0\rangle|1\rangle = |01\rangle \ ; \ |1\rangle|0\rangle = |10\rangle \ ; \ |1\rangle|1\rangle = |11\rangle$$

The basis states are given by a tensor product between the basis states of the two particle spaces, $V_1 \otimes W_2$:

$$|00\rangle = |0\rangle \otimes |0\rangle \ ; \ |01\rangle = |0\rangle \otimes |1\rangle \ ; \ |10\rangle = |1\rangle \otimes |0\rangle \ ; \ |11\rangle = |1\rangle \otimes |1\rangle$$

For example: $|01\rangle = \begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} = \begin{pmatrix}0\\1\\0\\0\end{pmatrix}$

Therefore, we get the vectors for the computational basis states of a two-qubit system:

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \ ; \ |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \ ; \ |10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \ ; \ |11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

The four combinations together make an identity matrix. Since it uses the binary base, it can be seen that the first one, corresponds to the number zero, the second one to the number one, the third to the number two and the fourth to the number three, as in classical computation.

A two-qubit system can, and probably will, have a superposition of states, just as the case of a single qubit. This means the system state will be, in general:

$$|\psi\rangle = \alpha_{00} |00\rangle + \alpha_{01} |01\rangle + \alpha_{10} |10\rangle + \alpha_{11} |11\rangle$$

As before, the probability amplitudes α_{ij}, obey a normalization condition:

$$|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10} + |\alpha_{11}| = 1$$

2.2.2.3 The case of a 3 qubit system

We have seen that the 2 qubit system has 4 possible combinations of single qubit basis states. With 3 qubits, we will have 8 possible combinations, as follows:

$$|0\rangle |0\rangle |0\rangle \ ; \ |0\rangle |0\rangle |1\rangle \ ; \ |0\rangle |1\rangle |0\rangle \ ; \ |0\rangle |1\rangle |1\rangle$$
$$|1\rangle |0\rangle |0\rangle \ ; \ |1\rangle |0\rangle |1\rangle \ ; \ |1\rangle |1\rangle |0\rangle \ ; \ |1\rangle |1\rangle |1\rangle$$

Using the same notation for the system used before, we can consider:

$$|000\rangle \ ; \ |001\rangle \ ; \ |010\rangle \ ; \ |011\rangle \ ; \ |100\rangle \ ; \ |101\rangle \ ; \ |110\rangle \ ; \ |111\rangle$$

The tensor product for a vector space with 3 qubits has dimension 8, this means that to calculate it for all the combinations becomes laborious, for example:

$$|101\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

Note that $|101\rangle$, in binary basis, represents the decimal number 5. Also, the number 1 is placed after 5 zeros in the corresponding vector array. In the following ordered list, the first state vector represents the number 0 and the last one the number 7:

$$|000\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} ; \ |001\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} ; \ |010\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} ; \ |011\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

$$|100\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} ; \ |101\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} ; \ |110\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} ; \ |111\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

2.2.2.4 Final Notes about N qubit systems

As discussed for the 2 qubit example, if the system is in a state superposition and we measure the first qubit, the result will have, in the most general case, only half of the terms. For example, if we have a two-qubit system and we get 0 as the result for the first qubit, then there are only two possibilities left: $|\psi\rangle = \frac{\alpha_{00}|00\rangle+\alpha_{01}|01\rangle}{\sqrt{|\alpha_{00}|^2+|\alpha_{01}|^2}}$. Note that, unless α_{00} or α_{01} is zero, we still do not know the state of the second qubit, but half of the final state options are already destroyed. That is why this equation needs the denominator part, in order to obey the normalization condition.

Now lets consider a special state, named as a **Bell state (or EPR)** of the form:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{2.5}$$

This state is fully normalized, and it has the interesting property of only being dependent on two possible state combinations while having two qubits. As already seen, with two qubits in superposition, the state will have, in general, contributions from 4 basis states, because of the tensor product result. However, in this case, it is easy to check that the two qubit state cannot be written as a tensor product of two independent one qubit states:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \neq |\psi_1\rangle \otimes |\psi_2\rangle$$

Instead, we have:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle_A \otimes |0\rangle_B + |1\rangle_A \otimes |1\rangle_B)$$

This state, discussed in detail in [23], is entangled. Being entangled means there is a strong correlation between the final state measurements of the two qubits: no matter how far apart the qubits are, the result of measuring the first qubit correlates with the result of the second qubit. This is in contrast with the non-entangled system, where a measurement of the first qubit does not provide information on the state of the second qubit. In the example of equation(2.5), if the measurement gives the state $|0\rangle$ as a result, the second qubit will be $|0\rangle$ as well.

The Bell states are vastly used in quantum computation, they are the basis to make some complex systems. The most famous example is the quantum teleportation system, which uses entanglement. Bell states can be made of other combinations, as listed below:

$$|\beta_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

$$|\beta_{01}\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$

$$|\beta_{10}\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$$

$$|\beta_{11}\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$$

2.2.3 Quantum Gates

Quantum gates are the representation of the logical operations used to construct algorithms. A quantum gate is not a measurement, it changes the qubit but does not convert it into a bit. The gate output is still, in general, a superposition of states. The concept of gates comes from classical computation. They are nothing more than operators that transform an entry state into an exit state. However, while classical are just constrained to perform a logic operation, quantum gates have additional requirements such as the condition to preserve the norm of the qubits and being reversible. Note that, quantum gates operations don't destroy the superposition of states.

The physical implementation of quantum gates depends on the physical nature of the qubits and it is a complex research area that exceeds the scope of this thesis. However, we can discuss the mathematical description of the gates, that is, the effect that they have on the entry states, and why that is so useful. It is a very important subject to discuss, the mathematical theoretical approach is convenient to predict the expected results from a certain system, and its efficiency. Therefore, the study of algorithms, gates and circuits, has developed independently from the physical systems. Actually, some of the algorithms are more than 20 years old, whereas the corresponding physical systems just became possible to implement in recent years, for systems with a few qubits.

Regarding mathematical single qubits gates, they are represented by a transformation matrix (2.6), which acts upon a qubit (the so called entry vector) (2.7) and gives as a result an exit vector, the output of the system (2.8). For example, the NOT operation is represented by the matrix:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2.6}$$

If the entry of the system is:

$$\alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{2.7}$$

Then, the output of the NOT operation on the entry is:

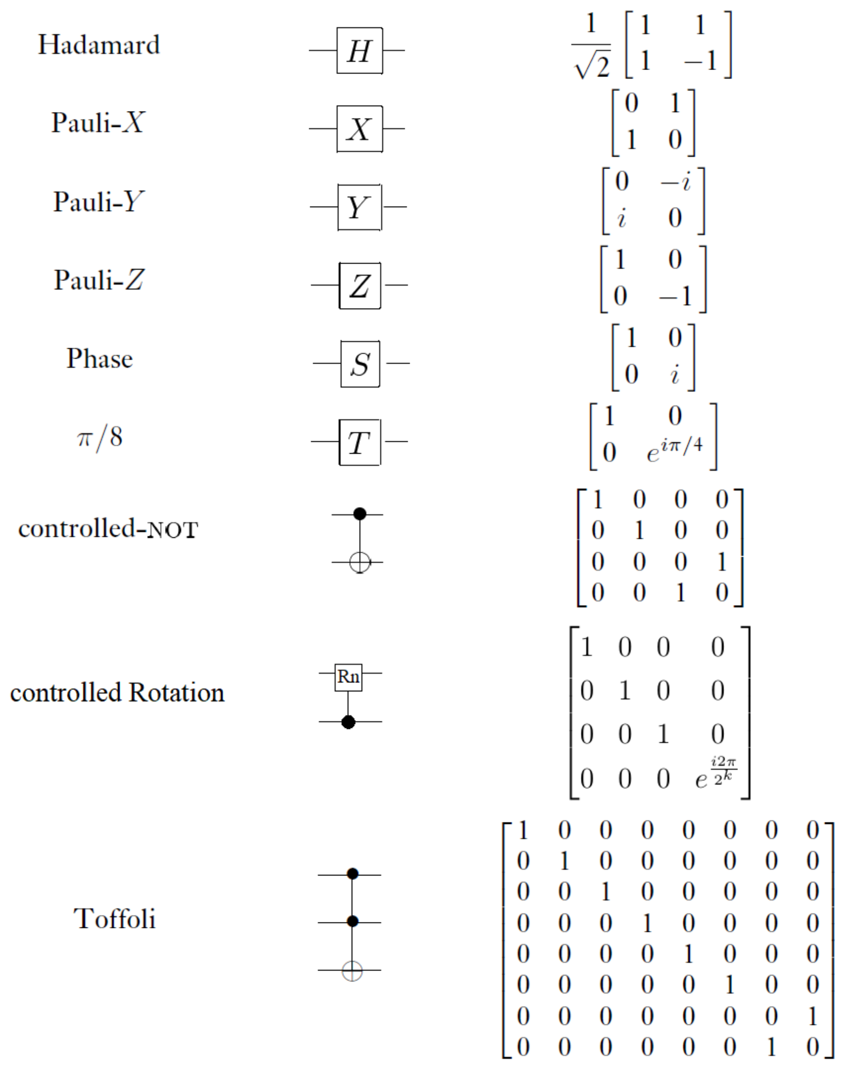
$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \alpha|1\rangle + \beta|0\rangle = \begin{bmatrix} \beta \\ \alpha \end{bmatrix} \tag{2.8}$$

It's important to mention again that quantum gates must obey the following conditions: they **must be reversible** and they **must be represented by a unitary operator**.

Being reversible means that from the output of the system we can recover the input. This question is extremely important in quantum computers, since laws of physics are reversible. Therefore quantum computers must also process information in a reversible way, to be able to simulate a real quantum system inside a quantum computer. This property is useful since quantum states can't be copied and any measurement destroys them. Being reversible allows us to recover the initial state which was not previously duplicated nor saved anywhere. Classical computation must also obey the laws of physics and still is usually performed in an irreversible way. It does not violate the laws of physics since there is a change in entropy [24]. This irreversibility has a cost in information loss that quantum computation wants to avoid.

The condition that gates are represented by an unitary operator relates to the need to preserve the norm of the state vector. Since the state of the system is quantum it must be normalized, as already discussed. As the quantum gates perform a logic operation on the state, it must assure that it keeps it normalized. This condition is assured if the gate is represented by an unitary operator. In linear algebra, a matrix is unitary if its conjugate transpose (named as its Hermitian conjugate in an Hilbert space) is equal to its inverse, *i.e.* $A^\dagger A = A^{-1}A = I$.

Figure (2.3) as some examples of gates. Gates X, Y and Z, are the Pauli operators. Hadamard and rotation gates are extremely important in quantum computation. If the Hadamard gate is applied twice it doesn't affect the qubit, since the inverse of the Hadamard is equal to itself, $H = H^{-1} = H^\dagger$. The phase and $\pi/8$ gates are merely special cases for the general rotation gate. The phase gate, also called S, causes a relative phase change of $\pi/2$ whereas $\pi/8$ gate, also called T, causes a



Hadamard	$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$
Pauli-X	$\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$
Pauli-Y	$\begin{bmatrix}0 & -i\\ i & 0\end{bmatrix}$
Pauli-Z	$\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$
Phase	$\begin{bmatrix}1 & 0\\ 0 & i\end{bmatrix}$
$\pi/8$	$\begin{bmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{bmatrix}$
controlled-NOT	$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}$
controlled Rotation	$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&e^{\frac{i2\pi}{2^k}}\end{bmatrix}$
Toffoli	$\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&1&0\end{bmatrix}$

Figure 2.3: Representation of some important quantum gates. The first column has the gate name, the second its circuit representation and the third the matrix operator of the gate. This image was adapted from [1]

relative phase change of $\pi/4$. The examples mentioned above are all single-qubit gates.

Multiple qubit gates are very interesting because they may introduce important relations between qubits. An important example in figure (2.3) is the controlled-NOT operation, the CNOT gate. For two qubits entries, CNOT gate applies the NOT operation on the second qubit, controlled by the first one. If the first qubit is $|1\rangle$, then the second qubit will be inverted. If the first qubit is $|0\rangle$ the second will suffer no change at all. This means that if the input of the system are bits, the gate behaves like the classic gate XOR. If the state is in a superposition state, the result will be more complex, in general it will be an entangled state. The figure (2.4) has the CNOT gate with the entry states $|A\rangle$, controlling the operation, and $|B\rangle$, where it is applied. The output states are $|A\rangle$ and the modulo 2 addition $|A \oplus B\rangle$, which is the result of the operation discussed above. Modular addition is an important concept which will be discussed and used in chapter 4. From the output qubits, we can reverse the circuit to recover the input information. It's possible to do so because we have in the output both the information about the original $|A\rangle$ and the relation between A and B.

If the input entries of the CNOT gate are both bits and $|B\rangle = 0$, the output will be a copy of the input bit $|A\rangle$. This is similar to the copy operation in a classical system. However, if $|A\rangle$ is in a superposition state, that is if $|A\rangle = \alpha|0\rangle + \beta|1\rangle$ and $|B\rangle = 0$ then the output will be $|\psi\rangle = \alpha|00\rangle + \beta|11\rangle$, which is an entangled state. Since an entangled state cannot be separated in two independent single qubit states, no copy was produced. Thus, we can produce a copy of a bit but not of a qubit. This conclusion can be proved in general and is known as the no-cloning theorem, see subsection (2.2.3.1).

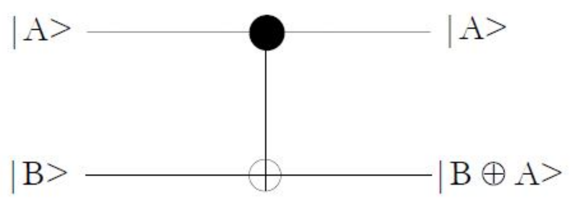


Figure 2.4: CNOT gate. Note that the output, $(B \oplus A)$, is the addition modulo 2 operation.

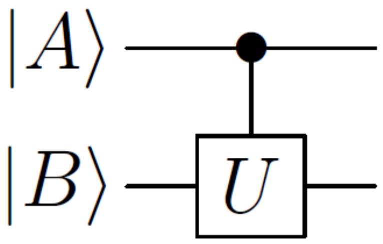


Figure 2.5: The general controlled U operation, where U is a general unitary operation.

Since CNOT works with two qubits, it has a wire connecting both qubits. The control qubit will have the closed dot symbol, and the target qubit, the one which changes if the control is $|1\rangle$, will have the cross symbol, or target symbol. If we need an inverted CNOT, this is, a CNOT that only changes the qubit when the control qubit is $|0\rangle$, then the closed dot is represented by an open dot. In the gate list of figure (2.3), we can see the Tofolli gate with two control qubits and one target. The controlled Rotation has a control qubit, and the target is the rotation, so instead of the cross symbol, it has the rotation gate symbol.

The concept of the controlled operation is very important and can be described for a general U operation being controlled by a qubit $|A\rangle$, as represented in the figure (2.5). If the control qubit is $|1\rangle$, then the operation U will act on qubit $|B\rangle$, otherwise, it does nothing. This controlled-U operation can be generalized to a controlled-U operation controlled by n qubits.

There is an infinite number of possible gates. As the number of gate entries increases, the corresponding matrices become huge. However, there is a small set, called **Universal set of gates (H, S, T and CNOT)** that can be used to build any computation process, disregarding the number of qubits at the entry of the circuit. All the large and complex processes can be built using this reduced number of gates, included in figure (2.3).

In classical computation, there is also a set of gates that can perform any operation. Gates like XOR, OR and NAND, represented in the figure (2.6), are extremely important in today's computation and they all have equivalent gates in the quantum realm.

The Tofolli gate is also an important gate because of its ability to reproduce classic gates using quantum reversible gates. This gate is the only one on the figure (2.3) with 3 qubits, and we can see that the matrix already has a considerable size. It behaves like a CNOT, but with an extra control qubit, only changing the state of the third qubit when both of the previous ones are $|1\rangle$. It can be used to make the classical NAND and FANOUT gates. With this two classical gates, it is possible then to simulate the other classic gates, allowing a conversion of classical non-reversible into quantum reversible gates.

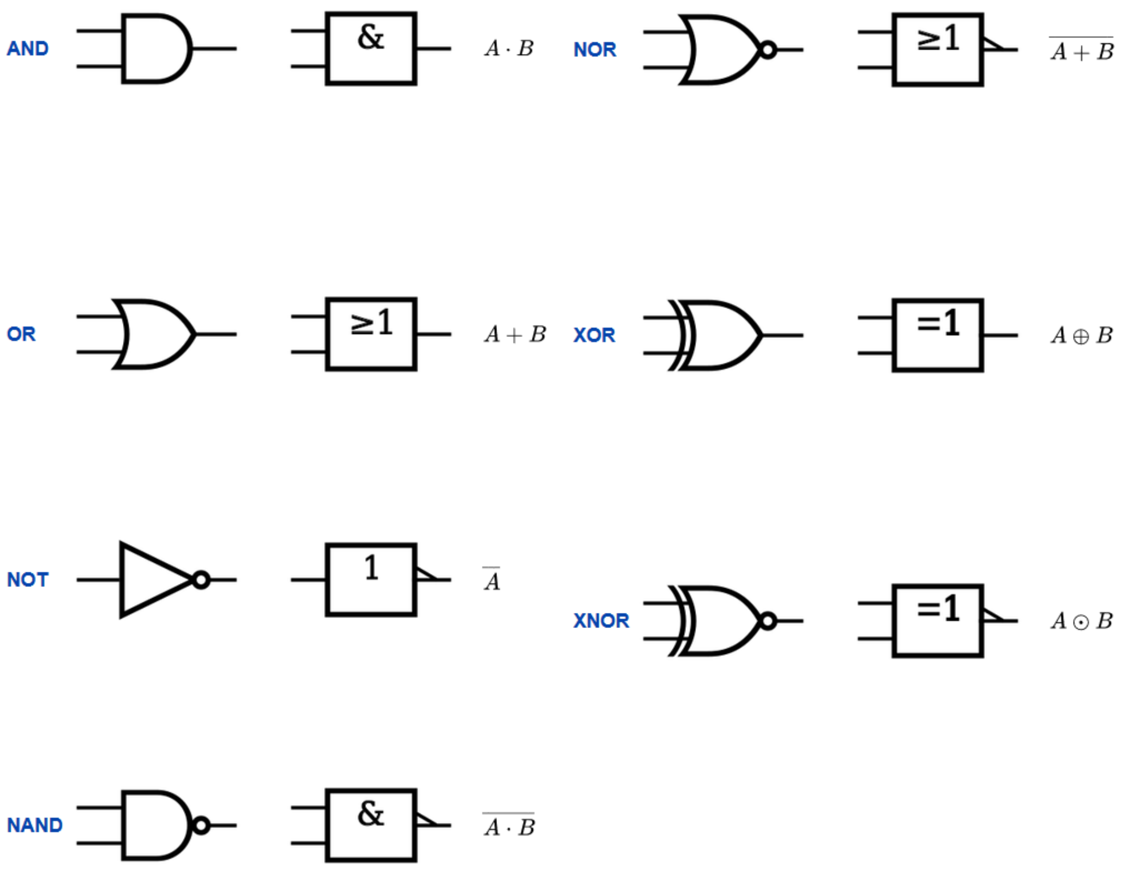


Figure 2.6: Representation of classical Logic Gates

2.2.3.1 No-Cloning Theorem

Cloning a qubit means to produce a similar copy of its state to another qubit. So, in order to clone a qubit with unknown superposition represented by the state $|\varphi\rangle$, we need another general qubit $|s\rangle$ ready to get the information from the first one. That way we have a two-qubit system:

$$|\varphi\rangle \otimes |s\rangle$$

Assuming the existence of a perfect unitary operator U which can clone a qubit, the following operation is possible:

$$|\varphi\rangle \otimes |s\rangle \xrightarrow{\text{U}} U(|\varphi\rangle \otimes |s\rangle) = |\varphi\rangle \otimes |\varphi\rangle$$

Now, let's assume the operator U also acts on another system, in order to do the same cloning process. In the end, we would have two qubits perfectly cloned, that is two identical $|\varphi\rangle$ states and two identical $|\psi\rangle$ states:

$$U(|\varphi\rangle \otimes |s\rangle) = |\varphi\rangle \otimes |\varphi\rangle$$

$$U(|\psi\rangle \otimes |s\rangle) = |\psi\rangle \otimes |\psi\rangle$$

However, if we take the inner product from the previous expressions, we get:

$$\langle\varphi|\psi\rangle = (\langle\varphi|\psi\rangle)^2$$

In order this condition to be true, either $\langle\varphi|\psi\rangle = 1$ or $\langle\varphi|\psi\rangle = 0$, for all the possible $|\varphi\rangle$ and $|\psi\rangle$. In other words, this means that either $|\varphi\rangle = |\psi\rangle$ or $|\varphi\rangle$ and $|\psi\rangle$ are orthogonal. That is true for bits $|0\rangle$ and $|1\rangle$ but not for qubits, in general. Therefore, it is impossible to clone qubits, unless they are bits.

2.2.4 Quantum Circuits

A quantum circuit implements a given operation and is an equivalent to a Quantum Turing Machine. They have the same goals, but circuits are easier to work with and to understand. As already mentioned, the operations performed on the quantum states are quantum gates, but it is important for the quantum computer to work with classical computers too. So, the circuit will include a series of quantum gates in order to make the necessary operations to the qubit. In the end, it will measure the qubit, and the output will be processed by the classical part of the system. Therefore, quantum computers are always hybrids, they still include classical parts, before (for state preparation) and after.

Let's examine a very simple quantum circuit, represented in the figure (2.7). The first thing worth mentioning are the lines, they are called wires, and represent connections on the system. However, it may not be a physical wire, it may also represent the passage of time or the movement of a certain particle. The circuit is always read from the left to the right. Each qubit entry in the system is represented at the left and then starts its own individual horizontal wire.

Wires connect operations, that is, gates. Gates are usually represented by a box in the wire with its own symbol. In figure (2.7) we can see gate CNOT and the gate Hadamard.

Lastly, at the right of the circuit, the boxes with an analogic measuring symbol,

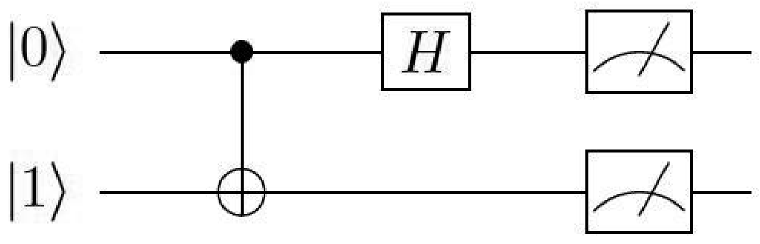


Figure 2.7: Example of a quantum circuit, includes an Hadamard gate and a CNOT gate. At the end, has the measurement operations.

represent the measure of the qubit. Of course, after the measurement the qubit will be destroyed, becoming only a bit. After this, the circuit needs to be connected to a classical computer in order to work with the bits.

2.2.5 Quantum Algorithms

The objective of quantum computation is the development of quantum algorithms and its implementation in a quantum computer. Quantum algorithms explore new solutions in order to be able to compute problems that are very complex and nearly impossible to compute in useful time, with classic computation algorithms.

Quantum algorithms are methods to solve difficult problems, such as the prime factorization of an odd positive integer (Shor's algorithm), using quantum gates and making quantum circuits.

Quantum algorithm design is an area of intense research effort, with good developments. It is a fairly complex thing to do, arranging the best mathematical way to solve a problem in an efficient way that can be implemented physically.

Some quantum algorithms already exist for decades, and only now we are starting to be used in quantum computers with capabilities to implement them. They are elegant ways to solve a problem, but sometimes quantum algorithms need to be improved in order to be more efficient. The field of quantum algorithms is constantly advancing despite being extremely complex.

There are two big classes regarding quantum algorithms, represented in the figure (2.8). One is the Quantum search class, introduced by Grover, which improves the classic search-based algorithms adopting some of its principles, but can perform with a quadratic speedup.

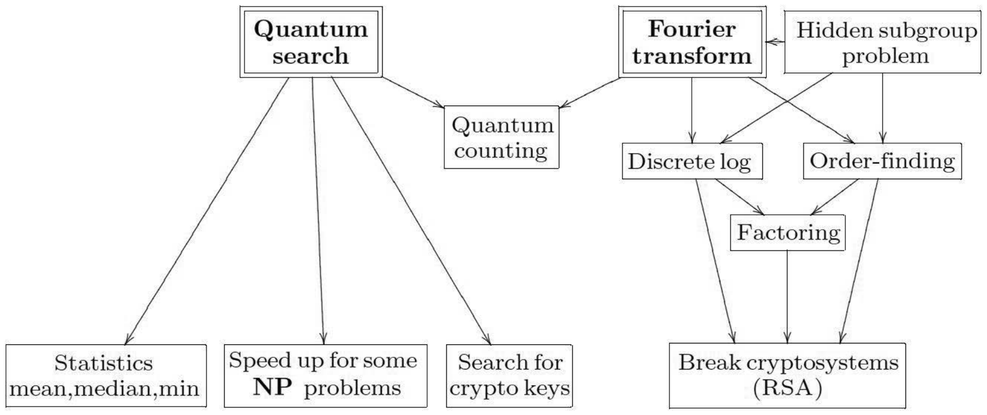


Figure 2.8: Quantum algorithm group tree. Two big groups are represented: Quantum Fourier Transforms and Quantum Search. Source:[1]

The other big class of quantum algorithms uses the **Quantum Fourier Transform** (QFT) and includes the very important Shor's algorithm. The QFT is the quantum variation of the Fourier Transform, which already has great importance in classical algorithms. Using QFT is very useful for problems of factorization, providing an exponential speedup over classical algorithms with the same objective. It can also be very advantageous for cryptography systems, which are having a huge development nowadays.

QFT is the class of algorithms studied in this thesis, being the basis to construct the modular addition algorithm and the Shor's algorithm.

2.2.5.1 Quantum Fourier Transform

One useful way to solve complex problems is to transform the problem into one that already has a solution. This method is used extensively in classical computation. Fourier Transform is one of the most used techniques to do it. For example, a complicated convolution operation can be simplified by doing the multiplication in the transform domain.

So, if used properly, the Quantum Fourier Transform, will have enormous speedup benefits to solve problems that otherwise would be extremely hard to compute.

In order to comprehend how the QFT works, we start with the classical discrete

Fourier Transform expression, represented by the equation (2.9).

$$y_K \equiv \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{2\pi i jk/N} \tag{2.9}$$

Where x_j are the input of the equation and y_k, represents a transform of domain. $N = 2^n$, n being the number of bits. In Fast Fourier transforms, it commonly changes from the time domain into the frequency domain.

The QFT can be viewed as a discrete Fourier Transform of the amplitudes of a quantum state. If the input states are the basis states, $|j\rangle$, the QFT can be defined as the map:

$$|j\rangle \longrightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i jk/N} |k\rangle \tag{2.10}$$

The QFT operator is unitary (for a demonstration see [1]). Since it is a unitary operation, we can construct a general circuit for the QFT. Using some algebra, it is possible to demonstrate (see [1]) that the general circuit for QFT of a n qubit system is the one represented in figure (2.9). The operator R_n in figure (2.9) is:

$$R_n = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix} \tag{2.11}$$

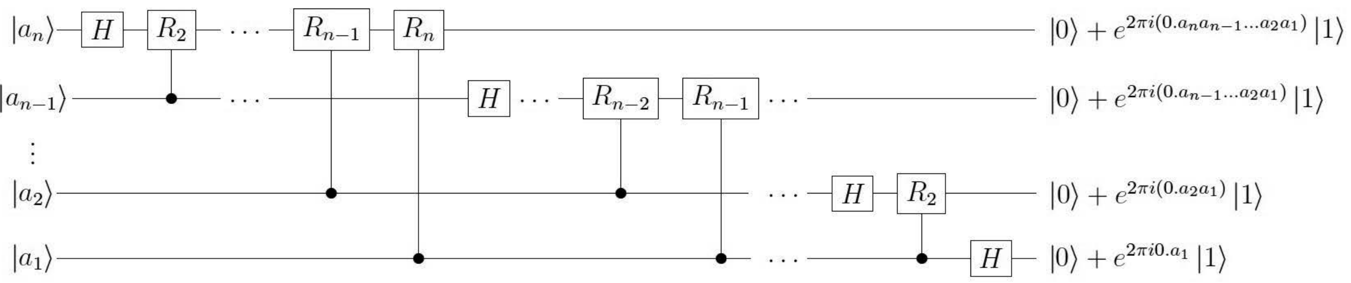

$|a_n\rangle$
$|a_{n-1}\rangle$
$|a_2\rangle$
$|a_1\rangle$
H
R_2
R_{n-1}
R_n
R_{n-2}
$|0\rangle + e^{2\pi i(0.a_n a_{n-1} \ldots a_2 a_1)} |1\rangle$
$|0\rangle + e^{2\pi i(0.a_{n-1} \ldots a_2 a_1)} |1\rangle$
$|0\rangle + e^{2\pi i(0.a_2 a_1)} |1\rangle$
$|0\rangle + e^{2\pi i 0.a_1} |1\rangle$

Figure 2.9: Circuit representation of QFT for a general n qubit system.

.

In order to understand better how the QFT works, let's see a 3 qubits example.

First, let's understand how a binary number is represented as a qubit. Let a be an integer number which is a binary representation, it has 3 bits a_3, a_2 and a_1:

$$|a\rangle = |a_3\, a_2\, a_1\rangle = |a_3\rangle \otimes |a_2\rangle \otimes |a_1\rangle \tag{2.12}$$

The number a, in a decimal representation is given by:

$$a = a_3\, 2^2 + a_2\, 2^1 + a_1\, 2^0 \tag{2.13}$$

For example, if $a = 3$, the binary representation is 011, which relates to the decimal three by $3 = 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$. Therefore, $|3\rangle = |011\rangle = |0\rangle \otimes |1\rangle \otimes |1\rangle$

In order to make it concise, let's adopt the **binary fraction** notation $(0.a_k\, a_{k-1} \cdots a_1)$ as follows:

$$(0.a_k\, a_{k-1} \cdots a_1) = \frac{a_k}{2^1} + \frac{a_{k-1}}{2^2} + \cdots + \frac{a_1}{2^k} \tag{2.14}$$

Now lets see the 3 qubit circuit, and analyse its output.

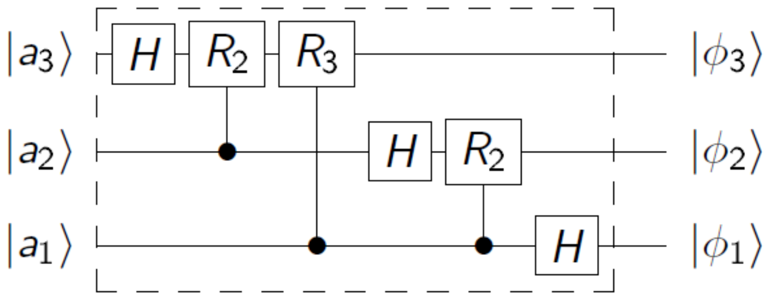


Figure 2.10: Circuit representation for a 3 qubit system.

