Delft University of Technology
Faculty of Electrical Engineering, Mathematics & Computer Science
Delft Institute of Applied Mathematics

Implementation and Analysis of an Algorithm on Positive Integer Addition for Quantum Computing

Report on behalf of
Delft Institute of Applied Mathematics
as part of the acquisition

of the degree

BACHELOR OF SCIENCE
in
APPLIED MATHEMATICS

by

M.A.J. LOOMAN

Delft, Netherlands
August 2018

BSc project APPLIED MATHEMATICS

"Implementation and analysis of an algorithm on positive integer addition for quantum computing"

Menno Looman

Delft University of Technology

Mentor

Dr. M. Möller

Committee Members

Prof.dr.ir. C. Vuik

Dr. J.G. Spandaw

August, 2018

Delft

Abstract. In this project, an positive integer addition algorithm for quantum computing is analyzed which does not makes use of carry bits. The paper shows a general parallelization method. Using this method, the circuit runs in $O(n)$ time instead of $O(n^2)$. The algorithm is executed with different input sizes and error rates such that usable values are found.

Introduction

Every once in a while the news talks about a new breakthrough in the progress towards a quantum computer. The first time this happened was at the end of 1959 when Richard Feynmann talked about the possibilities. In his lecture "There is Plenty of Room at the Bottom" he describes what could be possible to build a computer. In 2000, David P. DiVincenzo from IBM published 5 criteria, which describe what is required to build a successful quantum computer. Since 2010 several universities and companies have made chips based on quantum optics to run one specific algorithm and huge steps were made towards the criteria of DiVincenzo. 6 years later the first quantum computer was developed that is reprogrammable and can run any algorithm using a set of universal gates. In 2017 IBM launched two 5 qubit quantum computers which are available to everyone on their website. This year, Google announced a quantum computer with 72 qubits, which is the most ever produced.

Since only a few people have a quantum computer, multiple simulators are made. This is possible since we have more knowledge about how a quantum computer works. These simulators are a way for people to learn things about programming a quantum computer. At the TU Delft, the Qx-simulator[6] is developed by Nadar Khammassi. The benefits of this simulator over that of an online simulator is that you can program it in such a way that the process gets scalable. Of course, it is not efficient to simulate a quantum computer and preform algorithms, but it is a way to see if your programs work without using one of the few quantum computers.

We can conclude that the quantum computer is here to stay, but why is all this time invested in building a quantum computer? In the paper that David P. DiVincenzo published in 2000: "The Physical Implementation of Quantum Computation", he gave an answer what, probably, has driven people as Richard Feynmann to make the first steps in the direction of quantum computing:

> *"But the final physical theory of the world is not Newtonian mechanics, and there is no reason to suppose that machines following the laws of quantum mechanics should have the same computational power as classical machines; (...) quantum machines can only have greater computational power than classical ones."* (David P. DiVincenzo 2000, p. 1)

Therefore it would be more logical to build a quantum computer instead of a classical computer, simply because the theory of quantum mechanics is more complete in comparison with Newtonian mechanics. Since the qubits of the quantum computer could be in the states $|0\rangle$ and $|1\rangle$, all classical algorithms can be performed by just not using the quantum property of the qubits. When this property is used, this computer could make much different calculations through different algorithms. However, this does not guarantee that everything can be performed faster. For some processes, people have developed algorithms for a quantum computer that are much faster then the current process on a classical computer. For example:

Example 0.1. *Prime factorization of an n digit number*
On a classical computer, there is not jet an algorithm known that can do this in polynomial time (time complexity of $O(n^k)$ for some $k \in \mathbb{N}$), but only algorithms that run in $O((1+\alpha)^n)$ with $\alpha \in \mathbb{R}_{>0}$. While for a quantum computer, Shor's algorithm can do this in polynomial time. This means that the algorithm is exponentially faster. The efficient part is the Shor's Quantum Fourier Transform (QFT). This algorithm is much faster as the input number gets larger.

Shor's algorithm, just like other quantum algorithms, include steps that can be performed on a classic computer. At one step in the algorithm they need to make a specific calculation that can be done much faster using a quantum computer. When this is calculated, they use the result again on a classic computer. This same principle may be true for other algorithms. Therefore I think that, as long as quantum computers are expensive, they will be used in combination with the classical computer.

One day, everyone might have a quantum computer at home or at work. Therefore there needs to be software on these computers to make it easy to reprogram and run your own calculations. One of the building blocks to this software is basic operators like: $+, -, \times, \div$. The main goal of this paper is to answer the question: How to implement the + operator for positive integers on a quantum computer and what is the performance? There are multiple algorithms to implement the + operator, but I will consider only one specific algorithm that makes use of the QFT. This algorithm is described in chapter 5. Before that, there will be an introduction on qubits in chapter 1 and on gates in chapter 2. The simulations of this algorithm are made using the Qx-simulator. This could only be done by making a few changes to the algorithm since not every gate was already programmed in the Qx-simulator. This is further explained in chapter 5. in this chapter there is a gate decomposition given from the universal set of rotational gates and the c-NOT gate. This decomposition can differ from machine since a different quantum computer may have a different set of universal gates. The algorithm is also analyzed with a given error rate. Therefore I also came up with a method to parallelize a random circuit. This is explained in chapter 4 and used later with the analysis of the algorithm.

The variant of addition algorithm which is used is described by a paper from Thomas G. Draper: "Addition on a Quantum Computer". A theoretical analysis is made in chapter 5. The circuit has a time complexity of $O(n^2)$, but I will show that this can be done in $O(n)$ time using a parallelization method. Which means that the time for errors to occur is less and thus the algorithm is less sensitive to noise.

The code is written in c++ and available in appendix B or on Gitlab[1]. The packages needed to execute the code are from the QX-simulator and are not available to the public. Still the code gives a great impression of the steps that were made, how certain processes work and how obstacles can be solved.

[1]The c++ code can also be found on `https://gitlab.com/majlooman/Addition_algorithm`

Contents

CHAPTER 1

Qubits

1. Bits

A classic computer keeps his information in a string of bits. Each bit is either a **0** or **1**. The **0** and **1** can be represented by different physical properties, as long as this property has two distinguishable states. For example, a single bit b_0 can be represented by a switch that is 'on' or 'off' at all times, even if you are not looking at the switch. A string of 20 bits $\{b_i : i \in \{0, 1, ..., 19\}\}$ is simply a line of switches next to each other. The computer looks at a specific switch and then decides what to do next.
When this classic principle is used to describe qubits, it can be difficult to imagine how this is possible. Let q_20 be a switch that is not just in a state 'on' or 'off' but a state in between 'on' and 'off'. What is even more difficult to imagine is the idea that there are infinite various states in between the two states 'on' and 'off'. But when someone looks at the switch q_20, the switch jumps with some chance p to the state 'on' and jumps with a chance of $(1 - p)$ to 'off'.

Definition 1.1 (Qubit). *Let q be a qubit. The state of q will be described by $|q\rangle$, which is a linear combination (superposition) of the states $|1\rangle$ and $|0\rangle$: $|q_i\rangle = \alpha|0\rangle + \beta|1\rangle$ with $\alpha, \beta \in \mathbb{C}$.*

Let $|q_1\rangle = \frac{1}{2}i|0\rangle + \frac{1}{2}\sqrt{3}|1\rangle$ then q_1 is in a superposition of $|0\rangle$ and $|1\rangle$. When q_1 is measured, the probability of $|q_1\rangle$ to become **1** is: $|\alpha|^2 = |\frac{1}{2}i|^2 = \frac{1}{4}$ and the probability to become **0** is: $|\beta|^2 = |\frac{1}{2}\sqrt{3}|^2 = \frac{3}{4}$. Therefore the extra condition on α and β is very natural:

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

such that the probability to become either **0** or **1** is 1. Notice that a 'measurement' is the same as looking at the switch in the example. So the computer has an active role in the calculations of the qubit since it changes the state of the qubit.

Since $|q_i\rangle$ is depending on $\alpha, \beta \in \mathbb{C}$, the state $|q_i\rangle$ can be written as: $|q_i\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \in \mathbb{C}^2$. Notice that the norm of q_0 is equal to 1 because of the restriction (1). Thus $|q_i\rangle$ will be a unit vector in $\mathbb{C}^2$.