Considering the entry $|a_1\rangle$ in the circuit of figure (2.10), only one Hadamard gate applies. If a_1=0, then $H\,|a_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. If $a_1 = 1$, $H\,|a_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Both results can be written in a concise form as:

$$|\phi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{\pi\, i\, a_1}\,|1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{2\pi\, i\,(0.a_1)}\,|1\rangle) \tag{2.15}$$

For the qubits $|a_2\rangle$ and $|a_3\rangle$, we must also consider the operators R_2 and R_3:

$$R_3 = \begin{pmatrix} 1 & 0 \\ 0 & \mathrm{e}^{i\pi/4} \end{pmatrix} \quad R_2 = \begin{pmatrix} 1 & 0 \\ 0 & \mathrm{e}^{i\pi/2} \end{pmatrix}$$

As represented in the circuit, the qubit $|a_2\rangle$ will have a controlled rotation (R_2) after an Hadamard gate operation. The total operation is represented by $R_2(H\,|a_2\rangle)$ and we can write the final state as:

$$|\phi_2\rangle = \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{2\pi\, i(\frac{a_2}{2}+\frac{a_1}{4})}\,|1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{2\pi\, i\,(0.a_2\,a_1)}\,|1\rangle) \tag{2.16}$$

Note the differences between the equations (2.15) and (2.16). The term of the expression that relates to the Hadamard operation has the entry bit as a factor in the phase (a_1 bit in the case of equation (2.15); a_2 bit in the case of equation (2.16)). The term that relates to the controlled rotation has the bit which is controlling the rotation as a factor in the phase (a_1 bit in the case of equation (2.16)). Now, the third qubit has an additional controlled rotation R_3 after Hadamard and controlled rotation R_2, giving the result shown in equation (2.17).

$$|\phi_3\rangle = \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{2\pi\, i(\frac{a_3}{2}+\frac{a_2}{4}+\frac{a_1}{8})}\,|1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{2\pi\, i\,(0.a_3\,a_2\,a_1)}\,|1\rangle) \tag{2.17}$$

This example of three qubits can be generalized to the QFT circuit in figure (2.9), for n qubits.

The general circuit of the QFT (figure 2.9) only uses the controlled rotation gate and Hadamard gates. For each of the n qubits, seen on the left, starting in the basis $|a_n\rangle$, the first operation applied to every qubit is the Hadamard gate. The Rotation gate R_n ($n = 2$ to n) is controlled by $|a_j\rangle$ with $j = n - 1$ to $j = 1$. Note that there are other notations for the binary fractions regarding which bit is defined as the most significant bit (a_n or a_1) , which causes a change in the order of the input bits in the circuit relative to the circuits considered here. In those cases, a SWAP operation needs to be applied at the end of the circuit to get the same order. We can understand the effect of QFT by observing the final states of the circuit, which have the bits j_n in the qubits phase. It is a very powerful tool if it's used in a problem where operation can be performed more easily in the phase domain, for example, phase estimation.

It is important to mention that Inverse Quantum Transform (IQFT), as the name says, is the operation that will take us from the phase domain into the original domain. For the 3 qubit example given, we have the final states given by the expressions (2.15) to (2.17). The IQFT will bring us back to the initial states $|a_1\rangle$, $|a_2\rangle$ and $|a_3\rangle$. The operations are the algebraic inversions of the ones given. Note that the gate order is not the same as before since $(AB)^{-1} = B^{-1}A^{-1}$. Note also that the inverse of the Hadamard is the Hadamard itself ($H = H^{-1} = H^{\dagger}$). The IQFT circuit for three qubits is represented in figure 2.11.

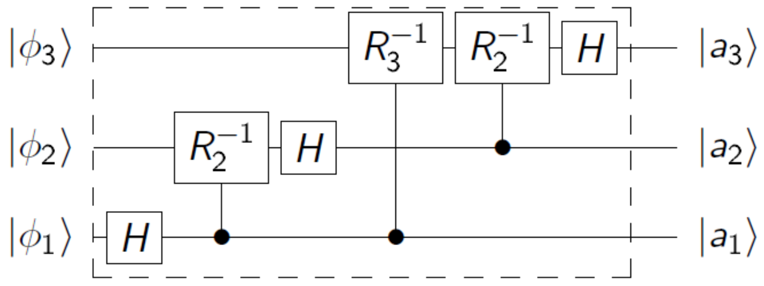


Figure 2.11: Circuit representation of the 3 qubit IQFT.

2.2.5.2 Shor's Algorithm

What is today called Shor's Algorithm, was introduced by Shor in 1994 [17] as a method to solve the factoring problem of large integer numbers. It is one of the most important algorithms in quantum computation and its speedup capability is due to the use of QFT to solve the problem. Its importance derives from the utility of prime number factorization, which is used in the RSA cryptosystem. It's not possible to obtain the private key of the RSA protocol in useful time with a classical computer. However, Shor's algorithm suggests that it could be possible with a quantum computer. This meant that an incredible desire to build a fully functional quantum computer started to grow [25], and now it's starting to give some results.

Shor's algorithm is not a single algorithm, is a sequence of algorithms (subroutines). It decomposes the factoring problem into an order-finding problem which uses a phase estimation tool. QFT is really good at doing phase estimation for periodic states, and this makes it possible to use it in this algorithm, giving a speedup advantage.

First things first: factorization of an integer number N means to discover the largest prime number k which divides exactly N, which can't be equal to 1 or N. If we keep dividing the previous division result by another factors, we will eventually have a number N factored by a product of different factors k_i.

The phase estimation is the process of estimating the phase φ in the eigenvalue $e^{2\pi i\varphi}$. This is the eigenvalue of the eigenvector $|u\rangle$ of the unitary operator U, that is $U|u\rangle = e^{2\pi i\varphi}|u\rangle$. To do the estimation we need some kind of *black box* [2], to prepare the state and perform the controlled U^{2^j} gate. We also need two registers of qubits. The first register has a number of qubits t in the state $|0\rangle$. The second register has the initial state $|u\rangle$ represented by as many qubits as the process needs. Now, the first register will suffer the action of Hadamard gate into all the qubits. After the Hadamard, the first register acts as the controller for the U operations. After these two operations the states from the first register, go from $|0\rangle$ to:

$$\frac{1}{2^{t/2}} \sum_{k=0}^{2^t-1} \mathrm{e}^{2\pi i \varphi k} |k\rangle$$

This is the state of the qubits from register one, which means the register that started as $|0\rangle$ now has the information about the phase of the system. We can see that the state is similar to the state after a QFT (equation (2.10)), so if we apply the inverse QFT, we will get the value of φ.

The order-finding problem tries to find the solution to $x^r = 1(\mod N)$. First, this implies modular arithmetic, which will be discussed in the modular addition chapter 4. The order that we are talking about is the number r. Having a number x, another N and the condition of $x < N$, the order will be r, for the operation x *modulo* N. One example would be, with $x = 5$ and $N = 21$, the r that makes the equation $5^r \mod 21 = 1$, which is $r = 6$. Now we can see how order-finding subroutine works.

This subroutine uses the phase estimation as the way to solve the problem, applies this algorithm with an unitary operator U which must be implemented efficiently. The unitary operator U will give us the states we desire, with $0 \leq y \geq N - 1$:

$$U|y\rangle \equiv |xy(\mod, N)\rangle$$

This being true, we can apply the operator U to the $|u_s\rangle$ states. This states are represented by:

$$|u_s\rangle \equiv \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \mathrm{e}^{\frac{-2\pi isk}{r}} |x^k \mod N\rangle$$

Now, applying the U operator, we get the result that matters to the problem:

$$U|u_s\rangle \equiv \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \mathrm{e}^{\frac{-2\pi isk}{r}} |x^{k+1} \mod N\rangle \;=\; \mathrm{e}^{2\pi i \frac{s}{r}} |u_s\rangle$$

[2]A *black box*, can also be called *oracle*, is an abstract way of describing an operation. It is used in an algorithm that includes a process that does a particular job and we are not interested in detailing how the job is done. The operation in the oracle can be described in detail later, with some physical implementation in mind.

Thus, u_s is an eigenstate of U with eigenvalue $e^{2\pi i \frac{s}{r}}$. From this point, if we have a state $|u_s\rangle$, we can just use the phase estimation problem in order to see the result for the eigenvalue, which has the order r in it. However, in order to do this step, we need to have a state $|u_s\rangle$, but this state is unknown and can't be prepared to meet this conditions. Fortunately, the condition:

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

allows us to initiate the states from the second register on the state $|1\rangle$. The proof of this condition will not be discussed here, see [26] for more information.

The only thing that is left is to retrieve the order r after measurement since the eigenvalue is a fraction. When measuring the first register the result will be approximately s/r, then another algorithm is needed to retrieve r from the fraction, see [1].

The Shor's Algorithm will transform the problem of factoring a number N, in the order finding problem, and this will transform it into the phase estimation problem. It does this process with two theorems that come from number theory, which will not be discussed here, we will just state the conclusions. First, it's the theorem that says that we can always find a factor of N if we find a non-trivial solution for the equation $x^2 = 1(\mod N)$. Then, at least one of the greatest common divisor between $(x-1,N)$ or the $gcd(x+1,N)$ is a non-trivial factor of N. Now, according to the second theorem, if we choose randomly a number y, co-prime with N, the probability of this number having an order r that is even and a non-trivial solution to the equation mentioned above, is quite high. This means that $x \equiv y^{r/2}(modN)$. At this point, the problem can be included in the other algorithms. This algorithm can factor a number per run, the only thing that is needed to find the other factors, is to re-run the algorithm. The image (2.12) shows us the circuit representation for the order-finding problem. In terms of the circuit, this is equivalent to Shor's algorithm, because the first part of the problem can be made in a classical computer.

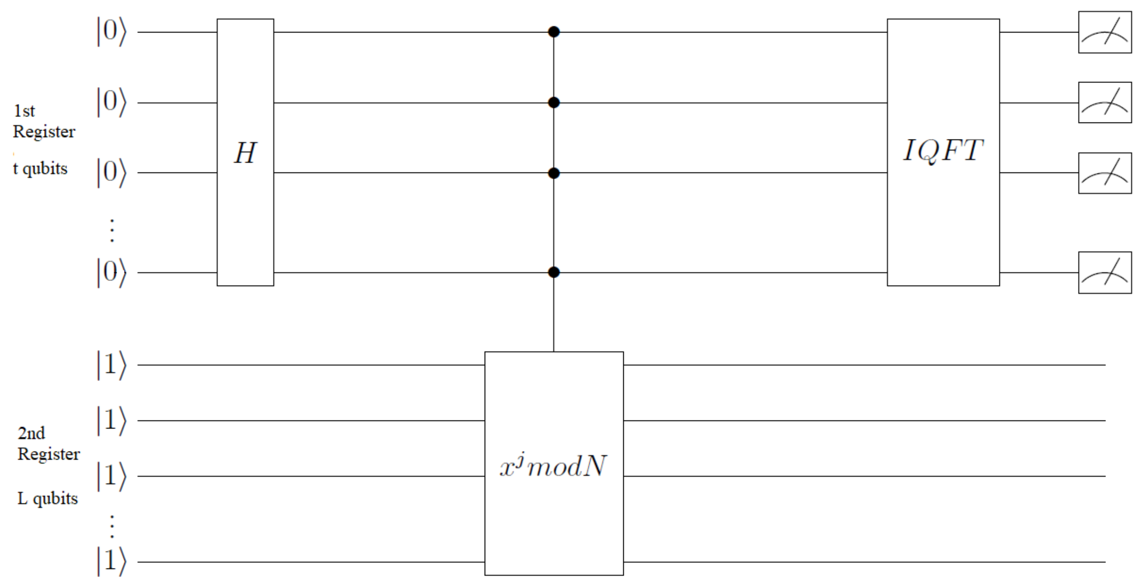


Figure 2.12: Circuit representation of the algorithm for the order-finding problem, hence also the circuit for the algorithm of Shor.

As we can see in the circuit, the first register gets the order r, more precisely the fraction s/r, after the IQFT. That's why upon measuring, it gives an estimative that need to use the continued fractions algorithm to give the real value r. The controlled operation U, is represented by $x^j \mod N$, because is the operation that it maps. The operator has an eigenstate which is in the second register and its eigenvalue is $e^{2\pi i \frac{s}{r}}$.

It's important to note that the operation $x^j \mod N$ is applied in the transform phase domain to the first register. So, logic dictates to implement QFT at the begging of the circuit, in order to transform into phase domain. However, since the initial states are all $|0\rangle$ in the first register, the QFT operation is simplified into only the Hadamard gates. The controlled rotations will have no effect. Note that at the end of the circuit the IQFT is applied.

The general outline of the algorithm is represented in figure (2.13).

General outline for Shor's Algorithm

Input: Integer N.

Output: Factor of N.

Principal Stages:

1. First thing to do is verify if the number N can be factorized by a classical process. If it is an even number the first factor will be 2, and in some cases it is simpler to do the factorization using a classical algorithm.
2. At this point, the true algorithm begins, so it's time to choose a number x in the range of 1 to $N-1$.
3. Time to use the order-finding subroutine. Solve the equation $x^r = 1(\mod N)$, find r.
4. Start the first register with t qubits in the state $|0\rangle$ and start the second register with N qubits in the state $|1\rangle$.
5. Apply the general circuit of the figure (2.12), using H, controlled-U and IQFT operations.
6. Measure the first register to obtain an approximation of s/r.
7. Use a subroutine to get the best estimation of r. Output $FAIL$, if no good values for r were found.
8. If the order r makes $x^{r/2}$ not even or trivial, output $FAIL$. If $gcd(x^{r/2}-1{,}N)$ or $gcd(x^{r/2}+1{,}N)$ is a non-trivial factor of N, output $SUCCESS$ and the factor. Else, output $FAIL$.
9. Repeat the algorithm for the other factors of N.

Figure 2.13: General outline for the Shor's Algorithm.

2.3 Some basic principles of Computational Science

2.3.1 Quantifying Computational Resources

The great objective of quantum computers is to be able to run algorithms in a short amount of time. That's the fundamental question of the computation science, always improving the algorithms and the hardware to do more in less time. However, it's not that easy to quantify the time that takes an algorithm to run, since a classical computer has a lot of factors that might affect the running time, and so does a quantum computer. For the study of algorithms, what really matters is the general behaviour of the algorithm, namely, how the running time scales with the size of the problem. So, there is the need to quantify the efficiency of the algorithms, that's why there is the **asymptotic notation**.

Using the **asymptotic notation**, the only thing that matters is not the time itself, but how many time steps the algorithm takes to solve the problem. Time steps are the operations that take time to execute, this is, the operations needed to do the algorithm. In the case of quantum computation these operations are gates. If we see a circuit of a quantum algorithm, the only thing that takes important amounts of time are the gates. These operations are executed, usually with computational steps that take almost always the same time. So analysing the number of operations is a good way to understand how the function behaves as it grows.

In order to understand how the function grows, we want to have it dependent on one of its variables. For example, adding two numbers of n bits. The algorithm may take $24n + 2[\log\ n] + 16$ operations to give a solution. Well, in the grand scheme of things, the n term is the most important because it's the biggest one, $\log\ n$ does not grow as fast. Constant values are not really important, so we might say that this algorithm depends on n, and scales with it.

We already established how the basic principle of this notation works. Now let's consider the three different symbols used:

***O* (big O)** gives the absolute maximum of the number of operations that the function requires, it's the upper bound. Giving one example of its use, if we have two functions, $f(n)$ and $g(n)$, we can say that $f(n)$ belongs to the class of functions $\boldsymbol{O}(g(n))$ as long as the function f is always smaller or equal to the g function times a constant. This is the same as saying that $g(n)$ is the upper bound of the function $f(n)$, *i.e.* it's the worst performance possible for that function. This is a very useful tool for the analysis of algorithm performance.

Ω (big Omega) notation is used to refer to the lower bounds of functions, in the same line of thought as the big O scenario, but this time, the function $g(n)$ acts as the lower bound for the $f(n)$ function. The notation is used exactly in the same way, we can say that the function f is of the kind of $\boldsymbol{\Omega}(g(n))$, where for large n, the performance will be the performance of g. It's a best-case scenario, this algorithm can't have better results than this without any improvements.

Θ (big Theta) notation is used to indicate one function with the same behaviour of another. In this case, the function $g(n)$ will not be upper bound nor lower bound, they will have the same performance. It is used like before, if f is of $\boldsymbol{\Theta}$ $(\boldsymbol{g(n)})$ then $\boldsymbol{g}$ is both the upper and lower bound, which makes it $\boldsymbol{O(g(n))}$ and $\boldsymbol{\Theta(g(n))}$.

2.3.2 Computational Complexity

It was said that quantum computers will overpower their classical peers. However, it's hard to evaluate how much difference there is and how it will be translated into more efficient problem-solving. That's why it is important to the know the classes of problems that can have efficient solutions and the time they require. It's rather important that we study the resources of space and time that each algorithm takes in order to understand which one will be better for each use.

Computational complexity is the area devoted to the study of the time and space resources. It's very important and allows us to understand how hard can it be to compute an algorithm in a classical or quantum computers. It can make it easier to analyse the question of which algorithms will perform best in which kind of computers.

This area of study is intrinsically connected to the design of algorithms, since one of its fundamental steps is to define the most efficient way to do the algorithm, regarding resources. So, the **computational complexity theory** provides us with the answer for the lower bounds of the time and space resources of the algorithm. This is an area that started in classical computation and it's used today in quantum computation. Quantum computation complexity has some unique classes, as can be seen in [27].

It's important to define a complexity class, but we will only describe them briefly. This is a huge area of study, and exceeds the scope of this thesis. For more information on computational complexity, is suggested the complexity zoo [28], which is a gathering of complexity classes in a form of a zoo, or the book [29].

A complexity class can be seen as a group of problems that will take the same amount of resources to be solved. Since this is a theory to help designing algorithms, it takes sometimes the Turing machine as the problem-solving machine, because it is a perfect theoretical way to understand the needed resources. The problems may also make use of the terminology of formal languages. A complexity class is defined based on three different aspects: the type of resource (time, space); the type of problem; and the computational model. Some important complexity classes are:

P group of problems that can be computed in a short amount of time with a classical computer. A short amount of time is relative. In this case, it means it can solve the problem in polynomial time.

NP problems, for which the solution can be checked quickly in a classical computer. It's not the same class as **P**. **NP** includes problems that can't be solved on a classical computer in a polynomial time, but their solution can be checked in such time.

PSPACE refers to the physical space needed to solve problems. Includes all problems that can be solved with a polynomial number of bits without time limits. This means that they can be solved in a relatively small hardware, even if it takes very long time to solve.

BPP this class includes all the problems that can be solved with algorithms using random methods in a polynomial time, as long as they can have a 1/4 of error probability to the solution. This can use a probabilist Turing machine, already discussed.

BQP So far the classes discussed were all classical. This class is the quantum complexity class equivalent to the **BPP** class.

As an example, we go back to the problem of factoring a prime number again. We can say that this is not solvable in a classical computer in polynomial time, in fact, it requires exponential time. So, it's believed that this problem does not belong to **P** class. However, we can check if a specific factor of the solution is correct in a classical computer, so it belongs to **NP**.

It is not yet proven that **NP** and **P** classes are all that different, it's hard to prove that $\mathbf{P} \neq \mathbf{NP}$. In theory **P** is a subset of **NP**, since that in order to solve a problem it is needed to check for its solution. In the example above, even though there is not a solution for factoring in polynomial time in a classical computer, it is not proven that it can't be done. So, there is a possibility that they are the same class [1].

PSPACE may be larger than **NP**, which makes it a bigger set of problems with **NP** as subset and **P** as sub-subset. How large is **PSPACE** and if this class has problems that are not from **P** is not known.

This theoretical discussion of classes and subsets of classes has a lot of undefined boundaries and a specific number of problems. It raises important questions concerning quantum computers, which problems it may solve in which class, and which boundaries it may break.

BQP has an undefined boundary, it is somewhere between **P** and **PSPACE**. Quantum computers can be efficient to solve the problems of **P**, but none out of **PSPACE**. So it's rather important to explore quantum computer algorithm design to understand what is exactly the scope of quantum computers.

Since this discussion is mostly theoretical, it's not required to fully understand all the complexity classes or their boundaries. It's important to discuss it to understand how these tools work in a general way, and to see what may be the benefits of quantum algorithms.

In a more practical way, the discussion in algorithm design lies in a classification of *exponential* versus *polynomial*. These terms are used to roughly describe the behaviour of the function when the number of entries grows to infinity. There are cases where the function grows faster than expected for a polynomial behaviour, but still not quite as fast as expected for an exponential behaviour. However, it is considered as an exponential type function. This is an easy and fast way to confront the performance of algorithms, and understand how efficient they may be.

As we will discuss in the next chapter, the polynomial algorithms may not be always more efficient than the exponential ones. In fact, for small values of the variables, most of the times the exponential type algorithm may work better. However, as the variables grow to infinity it becomes evident that polynomial behaviour is the answer. That's why quantum computers are developed, to complement classical computers in order to solve some problems for large values of the function variables.

Let us now consider the performance of the QFT and the Shor's algorithm, since now we have the tools to classify their efficiency.

The QFT needs $1 + 2 + \cdots + n = n(n+1)/2 = n^2/2 + 1/2$ gates to perform the circuit. Using the rules discussed above, this means the algorithm has boundaries of $\mathbf{\Theta}(n^2)$. There is an approximated QFT algorithm [20] that improves the original version to use only $n\log_2(n)$ operations.

The best classical Fourier Transform is the Fast Fourier Transform which needs $n2^n$ gates, thus is described by $\mathbf{\Theta}(n2^n)$. Looking to this expressions, the classical algorithm is of exponential type and the QFT is of polynomial type. And, as already discussed, the QFT is particularly useful for some specific applications.

Regarding Shor's algorithm, the best classical algorithm uses $\exp(\mathbf{\Theta}(n^{1/3}\log^{2/3} n))$ operations, which means, that this is the maximum number of operations. The quantum algorithm version, discussed before, can accomplish the same result with only a maximum of $\mathbf{\Theta}((\log N)^3)$ operations. The classical algorithm is exponential, while the quantum version reduces it to a polynomial version. That's why factoring prime numbers in a classical computer is very hard, the time needed increases exponentially with the size of the number to be factorized.

2.3.3 Quantum Parallelism

Quantum computation can overcome classical computation using some remarkable features, one of the most important is quantum parallelism. This is the ability to compute more than one variable at the same time. Unlike classical parallelism, which uses parallel circuits running at the same time, here we have true parallelism. The variables are computed at the same time, using the superposition of states. The Deutsch's algorithm was the first quantum algorithm that proved to be able to evaluate more than one entry at the same time.

Using a two-qubit example, represented in the figure (2.14), with an operation U performed by a black box, we have an input of two qubits $|x\rangle$ and $|y\rangle$. The output for this particular operation will be the qubit $|x\rangle$ and the state $|y \oplus f(x)\rangle$, $\oplus$ means addition modulo 2, involving the generic function $f(x)$.

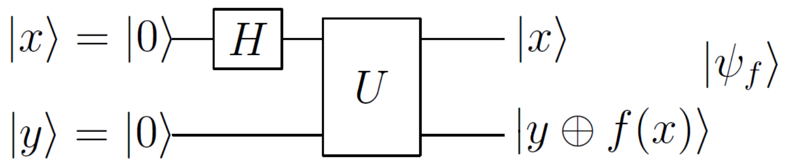


Figure 2.14: Circuit that relates the values of $f(0)$ and $f(1)$ at the same time, has $|x\rangle$ and $|y\rangle$ as input.

The input state $|x\rangle$ will suffer the effect of the Hadamard gate, into a superposition of states $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$. The final state of the system will be very interesting, with the addition modulo 2:

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

This result contains the information about $f(x)$ as supposed, but since $|x\rangle$ is in superposition, the final state will have information regarding the value of $f(0)$ and $f(1)$. The final state has two different values of the function $f(x)$, even though the operation was made only once. This is true parallelism, the values $f(0)$ and $f(1)$ were computed at the same time. However, the final state is still in superposition, which means measuring the final state will only give one of the values, $|0{,}f(0)\rangle$ or $|1{,}f(1)\rangle$.

This was an example for two qubits, but can be done with n qubits, giving the final state expression:

$$|\psi\rangle = \frac{1}{\sqrt{2^n}} \sum_x |x\rangle\, |f(x)\rangle$$

The method to retrieve information from the final state, is using the Deutsch's algorithm. Let's see the circuit implementation of the algorithm:

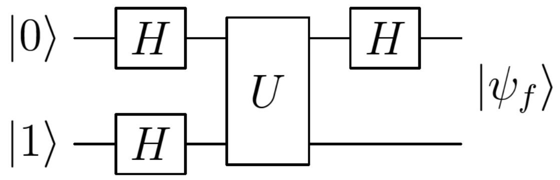


Figure 2.15: Circuit for the Deutsch's algorithm, inputs are $|0\rangle$ and $|1\rangle$. The output of the system is $|\psi_f\rangle$.

The figure (2.15), has input states $|0\rangle$ and $|1\rangle$, which is already different from the previous circuit. Now, applying the Hadamard gate to both states, we get the state $|\psi\rangle = [\frac{|0\rangle+|1\rangle}{\sqrt{2}}][\frac{|0\rangle-|1\rangle}{\sqrt{2}}]$, just before the U operation. After the U operation, and applying an Hadamard gate to the first qubit, we get the final state:

$$|\psi_f\rangle = \begin{cases} \pm\,|0\rangle\,[\frac{|0\rangle-|1\rangle}{\sqrt{2}}] & , if\ f(0) = f(1) \\ \pm\,|0\rangle\,[\frac{|0\rangle-|1\rangle}{\sqrt{2}}] & , if\ f(0) \neq f(1) \end{cases}$$

This can be written in a more concise form with the help of the modulo 2 operation, becoming the final state:

$$|\psi_f\rangle = |f(0) \oplus f(1)\rangle\, [\frac{|0\rangle - |1\rangle}{\sqrt{2}}]$$

So, having both the states $|x\rangle$ and $|y\rangle$ in superposition, we can obtain this final state, which upon one measurement yields $f(0) \oplus f(1)$. This result obtained after measurement contains a relation between the values $f(0)$ and $f(1)$, obtained with a single evaluation of $f(x)$.

The Deutsch's algorithm was the first demonstration that quantum parallelism can be used to process information in a more efficient way than a classic algorithm. It can retrieve information from multiple values of a function with only one evaluation, which is not possible in classical computation.

3

LIQUiD Language and Performance

3.1 Simulation

One important step of the process of designing and implementing an algorithm is its simulation, since it allows to verify results, to debug code and to discover new insights. This is the focus of this thesis: simulate a quantum circuit, in order to verify the expected results. Since constructing a quantum computer is still a huge endeavour, not to mention its cost, the feasible solution is to test a quantum algorithm in a classical computer. The results are limited but can lead us to the right path.

First of all, let us clarify the meaning of simulation. It should not be confused with the process called **quantum simulation**, which is the process of simulating a physical quantum system with a quantum device. Since a physical quantum system is fairly complex, it turns out that a classical computer is a really bad tool to simulate it, since it needs a lot of memory to store the information, which grows exponentially with the size of the system. Richard Feynman proposed in 1982 [30] that we could simulate a quantum system in a quantum device since they use the same physical laws and the resources needed will grow linearly with the size of the problem. Quantum simulation is used to understand complex systems, namely in quantum chemistry.

The original Moore's Law for classical computers is one of the reasons why is so important to invest in quantum computers. The Heisenberg principle gives Moore's law a limit [31] so there is the need to have an alternative, which is quantum computers. The original Moore's law states that the power of classical computers will double every two years, at least. However, simulating a quantum system in a classical computer has an exponential need for resources, which means computers would need to increment their power in half the time. Quantum computers only need to add one qubit every two years in order to have a similar power increase. Sometime in the future quantum simulation will overpower completely corresponding tasks in classical computers.

Quantum simulation will not be the focus of this thesis. Here the word simulation stands for the simulation of a quantum circuit. A quantum computer needs a code in order to implement a circuit. There are multiple layers of software between the algorithm and the hardware that are fundamental to the operation of the system. Even though simulating a quantum system in a classical computer is costly in terms of resources, it can tell us how the final algorithm will work and if there are any errors.

Circuit Simulation is available on classical computers via simulation environments, existing in a vast range of languages and libraries, and it's important to choose an effective environment. A quantum computer is a hybrid system, in the sense that it needs two different types of systems to work. It needs classical computers to help implementing the circuit, to control other secondary systems and to work with the results of the measurements, and a quantum part to run the circuits. In figure (3.1) we have a schematics of how a quantum computer system works, namely a stack of layers from the software to hardware. The algorithm is implemented in high-level languages in classical computers, and then it needs both systems running. The quantum part will be responsible to implement the algorithm into a circuit, after optimizing it and correcting errors using the Quantum Error Correction(QEC) code. Simulating environments will handle a small number of qubits but can perform all the tasks required prior to the hardware implementation. Since debugging a quantum code is hard and can have many different kinds of errors, it's very important to use simulation to optimize algorithms. The Microsoft research group has already made corrections to some theoretical algorithms using the environment they created, named LIQ$Ui\,|\rangle$ [32]. Their work also led to optimizations of the Shor's algorithm using Toffoli gates [12].

Reference [2] is a good review of available programming languages for quantum code, and how the whole system works in quantum computers. It's worth to note that this is a very active field, with constant changes. This article is from September 2017, but new tools are being developed every day. The article [33] has a good insight in software details, such as the optimization of code for error correction

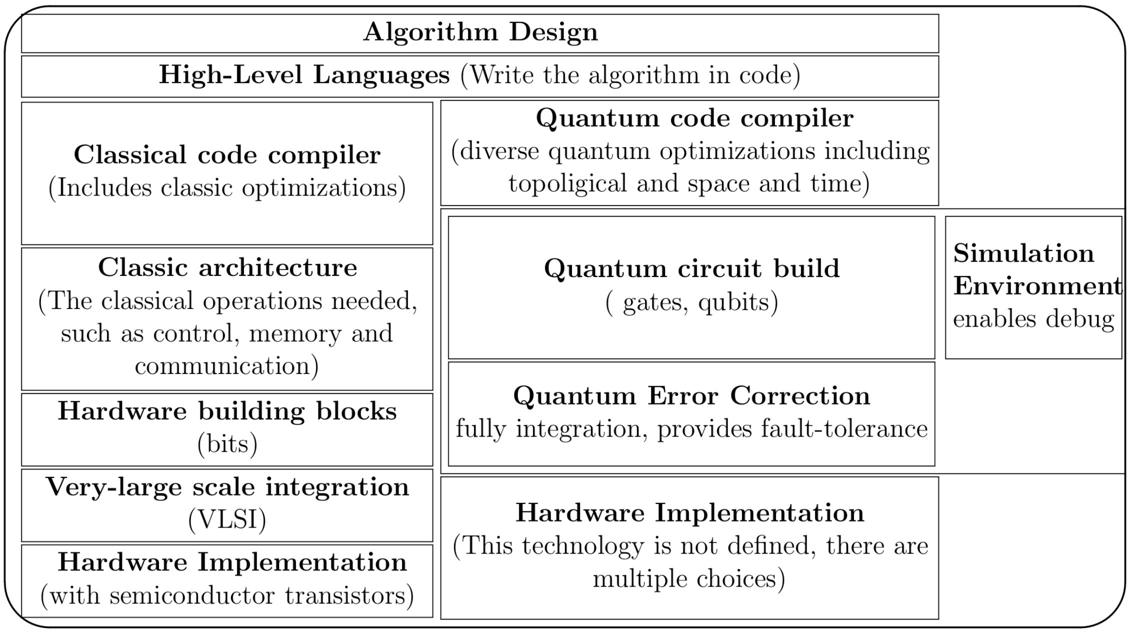

Algorithm Design
High-Level Languages (Write the algorithm in code)
Classical code compiler (Includes classic optimizations)
Quantum code compiler (diverse quantum optimizations including topoligical and space and time)
Classic architecture (The classical operations needed, such as control, memory and communication)
Quantum circuit build (gates, qubits)
Simulation Environment enables debug
Hardware building blocks (bits)
Quantum Error Correction fully integration, provides fault-tolerance
Very-large scale integration (VLSI)
Hardware Implementation (This technology is not defined, there are multiple choices)
Hardware Implementation (with semiconductor transistors)

Figure 3.1: Representation of the software to hardware flow, and the place of simulation environment as a viable option to explore some of the parts of hardware stack. Adapted from [2].

implementation and the topological optimization, which consists in reducing the volume of the circuits without affecting its function. The master thesis [34] is also a good review for the available languages at the time of its publication (2012), focusing on their performance.

The tool of choice for the circuit simulation in this thesis was the Microsoft library LIQ$Ui\,|\rangle$ [35]. It's a library based on the language F#, developed by the Microsoft research group, QuArC. When this decision was made, it was believed that it was the best choice available. However, shortly after this, the IBM quantum computer became available in the cloud with its own language. Later in the year, Microsoft announced a new language for quantum computers, the Q# [10]. So, it is likely that in the near future the LIQ$Ui\,|\rangle$ language, or better said, this library, will not be largely used. That does not affect the relevance of this work since it can be translated to a different programming language. It should be noted that despite the great development in recent years, quantum languages are still in their infancy.

3.2 The LIQ$Ui\,|\rangle$ Performance

3.2.1 A Brief Overview of the Language

LIQ$Ui\,|\rangle$ is a highly optimized language, which makes it a good option to run simulations. Since it's not completely open-source we don't have access to some of the explanations of optimizations. However, we have enough information to work with the language, make new functions and test them, so it's a pretty versatile option. All data types and functions are fully documented, we just don't have access to some of the code behind it.

It uses data types from quantum computation, which makes it easier to work with. For example, it has the data type of a classical *Bit*, and a *Qubit*, the fundamental unit of quantum computers. It has a set of most used gates already defined, and allows the definition of new ones, which is a very powerful option. It works with circuit modes, allowing the drawing, parallelization and optimization of the written algorithm. Since is based primarily in F#, it's a powerful language, because it allows high optimization regarding the mathematical functions, used mostly in academic domain and in industry.

LIQ$Ui\,|\rangle$ uses all the quantum properties already discussed, and has some useful tools that allows us to know the state of an entangled qubit without changing it. There is no need to measure the qubit if we only need to know its state, since the measurement causes a loss of information when the qubit becomes a bit.

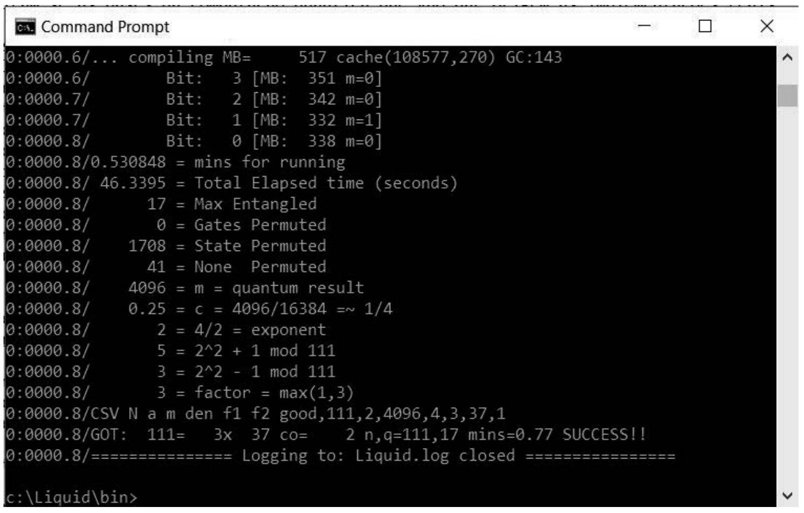
There are three simulators, see the manual for more details [36]:

Hamiltonian Simulator , which was not used in this work. It's used to simulate quantum systems (namwly in quantum chemistry applications).

Universal Simulator is the most versatile of the three, it enables a huge range of possibilities, allowing the addition of new gates and functions. It was used as the main simulator.

Stabilizer Simulator allows a great deal of optimization, using the so-called stabilizers. It can handle tens of thousands of qubits, since it groups them making a smaller circuit, and allows the usage of Error Correction Codes. This simulator was used in one part of this thesis work. We can adjust a circuit, built and tested in the Universal mode, to this one, optimizing it.

The code may be written in Visual Studio, and then run by LIQ$Ui\,|\rangle$ with multiple ways to test it. There is the option to run the code inside the Visual Studio, linked to the LIQ$Ui\,|\rangle$ library, or load the file into the LIQ$Ui\,|\rangle$ executable file. This runs the code with the tag [$< LQD >$] in the terminal (figure 3.2).



```
0:0000.6/... compiling MB=     517 cache(108577,270) GC:143
0:0000.6/        Bit:   3 [MB:  351 m=0]
0:0000.7/        Bit:   2 [MB:  342 m=0]
0:0000.7/        Bit:   1 [MB:  332 m=1]
0:0000.8/        Bit:   0 [MB:  338 m=0]
0:0000.8/0.530848 = mins for running
0:0000.8/ 46.3395 = Total Elapsed time (seconds)
0:0000.8/      17 = Max Entangled
0:0000.8/       0 = Gates Permuted
0:0000.8/    1708 = State Permuted
0:0000.8/      41 = None  Permuted
0:0000.8/    4096 = m = quantum result
0:0000.8/    0.25 = c = 4096/16384 =~ 1/4
0:0000.8/       2 = 4/2 = exponent
0:0000.8/       5 = 2^2 + 1 mod 111
0:0000.8/       3 = 2^2 - 1 mod 111
0:0000.8/       3 = factor = max(1,3)
0:0000.8/CSV N a m den f1 f2 good,111,2,4096,4,3,37,1
0:0000.8/GOT:  111=   3x  37 co=    2 n,q=111,17 mins=0.77 SUCCESS!!
0:0000.8/=============== Logging to: Liquid.log closed ================

c:\Liquid\bin>
```

Figure 3.2: Simulation example of one of the test modules. Here the Shor's Algorithm is being simulated to calculate the factors of the number 111 in the LIQ$Ui\,|\rangle$ terminal. Note that it gives the time of simulation in minutes.

The algorithms expressed by the code may be run in various ways. There is a test mode with some predefined algorithms to run. The script mode runs from a script file. More interesting is the function mode, it calls the functions of the code. Lastly, the circuit mode, which compiles the code into a circuit and then runs it. This mode has the advantage of using some circuit manipulation tools.

3.2.2 Testing the Shor's Algorithm

Having chosen the language, the first goal was to do a first "hands-on" test to evaluate the performance and to get used to the program. The test chosen was to run the Shor's algorithm, since it is a good example, as already discussed, as is part of the LIQ$Ui\,|\rangle$ library.

The LIQ$Ui\,|\rangle$ paper [35] talks about the performance of the system. It says that a 32 Gb of RAM on a computer should be able to run 30 qubits in the universal simulator, and it takes a petabyte of main memory to simulate 45 qubits. As already mentioned, quantum algorithm simulation on a classical computer needs a lot of resources, the growth rate is impressive. The biggest number ever factored with a quantum simulator used LIQUi $|\rangle$, a 14-bit number using 50 Gb of memory for 31 qubits, and took 30.1 days to process. That's a huge time to make this simulation, a classical computer can execute a classical algorithm to perform the same operation in a shorter computing time. Still, this is a good achievement for a quantum algorithm simulation, and it has a very optimized circuit running. The research group behind the language did various stages of optimization, rewriting packages. After the changes, a 13-bit number that would need three years to compute was done in four days. That's a huge improvement regarding simulation environments.

We also want to compare the results obtained with LIQUi $|\rangle$ to the output from a classical algorithm. So, we will run tests in LIQUi $|\rangle$ for factoring integers between 111 and 525 using Shor's algorithm, and, we will also run the classical algorithm Pollard's Rho.

The best classical algorithm for factoring integers is known to be the general number field sieve [37]. However, at the time it was easier to do a fast implementation of Pollard's Rho algorithm [38] in Java, which still is very optimized for classical computers, and perfectly suitable for this test.

3.3 Results

Figure (3.3) has the main results for the simulation of the Shor's algorithm. It has the results for the factorization of numbers from 111 to 525, with multiple runs in each one. The final plot represents the average of the results and its associated error. The fastest simulation corresponded to a minimum time registered of 0.63 min. The maximum time was a total of 76 min. Above this, the running time increases to non-practical values and it's not easy to run this kind of simulations unless we have a dedicated system, which is not the case.

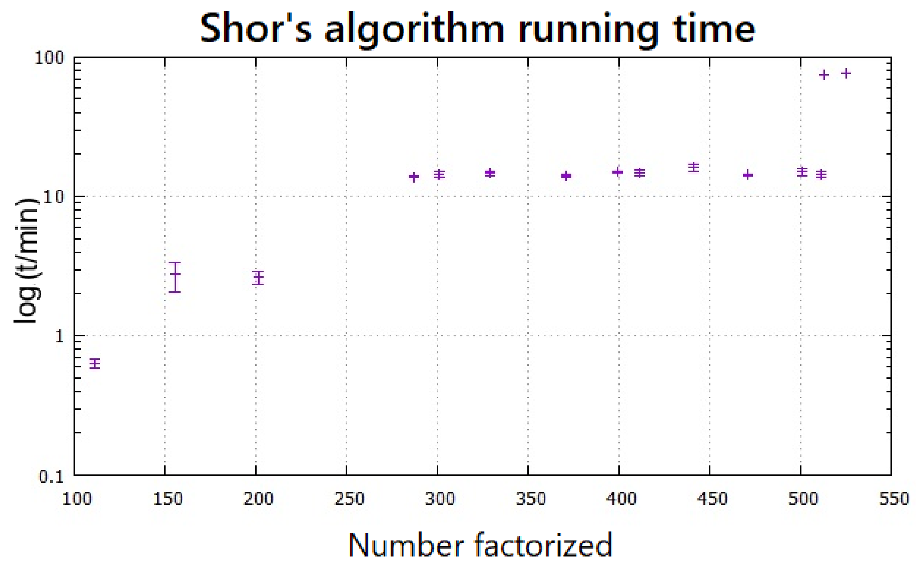


Figure 3.3: Results for factoring integer numbers up to 525 using Shor's algorithm, simulated in LIQUi |⟩ . The $y - axis$ is in logarithmic scale for better visualization and represents the computer running time t in min. The $x - axis$ represents the number to factorize.

The logarithmic scale on the $y - axis$ allows us to retrieve some information from this graph. The figure clearly shows three distinct regions: numbers lower than 220, will be relatively fast to compute; numbers between 220 and 513 reveal a step in the time duration, but not that much of an increment; for numbers higher than 513, however, the time needed to simulate the system increases a lot. The three regions correspond to three different time scales. If we analyse the data carefully, we conclude that the step increase, registered in the transition of regions, is related to an increase in the number of bits needed to represent the number being factorized. For example, the number 111 needs 7 bits to be represented, whereas the number 155 will need 8. It's important to note that in the same region the time variation is very small.

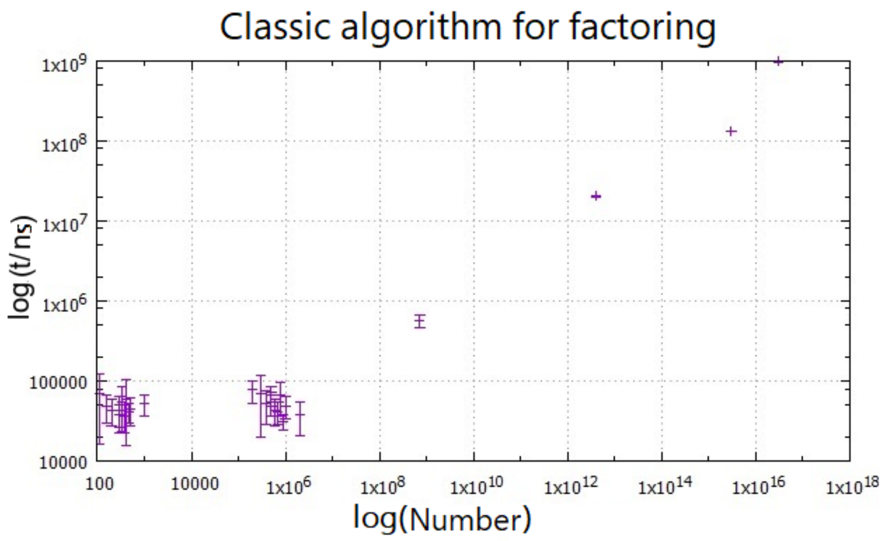
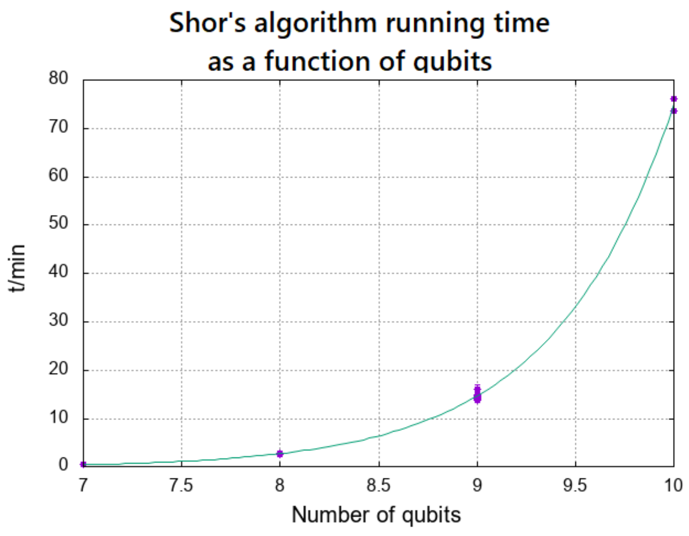


Figure 3.4: Factorization using a classic algorithm, the Pollard's Rho implemented in Java. Both x and $y - axis$ are in log scale. The number to factorize is in the $x - axis$, and the time t in ns is in the $y - axis$.

Figure 3.4 has the corresponding results of the classic algorithm Pollard's Rho, programmed and run in Java. Both of the axes are in logarithmic scale, for clarity. It's important to note that the unit of time here is the nanosecond, not the minute. Secondly, the numbers factorized went all the way up to values of the order of 10^9 to be able to see some difference in time. The maximum values were numbers in the order of 10^{16} and the corresponding time of execution was of the order of the second. This result exemplifies what we already discussed in chapter 2, classical algorithms have better results than the quantum ones when running on a classical computer. In the graph, with the logarithmic scale, we can clearly see that it has a linear progression, which corroborates the conclusion that the classical algorithm leads to an exponential behaviour with the size of the problem.

Since we concluded that the transition regions in figure (3.3) are due to an increment in the number of bits needed to represent the number, we should analyse the time growth as a function of the qubit growth for the Shor's algorithm. In figure (3.5) the running time of the algorithm is plotted as as function of the number of qubits. The graph at the left and the one at the right have the same data, the difference is that the one on the right has the $y - axis$ in a logarithmic scale of base 2.



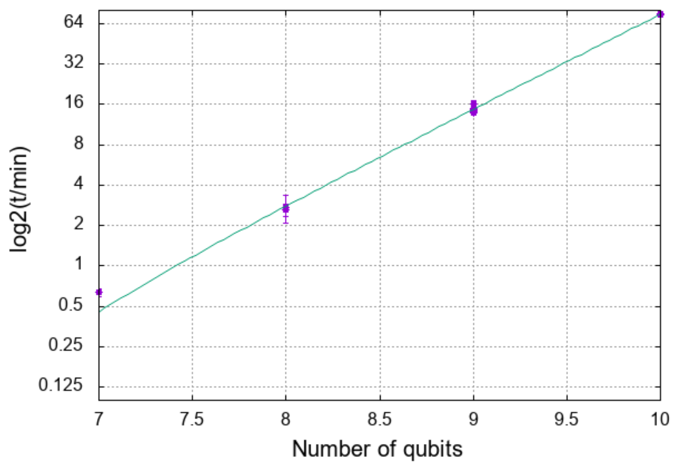


Figure 3.5: Representation of the LIQUi $|\rangle$ simulation, as a function of the number of qubits. the number of qubits is in the $x-axis$ whereas the computing time t in min is in the $y-axis$.The difference between the top and the bot plots is that in the latter the $y-axis$ is in $\log_2$ scale. The lines are a fitting to the data using the expression $y(x) = a * 2^{b*x}$.

$func:$	$y(x) = a * 2^{b*x}$	$\sigma_{y(x)}$
a =	$6.57e^{-6}$	$\pm 3e^{-6}$
b =	2.34	± 0.005

Table 3.1: Representation of the values of the fitting for the curve of the figure (3.5).

In figure (3.5) we can see that the data is well described by an exponential function of the type $y(x) = a * 2^{b*x}$. The exponential behaviour is evident in the bot plot since $\log_2(y(x)) = \log_2(a * 2^{b*x}) = \log_2(a) + b * x$ and the data $\log_2(y)$ shows a linear behaviour. The slope of the linear regression will be the factor $b = 2.34 \pm 0.005$. The corresponding fitting parameters are represented in table (3.1).

These test results show that the running time of the Shor's algorithm evolves with $\mathbf{O}(2^{2.34n})$ (n is the number of qubits), larger than the theoretical value that we discussed in section 2, $\mathbf{\Theta}((\log N)^3)$. However, we must note that the quantity plotted in the graphic is computing time. The theoretical value for the algorithm efficiency evaluates the number of operations, since the time for each operation should be a constant that varies with the computing device. Nevertheless, in order to obtain the time evolution we should only need to multiply by the time constant, and the growth behaviour should be similar. This would be valid for a quantum computer. However, in order to simulate an n qubit system in a classic computer, we need to store and process 2^n amplitudes of the states (we are working in the Hilbert space), which requires computing time and memory. Since the system grows exponentially, so does the time it takes to handle all the states [39]. The simulator needs to prepare all the states at the beginning of the test and then to perform the operations on them.

3.4 Important Insights

Comparing the results from this quantum simulation to a similar simulation presented in the master thesis of Johan Brandhorst-Satzkorn [34], we can say that the LIQ$Ui\,|\rangle$ language provides better results. He only could make factorizations up to the number 399, we had factorizations until 525. Regarding his maximum values, the times were between 31.8 min for the LibQuantum language and 74.6 min for the Quantum Computer Language. The number 399 is still on our second region of the figure (3.3). It has running times in the order of a little more than 10 min. The main reason for this difference is that LIQ$Ui\,|\rangle$ is a much more optimized language. However, the reason for which his tests reach only a maximum of 399 can be mostly explained by an older system with less memory. We can conclude that the choice of using this language was the right one at the time, since it is an environment more adequate for simulations, comparing with the previous ones.

One important idea explored in chapter 2, concerning quantum complexity was that the exponentially behaving classical algorithm (represented in the figure (3.4)) can be more efficient than the quantum one, as long as it only works with small numbers. The present results also show that the running time of a quantum algorithm has an exponential behaviour when simulated in a classic computer. In, addition, the values of time are some orders of magnitude larger than those obtained for the classical algorithm. The classical algorithm was capable of factoring numbers in the order of 10^{16} in only one second. Whereas the quantum algorithm needed 76 min to compute the factors of the number 525. This is an abysmal difference, which illustrates how much better a classical algorithm can perform in this scenario. It would be much more appropriate to compare the classical algorithm running in a classic computer, with a quantum algorithm running in a real quantum computer. This is explored in the article [40], where the results of a classic algorithm are confronted with the predictions of Shor's algorithm in a quantum computer. It predicts that a number of 375 binary digits, that takes hours in a classical computer to factorize, would be factorized in a matter of seconds in a quantum computer with a similar clock rate, exactly the reverse result that we get in our simulation.

Although the running time of our simulation has a behaviour of $\mathbf{O}(2^{2.34n})$, this simulation environment already gives solid results. It needs to simulate all the quantum system, but it does it in an optimized way, though we don't know the details of LIQ$Ui\,|\rangle$ optimization process. This test proves that quantum simulation is not easy to achieve since it needs to handle 2^n amplitudes of states of n qubits. That's one more reason why optimizing this process is such an important feature. Quantum systems, however, can handle this kind of information with ease.

Regarding the memory usage, for numbers larger than 513, the algorithm needs 23 qubits and this requires 2 Gb of RAM from the system, where the CPU is being used approximately at 100% during all the simulation, taking more than 70 mins. This may be a problem in terms of temperature of the computer device in a long-term use. Without a dedicated system with an adequate cooling system, the simulation has big limitations. All the simulations of this thesis were made with the personal laptop mentioned in the appendix C. All of them, except, of course, the ones from the IBM cloud.

4

Modular Addition Using Quantum Fourier Transforms

4.1 Modular Arithmetic

Modular arithmetic is very useful and vastly used in computation as the remainders of divisions. Dividing the integer x by the integer y, we get the quotient Q and the remainder R, where Q and R are both integers.

$$\frac{x}{y} = Q \text{ remainder } R \tag{4.1}$$

That is:

$$x = Q \times y + R \tag{4.2}$$

If our interest is in the remainder of the division, we deal with the arithmetic of remainders and we use the modulo operator (abbreviated as "*mod*"), defined as

$$x \bmod y = R \tag{4.3}$$

For example,

$$2 \bmod 3 = 5 \bmod 3 = 8 \bmod 3 = 11 \bmod 3 = 2$$

since 2, 5, 8 and 11 when divided by 3 all have a remainder of 2. For the number 5, for example, $5/3 = 1.6(6)$. If we use the integer part of the result, 1, we have $1 * 3 = 3$, and $5 - 3 = 2 = R$ which is the result of the $\bmod$ operation.

Modular arithmetics is sometimes called the math of clocks. A clock with the 12th number replaced by 0 gives us the $x \bmod 12$ operation. For example: $1 \bmod 12 = 1$, $10 \bmod 12 = 10$, $12 \bmod 12 = 0$. After a complete turn, the clock is reset. So, if $x = 13$, we count until 13 on the clock, which gives $R = 1$, that is $13 \bmod 12 = 1$.

We can do most operations using modular mathematics, as we do in normal mathematics. In this work we are mainly interested in the simple modular addition.

4.1.1 Modular Addition

Modular addition is the extension of normal addition:

$$(x + y) \text{ mod } C = (x \text{ mod } C + y \text{ mod } C) \text{ mod } C \tag{4.4}$$

where x, y and C are integers. For example, for x=2, y=3 and C=2 we have:

$$(2 + 3) \text{ mod } \ C = 5 \text{ mod } 2 = 1$$

where we used the left side of the equation (4.4). Using the right side of the same equation, we get

$$(2 \text{ mod } 2 + 3 \text{ mod } 2) \text{ mod } 2 = (0 + 1) \text{ mod } 2 = 1$$

This result exemplifies the rule that the remainder of the addition of two integers equals the remainder of the addition of the individual remainders.

The modular addition is particularly useful if the number C is related to x and y. If we have the relation of $C = 2^n$, n being the number of bits to represent both x and y. The modular addition becomes:

$$(x + y) \text{ mod } 2^n \tag{4.5}$$

Now, let's see the example for $x = 2$ and $y = 2$, being $n = 2$ bits. The modular addition is $(2 + 2) \text{ mod } 2^2 = 4 \text{ mod } 4 = 0$, where 4 is a 3 bit number. However, if we make $x = 1$ and $y = 2$ we get $(1 + 2) mod\ 4 = 3$ which is the same result as the normal addition. So, as long as the number of bits needed to represent $(x + y)$ is the same as n, the modular addition gives the same result as the normal addition. However, if the number of bits exceeds n, the result of the modular operation is the remainder. In other words, it is what is needed to add to C in order to get the normal addition result. For example, consider the case $x = 6$ and $y = 7$. We have $n = 3$, and the operation becomes $(6 + 7) \text{ mod } 8 = 5$. Since we know that $6 + 7 = 13$, we can see that the remainder plus C gives the result of 13 $(5 + 8 = 13)$.

In a common addition in computation, the numbers will be represented in binary, when the result of the operation needs more bits than the original numbers, the addition algorithm needs to carry the information of extra bits. Modular addition is used to help solving this need of extra bits.

Arithmetic operations are very important for computation systems. After all, the purpose of some of the first computers was to help in the process of mathematical operations, so it makes sense to study the first basic ways to do the same operations in quantum computers. Since quantum computers evolved from the classical ones, some of the first algorithms were created with the same basic principle as their classical equivalents. However, these algorithm had some modifications in order to be fully reversible, a requirement of quantum computers. These algorithms use at least $3n$ qubits.

Quantum Fourier Transform (QFT) can be used to improve the efficiency of arithmetic operations, namely the addition of two or more numbers, which can be converted in a modular operation. The use of QFT in an addition algorithm was first introduced by Draper in 2000 [20], it was a huge improvement and boosted the use of QFT in this kind of operations. The addition with QFT used in this thesis follows the Draper's QFT adder. However, there are some variations to this adder, using carry bits[1], introduced by Perez[11], which is also used in chapter 6.

4.2 Algorithm for Addition with QFT

The basic principle behind the QFT addition of two numbers is to work in the phase domain, first, by encoding the qubits of one number with QFT and then performing a group of phase rotations, that are controlled by the bits of the other number in the addition. The general circuit for this algorithm is represented in the figure (4.1).

In more detail, having the numbers x and y with n bits, we can use the notation discussed in the equations (2.13) and (2.12) to deal with binary numbers. x is represented by the bits x_1,$x_2 \dots x_n$, and y by y_1,$y_2 \dots y_n$. So, using quantum bits, we can represent x and y states by:

$$|x\rangle = |x_1\rangle \otimes |x_2\rangle \otimes \cdots \otimes |x_n\rangle$$

$$|y\rangle = |y_1\rangle \otimes |y_2\rangle \otimes \cdots \otimes |y_n\rangle$$

[1]A carry bit is the information generated in the addition that is transferred to a more significant digit. For example: to add $13 + 9$, first we add $9 + 3 = 12$, the digit 1 is transferred, as a carry bit, to the most significant one $1 + 1 = 2$, making the addition $13 + 9 = 22$.

The QFT will only act on the state $|x\rangle$, while $|y\rangle$ will remain unchanged. So QFT does the transformation:

$$QFT\,|x\rangle = |\phi(x)\rangle = \frac{1}{\sqrt{N}}\sum_{k=0}^{N-1} \mathrm{e}^{\frac{i2\pi xk}{N}}\,|k\rangle$$

with $N = 2^n$. At this point the x_i bits are in the phase domain, so the rotations controlled by the bits of y will perform the addition of their bits into the phase of $|\phi(x)\rangle$, giving $|\phi(x+y)\rangle$. After this point, the addition is already performed, but the information is in the phase domain. We need to apply an inverse QFT (IQFT), to perform the operation $IQFT\,|\phi(x+y)\rangle = |x+y\rangle$. Since y is a bit number and stays unchanged, this circuit shows a way to add classical information to a quantum state.

The controlled rotations in the circuit will perform an operation represented by:

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{i2\pi}{2^k}} \end{bmatrix} \tag{4.6}$$

In the section of QFT, (2.2.5.1), we have studied the effect of a QFT circuit. Now, we can use the results presented in that section in order to understand this addition algorithm. We can start with the states given by equations (2.15),(2.16) and (2.17), in the example of a 3 qubit QFT. The next step is to apply the necessary rotations for the modular addition operation. As in the QFT section, we start by considering the state which has fewer operations, that is, the output of $|x_1\rangle$ state after a QFT operation, which will be: :

$$|\phi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{\pi\, i\,(x_1)}\,|1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{\frac{2\pi\, i\, x_1}{2}}\,|1\rangle)$$

A rotation $R_1 = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi} \end{bmatrix}$ is applied to this state, controlled by $|y_1\rangle$, which is a bit number (1 or 0). The result of this operation can be written as:

$$|\phi(x+y)_1\rangle = R_1\,|\phi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{\pi\, i\,(x_1+y_1)}\,|1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{\frac{2\pi\, i\,(x_1+y_1)}{2}}\,|1\rangle)$$

Following the same logic, after the QFT, the state $|x_3\rangle$ will be in the state:

$$|\phi_3\rangle = \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{2\pi\, i(\frac{x_3}{2}+\frac{x_2}{4}+\frac{x_1}{8})}\,|1\rangle)$$

The modular addition operation is performed using three controlled rotations, R_1, R_2 and R_3, controlled by the bits $|y_3\rangle$, $|y_2\rangle$ and $|y_1\rangle$ respectively. The final state, after the controlled rotations, will be:

$$|\phi(x+y)_3\rangle = \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{2\pi\, i(\frac{x_3}{2}+\frac{x_2}{4}+\frac{x_1}{8}+\frac{y_3}{2}+\frac{y_2}{4}+\frac{y_1}{8})}\,|1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{2\pi\, i\,(0.x_3\,x_2\,x_1+0.y_3\,y_2\,y_1)}\,|1\rangle)$$

This example can be generalized to two numbers of n qubits, x and y, giving the state:

$$\begin{aligned} |\phi(x+y)_n\rangle &= \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{2\pi\, i(\frac{x_n}{2^k}+\frac{x_{n-1}}{2^{k+1}}+\dots\frac{x_1}{2^n}+\frac{y_n}{2^k}+\frac{y_{n-1}}{2^{k+1}}+\dots\frac{y_1}{2^n})}\,|1\rangle) \\ &= \frac{1}{\sqrt{2}}(|0\rangle + \mathrm{e}^{2\pi\, i\,(0.x_n\,x_{n-1}\dots\,x_1+0.y_n\,y_{n-1}\,\dots y_1)}\,|1\rangle) \end{aligned} \tag{4.7}$$

The second line in the previous equation is the representation using the binary fraction, defined in equation (2.14). The state $|\phi(x+y)_n\rangle$ has the information on $x+y$ in the phase domain. Applying the IQFT, the state $|\phi(x+y)_n\rangle$ is converted into the final state $|x+y\rangle$, which has the information on $(x+y)$ mod 2^n. The circuit for modular addition is shown in figure (4.1).