2. Notation

To make calculations with qubits, some notation needs to be introduced. In the previous section, it was made known that the state of a qubit q_0 is an unit vector in $\mathbb{C}^2$. The norm of a vector can be defined through the inner product: $|||q_0\rangle|| = \langle |q_0\rangle, |q_0\rangle\rangle = |q_0\rangle^\dagger |q_0\rangle$, where $|q_0\rangle^\dagger$ is the state of q_0 transposed and taken the complex conjugate. To make this notation more useful and less messy, introduce $\langle q_0| = |q_0\rangle^\dagger$ and define $\langle q_0|q_0\rangle$ such that:

$$\langle |q_0\rangle, |q_0\rangle\rangle = \langle q_0||q_0\rangle = \langle q_0|q_0\rangle \tag{2}$$

Note that in equation (2), the left part is the inner product of the vector $|q_0\rangle$. The middel part is the multiplication of the vector $|q_0\rangle$ with $|q_0\rangle^\dagger$. The right part is the new notation that is introduced.

Definition 2.1. *When A is a $n \times n$ matrix for some $n \in \mathbb{N}$, let $A^\dagger$ be a $n \times n$ matrix such that for all $x, y \in \mathbb{C}^n$ the following holds: $\langle x, Ay\rangle = \langle A^\dagger x, y\rangle$*

notice that $A^\dagger$ is the conjugate transpose of A, a proof can be found in appendix A. This can be shown by writing out the inner product. The conjugate transpose means that $A^\dagger$ can be

achieved by transposing the matrix A and take the complex conjugate of every value in A. Now let A be an 2×2 matrix and q_0 and q_1 be two qubits. Then the following notation is introduced:

$$\langle q_0|Aq_1\rangle = \langle q_0|A|q_1\rangle \tag{3}$$

Notice that when definition (2.1) and equations (2) and (3) are combined, the following holds:

$$\langle A^\dagger q_0|q_1\rangle = \langle q_0|Aq_1\rangle = \langle q_0|A|q_1\rangle = \langle |q_0\rangle, A|q_1\rangle\rangle = \langle A^\dagger|q_0\rangle, |q_1\rangle\rangle$$

From now on, this equality will be used as a calculation rule in further arithmetic.

3. Bloch sphere

The Bloch sphere is a representation of the state of a single qubit. Recall that the state of a qubit q_i can be written as an unit vector in $\mathbb{C}^2$. The Bloch sphere representation projects the unit vectors in $\mathbb{C}^2$ $\{x \in \mathbb{C}^2 : ||x|| = 1\}$ on to $\{y \in \mathbb{R}^3 : ||y|| = 1\}$ the unit vectors in $\mathbb{R}^3$. Notice that the projection is not injective (and thus not bijective). In order to understand why this representation can be of any use, there needs to be more definition around a measurement and how to make calculations with them. As stated in: "Quantum Computation and Quantum Information" by Michael A. Nielsen & Isaac L. Chuang (2010, p. 84) postulate 3 reads:

> **Postulate 3:** Quantum measurements are described by a collection $\{M_m\}$ of *measurement operators*. These are operators acting on the state space of the system being measured. The index m refers to the measurement outcomes that may occur in the experiment. If the state of the quantum system is $|\psi\rangle$ immediately before the measurement then the probability that result m occurs is given by
>
> $$p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle, \tag{4}$$
>
> and the state of the system after the measurement is
>
> $$\frac{M_m|\psi\rangle}{\sqrt{\langle\psi|M_m^\dagger M_m|\psi\rangle}}. \tag{5}$$
>
> The measurement operators satisfy the *completeness equation*,
>
> $$\sum_m M_m^\dagger M_m = I. \tag{6}$$
>
> The completeness equation expresses the fact that probabilities sum to one:
>
> $$1 = \sum_m p(m) = \sum_m \langle\psi|M_m^\dagger M_m|\psi\rangle. \tag{7}$$

Notice that equation (7) is a combination of equations (4) and (6). Nevertheless it is a really nice result because the chances of the measurements add up to 1. What is necessary for a probability mass function to be well defined. Even more, this postulate allows us to calculate what a measurement does with state of a qubit.

DEFINITION 3.1. *Let q_0, q_1 be qubits with states $|q_0\rangle, |q_1\rangle$. Then $|q_0\rangle \equiv |q_1\rangle$ when the difference between $|q_0\rangle$ and $|q_1\rangle$ can not be measured.*

In other words, for all measurements $M_m \in \{M_m\}$: $\langle q_0|M_m^\dagger M_m|q_0\rangle = \langle q_1|M_m^\dagger M_m|q_1\rangle$. Then the difference between $|q_0\rangle$ and $|q_1\rangle$ can not be measured. If two qubits would act physically different from each other, then it would be possible to measure the difference simply by 'making' them act in this different way. However, this implies that if it is not possible to measure the difference, that they will act perfectly the same. If two things act perfectly the same, what is the difference between them? Are they exactly the same? In the quantum theory they are, and that is how they are treated.

THEOREM 3.2 (Global phase). *Let q be a qubit with state $|q\rangle$ and let $\varphi \in \mathbb{R}$. Then the global phase $e^{i\varphi}$ is not measurable: $|q\rangle \equiv e^{i\varphi}|q\rangle$.*

PROOF. Let q_0 be a qubit. Then there exists $\alpha, \beta \in \mathbb{C}$ such that: $|q_0\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix}$. In section (1) we have seen that this must a unit vector in $\mathbb{C}^2$. Let q_1 be a qubits such that $|q_1\rangle = \begin{pmatrix}e^{i\varphi}\alpha\\ e^{i\varphi}\beta\end{pmatrix} = e^{i\varphi}|q_0\rangle$. Let $\{M_m\}$ be a collection of measurement operators that satisfy the equations from postulate 3. Then, the chance that when q_0 is measured an outcome m will occur, is given by equation (4): $p(m) = \langle q_0|M_m^\dagger M_m|q_0\rangle$. When q_1 is measured, the same index m will occur with a chance: $p(m) = \langle q_1|M_m^\dagger M_m|q_1\rangle$. Recall that q_1 was chosen such that $|q_1\rangle = e^{i\varphi}|q_0\rangle$. Due to equation (2): $\langle q_1| = \begin{pmatrix}e^{i\varphi}\alpha\\ e^{i\varphi}\beta\end{pmatrix}^\dagger = \left((e^{i\varphi}\alpha)^*, (e^{i\varphi}\beta)^*\right)$. Where * is the complex conjugate operator. So we have that:

$$\begin{aligned}\langle q_1| &= \left((e^{i\varphi}\alpha)^*, (e^{i\varphi}\beta)^*\right)\\ &= \left(e^{-i\varphi}\alpha^*, e^{-i\varphi}\beta^*\right)\\ &= e^{-i\varphi}\left(\alpha^*, \beta^*\right)\\ &= e^{-i\varphi}\langle q_0|\end{aligned}$$

Let us again look at the chance to measure m when q_1 is measured:

$$p(m) = \langle q_1|M_m^\dagger M_m|q_1\rangle = e^{-i\varphi}\langle q_0|M_m^\dagger M_m e^{i\varphi}|q_0\rangle = e^{-i\varphi}e^{i\varphi}\langle q_0|M_m^\dagger M_m|q_0\rangle = \langle q_0|M_m^\dagger M_m|q_0\rangle$$

Notice that this is allowed since our matrices, vectors and scalars are complex and thus commutative. This result is exactly the same chance of measuring m when q_0 is measured. Since the measurement operator M_m and the index m were chosen randomly, it can be concluded that: $|q_0\rangle \equiv |q_1\rangle = e^{i\varphi}|q_0\rangle$. Since q_0 was chosen randomly, this applies to all qubits. □

Let q be a qubit. Let $\alpha, \beta \in \mathbb{C}$ such that $|q_0\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix}$. Since α and β are complex values, they can be expressed in real polar coordinates. Let $\varphi_\alpha, \varphi_\beta \in [0, 2\pi)$ and $r_\alpha, r_\beta \in \mathbb{R}_{\geq 0}$ such that $\alpha = e^{i\varphi_\alpha}r_\alpha, \beta = e^{i\varphi_\beta}r_\beta$. Since theorem (3.2) we know that:

$$|q_0\rangle = \begin{pmatrix}e^{i\varphi_\alpha}r_\alpha\\ e^{i\varphi_\beta}r_\beta\end{pmatrix} \equiv e^{-i\varphi_\alpha}\begin{pmatrix}e^{i\varphi_\alpha}r_\alpha\\ e^{i\varphi_\beta}r_\beta\end{pmatrix} = \begin{pmatrix}r_\alpha\\ e^{i(\varphi_\beta-\varphi_\alpha)}r_\beta\end{pmatrix}$$