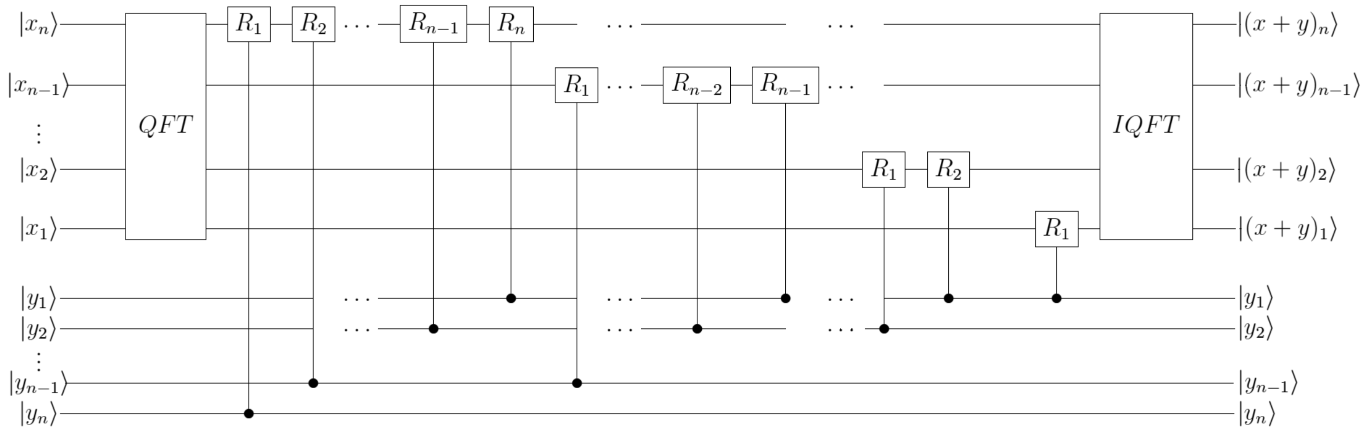


Figure 4.1: Circuit representation for the modular addition $(x+y)$ mod 2^n of two n bit numbers, x and y, using QFT.

This is the basic structure of the algorithm which is implemented in this work, as discussed in the next sections. It's the original circuit version presented in the article from Draper [20]. The output coincides with normal addition if $x+y$ has the same number of bits of x and y, that is n bits. When the solution has more bits than n, the output is the modular addition. It's important to stress that this algorithm does not need any intermediate carry bits, which is in great contrast with the classical algorithm where all the intermediate carry bits have to be transported and used. The first quantum addition algorithms use $3n$ qubits as input to perform the operation for two n qubit numbers, since it needs to manipulate intermediate carry bits. The Draper algorithm uses only $2n$ qubits, i.e., just the two numbers to be added, each with n qubits.

The addition algorithm using QFT requires that the numbers to be added are in binary representation. As discussed in the section (2.3.2), the theoretical number of gates needed for the QFT is obtained summing the number of gates in all the lines, which in this case corresponds to the series $1+2+\cdots+n = n(n+1)/2 = n^2/2+n/2$. The addition algorithm takes a QFT, and IQFT, which means doubling the number of gates. The addition part of the circuit follows the same reasoning, requiring also $n/2(1+n)$ gates. This makes the total number of gates for this algorithm $3n/2(1+n) = 3/2n^2 + 3/2n$. Therefore, this algorithm is of the form $\mathbf{O}(n^2)$.

The elegance of this algorithm comes from the absence of carry bits. However, it has the limitation discussed before that the result is only a modular addition in cases where $x+y$ has a number of bits larger than n. There is a modification that can be made to the circuit in order to provide a normal addition even in those cases, which was suggested by Perez, [11]. Adding only a line on top of the circuit, with the state $|0\rangle$, the final state will be $|(x+y)_{n+1}\rangle$, which is the carry bit for the most significant digit. This allows the normal addition of two numbers in all cases. This extra line is on top of the circuit, the rest is exactly the same, as represented in the figure (4.2). This circuit only needs one more entry qubit, in a total of $2n+1$. It needs the same number of gates as before, plus n, that is $3/2n^2 + 3/2n + n = 3/2n^2 + 5/2n$ number of gates.

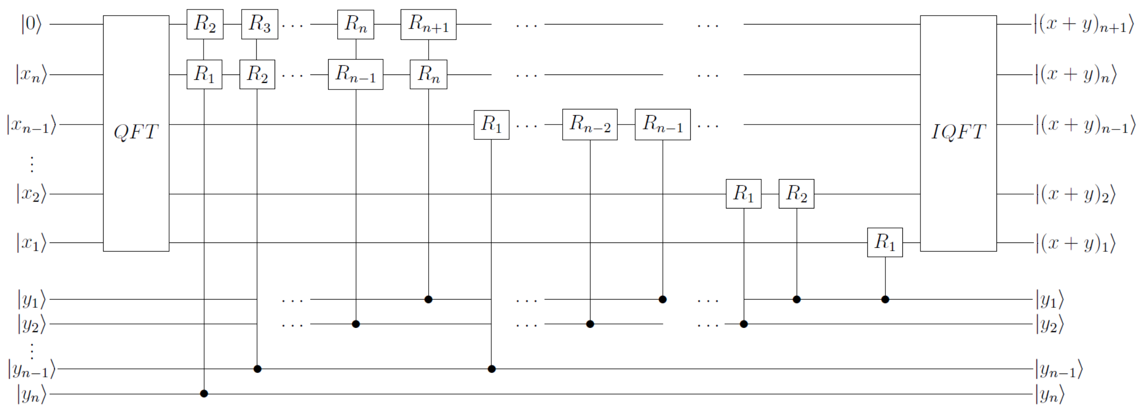


Figure 4.2: Representation of a circuit to do an addition of two numbers x and y with n qubits each. The first state is the carry bit and starts as $|0\rangle$.

4.2.1 LIQUiD implementation

Following the Draper addition circuit, the code was implemented in the LIQ$Ui\,|\rangle$ environment. The first thing to do was to implement the QFT and IQFT functions, not originally available by LIQ$Ui\,|\rangle$.

In this particular case, we used the universal simulation engine of LIQ$Ui\,|\rangle$, the most versatile one, which allows the implementation of various functions. Inside this simulation engine, there are a few different ways to work, hence the versatility. However, this function mode its not the best for optimizing the code, so in order to use the optimization options, we will work both with the function mode and the circuit mode, exploring the best options in both.

4.2.1.1 Function Mode

This is the mode of working with LIQ$Ui\,|\rangle$ which resembles a classical environment, where we implement functions which are run consecutively. It has a wide range of gates available and can be expanded.

In this particular case, we will work with QFT and IQFT. These operations are not originally available in LIQ$Ui\,|\rangle$, so we will create them as functions, but they can also be saved as gate operations. Saving them as gates may be done using the matrix of the operation, or the function itself. We decided to implement directly the function since we will work extensively in function mode. Saving as a gate does not seem to influence the speed of the computation. However, it can be very useful for future use with other programs, since it can be saved to be accessed later. The code implemented to do a QFT is as follows:

```
let qft (qs :Qubits)=
        //let_ =k.Single()
        //let qs=k.Qubits
        let n=qs.Length
        let mutable z=0
        for q in qs do
                H [q]
                let mutable qi=z+1
                for i in 2 .. (n-z) do
                        Cgate (R i) [qs.[qi];q]
                        qi<-qi+1
                z <-z+1
        z<-0
        //let mutable z=0
        //for q in qs do
        //   while z<n/2 do
        //       let g=(n-z)-1
        //       //show "test + z:%i" z
        //       SWAP [q;qs.[g]]
        //       z<-z+1

```

The Hadamard gate is applied to all the qubits, followed by the controlled rotation gates. The second half of the code is deactivated because it was not used in this scenario. It represents the SWAP part of the QFT, which is not always needed. The IQFT function is based on the same principle of the QFT, but since it needs to invert the order of the gates, it is a little trickier to implement.

Since LIQ$Ui\,|\rangle$ is based on F#, it is an indented language. It does not use curly brackets, which can facilitate mistakes. The gates that are predefined are represented by their name, for example, the Hadamard gate is H. Variables are not mutable by nature, so they need to be declared to be mutable. $Cgate$ is the operation to control gates. It must be used with caution to avoid errors, because of the parameters it needs in order to work. $Stopwatch()$ function allows us to get the running time of the simulation. LIQ$Ui\,|\rangle$ stores qubits in kets and the command $Dump(showInd)$ gives the state of the qubits even if they are entangled, without changing its state.

The simulation engine in this mode is useful but it's not the most optimal option. In order to improve with optimization, we run the circuit mode.

4.2.1.2 Circuit Mode

The circuit mode transforms the functions previously created into a circuit. Using this mode the circuit can be printed into an image or LaTeX file, and we can do circuit optimization with some simple functions. It's an independent mode, which means it doesn't need the function mode to run in order to be able to run the circuit. In fact, the simulation can be done directly in circuit mode. The computing times when running circuit mode directly, or running the function mode first will be discussed in the next section.

Circuit mode needs to be compiled, by the function *Circuit.Compile*. Since a circuit can be broken into parts, our modular addition has one QFT, one adder and one IQFT. Each function can be compiled individually and then make a bigger circuit. There is an option to compile all the functions at once. However, this only works in some circumstances, depending on the code cycles and variables. For example, *RenderHT*() will draw the circuit into an HTML file. The figure (4.3) is the representation of the QFT given by the program, generated by the implementation into circuit mode of the code discussed above.

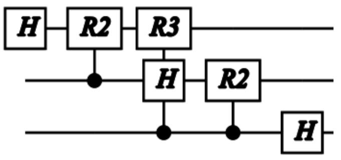


Figure 4.3: The circuit for a QFT of 3 qubits given by the LIQ$Ui\,|\rangle$ circuit mode with the qft function.

The $Fold()$ command manages the gates to display in the circuit image, since some can be grouped. $GrowGates()$ is the function that optimizes the circuit already compiled. This is a great feature using an algorithm to group gates and wires. It aggregates unitary gates into larger ones, making a larger matrix instead of many matrices, which makes it faster to run. We can see one example in figure (4.4), where it shows that it grouped all the 7 gates required to run the circuit on the left into 1 bigger gate with a internal code and 3 wires, on the right.

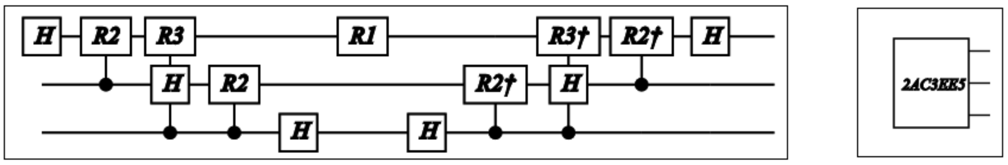


Figure 4.4: Representation of the circuit with $x = 2$ and $y = 2$, a 3 qubit circuit. On the right is the optimization of the circuit done with the $GrowGates()$ function. y is not displayed.

Some of the most important parts of the code will be on appendix (A) and are enough to understand how the circuit simulation was done. The total code is too long and therefore is not shown.

Figure (4.5) represents an example where we have two 3 qubit numbers being added. In this case all bits of y are $|1\rangle$, that is, $|y\rangle = |7\rangle = |1\rangle\,|1\rangle\,|1\rangle$. The figure is the result of the code done to implement the algorithm. It's important to mention that the number y is not represented in the circuit. In this circuit, all the controlled gates are present. If the bits from y are $|1\rangle$, then the rotation operations controlled by y must appear in the circuit. But if they are $|0\rangle$, the program will not represent them, because they have no effect in the circuit. However, controlled rotations belonging to

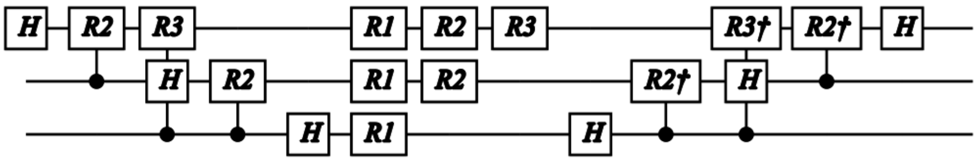


Figure 4.5: Representation of the 3 qubit circuit for modular addition. The bits corresponding to the number $y = 7$ ($y_1 = y_2 = y_3 = 1$) are not represented.

the QFT and IQFT part of the circuit are controlled by x, which is an entry bit. The code will implement the controlled gates even if they are controlled by $x = 0$, this means still appear in the circuit, even though they have no effect. That's something that can be optimized. So, we improved the algorithm code for QFT and IQFT by considering controlled rotations on x only if x is not $|0\rangle$ at that instance. This saves time and resources. An example can be seen, in the appendix (A.3), to better understand the improvement made, comparing the figure (A.3), the normal circuit, to the figure (A.4), the improved circuit. The last one has much less gates, being much faster to compute.

4.2.2 Results

In order to understand how the algorithm and the available optimization options work, we have run simulations with the code described, in its various forms. We simulated the different configurations multiple times and recorded the time that took to compute, the number of gates and the computer resources, namely the memory and CPU usage.

All the tests simulate the algorithm in function mode without optimizations. Then we run the circuit mode with optimizations, and without them. After that we can run the algorithm improved by us, optimizing the number of gates, this can be run in function and circuit modes, the last one with an option for optimizations. This is a sort of doubly optimized simulation.

There are two main sets of tests that were made, the first set was regarding the gate evolution and the second regarding the time evolution. The first set had the intention to see how the number of gates grew in order to compare it to the theoretical values. So, there are two main options to test: the case where $y = 2^{n-1}$, will give the results for the minimum gate number (figure (4.6)); the case with $y = Ones$, that is when all the bits from y are 1 , which gives the results for a maximum number of gates (figure 4.7). In the case where all the y bits are *one*, all the controlled rotations will have an effect on the circuit.

Figure (4.6) shows the case $y = 2^{n-1}$, using simple function mode, optimizations, improved circuit and respective optimizations. For example, since $y = 2^{n-1}$, $n = 3$ implies $y = 4$, which is $|y\rangle = |100\rangle$ in binary form. From the circuit in figure (4.1) we can conclude that it will only have one controlled rotation, R_1, with an effective role. This is the minimum number of gates possible. The expected number of gates will then be the gates from QFT plus IQFT plus one gate, which gives a total of $n(1 + n) + 1 = n + n^2 + 1$. Using our algorithm optimization in QFT and IQFT, a gate is not included if no effect is expected in the circuit. The expected minimum value of gates in QFT and IQFT correspond to the case when only Hadamard gates apply, so we should only have n gates for QFT/IQFT and the total minimum should be $2n + 1$ gates.

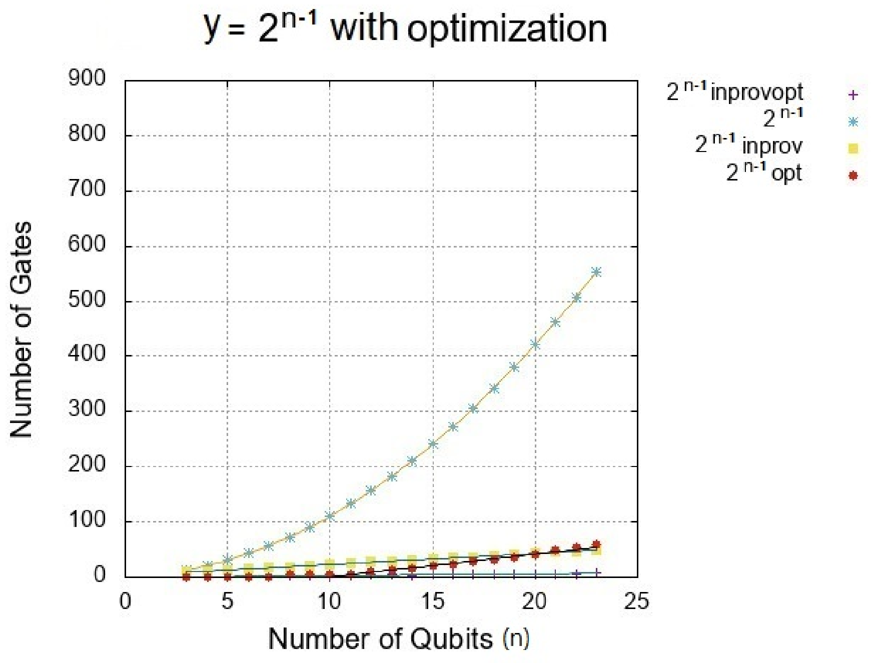


Figure 4.6: Graphical representation of the number of gates growth with the number of qubits for a circuit with y evolving with 2^{n-1}. 2^{n-1} represents the normal circuit for $y = 2^{n-1}$, and $2^{n-1}opt$ represents the LIQUi $|\rangle$ optimization of this circuit. $2^{n-1}inprov$ is the improvement done to the circuit. $2^{n-1}inprovopt$ represents the LIQUi $|\rangle$ optimized version of the improved circuit.

Fit values of the graph (4.6) are represented in the table (4.1). From this table we can see that the fit expression for the improved version of the circuit, $h(n)$, is a linear expression of the form $2n + 3$, making the algorithm of the form $\mathbf{\Omega}(n)$. Comparing it with the theoretical value, we see that the obtained results have two extra gates. This is justified by the fact that this test was run with $x = 2^{n-2}$, which

2^{n-1}	$g(n) = a1n^2 + b1n + c1$	$\sigma_{g(n)}$	2^{n-1}_{opt}	$y(n) = an + b$	$\sigma_{y(n)}$
a1=	1	$\pm 4e^{-017}$	a=	4.11	± 0.05
b1=	1	$\pm 1e^{-015}$	b=	-40.44	± 0.80
c1=	1	$\pm 5e^{-015}$			
2^{n-1}_{inprov}	$h(n) = tn + f$		$2^{n-1}_{inprovopt}$	$yy(n) = aa * n + bb$	$\sigma_{yy(n)}$
t=	2		aa=	0.31	± 0.0091
f=	3		bb=	-0.35	± 0.13

Table 4.1: Expressions of the fit for the graph 4.6

means that the QFT will get an extra gate, the controlled R_2, so the IQFT will have another gate too. This makes for the extra two gates, which means we can verify the theoretical value with these results.

The case of $2^{n-1}opt$ gives a linear fit, as the improved case, but the slope is twice as big (4.11). Considering the optimization to our improvement, the linear expression will be affected by a factor of 10, the slope becomes 0.31, which is a significant improvement. Note that for the optimization tests, the number of gates will be one if n is smaller than 8. The fit to the graphics regarding the optimized runs was only made for the part where it grows, that's why we can justify the value -40.44 of the $2^{n-1}opt$. This fit expression is only valid for $n \geq 8$, if $n < 8$, then $y(n) = 1$. For the optimization of the improvement, the fit is only valid if $n \geq 9$, if $n < 9$ then $yy(n) = 1$.

The graph of the figure (4.7) has the representation of the case of $x = y = Ones$, which gives the maximum number of gates ($n^2/2 + n/2$), discussed above in the theoretical discussion of this section. The case with only $y = Ones$ is intended to illustrate that in this case our improvement strategy has no effect on the number of gates, but LIQ$Ui\,|\rangle$ optimization is effective.

Table (4.2) has the expressions and parameters of the fit of the figure (4.7). From the expression for the case $x1y1$, we can see that the parameters of $1.5n^2 + 1.5 * n$ are exactly what we were looking for, the parameter C can be considered as zero since it is too small. The optimization fit gives a linear expression $t(n) = 2.86 * n - 17.63$. If we compare it with the previous case, with the improvement version, we see that the slope 2.86 is close to 2, which means that the optimization for a maximum number of gates case gave a similar result to an improvement version for a minimum case. However, this result reflects only the LIQ$Ui\,|\rangle$ strategy to aggregate gate operations, we analyse the impact of this strategy on computation time as well.

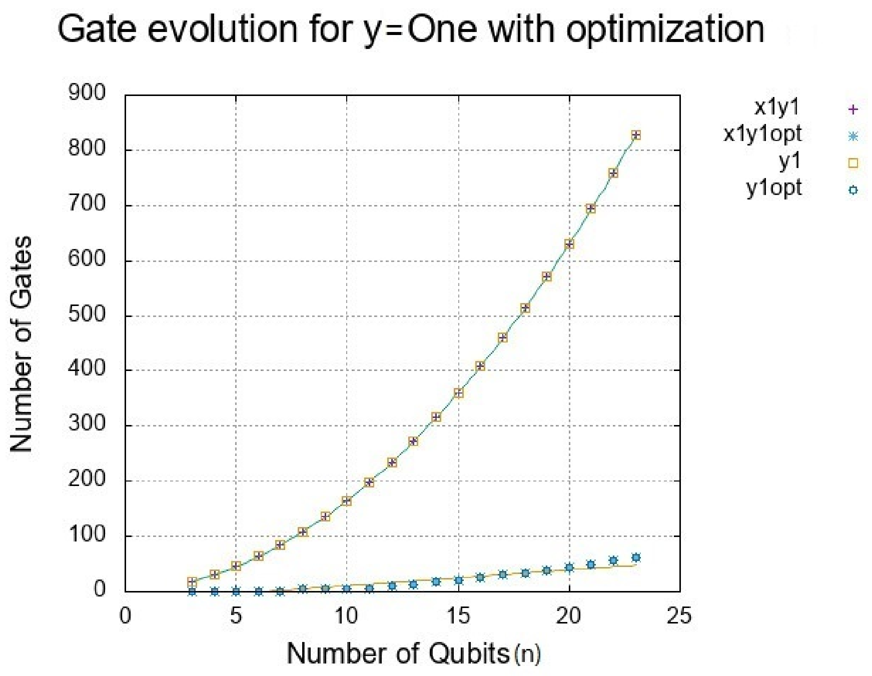


Figure 4.7: Representation of the gate evolution with the number of qubits for the case where y is always $Ones$, that is, in terms of binary numbers always represented by an array of bits of 1. $x1y1$ is the case for both x and y being Ones, for $y1$ only y is $Ones$, they represent the same number of gates at all times; the *opt* versions are the Liquid optimized versions of the previous cases.

$x1y1$	$y(n) = a * n^2 + b * n + c$	$\sigma_{y(n)}$	$x1y1_{opt}$	$t(n) = r * n + p$	$\sigma_{t(n)}$
a=	1.5	$\pm 2.7e - 015$	r=	2.86	± 0.069
b=	1.5	$\pm 3.8e - 014$	p=	-17.63	± 0.94
c=	7.44e-012	$\pm 3.15e - 014$			

Table 4.2: Fitting functions for graph 4.7

It is important to note that when a algorithm uses QFT all the corresponding gates are displayed and cannot be discarded because in general the control is done by qubits, not bits like in a addition algorithm. Therefore, we usually have a considerable number of gates as the qubit number increases. That's why our improvement to the addition circuit is important, reducing the gate number. Figure (4.8) summarizes the major cases for gate growth we analysed before, considering only our gate reduction strategy for improvement (without system optimizations). This graph allows us to verify, once again, the effect of the improvement in a best-case scenario, confronted with the maximum and minimum cases. Between this two latter cases, there is also a noticeable difference in the growth rate, since the linear and quadratic parameters are 1.5 for maximum case, while the corresponding parameters are 1 for the minimum case.

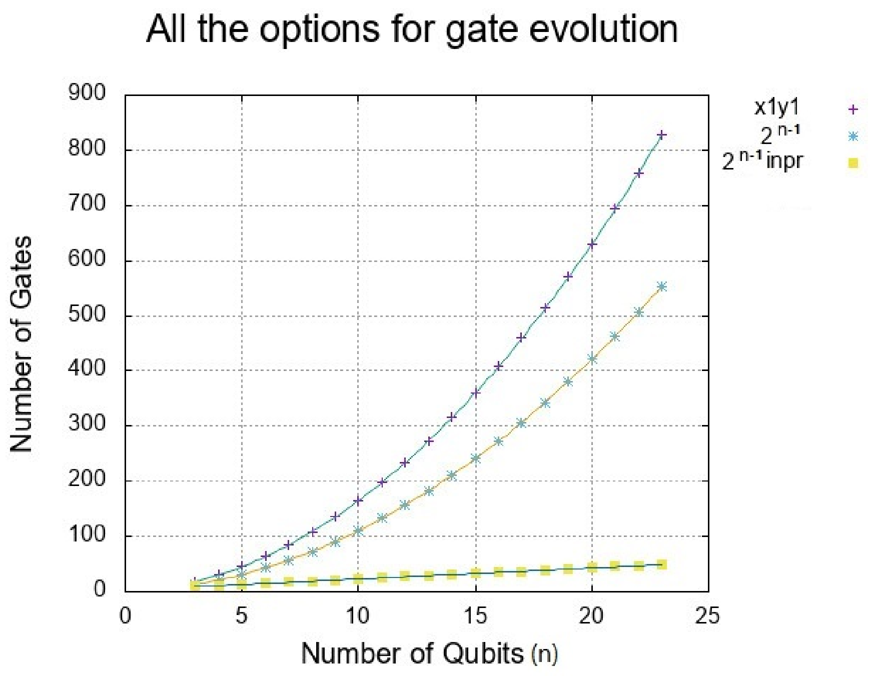


Figure 4.8: This represents the true gate evolution with the qubit number for 3 different cases without optimization. $x1y1$ is the already discussed case of x and y being $Ones$. 2^{n-1} y evolves with 2^{n-1}. And the $2^{n}inpr$ is the improved version of the previous one. This last one is an specific case with outstanding results.

Now let's see the results of the second main set of tests, regarding the time evolution as a function of the number of qubits. The tests were made considering the same cases as before, with the same kind of optimizations.

There are four graphics for the representation of all possible cases, concerning the time evolution. Since this is a simulation on a classical computer, we expect in all cases an exponential behaviour of the form $a * b^n$ for the running time, as discussed in chapter 3.

Figure (4.9) shows the running time as a function of the number of qubits for the case $y = 2^{n-1}$, using the three different modes provided by LIQ$Ui\,|\rangle$. It shows that the optimized circuit mode gives a huge advantage in running time when compared with function and circuit mode. Figure (4.10), shows the running time as a function of the number of qubits using our improved circuit strategy when $y = 2^{n-1}$. Comparing the two graphics, we can see that the original function without improvements, has a maximum order of 350000 ms for the same number of qubits. The version with improvements has an order of 35000 ms. The difference between them is a factor of 10, which is a great gain. Comparing the optimized version in both cases we can verify that the results go from the order of 50000 ms to 15000 ms. Comparing the normal case from the first graphic, with the optimized improved version of the second graph, we see a reduction in the running time of 95.7%, which is an abysmal difference.

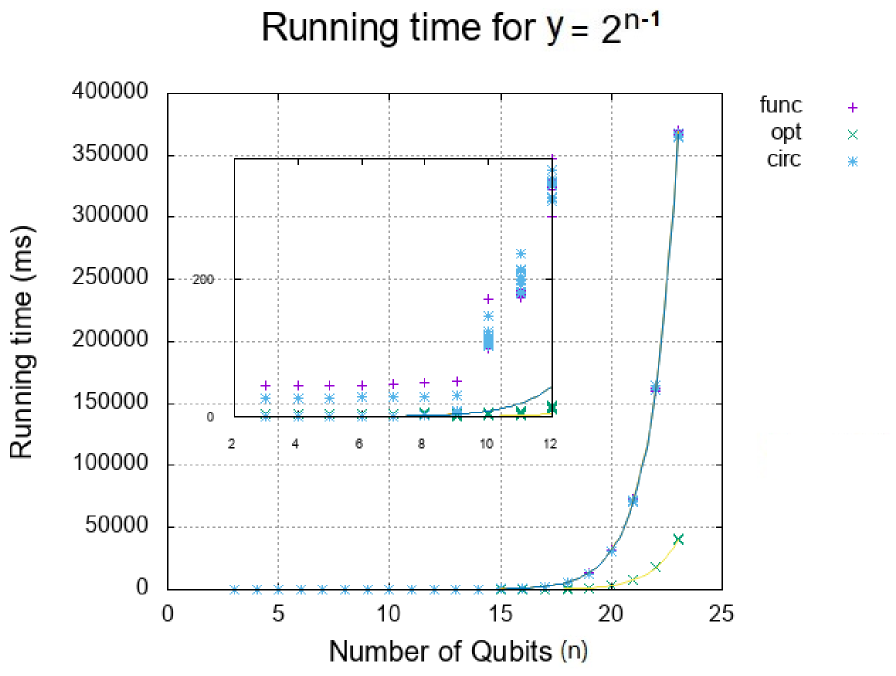


Figure 4.9: The running time of the case with $y = 2^{n-1}$ with n being the number of qubits. It has represented the 3 ways of running it: the function mode($func$), the circuit mode ($circ$) and the optimized circuit mode (opt). It has a close up of the initial qubits in order to see the difference between the function and circuit mode.

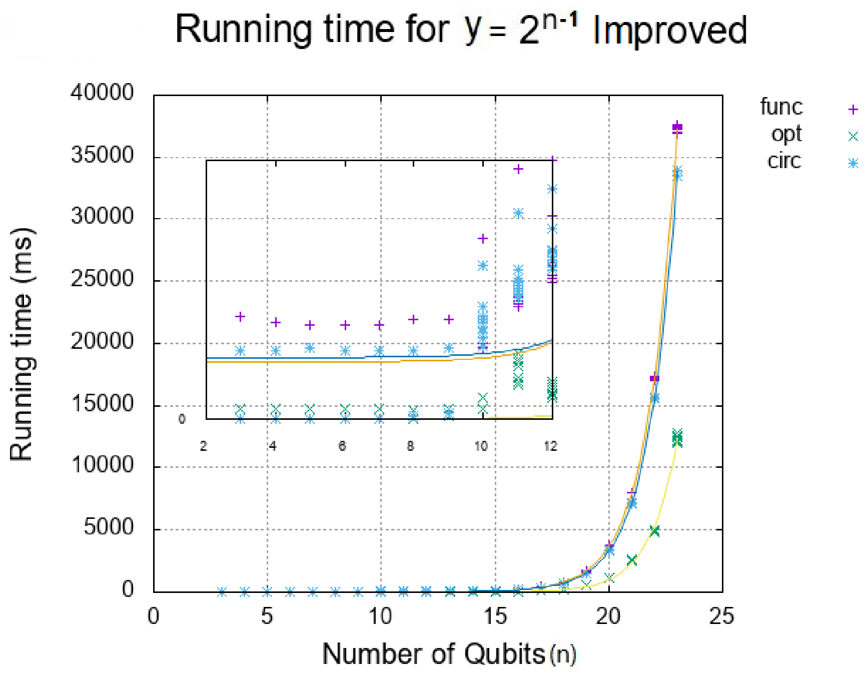


Figure 4.10: This figure has the graph of the running time for $y = 2^{n-1}$, with the improved versions of the same graph as represented in figure (4.9).

$func$	$y(n) = a * b^n$	$\sigma_{y(n)}$
a =	0.00235	±0.00004
b =	2.271	±0.002
$circ$	$y2(n) = a2 * b2^n$	$\sigma_{y2(n)}$
a2 =	0.00232	±0.00004
b2 =	2.272	±0.002
opt	$y1(n) = a1 * b1^n$	$\sigma_{y1(n)}$
a1=	0.00019	±0.000006
b1=	2.304	±0.003

Table 4.3: Fitting functions of figure (4.9).