Define $\varphi = \varphi_\beta - \varphi_\alpha$ since this is a constant value. Recall equation (1) which holds for every qubit:

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

So r_β and can be fully described by r_α since both values are positive. A nice way to do this is to make a parametrization. Let $\theta \in [0, 2\pi]$ such that $r_\alpha = \cos\theta$. Notice that this is possible since: $r_\alpha^2 + r_\beta^2 = 1$, and thus: $r_\alpha^2 \leq 1$ since $r_\beta^2 \geq 0$. Now $r_\beta^2 = 1 - r_\alpha^2 = 1 - \cos\theta$ and thus r_β can be described as follows: $r_\beta = \sin\theta$. Notice that for $\theta \in [0, \frac{\pi}{2}] : \sin\theta \geq 0$. So all possible states $|q_0\rangle$ can be described by $\theta \in [0, \frac{\pi}{2}]$ and $\varphi \in [0, 2\pi)$: $|q_0\rangle = \begin{pmatrix}\cos\theta\\ e^{i\varphi}\sin\theta\end{pmatrix}$. This is exactly where the Bloch sphere representation is based on. This Bloch sphere is a map from the unit vectors in $\mathbb{C}^2$ to the unit vectors in $\mathbb{R}^3$ described by:

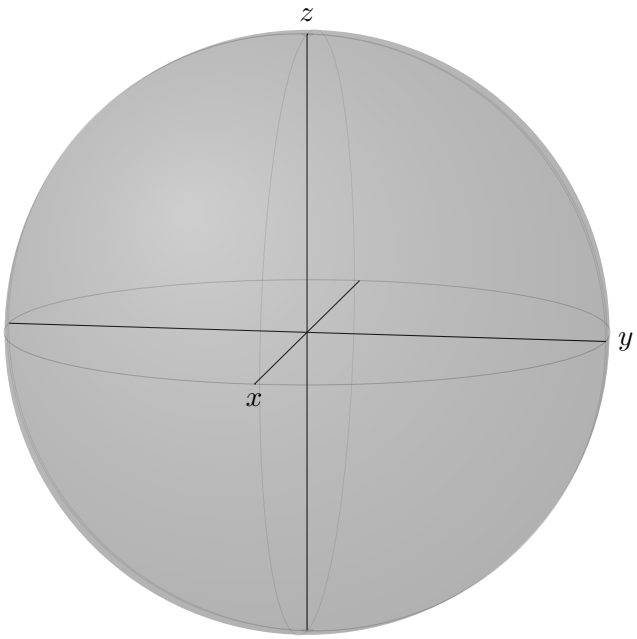
$$\begin{aligned}x &= \sin(2\theta)\cos\varphi\\ y &= \sin(2\theta)\sin\varphi\\ z &= \cos(2\theta)\end{aligned}$$

Here are some useful examples. Let q_0, q_1, q_2 be qubits with the states:

$$|q_0\rangle = |0\rangle = \begin{pmatrix}1\\ 0\end{pmatrix}, \qquad |q_1\rangle = |1\rangle = \begin{pmatrix}0\\ 1\end{pmatrix}, \qquad |q_2\rangle = \begin{pmatrix}-\frac{i}{2}\sqrt{3}\\ \frac{1}{4}\sqrt{2} - \frac{i}{4}\sqrt{2}\end{pmatrix}$$

Indeed these are legal states since equation (1) is satisfied.

FIGURE 1. The Bloch Sphere



· Evaluate q_0:
Notice that $\alpha = 1 \in \mathbb{R}_{\geq 0}$. Thus there exist a $\theta \in [0, \frac{\pi}{2}]$ such that $1 = \cos\theta$. This is satisfied for $\theta = 0$. Now $\beta = e^{i\varphi}\sin\theta = e^{i\varphi}\sin 0 = 0$ for every value $\varphi \in [0, 2\pi]$. Using the Bloch sphere representation, the state of the qubit can be visualized as:

$$\begin{pmatrix} \sin(2\cdot 0)\cos\varphi \\ \sin(2\cdot 0)\sin\varphi \\ \cos(2\cdot 0) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

· Evaluate q_1:
Notice that $|q_1\rangle$ can be derived in the same way as the evaluation of q_0 with $\theta = \frac{\pi}{2}$. Using the Bloch sphere representation, the state of the qubit can be visualized as:

$$\begin{pmatrix} \sin(2\cdot \frac{\pi}{2})\cos\varphi \\ \sin(2\cdot \frac{\pi}{2})\sin\varphi \\ \cos(2\cdot \frac{\pi}{2}) \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

· Evaluate q_2:
In this case $\alpha = -\frac{i}{2}\sqrt{3} \notin \mathbb{R}_{\geq 0}$. Remember that $\alpha = e^{i\varphi_\alpha} r_\alpha$ for some values $\varphi_\alpha \in [0, 2\pi]$ and $r_\alpha \in [0, 1]$. Notice that: $i\alpha \in R_{\geq 0}$, so $e^{-i\varphi_\alpha}$ must be equal to i. multiplying the state $|q_2\rangle$ by $e^{-i\varphi_\alpha}$ results in: $i\begin{pmatrix} -\frac{i}{2}\sqrt{3} \\ \frac{1}{4}\sqrt{2} - \frac{i}{4}\sqrt{2} \end{pmatrix} \equiv \begin{pmatrix} \frac{1}{2}\sqrt{2} \\ \frac{i}{4}\sqrt{2} + \frac{1}{4}\sqrt{2} \end{pmatrix}$. Now $i\alpha = \cos\theta = \frac{1}{2}\sqrt{3}$ is satisfied for $\theta = \frac{1}{6}\pi$. Let us take a look at β:

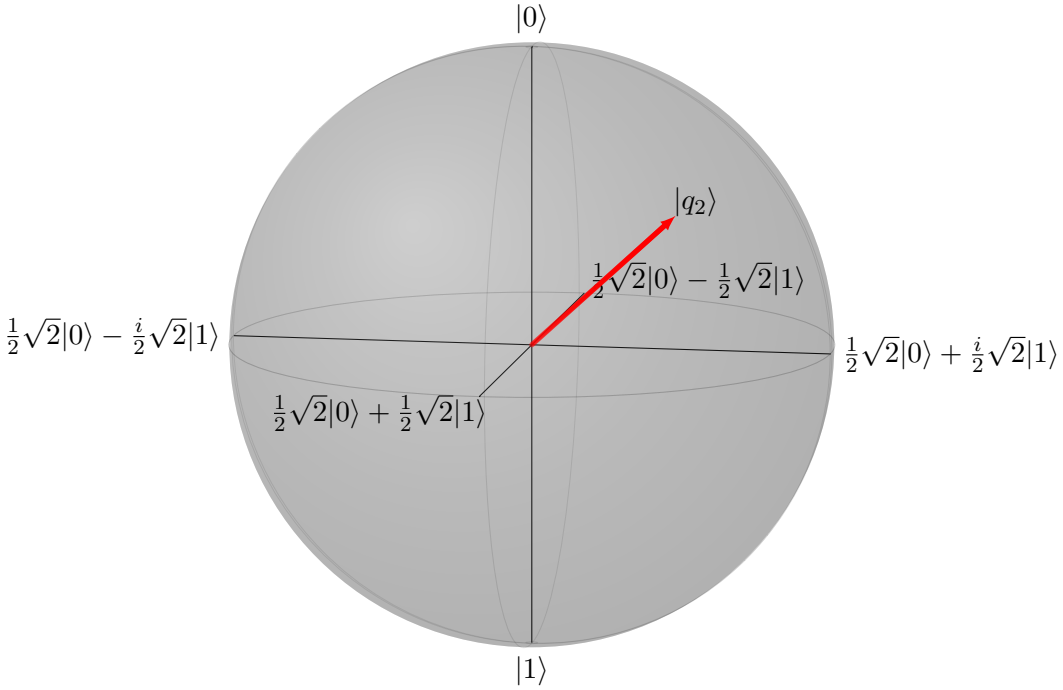
$$i\beta = \frac{i}{4}\sqrt{2} + \frac{1}{4}\sqrt{2} = e^{i\varphi} r_\beta = e^{i\varphi}\sin\theta = e^{i\varphi}\frac{1}{2}$$

. So by dividing $\frac{1}{2}$ we achieve the following: $e^{i\varphi} = \cos(\varphi) + i\sin(\varphi) = \frac{1}{2}\sqrt{2} + \frac{i}{2}\sqrt{2}$, So φ must be equal to $\frac{1}{4}\pi$. Using the Bloch sphere representation, the state of the qubit can be visualized as:

$$\begin{pmatrix} \sin(2\cdot \frac{1}{6}\pi)\cos\frac{1}{4}\pi \\ \sin(2\cdot \frac{1}{6}\pi)\sin\frac{1}{4}\pi \\ \cos(2\cdot \frac{1}{6}\pi) \end{pmatrix} = \begin{pmatrix} \frac{1}{4}\sqrt{6} \\ \frac{1}{4}\sqrt{6} \\ \frac{1}{2} \end{pmatrix} \tag{8}$$

This is also visualized in the bloch sphere in figure 2.

FIGURE 2. The Bloch Sphere with states



4. Quantum Register

So far, we only considered the states of single qubits in a superposition. These states could be visualized using a Bloch sphere. A system of multiple qubits can not always be described by the state of the individual qubits. To understand the reason why this is not possible, we need some knowledge about gates, which can be found in chapter 2.

Recall that the state of a qubit q_0 could be described by an unit vector in $\mathbb{C}^2$. The state of the system of two qubits q_0, q_1 (i.e. quantum register) will be defined by $|q_0q_1\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$ and therefore these states can be described with an unit vector in $\mathbb{C}^4$. In general, the state of a system with n qubits can be described by an unit vector in $\mathbb{C}^{2^n}$. Notice that the state of $|q_0\rangle$ can not be separated form $|q_0q_1\rangle$ in any general way, but the chance that q_0 will be 0 or 1 can be derived since the probability of the four separate states are known. This will be further explained at the end of this section.

The notation introduced in section 1.2 can be scaled up. Let $q_0, \ldots, q_n$ be qubits. Then the restriction on the qubits is:

$$\langle q_0, \ldots, q_n | q_0, \ldots, q_n \rangle = 1 \tag{9}$$

This is a generalization of equation 1 and ensures that the total probability of all the states is equal to 1. Notice that in total there are 2^n possible states.

Also theorem 3.2 holds, so a quantum register is equivalent under a global phase shift.

From the states of the qubits, the state of the register can be derived. However, in most cases this is not possible the other way around. For example, let q_0 and q_1 be two qubits. When the state of q_0 is equal to $\begin{pmatrix}\alpha\\ \beta\end{pmatrix} \in \mathbb{C}^2$ and the state of q_1 is equal to $\begin{pmatrix}\gamma\\ \delta\end{pmatrix} \in \mathbb{C}^2$ then the state of the system is the tensor product of the two individual states: $|q_0q_1\rangle = \begin{pmatrix}\alpha\gamma\\ \alpha\delta\\ \beta\gamma\\ \beta\delta\end{pmatrix} \in \mathbb{C}^4$. We will not

discuss the tensor product in greater detail, since we are only using it in this section. First, we will check if this state is allowed. Assume that $|\alpha|^2 + |\beta|^2 = 1$ and $|\gamma|^2 + |\delta|^2 = 1$. Now,

$$\begin{aligned}\langle q_0 q_1 | q_0 q_1 \rangle &= |\alpha\gamma|^2 + |\alpha\delta|^2 + |\beta\gamma|^2 + |\beta\delta|^2 \\ &= |\alpha|^2|\gamma|^2 + |\alpha|^2|\delta|^2 + |\beta|^2|\gamma|^2 + |\beta|^2|\delta|^2 \\ &= |\alpha|^2(|\gamma|^2 + |\delta|^2) + |\beta|^2(|\gamma|^2 + |\delta|^2) \\ &= |\alpha|^2 + |\beta|^2 = 1\end{aligned}$$

To prove that the contrary is not always true, let us consider the following situation. Let $|q_0 q_1\rangle$ be in one of the bell states: $|q_0 q_1\rangle = \begin{pmatrix} \frac{1}{2}\sqrt{2} \\ 0 \\ 0 \\ \frac{1}{2}\sqrt{2} \end{pmatrix}$. When we try to separate this state into two independent ones, we get a contradiction. Suppose $|q_0\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$ and $|q_1\rangle = \begin{pmatrix} \gamma \\ \delta \end{pmatrix}$ then $|q_0 q_1\rangle = \begin{pmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{pmatrix} = \begin{pmatrix} \frac{1}{2}\sqrt{2} \\ 0 \\ 0 \\ \frac{1}{2}\sqrt{2} \end{pmatrix}$. Notice that $\alpha\gamma = \frac{1}{2}\sqrt{2}$ so α can not be equal to 0. The same reasoning can be applied to δ since $\beta\delta = \frac{1}{2}\sqrt{2}$. Therefore $\alpha\delta \neq 0$ because there are both not equal to zero. However, from the given state we derive $\alpha\delta = 0$, which is a contradiction.

CHAPTER 2

Gates

1. Purpose

In the previous chapter we have discussed what qubits are and introduced some notation and visualization, but how can the state of a qubit be modified? That is where gates will be useful. Recall that a state of a system of n qubits could be interpreted by a unit vector: $|q_0 q_1 \dots q_{n-1}\rangle \in \mathbb{C}^{2^n}$. A gate that acts on this system will be a complex $n \times n$ matrix. When the gates acts on the qubit, the state of the qubit changes. The new state becomes the matrix U multiplied with the vector $|q_0 q_1 \dots q_{n-1}\rangle$. Recall the restriction in equation (9), this still needs to hold after the matrix U is applied.

$$\langle q_0, \dots, q_n | q_0, \dots, q_n \rangle = 1 \implies \langle U q_0, \dots, q_n | U q_0, \dots, q_n \rangle = 1 \tag{10}$$

Notice that $\langle U q_0, \dots, q_n | U q_0, \dots, q_n \rangle = \langle q_0, \dots, q_n | U^\dagger U | q_0, \dots, q_n \rangle$. Since the qubits were randomly chosen, this needs to hold for all possible qubits and thus must $U^\dagger U = I$. Then $\langle q_0, \dots, q_n | I | q_0, \dots, q_n \rangle = \langle q_0, \dots, q_n | q_0, \dots, q_n \rangle = 1$. A matrix with the property that $U^\dagger U = I$ is called unitary. So every gate must be some unitary matrix. Define $U(n)$ as the set of complex unitary $n \times n$ matrices.

2. Unitary Matrix

Instead of checking every matrix if it is unitary, we will find a general form. From this general form, it will be easy to build a unitary matrix. There will also be a method to find the general form. We do this only for matrices in $U(2)$ since we only need this for the decomposition in chapter 3.
Notice that if $U^\dagger U = I$, then $U U^\dagger = I$. The proof can be found in the appendix A. These equations give the following system:

	a	**b**
(11)	$a\overline{a} + b\overline{b} = 1$	$a\overline{c} + b\overline{d} = 0$
(12)	$c\overline{c} + d\overline{d} = 1$	$c\overline{a} + d\overline{b} = 0$
(13)	$a\overline{a} + c\overline{c} = 1$	$b\overline{a} + d\overline{c} = 0$
(14)	$b\overline{b} + d\overline{d} + 1$	$a\overline{b} + c\overline{d} = 0$

By subtracting equation **12a** from **13a**: $a\overline{a} = d\overline{d}$ is obtained. Similarly with **12a** and **14a** gives: $b\overline{b} = c\overline{c}$. Every number in $\mathbb{C}$ can be expressed in polar coordinates: $a = e^{i\varphi_a} r_a$ where $r_a \in \{\mathbb{R} \geq 0\}$ denotes the distance from 0 to a and $\varphi_a \in [0, 2\pi)$ the angle from the positive x-axis. Notice that $a\overline{a} = \mathrm{Re}(a)^2 + \mathrm{Im}(a)^2 = r_a^2$. For example, Expression **11a** now becomes: $r_a^2 + r_b^2 = 1$. This also applies to the other values: $r_a^2 = r_d^2$ and $r_b^2 = r_c^2$. Since $r_a \geq 0$ this means that: $r_a = r_d$ and $r_b = r_c$.
11b can be rewritten as:

$$r_a r_c e^{i\varphi_a} e^{-i\varphi_c} = -r_b r_d e^{i\varphi_b} e^{-i\varphi_d}$$

Substitute $r_a = r_d$ and $r_b = r_c$, this gives the following equation:

$$r_a r_b e^{i\varphi_a} e^{-i\varphi_c} = -r_b r_a e^{i\varphi_b} e^{-i\varphi_d}$$

So $r_a r_b$ can (unless one of them is zero) [1]be divided from both sides to get:

$$e^{i(\varphi_a-\varphi_c)} = -e^{i(\varphi_b-\varphi_d)} \iff e^{i(\varphi_a+\varphi_d)} = -e^{i(\varphi_b+\varphi_c)}$$