$func$	$y(n) = a * b^x + c$	$\sigma_{y(n)}$
a =	0.00071	±0.00001
b =	2.166	±0.002
c =	21.78	±4.664
$circ$	$y2(n) = a2 * b2^n + c2$	$\sigma_{y2(n)}$
a2 =	0.00061	$\pm 1e^{-5}$
b2 =	2.171	±0.002
c2 =	23.56	±3.30
opt	$y1(n) = a1 * b1^n$	$\sigma_{y1(n)}$
a1=	$5.57e^{-5}$	$\pm 7.85e^{-6}$
b1=	2.31	±0.01

Table 4.4: Fitting functions of figure (4.10).

Table (4.3) has the expression and respective parameters for the fitting functions in figure (4.9); Table (4.4) has a similar content, but related to figure (4.10). The first thing to notice is that the expressions in the tables corroborate our choice for exponential fitting functions. With the exception of $y(n)$ and $y_2(n)$ in table (4.4), none has a correction factor c, indicating that the exponential behaviour is dominant. The main difference between the original and the improved algorithms is a factor of the order of 10 reduction in the running time. This reduction is only noticeable for qubit numbers above $n = 10$, creating the need for the correcting term c in the improved function mode and improved circuit mode expressions. Because the initial values for these modes will have a bigger weight for the fit. The original optimized mode already has a drastic reduction of the a_1 parameter, the initial values are very small and therefore the improved version doesn't need a correcting term c.

From the values in table (4.3) it is evident that the function and circuit modes have a similar behaviour whereas the optimized mode leads to a reduction in the running time of a factor of 10, expressed in the a_1 factor. In table (4.4) the parameters a_k, also show a reduction by a factor of 10 between the original and improved versions, for all the test modes. Again, when considering the running time for simulation, the improved version has astonishingly good results.

The last two cases that will be considered in the study of the time growth are the case named $y = One$, represented in figure (4.11), and the case named $x = One$ and $y = One$, represented in figure (4.12). The first one is the situation where the number y has an array of bits which all equal to one ($|y\rangle = |11\dots1\rangle$). This means we get the maximum number of gates, which at first sight should correspond to a maximum in time. However, if we consider the case when $x = One$ and $y = One$, meaning $|x\rangle = |11\dots1\rangle$ and $|y\rangle = |11\dots1\rangle$, we have the same number of gates as in the $y = One$ case, but since x is only ones, all the gates of the circuit need to be computed and this is expected to be the theoretical maximum value for the simulation time. This agrees with what we see in figure (4.11), the case with $x = One$ and $y = One$ has a larger maximum for the running time (of the order of 450 000 ms for 23 qubits), while the case $y = One$ (figure (4.12)) has the maximum in the order of 400 000 ms, for the same number of qubits. This seems to be the expected result. However, in contrast with the previous study of the gate growth as a function of qubit number, the circuit and optimized modes do not seem to show differences between the original and improved performances.

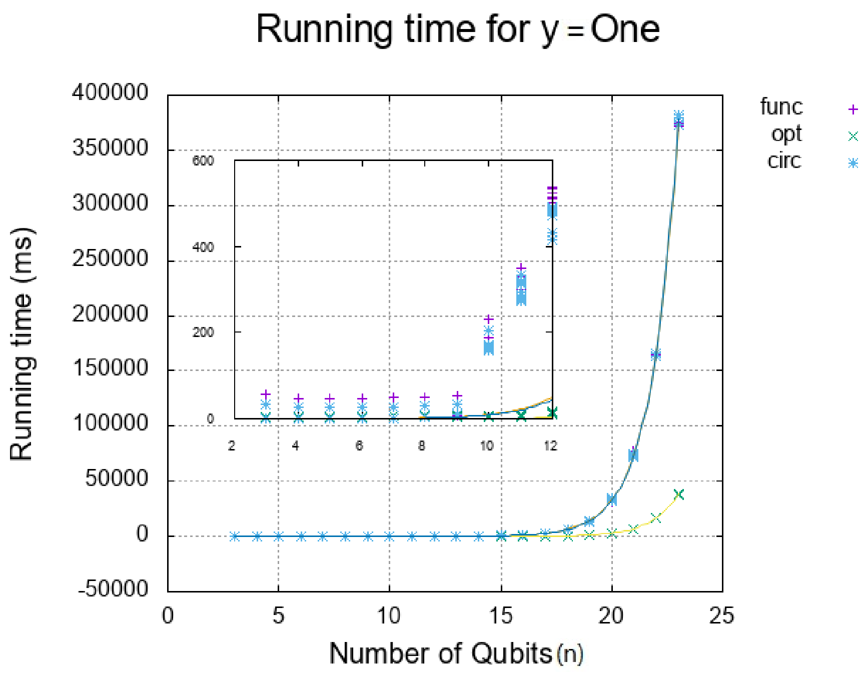


Figure 4.11: Running time as a function of the number of qubits for the case when the number y is always binary *Ones*. The 3 modes of execution are represented. The function and circuit mode have similar performances. Small differences are evident only for small qubit numbers, as shown in the inset plot.

$func$	$y(n) = a * b^n + c$	$\sigma_{y(n)}$
a =	0.00284	$\pm 6e^{-5}$
b =	2.254	± 0.002
$circ$	$y2(n) = a2 * b2^n + c2$	$\sigma_{y2(n)}$
a2 =	0.00232	$\pm 6e^{-5}$
b2 =	2.275	± 0.003
opt	$y1(n) = a1 * b1^n + c1$	$\sigma_{y1(n)}$
a1=	0.000162	$\pm 2e^{-6}$
b1=	2.3118	± 0.001

Table 4.5: Fitting functions of figure 4.11.

$func$	$y(n) = a * b^n + c$	$\sigma_{y(n)}$
a =	0.0021	± 0.0002
b =	2.297	± 0.007
$circ$	$y2(n) = a2 * b2^n + c2$	$\sigma_{y2(n)}$
a2 =	0.0034	± 0.0003
b2 =	2.237	± 0.009
opt	$y1(n) = a1 * b1^n + c1$	$\sigma_{y1(n)}$
a1=	0.000161	$\pm 4e^{-6}$
b1=	2.322	± 0.002

Table 4.6: Fitting functions of figure 4.12.

Table (4.5) has the fit expressions and respective parameters for the graph of figure (4.11). Table (4.6) has the same information regarding figure (4.12). As mentioned above, the function mode maximum will be higher for the $x = One$, $y = One$ case. However, the parameter a for the $y = One$ case is larger than for $x = One$ and $y = One$ case, a feature that was not expected. Nevertheless, since the parameter b

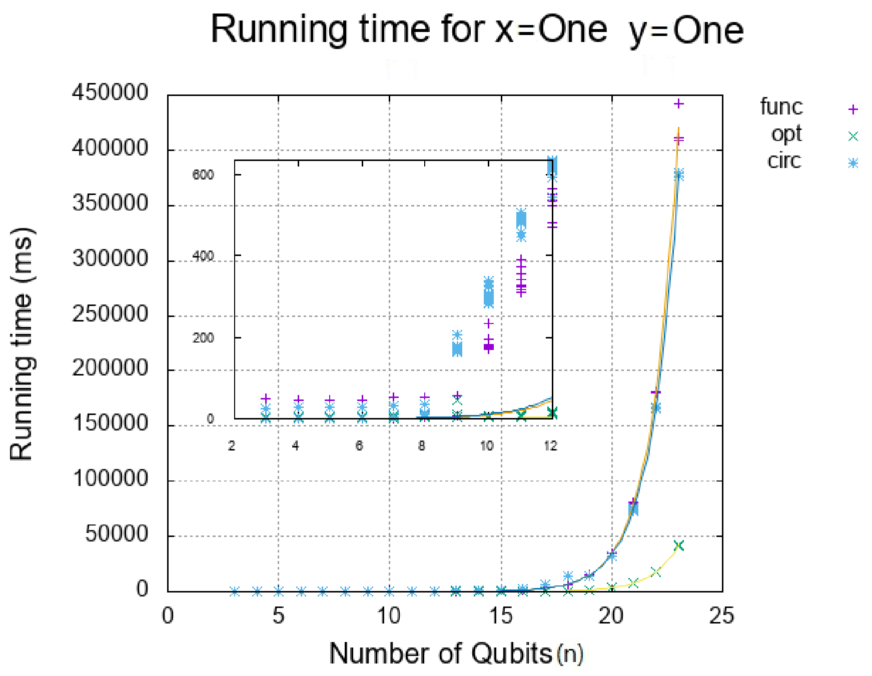


Figure 4.12: Running times as a function of the number of qubits for the case where x and y are binary *Ones* , using the three different modes of execution. Since this is the most heavy case to compute, the overall time is larger. The inset plot shows the difference between the function and circuit modes for small qubit numbers.

is a little larger for the $x = One$, $y = One$ case, it dominates the overall effect for large qubit numbers.

The circuit mode parameter $a2$ of the second graph is bigger than the one of the first graph. However, the first case will have slightly larger maximum values than the other case, because of the parameter $b2$. This was not the expected outcome, but it's not a relevant difference since both graph maximums will be slightly under 400 000 ms. Which justifies the observations from the graph, discussed above.

Comparing the function and circuit mode within the same graph, we see that for large qubit numbers the circuit mode performs better than the function mode in a heavy operation circuits, such as the $x = One$, $y = One$. However, in the case of $y = One$ the function mode performs slightly better than the circuit mode for large qubit numbers. For small qubit numbers, the circuit mode is slightly better than function mode in both $x = One$, $y = One$ and $y = One$.

Concerning the optimization mode, its parameters are the same, within errors, for both $x = One$, $y = One$ and $y = One$ cases. This suggests that for the LIQ$Ui\,|\rangle$ optimization there is no apparent difference between running one case or the other. Comparing optimization mode with function and circuits modes, the former has always the best performance and its particularly evident for large n, when the effect of b parameter dominates.

Now that we have seen all the separate cases regarding running time, it is instructive to review them all in the same graph. Figure 4.13 has the profiles for the previously discussed cases without optimizations, whereas figure 4.14 is a similar plot for the optimized cases. We can see that the optimization provided by LIQ$Ui|\rangle$ leads to important gains in performance but the optimized improved version can reduce the running time even further. These results emphasize the importance of the improvements made to the algorithm. These graphs also show that, without any doubt, the case with $x = One$, $y = One$ it's the most heavy to compute.

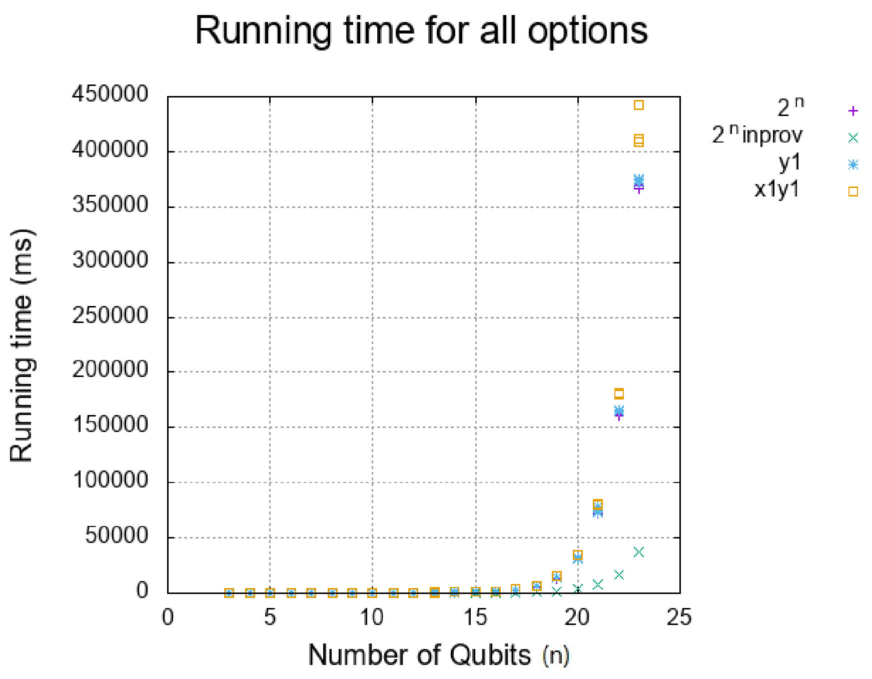


Figure 4.13: Running times for all the cases without optimized versions as a function of the number of qubits. The only case that stands out is the improved version of 2^n.

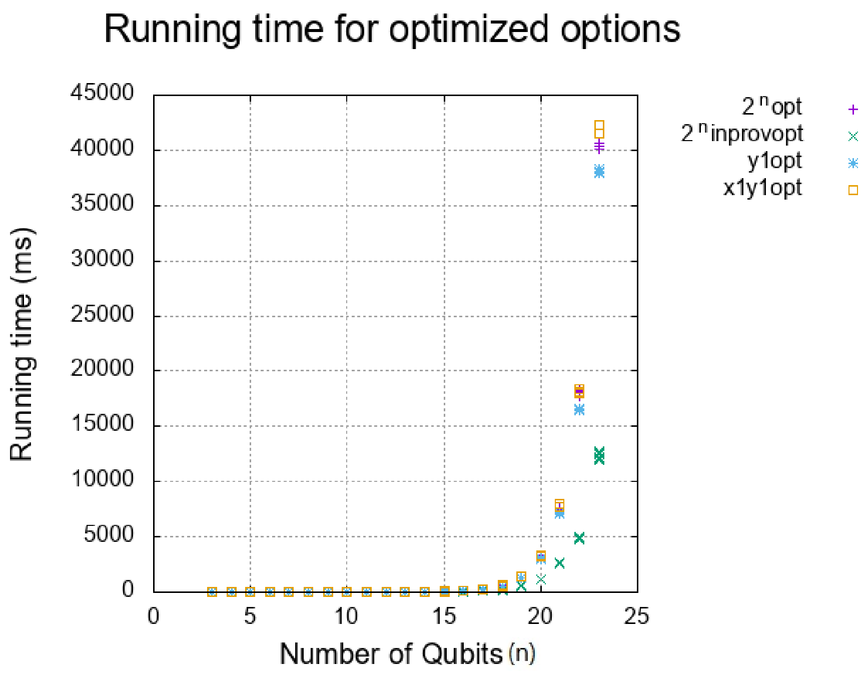


Figure 4.14: Similar to fig(4.13), but now with the optimized versions of the graphs. Once again the optimized version of the improved case was the one worth mentioning.

4.3 Discussion

We have seen from the graphs that the computation time has always an exponential behaviour with the number of qubits. The number of gates, however, has a polynomial dependence which may be quadratic at maximum and linear at minimum. So, the time to run a simulation on a classic computer increases exponentially, even if the gate count does not. This is because a classic computer will have to spend resources to make the operations which scale exponentially with the qubit number, as already discussed in section (3.3). Regarding time, the simulation behaves as $\mathbf{O}(b^n)$. The worst case is for x and y having all bits equal to one, for which the running time is described by 0.0021×2.3^n, where n is the number of qubits. The best case occurs for $y = 2^{n-1}$ in a version which is both optimized and improved, for which the form of the running time is described by the function 0.000057×2.31^n. The exponential behaviour is similar, but the pre-factor 0.000057 is about an order of magnitude smaller than corresponding factor for the worst case, which attenuates the exponential.

Concerning the option between function and circuit modes, we can say that the differences are relevant only in some cases, as discussed above. The graphs of simulation times have insets of the running times for the first 9 qubits. This is where there is a visible difference between both modes, the function mode takes clearly more time to run. However it becomes a very small difference as n grows, the circuit mode is faster, but not by much. When we analyse the graph for the worst case to simulate, $x = Ones$ and $y = Ones$ (figure (4.7)), there is a noticeable difference between the circuit and function mode. The latter will have reduced terms for the exponential 0.0021×2.297^n, comparing to the 0.00284×2.54^n. As discussed, the differences between these two modes of simulation may not be extremely relevant, and in some cases using function modes might have some advantages. Comparing the graphs for $x = One$, $y = One$ and $y = One$, we get the impression that in LIQ$Ui\,|\rangle$ environment it is different to represent the gates in function mode or actually have their effects on the circuit, in circuit mode, because of the way the function mode works with the matrices.

Figures 4.13 and 4.14 allows us to compare directly all the cases regarding running time, and put an emphasis on the fact that the improved version of the circuit has much better results. Comparing the case which grows with 2^{n-1} and the improved version, we get a reductio factor of 10, corresponding to a 90% improvement, which is an outstanding result. If we compare it to the optimized version of 2^{n-1}, we get a reduction to roughly 12000 ms, which accounts for an additional factor of 3 (97% of the original value) of time reduction. These results are much better than we ever imagined. This proves that improving a circuit with classical computer, before running it in a quantum computer, might speed up the process, and save resources. The original function took roughly 6 minutes to run, the optimized improved version took less than one minute, 37 seconds. The improved version uses the algorithm with the minimum number of gates possible.

The simulations were made several times for each qubit. However, the simulation terminal was not restarted for each one of them. Instead, we run the program multiple times, resetting the variables we needed. One interesting thing that happened was that for small qubit numbers, the simulation time varied a lot. For example, for 3 qubits, using this method and running the simulation 10 times, the first time may take 47 ms to run, whereas the next ones 0 ms. The number is zero when the program don't have enough resolution to indicate smaller values. When the qubit number is large enough (> 10), this stops happening. This may be because of how the program works: in the first simulation it needs to load all the libraries and variables, but in the second time and so on, it doesn't need to load anything, only to reset the variables. This is just an interesting detail but it has almost no influence on the growth profile. However, some fitting variables may have larger errors because of this effect.

Full circuits for the cases with $n = 23$ qubits, may be seen in the appendix A.3.

4.4 About Memory Usage

The amount of resources used in these tests is of major significance since it tells us how far we can achieve with quantum algorithm simulation within a classical computer. The LIQ$Ui\,|\rangle$ program did not allow simulations larger than 23 qubits because of the memory needed. We did not run out of memory in the simulations. However, the allocated memory was probably not optimized, since Windows needs to run a lot of services in the background. Windows needs almost 2 Gb of RAM only to run the system, when using programs and libraries it will require more. So, the truth is the available memory is not the same as the physical RAM of the computer.

We can get information about the used RAM, using native tools from windows. However, since LIQ$Ui\,|\rangle$ runs in a specific terminal with language packages in the background, we may not be able to see all the RAM that is being used. The LIQ$Ui\,|\rangle$ dump files provide the memory usage for the cases where it becomes important. We monitored the RAM usage with Windows and saw that this information, and the one given by the program is not always exactly the same. However, the Windows' tools are enough to see how the memory varies.

A graph of the memory usage provide by LIQ$Ui\,|\rangle$ dump files is shown in figure (4.15).

We can clearly see the exponential behaviour of the function as the memory starts to increase for large qubit numbers. The yellow line is the fit, used to demonstrate that the dependence is exponential. Even though the fit is not perfect, it is good enough to see the exponential behaviour. Interesting enough, the case which requires more memory is the improved circuit. Since it needs to run more processes and to store more information, it needs more memory. But most of that memory increase is for classical operations, which makes it pay off in the computation time, so it's worth it.

Regarding the CPU usage, we monitored it, and it usually spikes around 15 qubits, to very high values. When working with a large number of qubits, the CPU uses 100% of its capacity most of the time. If the code is more demanding, as the case for the improved circuit, the CP usage may spike earlier. The major concern with

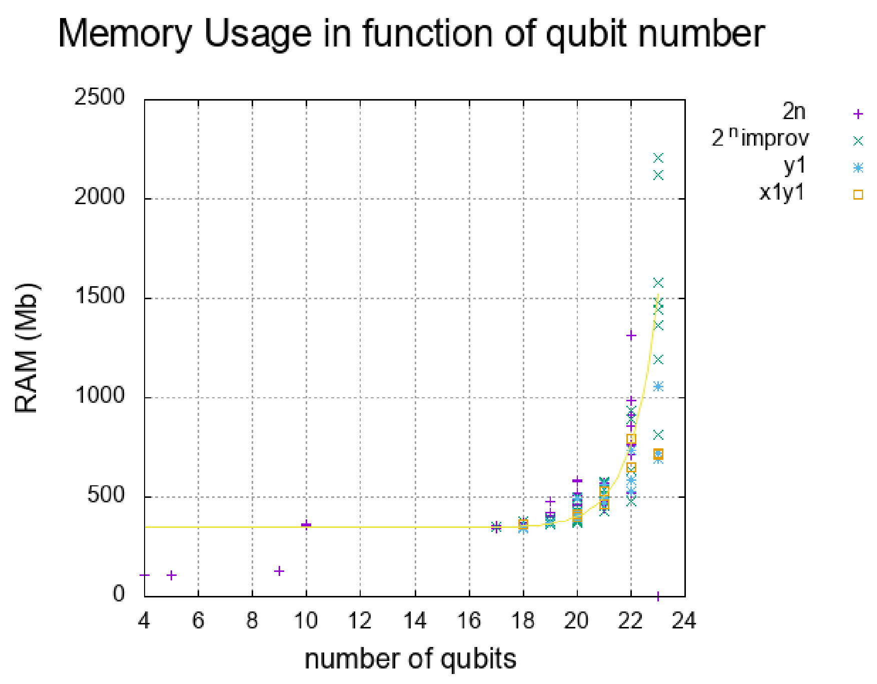


Figure 4.15: Memory usage in the simulations for all the different cases. The yellow line represents the fit to the data of 2^n_{improv} case, which is described by $f(x) = 8.4e^{-8} * 2.8^x + 347$.

the CPU usage is the heat, for long computations it may become a problem.

For this reasons, using a system dedicated only to this simulation, with a proper cooling system may give very good results, and since it is much cheaper than the quantum hardware, it is a viable solution.

5

Quantum Error Correction

5.1 Basic Principle

The most serious problem of a quantum computer is the loss of coherence of quantum states with time. It is known that the coherent states of qubits are extremely fragile, and the larger the system, the more fragile it is, as it loses the coherence more rapidly. The quantum computer uses physical gates that can introduce errors themselves. There are different sources of errors in quantum information that need to be corrected. A qubit may not only have bit flip errors (which can be described as the effect of gate X) but also has phase flip errors (gate Z). Note that in a Bloch Sphere, any undesired rotation, no matter how small, is an error to the system. However, knowing that $Y = iXZ$ and that any rotation can be described as a combination of these three gates (X,Y,Z), by correcting the two types of errors described X and Z gates corrects any rotation. This is the goal of Quantum Error Correction (QEC) codes.

QEC codes started by adapting the classical error correction techniques. From that point, it evolved to more complex codes with the objective of making the quantum computer a fault tolerant system[41]. The most basic code is the 3-qubit code.

5.1.1 3-Qubit Code

The principle behind this algorithm comes from classical computation. It only needs three qubits per original qubit of the system. A classic computer works with bits, and the only error that can occur is a bit 0 becoming 1 or vice versa. To be sure that the data doesn't have any errors of that kind we can try to encode it, work with it and decode it in the end. For example, the bit $|0\rangle$ becomes the logical zero,

$|0_L\rangle = |000\rangle$[1] , that is, the bit is repeated three times. Now, we may have some error inserted in the logical qubit along the way, as described in (5.1). In the example, the logical zero end up with an error in the second qubit ($|1\rangle$ instead of $|0\rangle$). In order to know how to correct this state, we must know which qubit is the right one. Using the method of majority voting, the majority of qubits in the same state must be the right ones, as long as the error probability is *small enough*[2].

$$|0_L\rangle = |000\rangle \xrightarrow{\text{Error Introduction}} |010\rangle \tag{5.1}$$

The first step of the 3 qubit code is to encode the qubits into the logical states. We must notice that the **no cloning theorem** prevents the cloning of quantum states, unlike the digital classical realm where copies of data can be made. The circuit in the figure (5.1) is able to create a logical state $|\psi_L\rangle$ without cloning the input state. The other states have similar information as the original state, in a redundant way, but are not the same as the original state.

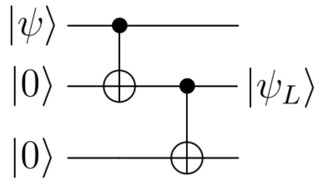


Figure 5.1: Representation of the circuit needed to encode a qubit in the state $|\psi\rangle$ into the logical state $|\psi_L\rangle$.

If the initial state is $|\psi\rangle = |0\rangle$ then the CNOT gates don't act and in the end we have 3 identical qubits $|000\rangle$. If the initial state is $|\psi\rangle = |1\rangle$ then the CNOT flips the 2 ancilla qubits to $|1\rangle$. The general state becomes: $|\psi\rangle = a\,|0\rangle + b\,|1\rangle \rightarrow |\psi_L\rangle = a\,|000\rangle + b\,|111\rangle$.

After the encoding, we need to have a circuit to verify if there is an error and where. This is called the **syndrome measurement**. This measurement gives us information on which bit there was an error, but it doesn't give any information about the state nor changes it. The operators listed bellow, are used to perform the

[1]As we represent the logical zero as $|0_L\rangle = |000\rangle$, we can do the same to represent the logical qubit 1 $|1_L\rangle = |111\rangle$.

[2]In this case a *small enough* probability means that only one error is allowed to appear in three bits logical state. If it appears more than one, then this method to correct the error does not work. The majority voting with two wrong bits gives the wrong bit as if it was correct. Being p the probability of flipping one bit, then the probability of error in more than one bit is $P_e = 3p^2(1-p) + p^3 = 3p^2 - 2p^3$ we need to assure that $P_e < p$ which happens if $p < 1/2$.

syndrome measurement on a 3 qubit code for bit flip, applying them to the encoded state and see where the error happened.

$$P_0 = |000\rangle\langle 000| + |111\rangle\langle 111|$$
$$P_1 = |100\rangle\langle 100| + |011\rangle\langle 011|$$
$$P_2 = |010\rangle\langle 010| + |101\rangle\langle 101|$$
$$P_3 = |001\rangle\langle 001| + |110\rangle\langle 110|$$

For example, the state $|\psi\rangle = \alpha|010\rangle + \beta|101\rangle$ has an error in the second qubit. In order to detect it with the syndrome measure, we apply the operators, as shown below:

$$p(0) = \langle\psi| P_0 |\psi\rangle = 0$$
$$p(1) = \langle\psi| P_1 |\psi\rangle = 0$$
$$p(2) = \langle\psi| P_2 |\psi\rangle = 1$$
$$p(3) = \langle\psi| P_3 |\psi\rangle = 0$$

The syndrome measurement gives the result $p(2) = 1$, so we conclude the error is in the second bit. The next step is to correct it by applying an inverting operation on that qubit.

This syndrome measure can also be implemented in the circuit mode, represented on the figure (5.2). With the syndrome circuit, there is no need to use the operator projection described above, it only needs to compare the qubits. With 2 ancilla qubits [3]and using CNOT gates, it compares the parity of the qubits, thus detecting the error. The comparison process goes like this:

$|\psi_1\rangle = |\psi_2\rangle$	and $|\psi_1\rangle = |\psi_3\rangle$	no error
$|\psi_1\rangle \neq |\psi_2\rangle$	and $|\psi_1\rangle \neq |\psi_3\rangle$	error on $|\psi_1\rangle$
$|\psi_1\rangle \neq |\psi_2\rangle$	and $|\psi_1\rangle = |\psi_3\rangle$	error on $|\psi_2\rangle$
$|\psi_1\rangle = |\psi_2\rangle$	and $|\psi_1\rangle \neq |\psi_3\rangle$	error on $|\psi_3\rangle$

Table 5.1: Representation of the syndrome measurement possible outcomes

After that, we measure the ancilla qubits, which will give the syndrome result. In turn, they will classically control a gate X to revert the error in the proper qubit.

[3]Ancilla qubits are qubits with a known initial state, usually $|0\rangle$, used to perform some auxiliary

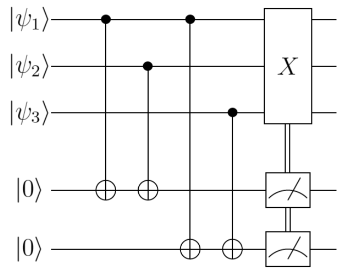


Figure 5.2: Circuit to perform the syndrome measurement with two ancilla qubits. After measuring this two ancilla qubits it performs a X gate with classic control to correct the error.

In the classical digital world, there are only bit flip errors. In the quantum domain, however, we may have both bit and phase flip errors[4]. This method only corrects bit flip errors. In addition, if there is more than 1 error, the syndrome measurement will not be able to identify with success which data bits contain errors. It assumes no error in the ancilla qubits nor in the CNOT gates, they are considered perfect qubits and gates, this method is therefore very limited. Let's examine the algorithm to correct the other type of errors, the phase flip.

5.1.2 3 Qubit code with phase errors

A phase flip error doesn't have an equivalent in classical computation. In a phase flip error, the state $a|0\rangle + b|1\rangle$ becomes the state $a|0\rangle - b|1\rangle$. This kind of errors can be corrected with the introduction of a different gate in the circuit, the gate Z.

If we apply a gate Z to the state $(|0\rangle + |1\rangle)/\sqrt{2}$, often called the state $|+\rangle$, the result will be the state $(|0\rangle - |1\rangle)/\sqrt{2}$, the so called state $|-\rangle$. There is a resemblance with the bit flip circuit and the classical bit flip, since applying the gate Z to the state $|+\rangle$ flips it to the state $|-\rangle$. So, this can be treated as a bit flip, but not in the basis $|0\rangle$ and $|1\rangle$ but in the basis $|+\rangle$ and $|-\rangle$. This means that the error detection for the phase flip can be converted to a simple bit flip with the same 3 qubit code. All it is needed to do is encode the qubits from the state $|0\rangle$ or $|1\rangle$ to the states $|+\rangle$ or $|-\rangle$ respectively. To do this, first we do the encoding as before into the logical states, and then we apply a Hadamard (H) gate. The Hadamard gate acts on the basis $|0\rangle$, $|1\rangle$ to change them into $|+\rangle$, $|-\rangle$, represented in the figure (5.3).

operations. In this case the ancilla qubits are the register to the syndrome measure results.

[4]Any unwanted rotation in the Bloch sphere is viewed as an error, but correcting bit flip and phase flip errors is enough to get the right state.

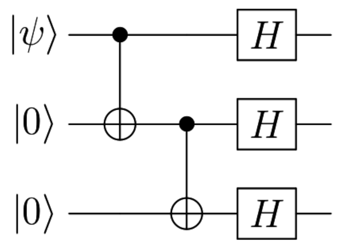


Figure 5.3: Representation of the circuit needed to encode a qubit in the state $|\psi\rangle$ into a logical state prepared for correcting errors of phase flip.

The error detection and correction will be done the same way as in the bit flip code, after adding a H gate to the circuit, put after the $CNOT$ gate. The error probability must be the same as before.