Since

$$\det(U) = ad - cb = r_a r_d e^{i\varphi_a} e^{i\varphi_d} - r_b r_c e^{i\varphi_b} e^{i\varphi_c} = r_a^2 e^{i(\varphi_a+\varphi_d)} - r_b^2 e^{i(\varphi_b+\varphi_c)}$$

Since $e^{i(\varphi_a+\varphi_d)} = -e^{i(\varphi_b+\varphi_c)}$ the following is obtained:

$$\det(U) = (r_a^2 + r_b^2) e^{i(\varphi_a+\varphi_d)} = e^{i(\varphi_a+\varphi_d)}$$

Recall equation **1a** that says $r_a^2 + r_b^2 = 1$. Define $\varphi \in [0, 2\pi)$ such that $\det(U) = e^{i\varphi}$, then $-e^{i(\varphi_b+\varphi_c)} = e^{i(\varphi_a+\varphi_d)} = e^{i\varphi}$ and thus the following two equations are obtained: $e^{i\varphi_d} = e^{i\varphi} e^{-i\varphi_a}$ and $e^{i\varphi_b} = -e^{i\varphi} e^{-i\varphi_c}$. Now the variable b and d are totally defined by the value of a, c and φ which depend from the determinant of the matrix.

$$r_a^2 + r_c^2 = 1 \qquad b = -r_c e^{-i\varphi_c} e^{i\varphi} \qquad d = r_a e^{-i\varphi_a} e^{i\varphi}$$

Thus a unitary matrix $U \in U(2)$ can be written as $U = \begin{bmatrix} a & -\overline{c}e^{i\varphi} \\ c & \overline{a}e^{i\varphi} \end{bmatrix}$.

Now we have the general form of a matrix, let us take a look at how the value of φ can be determined. Recall that $\det(U) = e^{i\varphi}$, therefore there are some easy methods to get the value of φ from a unitary matrix U modulo 2π:

- $\varphi = \arctan 2(\mathrm{Im}(-bc), \mathrm{Re}(-bc))$
- $\varphi = \arctan 2(\mathrm{Im}(\det(U)), \mathrm{Re}(\det(U)))$
- $\varphi = \mathrm{Im}(\ln(\det(U)))$
- $\varphi = -i \ln(\det(U))$

Define $SU(2)$ as the set of the complex 2×2 unitary matrices with determinant equal to 1. Notice that $SU(2) \subset U(2)$. We distinguish matrices that are in $SU(2)$ and $U(2)$ because it will be used in the decomposition in chapter 3. The general form of a matrix in $SU(2)$ is slightly simpler. Recall that matrix $U \in U(2)$ can be written as: $U = \begin{bmatrix} a & -\overline{b}e^{i\varphi} \\ b & \overline{a}e^{i\varphi} \end{bmatrix}$. Now $det(U) = (|a|^2 + |b|^2)e^{i\varphi} = e^{i\varphi}$. If $\varphi = 0$ then $det(U) = 1$, so the general form of a matrix U' in $SU(2)$ can be written as: $U' = \begin{bmatrix} a & -\overline{b} \\ b & \overline{a} \end{bmatrix}$. Notice that every matrix in $U(2)$ can be written as a matrix in $SU(2)$ multiplied by a matrix: $\begin{bmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{bmatrix}$.

3. Single-qubit gates

In this section we will give a short list of single-qubit gates that were used in the algorithm. The values of the general form and a short description are given for every gate. Recall that if $\varphi = 0$ then the gate is in $SU(2)$. There will be two rotation matrices given: $R_z(\phi)$ and $R_y(\phi)$. In chapter 3 there will be referred to these two by the rotation matrices.

[1]If r_a of is zero then φ_a is unknown since a is just equal to zero. The same applies to b

Single-qubit gates			
I	$\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$	$a=1,\ b=0,\ \varphi=0$	The identity matrix will sometimes be indicated with I_2 to clarify that it is the 2×2 identity matrix. This gate does not change the state of a qubit and can be left out of a circuit.
X	$\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$	$a=0,\ b=1,\ \varphi=\pi$	Also known as the NOT gate or Pauli-x gate. In the Bloch sphere representation it can be interpreted as a rotation of π around de x-axes.
H	$\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$	$a=\frac{1}{\sqrt{2}},\ b=\frac{1}{\sqrt{2}},\ \varphi=\pi$	The Hadamard gate. In the bloch sphere representation it can be interpreted as a rotation of π around the line $z=x$.
$Z(\phi)$	$\begin{bmatrix}1 & 0\\ 0 & e^{i\phi}\end{bmatrix}$	$a=1,\ b=0,\ \varphi=\phi$	A rotation of ϕ around the Z axis. This gate is equivalent with the $R_z(\phi)$ gate for single qubits. A proof can be found in appendix A
$R_y(\phi)$	$\begin{bmatrix}\cos\phi & -\sin\phi\\ \sin\phi & \cos\phi\end{bmatrix}$	$a=\cos\phi,\ b=\sin\phi,$ $\varphi=0$	A rotation matrix with $\phi\in\mathbb{R}$. It can be interpreted as a rotation of 2ϕ around the y-axis.
$R_z(\phi)$	$\begin{bmatrix}e^{-i\frac{1}{2}\phi} & 0\\ 0 & e^{i\frac{1}{2}\phi}\end{bmatrix}$	$a=e^{-i\frac{1}{2}\phi},\ b=0,\ \varphi=\phi$	A rotation matrix with $\phi\in\mathbb{R}$. It can be interpreted as a rotation of ϕ around the z-axis.

TABLE 1. List of single-qubit gates

4. Multi-qubit gates

Single-qubit gates are not the only gates we need for the algorithm, also 4 multi-qubit gates are needed. The gates we need are acting on 2 qubits. So let q_0, q_1 be qubits. Let $\alpha,\beta,\gamma,\delta\in\mathbb{C}$ such that $|q_0q_1\rangle=\alpha|00\rangle+\beta|01\rangle+\gamma|10\rangle+\delta|11\rangle\in\mathbb{C}^4$. A gate acting on this quantum system must be a matrix in $U(4)$. We have not discussed a general form for matrices in $U(4)$, but the gates we need a nice structure. Therefore before I will give a list of the gates, I will show a property of unitary matrices.
Let U be a matrix in $U(2)$ with elements $u_1,u_2,u_3,u_4\in\mathbb{C}$. Let I_k be the $k\times k$ identity matrix. Let A be a 4×4 matrix given by:

$$A=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & u_1 & u_2\\ 0 & 0 & u_3 & u_4\end{bmatrix}=\begin{bmatrix}I_2 & 0\\ 0 & U\end{bmatrix}$$

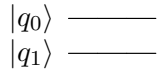
Recall that $A^\dagger$ is just the conjugate transpose of A, thus: $A^\dagger=\begin{bmatrix}I_2 & 0\\ 0 & U^\dagger\end{bmatrix}$. Since U is unitary, A is also unitary: $AA^\dagger=\begin{bmatrix}I_2 & 0\\ 0 & U\end{bmatrix}\begin{bmatrix}I_2 & 0\\ 0 & U^\dagger\end{bmatrix}=\begin{bmatrix}I_2 & 0\\ 0 & UU^\dagger\end{bmatrix}=I_4$.
We use it for a 4×4 matrix A, but it still holds for more general matrices. Let U be a matrix in $U(2)$. Let B be a $n\times n$ matrix given by: $\begin{bmatrix}I_j & 0 & 0\\ 0 & U & 0\\ 0 & 0 & I_k\end{bmatrix}$, such that $k+j+2=n$, then B is also unitary. The proof is the same as above for the $n=4$ case.

Multi-qubit gates			
c-X	$\begin{bmatrix} I_2 & 0 \\ 0 & U \end{bmatrix}$	U is the X gate from the single-qubit list	Also known as the c-NOT gate. The first qubit (q_0 in our case) is usually referred to as the control qubit and the other qubit as the target qubit.
c-Z	$\begin{bmatrix} I_2 & 0 \\ 0 & U \end{bmatrix}$	With U the $Z(\pi)$ gate from the single-qubit list.	The controlled Pauli-z gate. A controlled rotation of π around the z axis
c-R_k	$\begin{bmatrix} I_2 & 0 \\ 0 & U \end{bmatrix}$	With U the $Z(\frac{2\pi}{2^k})$ gate from the single-qubit list.	Also known by the controlled phase shift gate. In the Qx-simulator, This gate is only implemented for $k \geq 2$. Notice that instead of c-R_1 the c-Z gate can be used.
SWAP	$\begin{bmatrix} 1 & 0 & 0 \\ 0 & U & 0 \\ 0 & 0 & 1 \end{bmatrix}$	U is the X gate from the single-qubit list	This gate can be seen as the exchange of the states of the two qubits. Notice that this is our only gate that is not controlled, therefore it will get it's own representation in the next section.