5.2 The 9 Qubit Code (Shor)

The Shor's code of 9 qubits was one of the first most complete QEC codes because it can correct both bit and phase flip errors in the same circuit. It is a combination of the both codes already discussed. The phase flip runs first and needs 3 qubits to be encoded. After that, it needs 3 more qubits for each one, to run the bit flip code. So, the total number of qubits in the circuit for encoding is 9, hence the name. This is the basic principle behind the Shor's code. As long as there isn't more than 1 error per qubit, the code can correct it. This code is catalogued as a degenerate code because it doesn't need to know in which qubit the error appeared, all it needs is the block of qubits in question. The encoding process is represented in the figure (5.4), the qubits will be encoded in the phase domain, as discussed in the phase code, that is, the states $|0\rangle$ become $|+++\rangle$ and $|1\rangle$ become $|---\rangle$. After the bit encoding the logical states of the system become:

$$
\begin{aligned}
|0_L\rangle &\equiv \frac{(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)}{2\sqrt{2}} \\
|1_L\rangle &\equiv \frac{(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)}{2\sqrt{2}}
\end{aligned}
$$

For the error detection and correction, this code uses a similar principle to the previous codes syndrome measurement. For the bit flip, the syndrome circuit is applied to each block of three qubits independently, comparing each qubit with the others of the same block, as the circuit in figure (5.2). This code can have up to one

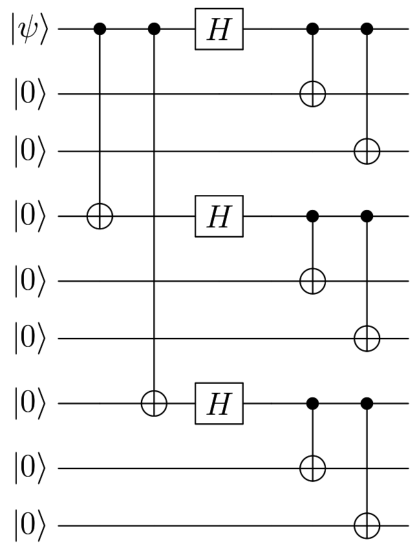


Figure 5.4: Representation of the circuit needed to encode a qubit in the state $|\psi\rangle$ into a logical state prepared for the 9 qubit Shor code.

error in each block, a total of three errors of bit flip.

For the phase flip, the error detection is done comparing the sign of the three blocks of qubits. Since the phase encoding is the first stage of the code, if there is a phase error in either of the three phase encoded qubits, then all the qubits in that block will get the same error sign. So, the syndrome measurement compares between blocks of qubits. First, it is made a comparison between the first two blocks, and later between the second and third block.

Comparison between blocks	Sign	Block Error
1st: $|000\rangle - |111\rangle$ 2nd: $|000\rangle + |111\rangle$	different	1st
2nd: $|000\rangle + |111\rangle$ 3rd: $|000\rangle + |111\rangle$	same	
1st: $|000\rangle + |111\rangle$ 2nd: $|000\rangle - |111\rangle$	different	2nd
2nd: $|000\rangle - |111\rangle$ 3rd: $|000\rangle + |111\rangle$	different	
1st: $|000\rangle + |111\rangle$ 2nd: $|000\rangle + |111\rangle$	same	3rd
2nd: $|000\rangle + |111\rangle$ 3rd: $|000\rangle - |111\rangle$	different	

Table 5.2: Possible combinations to detect the phase error in the 9 qubit code.

This means that for example, if the first and second blocks have different signs and the second and third blocks have the same sign, then the error is in the first block.

All the possible combinations of the syndrome measurement for the phase flip are represented in the table (5.2). The implementation of this kind of circuit is the same as discussed previously, using $CNOT$ gates to compare between blocks and H gates to work in the phase domain.

5.3 Liqui$|\rangle$ and the Quantum Error Correction (QEC) Codes

In the previous section we discussed the basic principle needed to understand the error correction codes. QEC codes are a hot topic nowadays and they are progressing. Even when using the basic codes, making QEC circuits for complex systems needs a large number of qubits. In order to implement the Shor's code it takes 9 qubits just for the encoding of a single qubit. It's worth reinforcing the idea that the objective of this chapter isn't to explore with great detail the QEC, instead to be somehow familiarized with them.

The implementation of the QEC codes can make the circuit very heavy, so it's better to work with solutions that make this process lighter. LIQ$Ui\,|\rangle$ has a suite for error correction code simulation. Using this mode of simulation, we can encode the states, then simulate the introduction of errors and its correction, and lastly check if the results are the ones expected. The error introduction is done by inserting gates in certain points of the circuit randomly. There is a certain probability of error introduction that is defined. So, an interesting analysis may be to test the ability of the circuit to correct errors as a function.

As was mentioned QEC codes need a lot of resources, one of the things that may need some improvement. For this reason, the LIQ$Ui\,|\rangle$ QEC suite doesn't use the Shor's code of 9 qubits, which need many qubits. Instead, it uses a more robust code, the Steane 7 qubit code.

5.3.1 Steane 7 qubit QEC

The Steane 7 qubit code belongs to the class of CSS codes (Calderbank-Shor-Steane) and is a QEC code used in a different category from the ones discussed so far, the category of stabilizers. However, it can be established without the stabilizer formalism, which is how it was created. In order to understand this code origin, it is needed to study classical linear codes which is not the scope of this thesis, see the references [1, 42] and [43] for more details. However, it is important to highlight some features. A linear code C that encodes k bits into an n bit code, does that using a generator matrix G (n by k), which is made of zeros and ones. The parity check matrix H makes $Hx = 0$ (x being a codeword), is used to perform the error-correction. This matrix is related to the matrix G. The concept of dual code is also important for CSS codes. Having a code C with matrices G and H, its dual would be defined as $C^{\perp}$, which has the transpose matrices of G and H.

The Steane 7 is a CSS dual code. The code C_1 is a [7,4,3][5]code, known as the Hamming Code (see [1] for details), and the code $C_2 = C^{\perp}$, a [7,3,3] code. The Steane 7 is constructed with this two codes, as a [n,$k_1 - k_2$,d]=[7,1,3] code. Which means that each 1 qubit is encoded by 7 qubits. The distance between basis states of 3 provides that the code can correct 1 error, according to equation (5.3). The logical states of the code are derived from the parity check matrix of the Hamming code [44], which are:

$$
\begin{aligned}
|0_L\rangle = \frac{1}{\sqrt{8}}(&|0000000\rangle + |1010101\rangle + |0110011\rangle + |1100110\rangle \\
&+ |0001111\rangle + |1011010\rangle + |01111000\rangle + |1101001\rangle) \\
|1_L\rangle = \frac{1}{\sqrt{8}}(&|1111111\rangle + |0101010\rangle + |1001100\rangle + |0011001\rangle \\
&+ |1110000\rangle + |0100101\rangle + |1000011\rangle + |0010110\rangle)
\end{aligned}
\tag{5.2}
$$

5.3.1.1 Stabilizer Formalism

This formalism uses operators to describe the quantum state instead of the state himself. It can be used to perform various operations, and it's not limited to error correction. However, is particularly useful for error correction because it can simulate the system with fewer resources.

The stabilizer formalism uses the concept that an operator K stabilizes a state $|\psi\rangle$,

[5][n,k,d] is the set of parameters that specify the code, which means that n qubits encode a k logical qubit, with d distance. The distance refers to how many qubits need to change in order to a codeword to become another different codeword, which translates the number of errors (t) it can correct.

$$t = [\frac{d-1}{2}] \tag{5.3}$$

as long as this state is a +1 eigenstate of K with:

$$K|\psi\rangle = |\psi\rangle$$

If we consider the EPR state $|\psi\rangle = \frac{|00\rangle+|11\rangle}{\sqrt{2}}$, we see that this state is stabilized by the operators $K^1 = X_1 \otimes X_2 = X_1X_2$ and $K^2 = Z_1Z_2$ (the subscript numbers refer to the qubit each operator acts upon), which belong to the Pauli group of operators. This translates into $X_1X_2|\psi\rangle = |\psi\rangle$ and $Z_1Z_2|\psi\rangle = |\psi\rangle$. These two operators are enough to stabilize this state, which means we only need the operators to refer to the state. In a computation perspective, it means it is only needed to save the group of operators and not all the state amplitudes. This is the great advantage of using stabilizer formalism, it becomes lighter to compute. For the Steane 7 code, we have a group of six operators:

K^1	$IIIXXXX$
K^2	$IXXIIXX$
K^3	$XIXIXIX$
K^4	$IIIZZZZ$
K^5	$IZZIIZZ$
K^6	$ZIZIZIZ$

Table 5.3: The group of six generators (operators) for the Steane 7 code.

It's worth mentioning that the operators have a similar structure to the parity check matrix that was used to construct the Steane 7 code. These operators are able to represent the logical states of the code, in the equation (5.2), but in a more compact representation. The stabilizer formalism establishes a subspace of reduced dimension. The Hilbert space for the encoded 7 qubits is 2^7, the subspace has the size of the number of qubits subtracted by the number of operators, giving a total subspace of $2^{7-6} = 2$. The stabilizer formalism works in a subspace that only has dimension 2 to work with seven qubits.

We saw how the stabilizer formalism works and how the Steane 7 code is defined within this formalism, now we must inspect the preparation of the logical states. Since the encoded logical states for the code are +1 eigenstates for the six operators, we need to assure this when encoding the qubits.

The encoding process uses a dedicated circuit. Theoretically, in order to prepare the logical state $|0_L\rangle$, we start by preparing the states $|0\rangle^{\otimes n}$ and an ancilla qubit in the state $|0\rangle$. The ancilla will have an Hadamard gate applied and then it controls the general operator K applied in the state system. After this, the ancilla has another

Hadamard gate and then it's measured. The outcome of this measurement makes the state an $+1$ eigenstate of K if the measure is $|0\rangle$, or -1 if it measures $|1\rangle$. The -1 eigenstates can be converted to the expected $+1$ eigenstates with the use of Z gates. This process projects the initial state, making it stabilized by the operator K, so it needs to be done for all the operators.

In practical terms, we can use more than one ancilla to accelerate the operator measurement process. So, for the Steane 7 code, we use three ancilla qubits for the three operators at the same time. It only needs to use the first three operators (regarding gates X). That's because when encoding the logical $|0_L\rangle$, we initiate a state $|0\rangle^{\otimes 7}$, which is already an eigenstate of the other three operators (regarding Z gates).

Steane 7 logical qubits can be prepared by a circuit using only Hadamard and CNOT gates. The circuit of the figure (5.5) is the preparation circuit for the Steane 7 logical qubit $|0_L\rangle$. Note that the figure (5.8) has the same circuit but was drawn directly with the LIQ$Ui\,|\rangle$ program, it is compacted. The bottom three qubits are the ancilla qubits. All the qubits of the circuit started in the $|0\rangle$ state. This circuit can do the logical preparation with only 7 qubits, while the theoretical example discussed above needs 10 qubits. Attending to the fact that the encoded operator X is $\bar{X} = X_1X_2X_4$, then we can encode the logical state using the rearranged operators $K^1 = X_5X_6X_7$; $K^2 = X_3X_6X_7$ and $K^3 = X_3X_5X_7$. Since the states start as $|0\rangle$, it can be guaranteed that they will be a $+1$ eigenstate of the operators. After this part, the circuit itself can be implemented, including the error-correcting part.

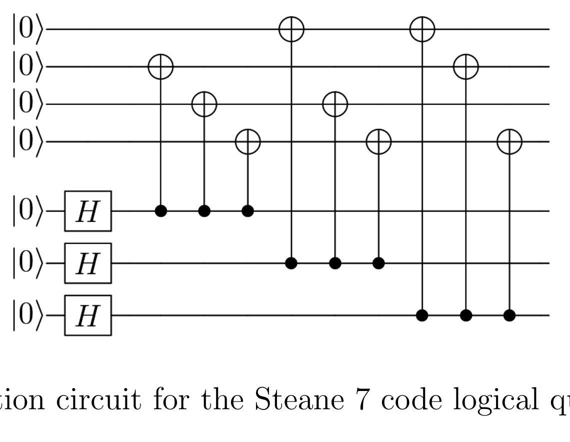


Figure 5.5: Preparation circuit for the Steane 7 code logical qubit $|0\rangle$.

One error in the system may be translated by an operator E. This error correcting code can correct errors of the type of X and Z operators, plus the operator $Y = iXZ$. Since the state is stabilized with the operator K, then the state with error will satisfy the equation:

$$KE\left|\psi\right\rangle_L = \pm E\left|\psi\right\rangle_L$$

The state is a +1 eigenstate if the error operator commutes with the stabilizer and -1 if it anti-commutes. What this means, in practical terms is that we can detect this types of errors simply by measuring the operators of the code, and detecting the anti-commuted eigenstate.

The errors of type Z anti-commute with the operators K^1, K^2 and K^3. The errors of the type X anti-commute with the operators K^4, K^5 and K^6. This makes the errors of the type Y anti-commute with both kinds of operators, since it is a X and Z combination. So, similar to the state preparation, we measure all the operators [41]. With the result, we know in which qubits there is an error, so applying the proper operators X and Z to the circuit corrects the error. This process needs to be done multiple times to correct all the errors, in a circuit that has multiple operations. In order to be fault-tolerant, we can do this process after every operation to the circuit. To measure the operators of the circuit we need 6 ancilla qubits, one for each operator. This circuit is represented in the figure (5.6), which has the first part to detect the type Z errors. The first ancilla qubit is relative to the operator K^1, the second to the operator K^2 and so on. Note that, once again, the first three operators (regarding X gates) can be implemented with the CNOT operations. The last three ancilla qubits control the operators of Z gates, which detect X errors. After measuring the ancilla qubits, we implement the error-correction, with the proper X and Z gates controlled by the measure results. These controlled operations are represented by two lines since it's a classical control. This can be seen in the figure (5.10), which has the syndrome measurement and correcting circuit from the LIQ$Ui\,|\rangle$ program. Note that the syndrome is the first part of the circuit, before measures, and is the same as represented in the figure (5.6) but in a compact representation.

5.3.2 Programming QEC

LIQ$Ui\,|\rangle$ has a library for QEC that is called Quantum Error Correction Codes (QECC). More precisely it has a *Steane*7 class that implements the Steane 7 code with [7,1,3].

This class needs to work in the Circuit mode, this means the circuit must be compiled and then converted to the Steane 7 environment and only then do the simulations. In order to obtain results, it is needed to run the compiled circuit. The run method

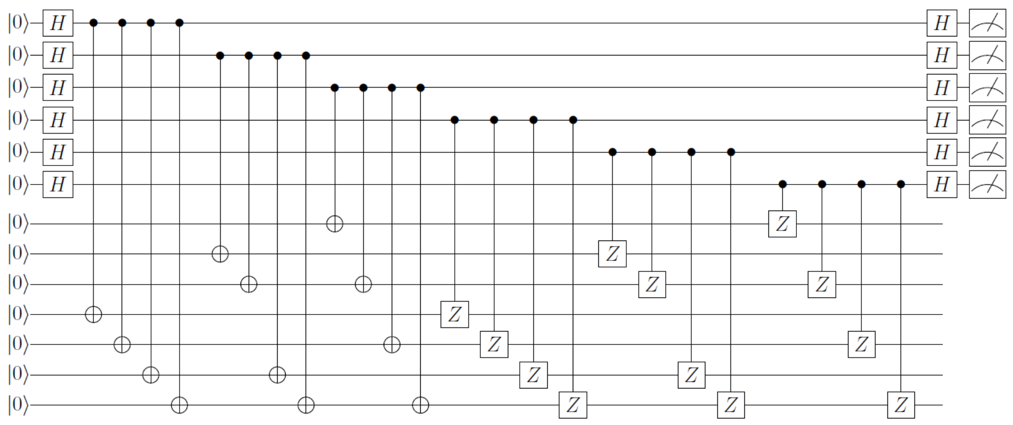


Figure 5.6: The syndrome circuit for the Steane 7 code. The first part detects type Z errors, and the second part detects X errors.

will change the qubits values in their own kets, this means that for the simulation we will need two identical kets. With one we run the normal function and with the other, we run the simulation with error introduction. In the end, we compare the results to see if the error-correction was successful.

5.3.2.1 Limitations

The LIQ$Ui\,|\rangle$ documentation has an example of how the *steane*7 class is done, in order to understand how it works. However, in code implementation, the only thing that is needed is to work directly with the class constructor and its methods. The constructor will encode the qubits, as discussed in the previous section, and then will compile the circuit already made into this class.

The QECC class has its own methods to transform the normal gates into gates prepared for the logical qubits. *Steane*7 is a different class that inherits all the QECC library, and his constructor will prepare the gates via Transverse operation into 7 qubits gates. From the *steane*7 example we know that it only supports the following gates: H, S, X, Y, Z, I, M. M is the measurement operation, the others are all single-qubit gates. Also supports the CNOT gate, a 2 qubit gate, since is a controlled operation.

All the initial qubits in the circuit need to begin in the state $|0\rangle$, in order to work with the *steane*7 class. This means that in the case of the addition or QFT circuit, only simple operations with $|0\rangle$ are possible, for example, the 0+0 addition.

Since the QFT and addition functions have a controlled rotation that isn't supported natively by the *steane*7 class, we will need to do it manually. This means, convert the QFT circuit with the transverse function. However, some complications were met trying to do this operation. Since the initial state needs to be $|0\rangle$ and the addition function has other limitations, it becomes a very complex circuit rapidly. Then it was decided that, since the main objective of this chapter was to explore and do some simulations with the QECC and *steane*7 class, the circuit used must be simplified. The final circuit that was used in the simulation was a QFT with only 2 qubits (figure (5.7)). This is enough to explore the error correction since the 2 qubit code has the basics to simulate the system, and it becomes a big circuit in the end.

Implementing the QFT function with more than 2/3 qubits will be a very large circuit, which is one of the biggest limitations of quantum error correction in general. The resources needed are frequently larger than the ones available.

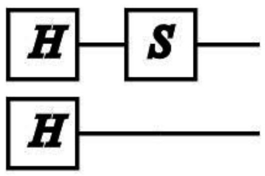


Figure 5.7: Qft circuit with 2 qubits implemented with steane7

5.3.2.2 Code

The full code used to implement the QECC will be in the appendix A, in this subsection we will explore some of the basic commands needed to simulate the *steane*7 class.

In order to measure the time of the simulation, we used the *Stopwatch*() function from the *System.Diagnostics* as represented in line 3 from the code. As already said the program will need 2 identical kets, initiated in the state $|0\rangle$, in this code they will be called k and b.

The first step is to compile the *tqft* circuit of 2 qubits, with the respective qubit lists. The compile function will prepare the circuit mode. In this mode, there are some interesting operations, namely the render function which will print the circuit to file. It's important to refer that Circuit mode only changes the ket states upon the run command. The *tqft* function is the same as QFT only simplified to 2 qubits, as follows:

```
1 let tqft (qs:Qubits)=
        let n=qs.Length
        let mutable z=0
        for q in qs do
                H [q]
                let mutable qi=z+1
                for i in 2 .. (n-z) do
                        S [q]
                        qi<-qi+1
                z <-z+1
        z<-0
```

Next, the compiled circuit will be implemented in the *steane*7 class. *let s*7 = *Steane*7(*qc*) command will prepare the qubits into logical ones with 7 qubits each and then convert the circuit gates to transverse ones. Now the circuit is prepared to do the error simulation.

The *s*7*.prep* command selects the encoding part of the circuit that can be rendered (figure 5.8). In order to render the encoding or syndrome part of *steane*7, the circuit needs to be compiled, as before. However, the *Circuit.Compile* command needs a qubit list, on which it operates. This means that to encode the circuit it needs a 7 qubit long list, but so far the *qs* list only has a length of 2. So, we created a new qubit list with 7 qubits with the code line: *let si* = *Ket*(7)*.Qubits*. In the case of the syndrome circuit, it needs 7 qubits plus 6 ancilla qubits for the measurement operations.

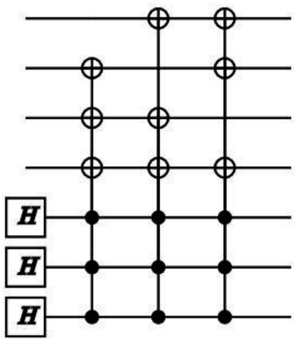


Figure 5.8: Circuit for logical qubit encoding in Steane 7 code.

At this point in the program, the circuits for the error correction are prepared, but there are no errors yet. The errors are injected with a probability p defined in the function calling section. This command line is *let err,stats* = *s7b.Injectp*, and the *Inject* command will introduce the error. This will make a new circuit, which is stored in the *err* variable. The *stats* variable stores how many errors and which types the circuit will have.

The error circuit is a very long one because the algorithm runs each syndrome measurement and correction for each gate in the circuit. So, in this case, we are talking about 3 times the same sequence. But this method allows for the errors to be detected in different parts of the circuit. The full circuit is represented in the Figure B.1, in the appendix B.2.

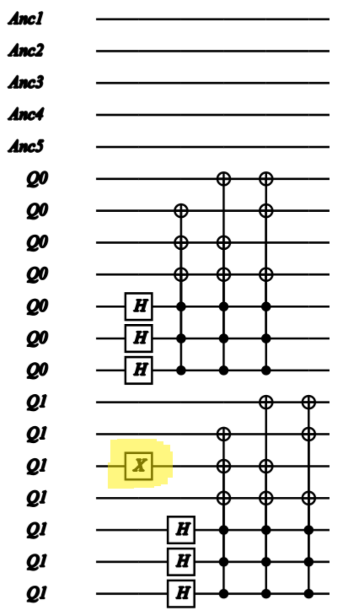


Figure 5.9: Detail of the figure B.1.The error gate X is highlighted in yellow.

The final step of the program is to run the circuits, both original and corrected, and display the results data. $s7f.Runs7.Ket.Qubits$ command will run the circuit without errors. The stopwatch will be started and stopped to measure the time, and after this, the program will display the data with the *dump* and *show* commands.

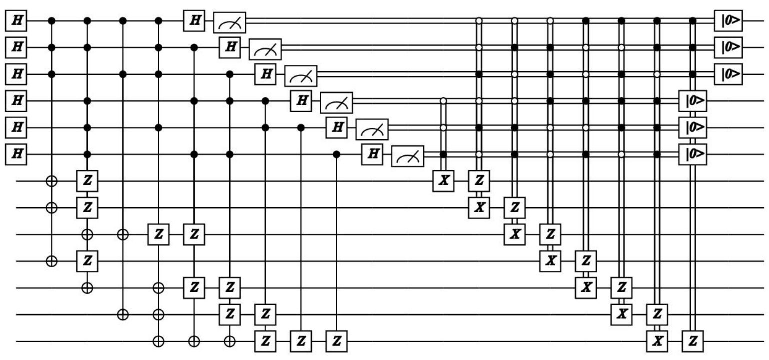


Figure 5.10: Circuit to perform syndrome measurement after encoding.

5.4 Results

In order to analyze the results, the data was treated in excel using macros in visual basic language, because the data from the LIQ$Ui\,|\rangle$ program, which was saved in a log file, has a lot of information non useful for this case. It was decided to run the program with a total of 11 different probabilities, 50 times each. So, in excel with the proper macros, the log files were cleaned to select the essential data. One of the test cases, with probability 0.009 is represented in the appendix B.3 as an example.

The graph from the figure (5.11) has the results from the simulations, and shows the success rate of the error correction as a function of the probability of error injection. When analysing the results, it's important to keep in mid that a change in the global phase has no physical meaning and does not represent an error.

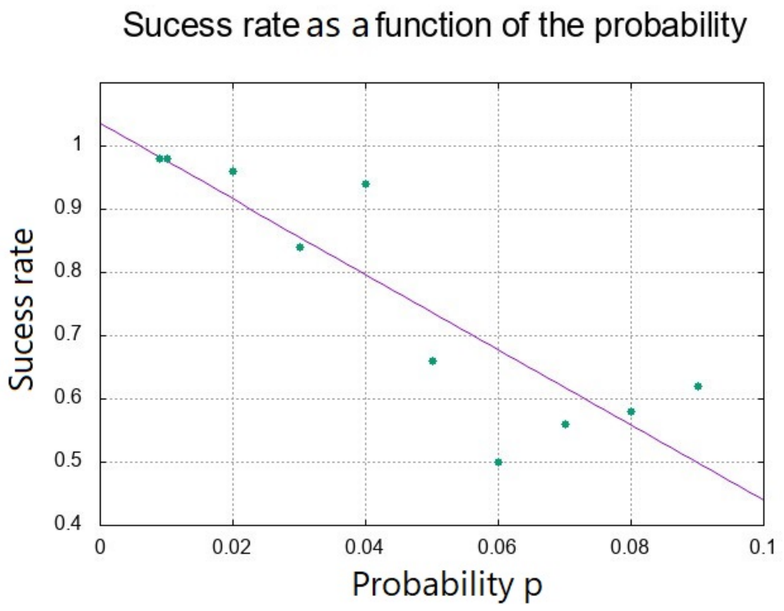


Figure 5.11: Representation of success rate as a function of the probability of error insertion. Includes a linear regression to show that it has a roughly linear behaviour.

It shows clearly that higher the probability of an error injection the worst the success of the correction. With a probability of 0.009, we have a 0.98 rate of success (49 of 50 cases). Increasing the probability to 0.02, the success rate will drop to 0.96. This probability is the limit in order to have results with 5% error. The first conclusion from the data is that the error correction method is only reliable for an error probability not larger than 2%. In theory, a realistic model for error correction should work with an error probability between 0.0001 and 0.001. There are theoretical values for some models which can deal with an error probability of 0.01 - 0.03

[45]. The simulation results indicate that this model has a good performance, sinc it is able to make corrections with an uncertainty of 5% with a maximum $p = 0.02$.

The QEC starts to fail when more than 2-3 error gates are introduced in the system, even if the probability of error in the gate will be only 0.02 for this case. Classical gate operations are extremely reliable, even without error correction, with one error in 10^{17} operations. So, maybe it is more efficient to make the physical gates more reliable, instead of creating long circuits to correct errors.

In this case, with a 2 qubit circuit, each encoding needs 7 qubits, and the syndrome measurement needs 6 more qubits, giving a total of 20 qubits to correct Z and X errors. Most of the physical systems available right now, only have qubits in the order of 10 to 20. For example, the IBM computer only has 5 qubits available for the public (16 by code platform) and 20 qubits to clients. In most cases the quantum computer circuits have to be implemented without error correction. It's an area that needs further development. The memory used to store each qubit grows exponentially with 2^n, with n being the qubit number. So, the running time of the simulation increases compared to the version without error correction, because the circuit doesn't need to be compiled and is much smaller.

Regarding the time of computation, there isn't a visible relation between the probability of error and the time elapsed. All the computations took longer than 208 and less than 450 ms. The variation here was not significant, because only one run took more than 350 ms. And this small variation was because of some shifts in the system, namely the background tasks running in the CPU. The figure (5.12) shows that there is no relation between the error probability and the elapsed time for this case scenario.

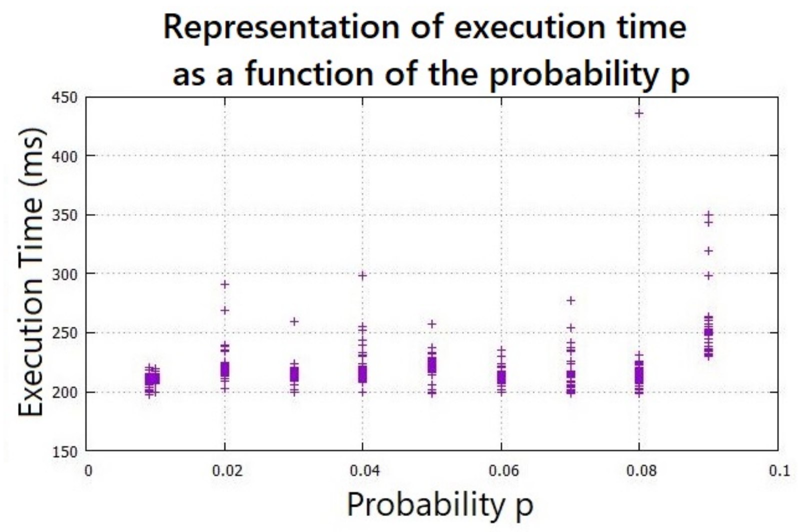


Figure 5.12: Representation of execution time (ms) as a function of the probability p.

5.5 Stabilizer Simulator

LIQ$Ui\,|\rangle$ has a different kind of simulation engine included, which can be the answer to some problems of memory and time that it takes to run the *steane*7 class. It is called stabilizers simulator. It has a different way to store and run circuits so it can give better results. They have the limitation of only working with a reduced list of gates and can't convert any others. The gates are the same ones as *steane*7, plus a few operations that may be useful, for example a controlled X gate [36]. Besides, the states need to be $|0\rangle$ or a state reachable within the gate list operations. This means the simulator is limited and only exists to analyse certain systems. Having the addition function as an example, in order to be implemented with this simulator it would need some serious change, including the derivation of controlled rotation gates into other gates from the list. This would make the circuit very complex to work with. However, the error correction is one of the things that can benefit from this simulator, because as the qubits are stored in a different way, it can handle a lot more qubits without expending the resources.

The stabilizer simulator uses the methods described in the article [46]. We have described its basic principles in previous sections. It's not the scope of this thesis to discussed these to their full extent. However, one of its benefits, for computation, is storing the qubit state in one tableau that can be read back. This is used by the simulator, and after the operations it gives a tableau representing the state of the system. Since it can handle thousands of qubits at once it can be exhausting trying to read and work with it.

As already discussed the stabilizer formalism, which includes the Steane 7 code, represents the state of one qubit by operators and not by amplitude. It must obey the condition that the state must be an eigenstate of the operator with eigenvalue +1. So, if the operators are represented in a tableau it may help significantly some simulations. This formalism, of writing the tableau of the unitary matrix that stabilizes the state, may not be the easiest to read or even the most efficient. It may take many bits to be stored classically, however, it can be simulated much faster. Plus, the system used by the LIQ$Ui\,|\rangle$ library uses an improved version of the stabilizer group. It changed the measurement operation so it only needs $O(n^2)$, instead of the previous n^3, this means it can handle more qubits faster. Noting that each generator needs to specify the state $|\psi\rangle$ expending $n(2n+1)$ bits, plus 2 bits for each n Pauli matrix and1 for the phase. The tableau will be representing the state stabilizer with binary values, so it can be faster for the simulation. When the system is asked for the state it will give a tableau (figure (5.13)) with the representation of the operators and a dot for each 0.

The code is available in the appendix A. It uses similar tools as the ones already described but redirected to the stabilizer simulator. The command *let stab = Stabilizer(err,s7b.Ket)* converts the circuit made with errors, into the simulator, and stores it in the variable *stab*. The *Stopwatch* needs to be reset and then the circuit is run in the stabilizer simulator. *ShowState* prints the tableau for the state, and after that, the various *show* lines, print data from the system.

The biggest advantage of the stabilizer simulator can be seen in the graph of the figure (5.14). It's the representation of the runtime in milliseconds of 100 simulations with the *steane*7 algorithm in the stabilizer simulator. Most of the runs had the same time, 2 ms, but there are some exceptions, a few spikes. These were related to the CPU and the operating system or to the fact that it was the first run. Actually, the simulation was initiated twice with 50 runs each, and each time the simulation started, the time was a bit longer, in the order of 25 ms. Two of the spikes in the graph, the first and the 51th are for this reason.