TABLE 2. List of multi-qubit gates

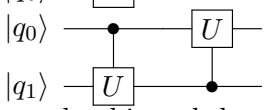
5. Representation

The representation of a quantum circuit is mostly done through a system of wires and boxes as shown below. Every wire symbolizes a qubit. The state of the qubit is given in the beginning of the circuit on the left. As time moves on, follow the wire of the qubit to see what gate is acting on the qubit.

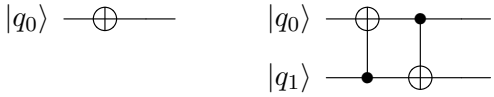


We dissociate three kinds of gates. The single-qubit gates, the controlled gates and the SWAP gate. Each of these kinds of gates has its own representation:

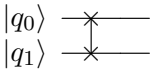
- Let U be a single-qubit gate: $|q_0\rangle$ — U —
- Let c-U be a controlled gate:



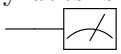
  The wire with the dot is the control qubit and the other one the target. Note that there is one exception, the X and c-X gates will be visualized using a circle and a cross:



- Let $\times$ be the SWAP gate:



There is one different kind of gate, a measurement gate. What a measurement exactly does is described in chapter 1. We will only put a measurement gate at the end of a circuit:



CHAPTER 3

Decomposition

1. Purpose

In chapter 2, we have discussed the definition of a gate and what conditions it should meet. In theory we are able to write down every gate which agrees to these conditions, unfortunately this is not the case in practice. In practice every quantum computer has a set of universal gates. A set of gates is called universal if and only if every unitary gate can be build using only the gates from the set. The universal set of gates which is used in this paper is a set of two rotation gates and the c-X gate defined in chapter 2. The single-qubit gates can be decomposed using only the two rotation gates. Notice that we only have two kinds of multi-qubit gates: the controlled single-qubit gates and the SWAP gate. A SWAP gate can be decomposed using three c-X gates. Since the SWAP gate is already implemented in the Qx-simulator, we will not further discuss this in the paper. The controlled single-qubit gates will be decomposed. This is also noticed in "Quantum Computation and Quantum Information" [2], but we will explain every step in a more detailed way.

2. Rotation Gates

In this section some properties of the rotation gates and the X will be shown. These properties are necessary for the decomposition explained in section 3.3 and 3.4. Before we show and proof the properties, we notice something about the X gate. Recall that $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. A simple matrix multiplication gives us:

$$XX = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I \tag{15}$$

. Therefore X is involutory, which means that the matrix is identical to his inverse. This property will be used in the following proofs. Recall the rotation gates from table 1. Note that instead of $\phi \in \mathbb{R}$, we could also have chosen to define the matrices for $\phi \in [0, 2\pi)$. From these definitions we are going to prove three different theorems: theorem 2.1 describes the *additive* property, theorem 2.2 describes the *identity* property and theorem 2.3 describes how the rotation gates acts with the X gate. This theorem is proven by using theorems 2.1 and 2.2. To avoid circular reasoning, this is the last theorem that is proven.

THEOREM 2.1 (Additivity). *Let $\theta, \varphi \in \mathbb{R}$ and let R be one of the rotation gates as described in table 1. Then $R(\theta)R(\varphi) = R(\theta + \varphi)$.*

PROOF. To proof this theorem, we first proof it for $R_z(\varphi + \theta)$ and afterwards for $R_y(\varphi + \theta)$. Let $\theta, \varphi \in \mathbb{R}$. Then

$$\begin{aligned} R_z(\theta)R_z(\varphi) &= \begin{bmatrix} e^{-i\frac{1}{2}\theta} & 0 \\ 0 & e^{i\frac{1}{2}\theta} \end{bmatrix} \begin{bmatrix} e^{-i\frac{1}{2}\varphi} & 0 \\ 0 & e^{i\frac{1}{2}\varphi} \end{bmatrix} = \begin{bmatrix} e^{-i\frac{1}{2}\theta}e^{-i\frac{1}{2}\varphi} & 0 \\ 0 & e^{i\frac{1}{2}\theta}e^{i\frac{1}{2}\varphi} \end{bmatrix} \\ &= \begin{bmatrix} e^{-i\frac{1}{2}(\theta+\varphi)} & 0 \\ 0 & e^{i\frac{1}{2}(\theta+\varphi)} \end{bmatrix} = R_z(\theta + \varphi) \end{aligned}$$

To proof that the same property for $R_y(\theta+\varphi)$ holds, some addition rules for trigonometric functions are needed, but we will not further discuss those. Again let $\theta, \varphi \in \mathbb{R}$. Then:

$$\begin{aligned} R_y(\theta)R_y(\varphi) &= \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} \cos(\varphi) & \sin(\varphi) \\ -\sin(\varphi) & \cos(\varphi) \end{bmatrix} \\ &= \begin{bmatrix} \cos(\theta)\cos(\varphi) - \sin(\theta)\sin(\varphi) & \cos(\theta)\sin(\varphi) + \sin(\theta)\cos(\varphi) \\ -\sin(\theta)\cos(\varphi) - \cos(\theta)\sin(\varphi) & cos(\theta)\cos(\varphi) - \sin(\theta)\sin(\varphi) \end{bmatrix} \\ &= \begin{bmatrix} \cos(\theta+\varphi) & \sin(\theta+\varphi) \\ -\sin(\theta+\varphi) & \cos(\theta+\varphi) \end{bmatrix} = R_y(\theta+\varphi) \end{aligned}$$

□

THEOREM 2.2 (Identity). *Let R be one of the rotation gates as described in table 1. Then $R(0) = I$.*

PROOF. Let I be the 2×2 identity matrix. First we will look at $R_z(0)$, then at $R_y(0)$. Consider the following:

$$\begin{aligned} R_z(0) &= \begin{bmatrix} e^{-i0} & 0 \\ 0 & e^{i0} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I \\ R_y(0) &= \begin{bmatrix} \cos(0) & -\sin(0) \\ \sin(0) & \cos(0) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I \end{aligned}$$

□

THEOREM 2.3 (Rotation and X gate). *Let $\theta \in \mathbb{R}$ and let R be one of the two rotation gates, then: $XR(\theta) = R(-\theta)X$.*

PROOF. Let $\theta \in \mathbb{R}$. Consider the following expression: $XR(\theta)XR(\theta)$. If we can show that this expression is equal to I, then $XR(\theta)$ is involutory. This is shown below for both $R_z(\theta)$ and $R_y(\theta)$.

$$\begin{aligned} XR_z(\theta) &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} e^{-i\frac{1}{2}\theta} & 0 \\ 0 & e^{i\frac{1}{2}\theta} \end{bmatrix} = \begin{bmatrix} 0 & e^{i\frac{1}{2}\theta} \\ e^{-i\frac{1}{2}\theta} & 0 \end{bmatrix} \\ XR_z(\theta)XR_z(\theta) &= \begin{bmatrix} 0 & e^{i\frac{1}{2}\theta} \\ e^{-i\frac{1}{2}\theta} & 0 \end{bmatrix} \begin{bmatrix} 0 & e^{i\frac{1}{2}\theta} \\ e^{-i\frac{1}{2}\theta} & 0 \end{bmatrix} = \begin{bmatrix} e^0 & 0 \\ 0 & e^0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ XR_y(\theta) &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} = \begin{bmatrix} \sin(\theta) & \cos(\theta) \\ \cos(\theta) & -\sin(\theta) \end{bmatrix} \\ XR_y(\theta)XR_y(\theta) &= \begin{bmatrix} \sin(\theta) & \cos(\theta) \\ \cos(\theta) & -\sin(\theta) \end{bmatrix} \begin{bmatrix} \sin(\theta) & \cos(\theta) \\ \cos(\theta) & -\sin(\theta) \end{bmatrix} \\ &= \begin{bmatrix} \sin^2(\theta) + \cos^2(\theta) & \sin(\theta)\cos(\theta) - \sin(\theta)\cos(\theta) \\ \sin(\theta)\cos(\theta) - \sin(\theta)\cos(\theta) & \sin^2(\theta) + \cos^2(\theta) \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{aligned}$$

Now we have for both $R_z(\theta)$ and $R_y(\theta)$ that the following holds: $XR(\theta)XR(\theta) = I$. When both sides of the expression are multiplied with $R(-\theta)X$ we get:
$XR(\theta)XR(\theta)R(-\theta)X = R(-\theta)X$. Because of the fact that X is involutory and due to theorems 2.1, 2.2 we obtain: $XR(\theta) = R(-\theta)X$. □

3. Decomposition U gate

In this section a decomposition is given of a gate $U \in U(2)$ into three rotation gates. Since we know that for some $U' \in SU(2)$ holds that $U = U' \begin{bmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{bmatrix} \equiv U'R_z(\varphi)$ we will first discuss the decomposition of a gate $U' \in SU(2)$. U can later be decomposed by applying $R_z(\varphi)$ after

the decomposition of U'.

Let $U' \in SU(2)$. In chapter 2 we have described that for some $a, b \in \mathbb{C}$, $U' = \begin{bmatrix} a & -\overline{b} \\ b & \overline{a} \end{bmatrix}$. Since $a, b \in \mathbb{C}$, there exists $r_a, r_b \in \mathbb{R}_{\geq 0}$ and $\varphi_a, \varphi_b \in [0, 2\pi)$ such that $a = r_a e^{i\varphi_a}$ and $b = r_b e^{i\varphi_b}$. Define $\alpha \in (-2\pi, 2\pi)$ by $\alpha = \varphi_b - \varphi_a$ and define $\beta \in (-4\pi, 0]$ by $\beta = -\varphi_a - \varphi_b$.
In chapter 2, we also mentioned that the restriction $r_a^2 + r_b^2 = 1$ must apply. This restriction can be used in combination with the knowledge that $r_a, r_b \geq 0$ to find that there exists a $\theta \in [0, \frac{1}{2}\pi]$ such that $r_a = cos(\theta)$ and $r_b = sin(\theta)$. A proof can be found in appendix A. Notice that for $r_a \neq 0$: $\frac{r_b}{r_a} = \frac{\sin(\theta)}{\cos(\theta)} = \tan(\theta)$. Therefore θ can easily be calculated by applying the following method: if $r_a = 0$ then $\theta = \pi$ else $\theta = \tan^{-1}(\frac{r_b}{r_a})$.
Notice: $\varphi_a = \frac{-\alpha-\beta}{2}$ and $\varphi_b = \frac{\alpha-\beta}{2}$. Now the matrix U' can be rewritten in terms of α, β and θ:

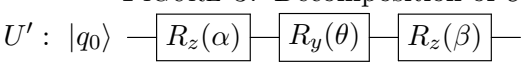
$$U' = \begin{bmatrix} a & -\overline{b} \\ b & \overline{a} \end{bmatrix} = \begin{bmatrix} e^{i\varphi_a} r_a & -e^{-i\varphi_b} r_b \\ e^{i\varphi_b} r_b & e^{-i\varphi_a} r_a \end{bmatrix} = \begin{bmatrix} e^{i\frac{-\alpha-\beta}{2}} \cos(\theta) & -e^{i\frac{\beta-\alpha}{2}} \sin(\theta) \\ e^{i\frac{\alpha-\beta}{2}} \sin(\theta) & e^{i\frac{\alpha+\beta}{2}} \cos(\theta) \end{bmatrix} \tag{16}$$

This can be decomposed in the rotation gates from table 1. By doing this, our decomposition of U' is complete:

$$\begin{aligned} U' &= \begin{bmatrix} e^{i\frac{-\alpha-\beta}{2}} \cos(\theta) & -e^{i\frac{\beta-\alpha}{2}} \sin(\theta) \\ e^{i\frac{\alpha-\beta}{2}} \sin(\theta) & e^{i\frac{\alpha+\beta}{2}} \cos(\theta) \end{bmatrix} \\ &= \begin{bmatrix} e^{-i\frac{\alpha}{2}} & 0 \\ 0 & e^{i\frac{\alpha}{2}} \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} e^{-i\frac{\beta}{2}} & 0 \\ 0 & e^{i\frac{\beta}{2}} \end{bmatrix} \\ &= R_z(\alpha) R_y(\theta) R_z(\beta) \end{aligned} \tag{17}$$

This can be decomposed in gates, this is visualized in figure 3. Now we will give a decomposition

FIGURE 3. Decomposition of U'

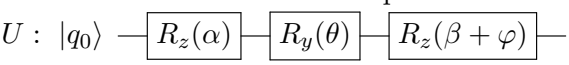


of $U \in U(2)$. Recall that $U \equiv U' R_z(\varphi)$. In section 2.2 several ways are given to obtain φ from a matrix U. The decomposition of U is therefore:

$$U \equiv R_z(\alpha) R_y(\theta) R_z(\beta) R_z(\varphi) = R_z(\alpha) R_y(\theta) R_z(\beta + \varphi) \tag{18}$$

The gate decomposition of the gate U is shown in figure 4.

FIGURE 4. Decomposition of U



4. Decomposition c-U gate

Let $U' \in SU(2)$ and $U \in U(2)$ such that: $U \equiv U' R_z(\varphi)$ with $\varphi = \arctan 2(\mathrm{Im}(\det(U)), \mathrm{Re}(\det(U)))$. From equations 17 and 18 we know that U' and U can be decomposed as:

$$\begin{aligned} U' &= R_z(\beta) R_y(\theta) R_z(\alpha) \\ U &\equiv R_z(\beta) R_y(\theta) R_z(\alpha + \varphi) \end{aligned}$$

Define c-$U' = \begin{bmatrix} I_2 & 0 \\ 0 & U' \end{bmatrix}$ and c-$U = \begin{bmatrix} I_2 & 0 \\ 0 & U \end{bmatrix}$ as two complex 4×4 matrices. In section 2.4 it is mentioned that c-U' and c-U are unitary since U' and U are unitary. These two matrices are applied on a system of two qubits q_0 and q_1 with their quantum register $|q_0 q_1\rangle \in \mathbb{C}^4$. Here q_0 is the control qubit and q_1 the target qubit. The goal in this section is to give a decomposition

of these two matrices c-U' and c-U. This decomposition is done using the two rotation gates and the c-X gate as the universal set. First, the focus will be on the decomposition of c-U'. Afterwards, it would not take much effort to show what a decomposition of c-U is. Since the c-X gate can be interpreted as: a X gate if the control qubit turns out to be 1 when measured; a I gate if the control qubit turns out to be 0. Therefore it is sufficient to only analyze the two cases where the control qubit q_0 is in the state $|0\rangle$ or $|1\rangle$.

THEOREM 4.1 (Controlled gate implementation). *$\forall\alpha,\beta\in\mathbb{R}$ and $\theta\in[0,\frac{1}{2}\pi]$ define $A=R_z(\alpha)R_y(\frac{\theta}{2})$, $B=R_y(-\frac{\theta}{2})R_z(\frac{-\alpha-\beta}{2})$ and $C=R_z(\frac{\beta-\alpha}{2})$. Then:*

$$A\ X\ B\ X\ C=R_z(\alpha)R_y(\theta)R_z(\beta)\qquad\text{and}\qquad A\ B\ C=I$$

PROOF. Let $\alpha,\beta\in\mathbb{R}$ and $\theta\in[0,\frac{1}{2}\pi]$ and define $A=R_z(\alpha)R_y(\frac{\theta}{2})$, $B=R_y(-\frac{\theta}{2})R_z(\frac{-\alpha-\beta}{2})$ and $C=R_z(\frac{\beta-\alpha}{2})$. Te theorems that are used are stated on the right side of the expressions. First we check if $A\ B\ C=I$:

$$\begin{aligned}A\ B\ C&=R_z(\alpha)R_y(\frac{\theta}{2})R_y(-\frac{\theta}{2})R_z(\frac{-\alpha-\beta}{2})R_z(\frac{\beta-\alpha}{2})\\&=R_z(\alpha)R_y(0)R_z(\frac{-\alpha-\beta}{2})R_z(\frac{\beta-\alpha}{2})&(2.1)\\&=R_z(\alpha)IR_z(-\alpha)&(2.2)\\&=R_z(0)=I&(2.1,2.2)\end{aligned}$$