Those results are indeed a great improvement related to the non-stabilizer simulation, reminding that those times were in the order of hundreds of milliseconds, more specifically 208- 450 ms. These are 104 to 225 times bigger than the stabilizer results. So, it's an outstanding result. Note that the stabilizer simulation did the same operations as the *steane*7, it's the same circuit, only optimized in a different simulator. As discussed previously, the Steane 7 code is of the category of stabiliz-

```
+X...................
+.X..................
+..X.................
+...X................
+....X...............
+.....X..............
+.....XZ.............
+....X..Z............
+....XX..Z...........
+...X.....Z..........
+..........Y.........
+...........Y........
+............Y.......
+..X..X.......Z......
+.X..X.........Z.....
+.XX.XX.........Z....
+X..X............Z...
+X.XX.X...........X..
+XX.XX.............X.
+XXXXXX.............X
---------------------
+Z...............YYYY
+.Z............YY..YY
+..Z..........Y.Y.Y.Y
+...Z.....YYYY...YYYY
+....Z..YY..YY.YY..YY
+.....ZY.Y.Y.YY.Y.Y.Y
+......Y....YY.......
+.......Y..Y.Y.......
+........Y.YY........
+.........YYYY.......
+.......ZZZZ.........
+......Z.ZZ.Z........
+......ZZ.Z..Z.......
+.............X....XX
+..............X..X.X
+...............X.XX.
+................XXXX
+..............ZZZZ..
+.............Z.ZZ.Z.
+.............ZZ.Z..Z
```

Figure 5.13: Representation of the tableau for the state of the system after the error correction.

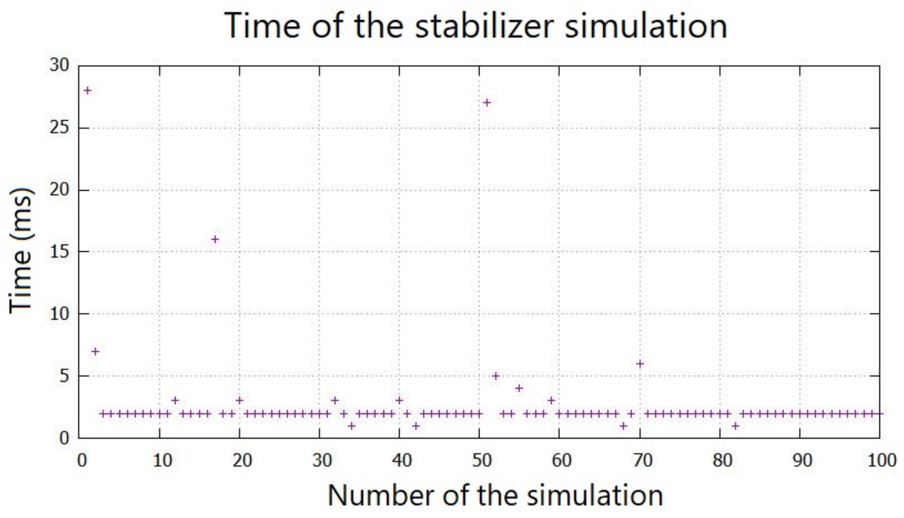


Figure 5.14: Representation of the time of 100 runs with the stabilizer simulator.

ers but can be defined without them. The LIQ$Ui\,|\rangle$ main simulator uses the class *steane*7 based in the stabilizer formalism. It prepares the circuit with this logic as described, but creates the circuit with the state amplitudes. The stabilizer simulator only works with the operators instead of the state, so it is working in the subspace of dimension 2.

The stabilizers are a very powerful simulation environment, however, are very limited, as proved. It's hard to retrieve the state to the normal simulator without any measures (decoding the state), so it becomes hard to do some kinds of circuits. Still, is a very interesting and useful tool in the LIQ$Ui\,|\rangle$ library. With some work in deriving gates, it can be used to simulate complex circuits with much less memory and computation time.

6

The IBM Q Experience

6.1 Introduction

The IBM Q Experience was open to the public in 2016 [13], and it was widely used during the year 2017, making it an excellent choice to do quantum experiments. This experience is based on the IBM Cloud. This means that we can write the code and run it in a simulator, or in the real quantum computer of IBM Research, via the internet platform. This platform has a circuit composer, which works by dragging the gates needed to the circuit, and then running it. At present it's also available a platform to write your own code. The Quantum Information Science Kit (QISKit), allows the use of the OpenQASM language based on Python.

Using the Composer is intuitive, but has many limitations, namely the reduced number of gates available. However, since they include the universal set of gates, we can decompose any gate into them. Writing the code may make this process easier.

At present, the IBM Q has 3 quantum computers available, two of them with 5 qubits, and one with 16 qubits. The latter is only accessible via the QISKit platform. There are two quantum chips with 20 qubits only accessible to partners of IBM, and in the near future, these might have 50 qubits. All of the chips are based on superconductor technology.

This is a unique opportunity to make some tests in a real quantum computer, so we implemented the addition algorithm in the Composer, in order to test it.

6.2 The Experience

Since the IBM Q uses a different language from LIQ$Ui\,|\rangle$, we will not go into the code writing process, that might be for a future work. We only want to see how the process works, and the results it gives. Having access to a 5 qubit quantum computer, we are limited to an addition of two numbers of two qubits each. This means that we are left with an extra qubit, so it's possible to make an addition with a carry bit. With two qubits and a carry bit available, we decided to implement the most common example of addition : $2 + 2 = 4$.

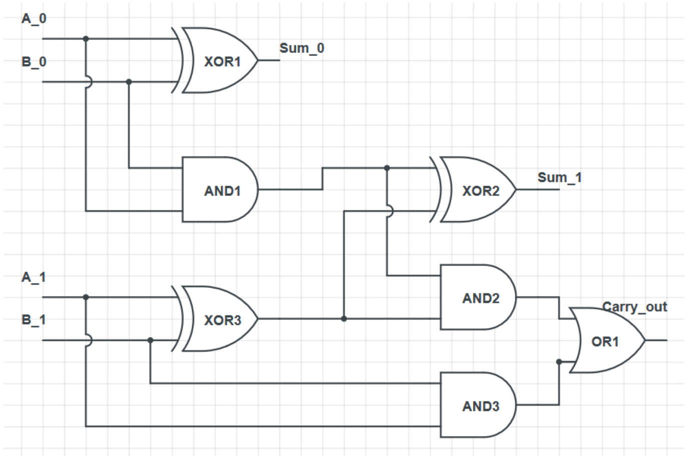


Figure 6.1: Representation of the circuit for the classic two bits adder.

Figure (6.1) shows the classic circuit for the addition of two bit numbers. One of the outputs will be the carry bit. This circuit uses only XOR and AND gates in a total of 7 gates to form a non-reversible circuit.

Using the quantum algorithm discussed, to make the addition of two numbers of two qubits, we get the circuit of the figure (6.2). This circuit has 9 gates, which is bigger than the classical one, however it's fully reversible. Besides, when comparing with a classic addition circuit, the gain of gates in the quantum addition circuit is only evident for a large number of qubits. So, up to now, the quantum circuit has more gates than the classic circuit, but using the improvement for the case we want, $2 + 2 = 4$, we can reduce it.

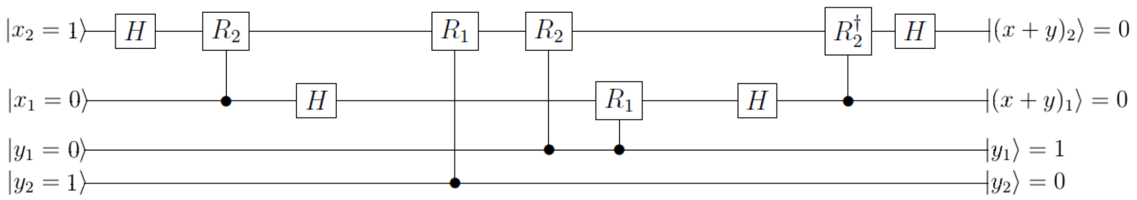


Figure 6.2: The circuit for the addition of two qubit numbers, following the quantum algorithm.

We want $x = 2$ and $y = 2$, so we have the quantum bits $|x\rangle = |10\rangle$, and $|y\rangle = |10\rangle$. This means that $y_1 = x_1 = 0$ and $y_2 = x_2 = 1$. With these input bits, the QFT part of the circuit will have no rotations, and the adder will only have the effect of the R_1. So there is no need to represent the gates that have no effect on the circuit. Note that the IBM Q Composer doesn't have the general R_n gates, however, we must remember that $R_1 = Z$ and $R_2 = S$ which are available. Thus, we can implement the shorter circuit to perform the operation $2 + 2 = 4$ in the IBM Q composer (see figure (6.3)). This circuit has only a total of 7 gates, and two of them are there to convert the state $|0\rangle$ into $|1\rangle$, since all the inputs in the IBM Q are the state $|0\rangle$. All the qubits must be measured, in contrast with the LIQ$Ui|\rangle$ simulation environment. It is important to note that, after improvement, the entries y_1 and y_2 could be omitted. However, it is instructive to keep them to evaluate the impact of errors on them in the real quantum computer circuit.

The addition $2+2 = 4$ written in binary is $10+10 = 100$. The result has three bits so the modular addition is expected to give $(2 + 2) \mod 4 = 0$.

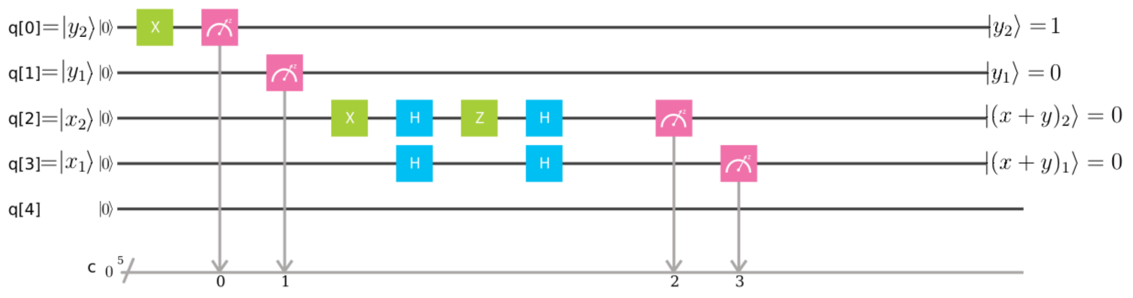

q[0]=$|y_2\rangle$ $|0\rangle$
q[1]=$|y_1\rangle$ $|0\rangle$
q[2]=$|x_2\rangle$ $|0\rangle$
q[3]=$|x_1\rangle$ $|0\rangle$
q[4] $|0\rangle$
c 0 5
X
H
Z
0
1
2
3
$|y_2\rangle = 1$
$|y_1\rangle = 0$
$|(x+y)_2\rangle = 0$
$|(x+y)_1\rangle = 0$

Figure 6.3: Circuit implementation for the modular addition of two qubit numbers. The X gate is used to convert $|0\rangle$ to $|1\rangle$. The qubit mapping of this circuit is made clear by the input and output labels. The bottom line, named C, contains the order of the output qubit register. In this case, from top to bottom, the qubit measurement register order is 0,1,2,3, which means that the top qubit will be the first to be measured, and the first entry bit of the result register. In other words, it will be the least significant bit of the result, that is the bit located at the far right in the output register. In this case, the output register is expected to be $|\, q[4]\ (x+y)_1\ (x+y)_2\ y_1\ y_2\rangle = |00001\rangle$.

Since the result of the addition $10+10 = 100$ has an extra qubit, we can implement it using a carry bit, as discussed in chapter 4 (figure (6.4)). This way, we can compute the usual addition $2 + 2 = 4$. The circuit using one carry bit was implemented using only gates that have an effect on the circuit, and is shown in the figure (6.5). Note that this circuit has a different order of the qubits, but it represents the same principle. Only the final register is affected by this order.

The carry bit, in figure (6.5) is represented by the qubit $q[2]$, that starts as $|x_3\rangle = |0\rangle$. In general, changing from modular addition to common addition, requires only one

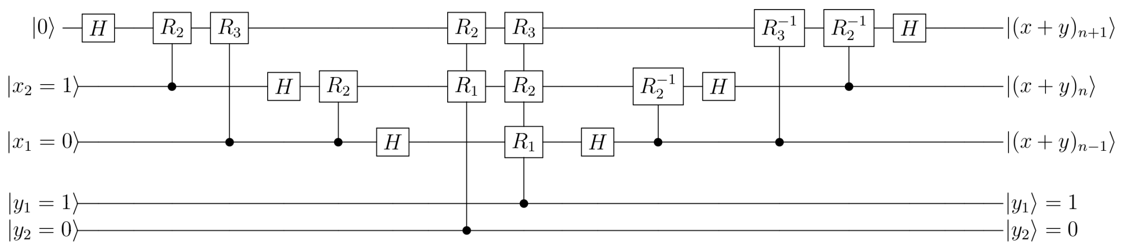
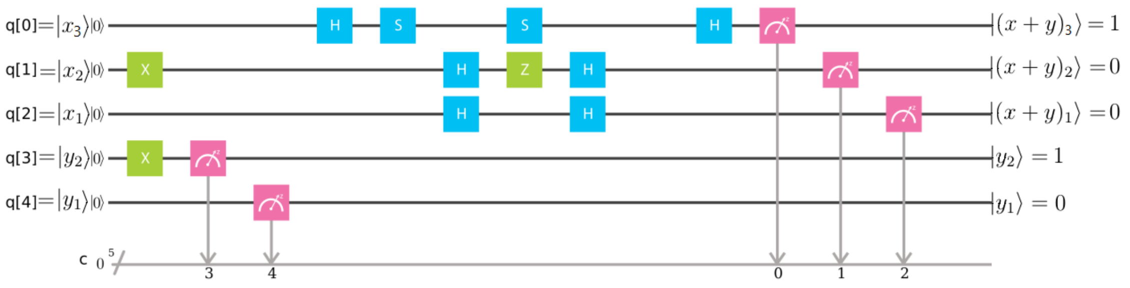


Figure 6.4: The circuit for the addition of two qubit numbers with carry bit, following the quantum algorithm for the use of carry bit.



q[0]=$|x_3\rangle|0\rangle$
q[1]=$|x_2\rangle|0\rangle$
q[2]=$|x_1\rangle|0\rangle$
q[3]=$|y_2\rangle|0\rangle$
q[4]=$|y_1\rangle|0\rangle$
c 0 5
X
H
S
Z
3
4
0
1
2
$|(x+y)_3\rangle = 1$
$|(x+y)_2\rangle = 0$
$|(x+y)_1\rangle = 0$
$|y_2\rangle = 1$
$|y_1\rangle = 0$

Figure 6.5: Implementation of the addition circuit with carry bit. The qubit q[0] is the carry bit: its input state is $|0\rangle$, the output will be $|(x+y)_3\rangle = 1$. The output register is expected to be $|(x+y)_1\ (x+y)_2\ (x+y)_3\ y_1\ y_2\rangle = |01001\rangle$.

extra line in the circuit, disregarding the number of entry qubits. This extra line will be on top of the QFT, adder and IQFT lines, converting a n qubit circuit into a $n+1$ circuit, but forcing the additional entry qubit to be $|x_{n+1}\rangle = |0\rangle$. The $|y_{n+1}\rangle = |0\rangle$ is not necessary since y qubits are used for control and a gate controlled by $|0\rangle$ has a null effect.

6.3 Results

Before running the circuits in the quantum computer, we have to run them in the IBM simulator to check for errors. Since there is a waiting queue to do the simulations, this process saves time.

The results for the simulation of modular addition $(2+2) \mod 4$ are in figure (6.6). The expected result is $(2+2) \mod 4 = 0$. As explained in the caption of figure (6.3), this should correspond to an output register 00001 with probability 100%, as observed in the simulation. The results of the quantum computer for the same circuit are in figure (6.7). The outcome is similar, but with noise. The register is 00001 with a probability of less than 75%. It's important to note that this is a real quantum computer with no error correction code implemented, so it's expected

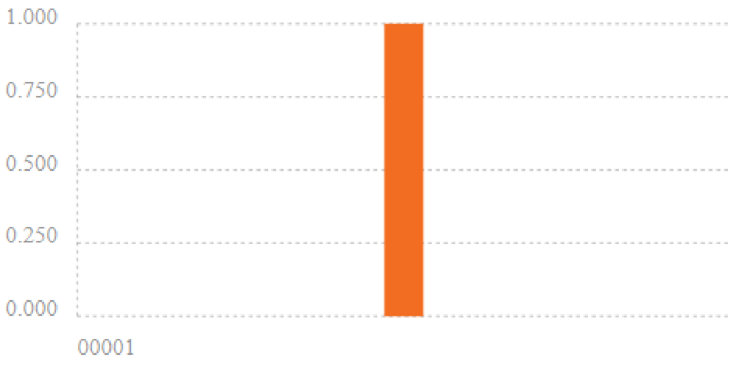


Figure 6.6: Output for the simulation of modular addition. The y-axis represents the probability of getting the bit register in the state shown in the x-axis.

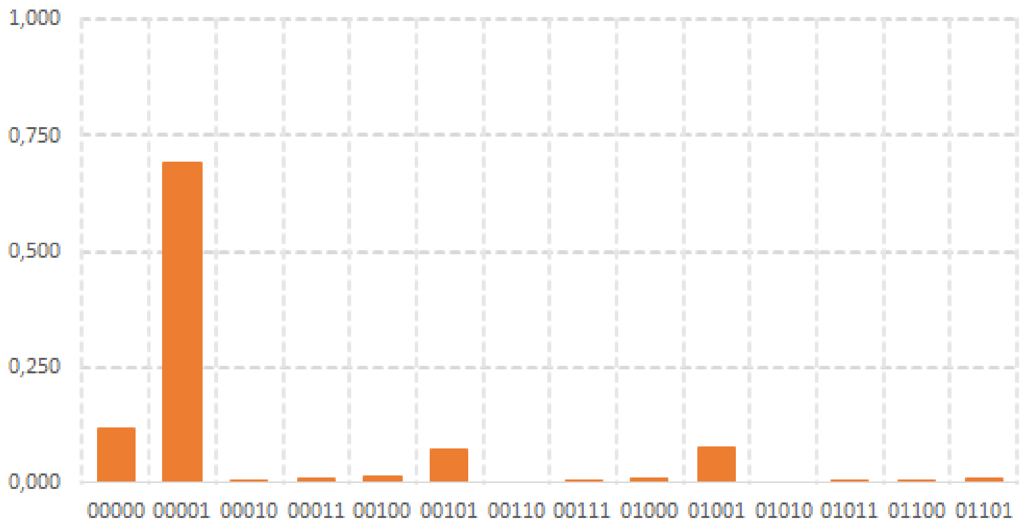


Figure 6.7: Output for the results of the modular addition circuit in a real quantum computer. The y-axis represents the probability of getting the bit register in the state shown in the x-axis.

to have noise in the system. All the states are measured at the end of the circuit. While the simulator engine measures the circuit 100 times, the quantum computer runs and measures the circuit 1024 times, which is a much larger number.

It is interesting to analyse which are the most frequent errors. Considering the outputs with error, we can say the most likely errors are the ones that affect only one qubit. This means they can be corrected by a QEC code, as discussed earlier. The corresponding register strings for the biggest error probability are, ordered from the higher to the lower error probability: 00000 (error in y_2), 01001 (error in $(x+y)_1$) and 00101 (error in $(x+y)_2$) . The observed very low probability of one single error in the bit y_1 (output string 00011) is not surprising, since there is no gate operation on y_1. But there seems to be no simple correlation between the error rate and the gate number. The most frequent error is one single error in y_2, which is an entry submitted to only one gate operation. These errors that affect only one qubit have a total probability of approximately 0.27.

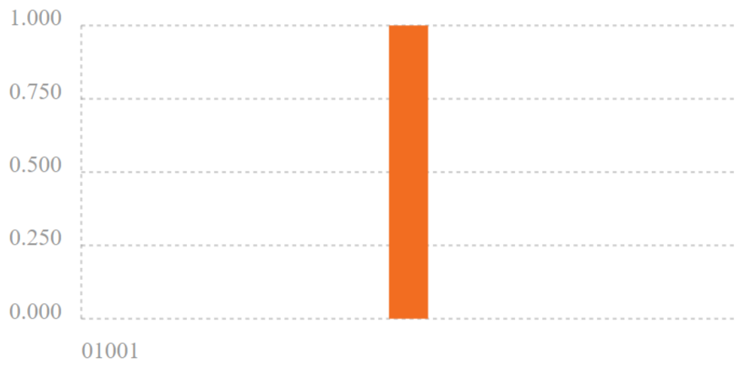


Figure 6.8: Output for the simulation of addition with carry bit. The y-axis represents the probability of getting the bit register in the in the state shown in the x-axis.

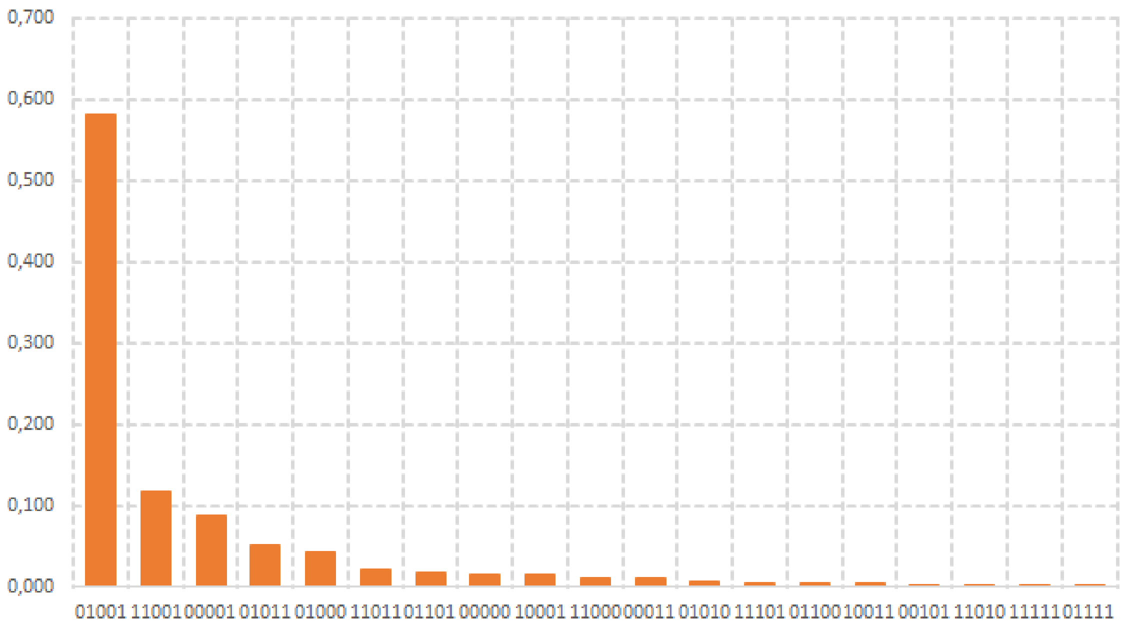


Figure 6.9: Output for the results of the circuit for addition with carry bit in a real quantum computer. The y-axis represents the probability of getting the bit register in the state shown in the x-axis.

We expect that an addition $2 + 2$ gives a result of 4 when the carry bit is included. The output for the number y will be the same as before, $y_2 = 1$ and $y_1 = 0$. For the number x, we expect $(x+1)_3 = 1$, $(x+1)_2 = 0$ and $(x+1)_1 = 0$, which corresponds to an output register 01001 (see figure (6.5) for the bit order). Figure (6.8) presents the results of the simulation, where it gives 100% probability of getting an output register 01001, as expected. The figure (6.9) has the results for the real quantum computer, which gives a smaller probability for the expected register output (around 58%).

As in the previous case, it is interesting to analyse which are the most likely errors. The most significant register with errors, in order, are: 11001 (error in y_1); 00001 (errors in y_2); 01011 (error in $(x+y)_2$); and 01000 (error in $(x+y)_3$). These four register strings together have a probability of approximately 30%. Concerning the group of four registers discussed, we can see that the error that appears more often is an error that affect only one qubit. Errors that affect more qubits have much less probability (at least half probability of the lower error register). In contrast with the previous case, now the qubit with fewer operations (y_1) has the biggest error, and the qubit with more operations ($(x+y)_3$), has the lower error.

6.4 Conclusions

The success rate of the computation in the modular addition was around 75% and in the addition with carry bit was around 58%, which is smaller. This is considerably high error rate in the results suggest that there are errors related with gate operations. However, our results do not allow to extract any simple correlation between the error rate and the gate number or the type of gate. It is possible that the the lack of stability in this quantum computer also plays a role, as suggested in the literature [47]. These quantum computers are fairly new, and need to improve the overall stability of the system. However, it's a good sign that most of the registers with error, only had one qubit affect by the error. It means they could eventually be corrected.

The IBM Q system does not give constant results over time. I.e., doing one simulation at two different occasions, may give entirely different results. Besides, we found that when changing the order of the qubits (figure (6.4), the results would vary. This means the qubits of the system are not equivalent.

Although the case $2+2=4$ is of no practical interest, it was an opportunity to see how the algorithm works in a real quantum computer. We could get a result quickly unless there were many people in the waiting queue. However, the results do have a significant error rate.

This was a real application to a real system, so it was good to see how the optimizations we made in the modular addition section worked well. Using classical computers to simulate quantum circuits allows us to create a simplified version before implementing the circuits in a real quantum computer. In this case, we only applied the gates that had some impact on the system, ignoring the ones controlled by $|0\rangle$. That simplification meant that we didn't have to apply large rotations.

The presence of errors in the real system is the reason why we need QEC codes. However, we only had 5 qubit entries. In order to implement a simple QEC code in a 2 qubit system, we need a total of 20 qubits. QEC codes need to be optimized in order to use less qubits. The QEC implementation in this circuits, with mostly one qubit errors, may be of great help.

Since the Composer also generates the QASM code, we post it here for the modular addition case, to complete this section. It's pretty straightforward to follow.

```
IBMQASM 2.0;
include "qelib1.inc";
qreg q[5];
creg c[5];
x q[0];
measure q[0] -> c[0];
measure q[1] -> c[1];
x q[2];
h q[2];
h q[3];
z q[2];
h q[2];
h q[3];
measure q[2] -> c[2];
measure q[3] -> c[3];
```

7 Conclusions

This work was structured in four different chapters, starting with a simple test on Shor's algorithm, and evolving to the core of this thesis, which is the simulation of a quantum algorithm for modular addition. After that, there was the study of how QEC works. Finally, a real implementation in the IBM Q quantum computer. The improvements in the modular addition circuit led to very good results, and there were some interesting insights emerging from this work.

The test of Shor's algorithm showed the algorithm runs in a classical computer simulation with a $\mathbf{O}(b^n)$ behaviour, a lower performance than predicted by the theory. An important conclusion of this test was that Shor's algorithm does not perform in a polynomial way but in an exponential way, when running on a classical computer. The theoretical comtyplexity class of Shor's algorithm is based on gate number, whereas the test evaluates the time of execution. Since Shor's algorithm works with qubits, which belong to a Hilbert space of dimension 2^n, in order to simulate the system in a classical computer all the 2^n amplitudes of the states need to be stored, causing an exponential increase of the execution time. When using a real quantum computer, instead of a simulation on a classical computer, this exponential time increase is not expected.

The quantum modular addition using QFT increases the performance of addition, when compared with the traditional addition algorithm. The simulation results show that the number of gates behaves as expected in theory, described by $\mathbf{O}(n^2)$. However, improving the circuit by disregarding gates that are not used in specific cases, can lead to a $\mathbf{\Omega}(n)$ behaviour in the best case scenario. The change in time, from the best to the worse optimized simulation, is an order of magnitude, although both have an exponential behaviour with execution time. This difference is clearly evident in the result graphs of running time as a function of the number of qubits.

Results also showed that the simulation execution time also depends on LIQ$Ui\,|\rangle$ operation mode. Circuit mode is faster than function to run a small number of qubits, and complex circuits. However, this difference is not very significant most of the times. The optimization of the circuit will always have the best results, especially for the most arduous simulations, making it the better choice.

Improving the circuit by identifying the cases where some gates are not necessary and removing them in those scenarios, leads to an interesting conclusion. Our performance results show that improving the circuit this way, followed by an optimization in LIQ$Ui\,|\rangle$, leads to running time reductions between 90% to 97%. A simulation that used to take 6 min to run, now takes about 37 seconds. These results are impressive and make simulation possible with a much larger number of qubits if there is enough memory available.

The LIQ$Ui\,|\rangle$ way of implementing QEC codes only supports circuits with simple operations, and the results show that we need 20 qubits to implement QEC codes in a two-qubit circuit. Using a version of a two-qubit QFT with QEC tells us that the system can effectively correct errors with a probability of 0.02 with 95% success rate. This ensures that the final results of computations are correct, but it makes the circuit so big it may not justify QEC implementation, if the system fidelity is reasonable. The stabilizers environment is very powerful but has a limited number of available gates. It can handle many qubits with very small simulation times.

Implementing a circuit to perform the basic $2 + 2$ operation in a real quantum computer allowed us to demonstrate the benefit of disregarding gates that are not used in specific cases. As the quantum computer lacks stability and gate fidelity, adding more gates to the system would increase errors. Implementing QEC codes would need a quantum computer with more qubits than the ones available today. The implementation without the QEC codes was an opportunity to observe the error rate in a real implementation, and its effects, which is a valuable result by itself.

This thesis was a first experience in quantum computing programming. Using the LIQ$Ui\,|\rangle$ environment was a good experience. Since programming a quantum computer is a very different experience from a classical one, it required learning its logic, including working with entangled state results. There was the need to create several functions, including those necessary for QFT, IQFT in order to implement the algorithm. Only a reduced set of built-in gates from the library was used, even the controlled rotation needed to be adapted from functions available in the platform. This was a valuable experience that will be useful in the future. The results of the work were good, the fact that a quantum circuit can be improved considering specific cases, using the classical programming and the system input information is a very valuable lesson. The fact that quantum computers are always dependent on a classical system can be used to optimize operations in a quantum computer.

7.1 Future Works

This work may be used in the future as a starting point experience in quantum computation. Using the experience in improving circuits, there is the possibility of trying to implement it to more complex systems.

Basic operations were studied and implemented in this thesis, and now algorithms can be explored to solve problems using them. For example, as discussed in reference [11], there is the possibility to implement algorithms that use the same principle as the ones we studied here. Quantum circuits for multiplications and weighted averages using QFT are two examples of circuits that could be explored, but it can easily be extended to the large class of problems that use QFT. Understanding how to improve this circuits using a classical computer, as we did in this thesis, is also important. Machine learning uses weighted averages and may benefit from this kind of algorithms optimization.

For the future work in this field, the kind of problems discussed above may be explored, with an implementation in available quantum computers like IBM Q. There is also the need to explore new languages, as, for example, the new Q# from Microsoft. We already know that LIQ$Ui\,|\rangle$ is great for simulations, so probably this next generation languages will be even better. Using quantum computers will also raise the need to find a good algorithm to implement QEC codes using the minimum possible number of qubits.

This area of investigation is vast and in great expansion. There is a lot of work to be done, and this is the perfect point to start working in this field, namely in problems like machine learning. Nevertheless, there are other areas that could be explored as well, namely quantum cryptography, which is used for cryptocurrency. This is a hot topic at the moment, and there are already some applications using quantum computer algorithms [48].

Bibliography

[1] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information.* Cambridge University Press, 2000.

[2] F. T. Chong, D. Franklin, and M. Martonosi, "Programming languages and compiler design for realistic quantum hardware," *Nature*, vol. 549, no. 7671, p. 180, 2017.

[3] A. Kandala, A. Mezzacapo, K. Temme, M. Takita, M. Brink, J. M. Chow, and J. M. Gambetta, "Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets," *Nature*, vol. 549, pp. 242–246, Sept. 2017.

[4] S. Debnath, N. M. Linke, C. Figgatt, K. A. Landsman, K. Wright, and C. Monroe, "Demonstration of a small programmable quantum computer with atomic qubits," *Nature*, vol. 536, pp. 63–66, Aug. 2016.

[5] B. Lekitsch, S. Weidt, A. Fowler, K. Mølmer, S. J. Devitt, C. Wunderlich, and W. Hensinger, "Blueprint for a microwave trapped ion quantum computer," *Science Advances*, vol. 3, p. e1601540, 02 2017.

[6] A. Paler, A. G. Fowler, and R. Wille, "Reliable quantum circuits have defects," *XRDS*, vol. 23, pp. 34–38, Sept. 2016.

[7] O. Bernhard, "A procedural formalism for quantum computing," Master's thesis, Technical University of Vienna, 1998.

[8] B. Öemer, *Structured quantum programming.* PhD thesis, Vienna University of Technology, 2003.

[9] J. Miszczak, "Models of quantum computation and quantum programming languages," *Bulletin of the Polish Academy of Sciences: Technical Sciences*, vol. 59, Jan 2011.

[10] K. M. Svore, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, A. Paz, and M. Roetteler, "Q#: Enabling scalable

quantum computing and development with a high-level domain-specific language," ACM, February 2018.

[11] L. Ruiz-Perez and J. C. Garcia-Escartin, "Quantum arithmetic with the quantum fourier transform," *Quantum Information Processing*, vol. 16, pp. 1–14, June 2017.

[12] T. Häner, M. Roetteler, and K. M. Svore, "Factoring using 2n+2 qubits with Toffoli based modular multiplication," *ArXiv e-prints*, 2017.

[13] IBMCorporation, "Ibm makes quantum computing available on ibm cloud." `https://www-03.ibm.com/press/us/en/pressrelease/49661.wss`, acessed in 2018.

[14] press realease, "Alibaba cloud and cas launch one of the world's most powerful public quantum computing services." `https://www.alibabacloud.com/press-room/alibaba-cloud-and-cas-launch-one-of-the-worlds-most`, acessed in 2018.

[15] Xinhua, "China's 712-km quantum communication line put into use." `http://www.xinhuanet.com/english/2016-11/20/c_135844035.htm`, acessed in 2018.

[16] J. Hsu, "Ces 2018: Intel's 49-qubit chip shoots for quantum supremacy." `https://spectrum.ieee.org/tech-talk/computing/hardware/intels-49qubit-chip-aims-for-quantum-supremacy`, acessed in 2018.

[17] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134, Nov 1994.

[18] V. Vedral, A. Barenco, and A. Ekert, "Quantum networks for elementary arithmetic operations," *Phys. Rev. A*, vol. 54, pp. 147–153, Jul 1996.

[19] P. Gossett, "Quantum Carry-Save Arithmetic," *eprint arXiv:quant-ph/9808061*, Aug. 1998.

[20] T. Draper, "Addition on a Quantum Computer," *eprint arXiv:quant-ph/0008033*, Aug. 2000.

[21] C. P. Williams and S. H. Clearwater, *Ultimate zero and one: computing at the quantum frontier*. Copernicus, 2000.

[22] D. Deutsch, "Quantum theory, the church–turing principle and the universal quantum computer," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 400, no. 1818, pp. 97–117, 1985.

[23] B. Zwiebach, "Multiparticle states and tensor products," *8.05 Quantum Physics II MIT Lecture Notes*, 2013.

[24] S. Lloyd, "Quantum information science," *MIT*, 2009.

[25] Y. Wang, S. Yan, and H. Zhang, "A new quantum algorithm for computing rsa ciphertext period," *Wuhan University Journal of Natural Sciences*, vol. 22, no. 1, pp. 68–72, 2017.

[26] P. Kaye, R. Laflamme, and M. Mosca, *An introduction to quantum computing.* Oxford University Press, 2007.

[27] J. Watrous, "Quantum Computational Complexity," *ArXiv e-prints*, Apr. 2008.

[28] S. Aaronson and G. Kuperberg, "Complexity zoo." `https://complexityzoo.uwaterloo.ca/Complexity_Zoo`, accessed in 2018.

[29] C. H. Papadimitriou, *Computational complexity.* Addison-Wesley, 1996.

[30] R. Feynman, "Simulating Physics with Computers," *International Journal of Theoretical Physics*, vol. 21, pp. 467–488, June 1982.

[31] J. R. Powell, "The quantum limit to moore's law," *Proceedings of the IEEE*, vol. 96, pp. 1247–1248, Aug 2008.

[32] D. Wecker, B. Bauer, B. K. Clark, M. B. Hastings, and M. Troyer, "Gate-count estimates for performing quantum chemistry on small quantum computers," *ArXiv e-prints*, vol. 90, p. 022305, Aug. 2014.

[33] S. Devitt, "Programming quantum computers using 3-D puzzles, coffee cups, and doughnuts," *ArXiv e-prints*, Sept. 2016.

[34] J. Brandhorst-Satzkorn, *A Review of Freely Available Quantum Computer Simulation Software.* PhD thesis, LiTH-MAT-EX, 2012.

[35] D. Wecker, K. M. Svore, and K. M. Svore, "Liquid: A software design architecture and domain-specific language for quantum computing." February 2014.

[36] D. Wecker, "Quantum architectures and computation liquid users manual," *Microsoft Corporation*, 2016.

[37] C. Barnes, "Integer factorization algorithms," *Oregon State University*, 2004.

[38] J. M. Pollard, “A monte carlo method for factorization,” *BIT Numerical Mathematics*, vol. 15, pp. 331–334, Sep 1975.

[39] A. Milne and A. Simmons, “Solving maximally-constrained 1-sat problems with oracular access,” *Liquid Quantum Challenge*, May 2016.

[40] S. M. Hamdi, S. T. Zuhori, F. Mahmud, and B. Pal, “A compare between shor’s quantum factoring algorithm and general number field sieve,” in *Electrical Engineering and Information & Communication Technology (ICEEICT), 2014 International Conference on*, pp. 1–6, IEEE, 2014.

[41] D. Bouwmeester, A. Ekert, A. Zeilinger, *et al.*, *The physics of quantum information*, vol. 3. Springer, Berlin, 2000.

[42] S. J Devitt, W. Munro, and K. Nemoto, “Quantum error correction for beginners,” *Reports on Progress in Physics*, vol. 76, p. 076001, 06 2013.

[43] A. O. Pittenger, *An introduction to quantum computing algorithms*. Birkhauser, 2000.

[44] P. Majek, “Quantum error correcting codes,” Master’s thesis, Comenius University, Bratislava, 2005.

[45] A. M. Steane, “Overhead and noise threshold of fault-tolerant quantum error correction,” *Phys. Rev. A*, vol. 68, p. 042322, Oct 2003.

[46] S. Aaronson and D. Gottesman, “Improved simulation of stabilizer circuits,” *Phys. Rev. A*, vol. 70, p. 052328, Nov 2004.

[47] D. Alsina and J. I. Latorre, “Experimental test of Mermin inequalities on a five-qubit quantum computer,” *Physical Review A*, vol. 94, p. 012314, July 2016.

[48] D. Aggarwal, G. K. Brennen, T. Lee, M. Santha, and M. Tomamichel, “Quantum attacks on Bitcoin, and how to protect against them,” *ArXiv e-prints*, Oct. 2017.

Appendices

A

Modular addition

A.1 Code for simple modular addiction

```
let modular (k:Ket, b:Ket)=
        let sw=System.Diagnostics.Stopwatch()
        let _ =k.Single()
        let qs=k.Qubits
        let __=b.Single()
        let bb=b.Qubits
        show "----------------before␣x="
        k.Dump(showInd)
        show "\n--------------------------------y="
        b.Dump(showInd)
        sw.Start()
        qft qs

        M >< bb
        let n=qs.Length
        for i in 0 .. (n-1) do
            let cb=bb.[i]
            for j in 1 ..(n-i) do
                BC (R (j)) [bb.[j-1+i];qs.[i]]

        qfti qs
        sw.Stop()
        show"---------------------␣after␣iqft"
        k.Dump(showInd)
        show "\nTime␣elapsed=␣%d␣ms" ( sw.ElapsedMilliseconds)
```

A.2 Code for circuit mode for modular addition with optimizations

```
let modularcom (k:Ket, b:Ket)=
        let sw=System.Diagnostics.Stopwatch()
        let _ =k.Single()
        let qs=k.Qubits
        let __=b.Single()
        let qb=b.Qubits
        let qf=Circuit.Compile qft qs
        let qfi=Circuit.Compile qfti qs
        show "---------------before x="
        k.Dump(showInd)
        show "\n-------------------------------y="
        b.Dump(showInd)

        M >< qb
        let n=qs.Length
        let sum (q:Qubits)=

            k.Dump(showInd)
            //show" qb s=%i" qb.Length
            for i in 0 .. (n-1) do
                let cb=qb.[i]
                for j in 1 ..(n-i) do
                    BC (R (j)) [qb.[j-1+i];q.[i]]

        let sumc=Circuit.Compile sum qs
        let CF=Circuit.Seq [qf;sumc;qfi]
        let numgates= CF.Fold(true).GateCount()
        CF.Fold().RenderHT("Sumf.html")
        show "Gate Count= %i"numgates
        show "Optimizing----------"
        let CFG=CF.GrowGates(k)
        sw.Start()
        CFG.Run qs
        sw.Stop()
        let numgates= CFG.Fold(true).GateCount()
        CFG.Fold().Render("SUMF.html")
        show "\nTime elapsed= %d ms" ( sw.ElapsedMilliseconds)
        show "Gate Count= %i"numgates
        show"--------------------- after iqft"
        k.Dump(showInd)
```

A.3 Full Modular Addition Circuits

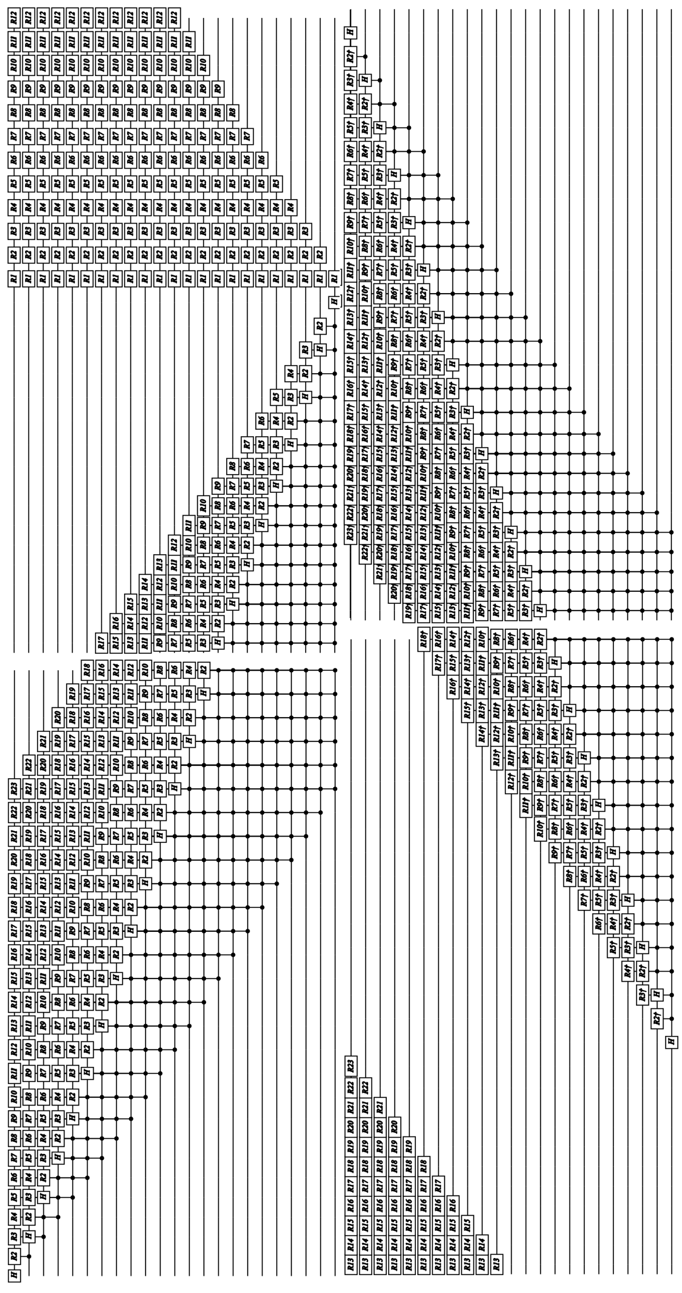


Figure A.1: The full circuit for modular addition for the case with y=Ones, without any optimization nor improvements. n=23 qubits. Horizontal orientation.

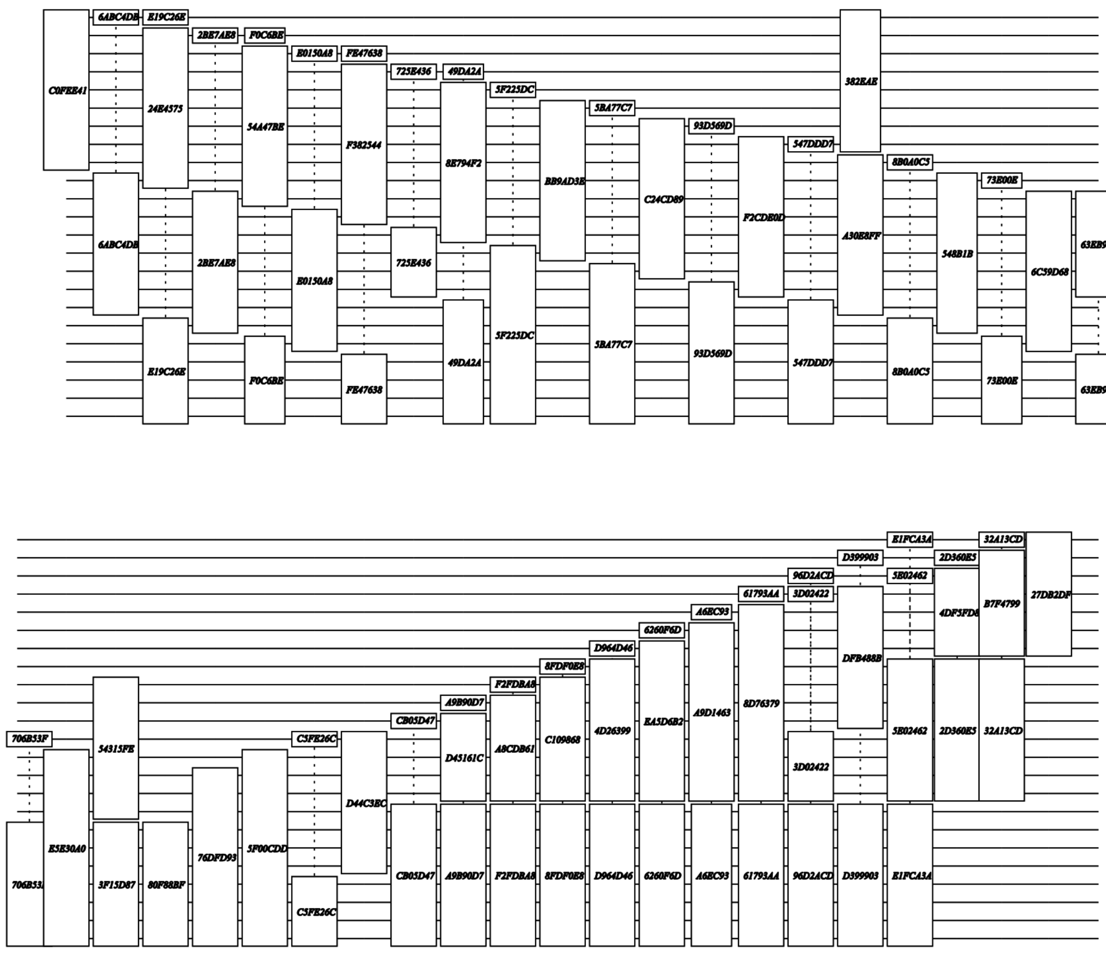


Figure A.2: Full circuit for the modular addition with n=23 qubits and the case with y=Ones. It includes optimization, so we can see that it needs much less gates. And that the gates names have a proper organization, the circuit symmetry is similar to the one without optimization.

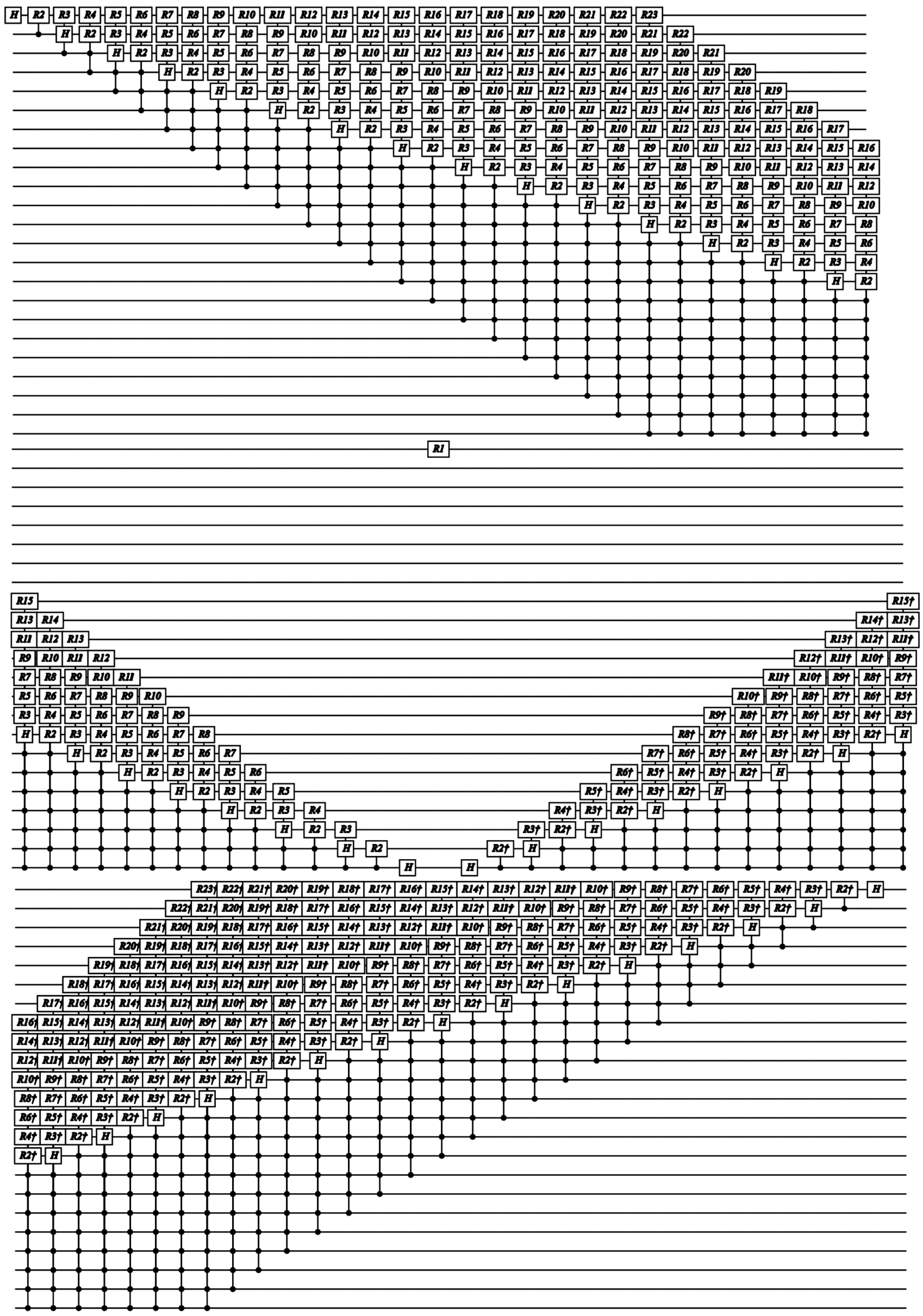


Figure A.3: Full circuit for the modular addition with 23 qubits and for the case with $y = 2^{n-1}$. We can verify that there is only one rotation controlled by y applied to the circuit. The QFT and IQFT controlled rotations are all applied, even though $x = 2^{n-2}$.

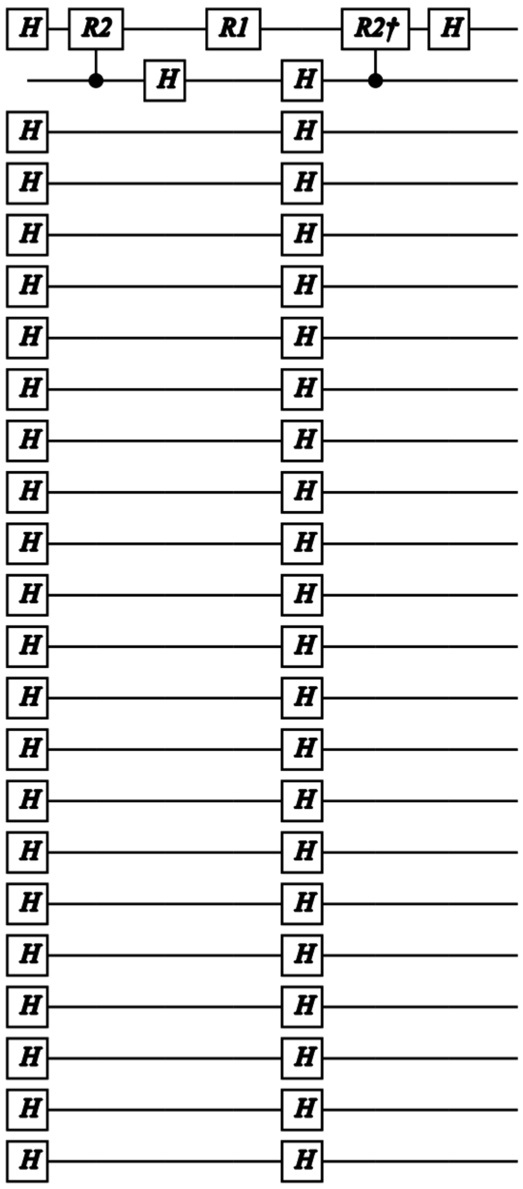


Figure A.4: The improved circuit with 23 qubits for the case where $y = 2^{n-1}$ and $x = 2^{n-2}$. We can see that all the non-essential gates were deleted. Comparing to the circuit of the figure (A.3), we can see exactly the improvements done, in the QFT and IQFT parts.

B

Quantum Error Correction

B.1 Full QEC code

```
let QEC (k:Ket, b:Ket, p:float , v:Boolean)=
    let sw=System.Diagnostics.Stopwatch()
    let _ =k.Single()
    let qs=k.Qubits
    let __=b.Single()
    let bb=b.Qubits
    if (v=true) then
    show "----------------------before x="
    k.Dump(showInd)
    show "\n----------------------------------------------------y="
    b.Dump(showInd)
    let qc=Circuit.Compile tqft qs
    let qb=Circuit.Compile tqft bb
    if (v=true) then
        qc.Fold().Render("qft.html")
    let ccnt=7
    let s7=Steane7(qc)
    let s7b=Steane7(qb)
    //let tt=Transverse 7 (CR 2.) qs
    let pr=s7.Prep
    let si=Ket(7).Qubits
    let qfts7= Circuit.Compile pr si
    //qfts7.Dump(showInd)
    //qfts7.Fold().Render("qfts7prep.html")
    let sy=s7.Syndrome
    let acc=Ket(6+7)
    let cc=Circuit.Compile sy acc.Qubits
    //cc.Fold().Render("qecc_cc.html")
    //let ns7= s7.Replace (H qs)
    let s7f= s7.Circuit
    let s7fb=s7b.Circuit
```

```
                let err,stats=s7b.Inject p
                if (v=true) then
                        err.Fold().Render("qfterrors.html")
                s7f.Run s7.Ket.Qubits
                sw.Start()
                err.Run s7b.Ket.Qubits
                sw.Stop()
                show "------------------------------------------FINAL"
                show "\n--------------x="
                s7.Ket.Dump(showInd)
                show "\n--------------y="
                s7b.Ket.Dump(showInd)
                show "\n␣Injected␣errors:"
                show "X␣gates:␣%i"stats.[0]
                show "y␣gates:␣%i"stats.[1]
                show "z␣gates:␣%i"stats.[2]
                show "fixes␣needed:␣%i" s7b.NumFixed
                show "gate␣number=␣%5d" (err.GateCount(true))
                show "time␣elapsed=␣%dms" ( sw.ElapsedMilliseconds)
                if (v=true) then
                        s7f.Fold().Render("qecf.html")
                        let stab= Stabilizer(err,s7b.Ket)
                        sw.Reset()      4
                        sw.Start()
                        stab.Run()
                        sw.Stop()
                        stab.ShowState showInd 0
                        let bit0=s7b.Log2Phys 0 //|> s7b.Decode
                        let bit1=s7b.Log2Phys 1 //|> s7b.Decode
                        let _,bi0=stab.[0]
                        let _,bi1=stab.[1]
                        show"qubit␣0=\n"
                        (bi0.Dump(showInd))
                        for q in bit0 do
                        show "%s" (q.ToString())
                        show"qubit␣1=\n"//(bit1.ToString())
                        for q in bit1 do
                        show "%s" (q.ToString())
                        show "\n␣Injected␣errors:"
                        show "X␣gates:␣%i"stats.[0]
                        show "y␣gates:␣%i"stats.[1]
                        show "z␣gates:␣%i"stats.[2]
                        show "fixes␣needed:␣%i" s7b.NumFixed
                        show "time␣elapsed=␣%dms" ( sw.
                         ElapsedMilliseconds)
```

B.2 Full circuit of error correction

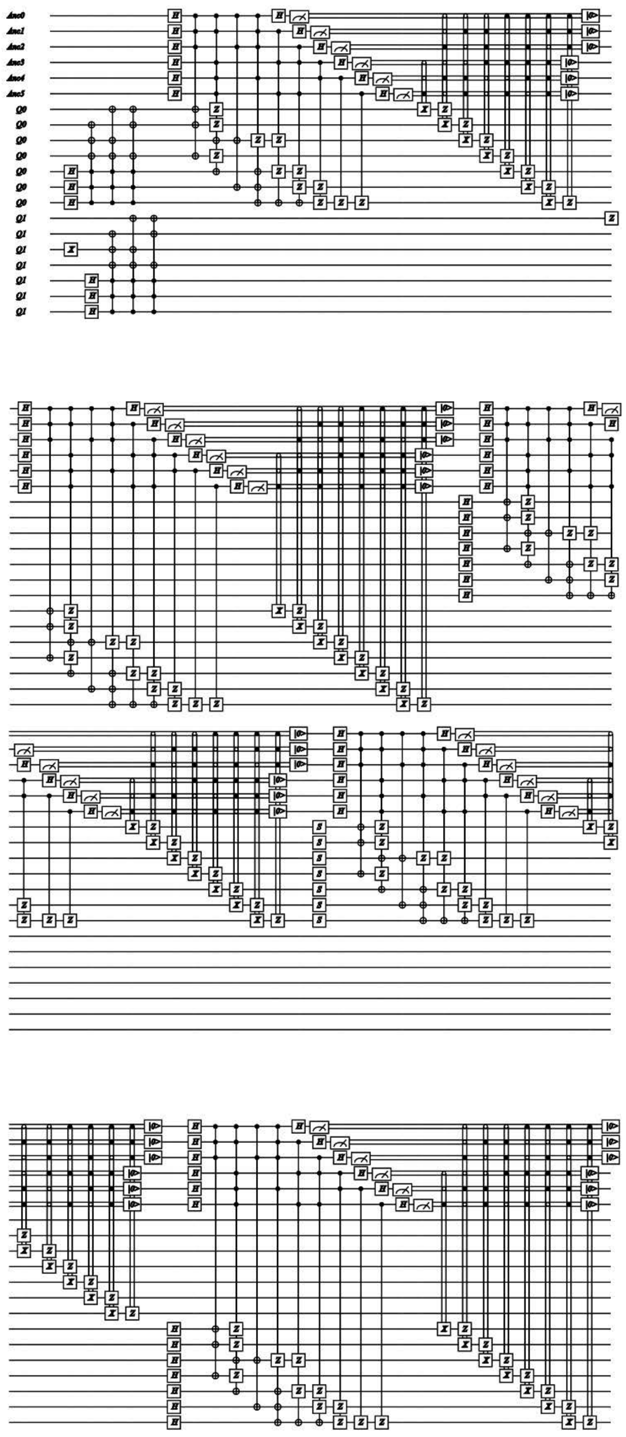


Figure B.1: Full circuit for the Steane7 error correction code, with probability or errors of 0.009

B.3 Tables

Sucess of correction	X gate error	Y gate error	Z gate error	Fixes done (cumulative)	Number of gates
0	2	0	1	2	358
0	0	0	0	2	355
0	1	1	1	5	358
0	0	0	0	5	355
0	0	0	1	6	356
0	0	0	0	6	355
0	0	0	0	6	355
0	1	0	0	7	356
0	0	0	0	7	355
0	0	0	0	7	355
0	1	0	0	8	356
0	0	1	0	9	356
0	1	0	0	10	356
1	0	2	0	12	357
0	0	0	0	12	355
0	0	0	0	12	355
0	0	0	0	12	355
0	0	0	0	12	355
0	0	0	0	12	355
0	1	1	0	14	357
0	0	0	0	14	355
0	0	0	0	14	355
0	0	0	0	14	355
0	0	0	0	14	355
0	0	1	1	17	357
0	1	0	2	19	358

0	0	0	1	20	356
0	0	0	0	20	355
0	1	0	2	22	358
0	3	0	0	25	358
0	0	1	0	27	356
0	1	0	0	29	356
0	0	0	1	30	356
0	0	0	0	30	355
0	0	0	1	30	356
0	1	0	0	31	356
0	0	1	1	33	357
0	0	0	1	34	356
0	1	0	0	35	356
0	0	1	0	36	356
0	0	0	0	36	355
0	0	0	0	36	355
0	0	0	0	36	355
0	0	0	0	36	355
0	0	0	0	36	355
0	0	0	0	36	355
0	0	0	0	36	355
0	0	0	0	36	355
0	0	0	0	36	355
0	0	0	0	36	355

Table B.1: Table with the data from the simulation run with 0.009 probability. The success of the correction gives the value 1 if it was unsuccessful and 0 if it was successful. The fixes done are cumulative.

C

Computer Characteristics

CPU	Intel Core i7-4710MQ
Speed	2.5GHz-3.5 Ghz (Overclock)
RAM	1 slot 8 GB DDR3 (1333MHz)
Available RAM	$\sim 5.9GB$
Graphic Card	Intel HD Graphics 4600
Dedicated Graphic Card	NVIDIA GeForce GTX 960 M
Memory	2.0 GB
Core Speed	1097 MHz
Hard Drive	SSD Micron M600 128 GB
Sequential Read	560 Mb/s
Sequential Write	400 Mb/s