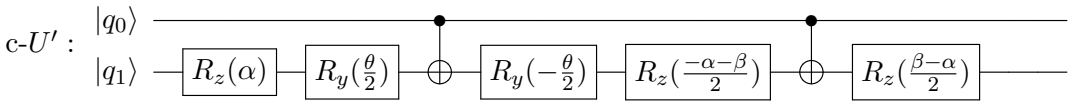
So indeed $A\ B\ C=I$. Now we check if $A\ X\ B\ X\ C=R_z(\alpha)R_y(\theta)R_z(\beta)$:

$$\begin{aligned}A\ X\ B\ X\ C&=R_z(\alpha)R_y(\frac{\theta}{2})XR_y(-\frac{\theta}{2})R_z(\frac{-\alpha-\beta}{2})XR_z(\frac{\beta-\alpha}{2})\\&=R_z(\alpha)R_y(\frac{\theta}{2})R_y(\frac{\theta}{2})XXR_z(\frac{\alpha+\beta}{2})R_z(\frac{\beta-\alpha}{2})&(2.3)\\&=R_z(\alpha)R_y(\theta)XXR_z(\beta)&(2.1)\\&=R_z(\alpha)R_y(\theta)R_z(\beta)\end{aligned}$$

□

We have already shown in which way the values α, β and θ can be obtained from a matrix U'. By using theorem 4.1 with these values α, β and θ we find that the gate decomposition of c-U'. This is visualized in figure 5. Now we will show the gate decomposition of the c-U

FIGURE 5. Decomposition of c-U'



gate. First of all, notice that unfortunately c-$U\not\equiv$ c-U'c-$R_z(\varphi)$. Even when $U\equiv U'R_z(\varphi)$, this equivalence does not hold when a matrix is 'scaled' up. To show that this equivalence does not hold, let $\alpha_0,\alpha_1,\alpha_2,\alpha_3\in\mathbb{C}$ such that: $|q_0q_1\rangle=\alpha_0|00\rangle+\alpha_1|01\rangle+\alpha_2|10\rangle+\alpha_3|11\rangle$. When theorem 4.1 is applied to the values α, $(\beta+\varphi)$ and θ from the matrix U, it gives us the same as applying the matrix c-$U'R_z(\varphi)$ to the register $|q_0q_1\rangle$. This results in:

$$\begin{bmatrix}I_2 & 0\\0 & U'R_z(\varphi)\end{bmatrix}\begin{pmatrix}a_0\\a_1\\a_2\\a_3\end{pmatrix}=\begin{pmatrix}a_0\\a_1\\a_2'\\a_3'\end{pmatrix}$$

where $\begin{pmatrix} a_2' \\ a_3' \end{pmatrix} = U' \begin{pmatrix} a_2 e^{-i\frac{1}{2}\varphi} \\ a_3 e^{i\frac{1}{2}\varphi} \end{pmatrix}$. Notice that $\begin{pmatrix} a_2' e^{i\frac{1}{2}\varphi} \\ a_3' e^{i\frac{1}{2}\varphi} \end{pmatrix} = U \begin{pmatrix} a_2 \\ a_3 \end{pmatrix}$. The register is invariant under a global phase shift, so:

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2' \\ a_3' \end{pmatrix} \equiv e^{i\frac{1}{2}\varphi} \begin{pmatrix} a_0 \\ a_1 \\ a_2' \\ a_3' \end{pmatrix} = \begin{pmatrix} a_0 e^{i\frac{1}{2}\varphi} \\ a_1 e^{i\frac{1}{2}\varphi} \\ a_2' e^{i\frac{1}{2}\varphi} \\ a_3' e^{i\frac{1}{2}\varphi} \end{pmatrix} \neq \begin{pmatrix} a_0 \\ a_1 \\ a_2' e^{i\frac{1}{2}\varphi} \\ a_3' e^{i\frac{1}{2}\varphi} \end{pmatrix} = \begin{bmatrix} I_2 & 0 \\ 0 & U \end{bmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \text{c-}U\ |q_0 q_1\rangle$$

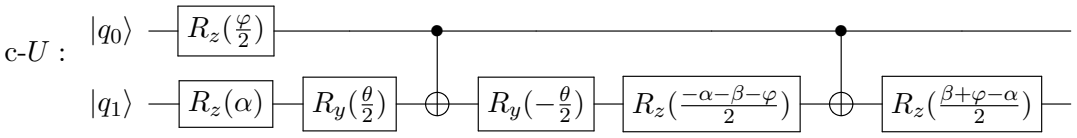
Due to the inequality above, let us go back to the definition. Recall that $U = U' \begin{bmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{bmatrix}$. We can write U using the $R_z(\varphi)$ gate such that:

$$U = U' R_z(\varphi) \begin{bmatrix} e^{i\frac{1}{2}\varphi} & 0 \\ 0 & e^{i\frac{1}{2}\varphi} \end{bmatrix} = \begin{bmatrix} e^{i\frac{1}{2}\varphi} & 0 \\ 0 & e^{i\frac{1}{2}\varphi} \end{bmatrix} U' R_z(\varphi)$$

Since we already know how the c-$U' R_z(\varphi)$ gate performance on the quantum register, we are interested in how the c-$\begin{bmatrix} e^{i\frac{1}{2}\varphi} & 0 \\ 0 & e^{i\frac{1}{2}\varphi} \end{bmatrix}$ gate works, we will check in which way this gate performance on the register and how it should be implemented. Notice that this gate is still invariant for qubit q_1, so it could only change the state of the control qubit q_0. This gate is equivalent to I_2 if q_0 is in the state $|0\rangle$ and multiplies the state α_2 and α_3 with a factor $e^{i\frac{1}{2}\varphi}$ if q_0 is in the state $|1\rangle$. Just like before it is sufficient to only check these two states. From these to statements the c-$\begin{bmatrix} e^{i\frac{1}{2}\varphi} & 0 \\ 0 & e^{i\frac{1}{2}\varphi} \end{bmatrix}$ gate should be equivalent to $\begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{1}{2}\varphi} \end{bmatrix}$ on the control qubit q_0. Since this is a single qubit gate, it is equivalent to use a $R_z(\frac{\varphi}{2})$ gate on the control qubit because they are equivalent under a global phase shift.

So theorem 4.1 (replace β with $(\beta + \varphi)$) gives us almost all the information we need for the circuit. The only thing we still need to do is apply a $R_z(\frac{\varphi}{2})$ on the control qubit. Notice that the position of the $R_z(\frac{\varphi}{2})$ gate does not matter since it could be performed before or after a c-X gate. For example, the gate decomposition of the c-U is visualized in figure 6.

FIGURE 6. Decomposition of c-U



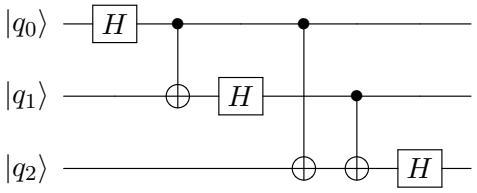
CHAPTER 4

Parallelization

1. Purpose

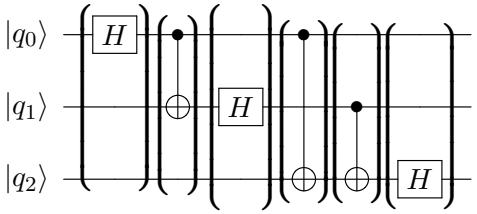
As mentioned in section 2.5, a circuit consist of a number of qubits where gates are being performed on. For example, let us look at the circuit below. In this circuit an algorithm is

FIGURE 7. Basic circuit



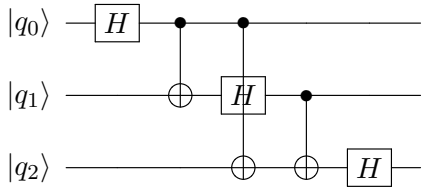
visualized. The algorithm performs a c-X on every qubit for all the qubits above it as the control. After that a H gate is performed. Recall that the gates on the left are applied before the gates on the right. Therefore, the time a circuit takes to execute is the length of the circuit. The measuring unit of time will be 'ticks'. The circuit in figure 7 is performed in a time of 6 ticks. This is visualized in figure 8. But we can do better. Gates that get performed on

FIGURE 8. Basic circuit in slices



different (control and target) qubits, can be applied simultaneously. We will call this process the parallelization of a circuit. The parallelization of figure 7 is given below in figure 9.[1] This

FIGURE 9. Parallelization of basic circuit



circuit only takes 5 ticks to execute. In this example the amount of ticks is reduced from 6 to 5. For a small circuit like this it does not make a big difference, but when algorithms get longer, more gates can be reduced. Depending on the algorithm, this can make a big difference.
The reason we want to reduce the size of a circuit is because of something we have not explained

[1]Notice that the visualization of a parallel circuit is quite confusing, but the same gates as the original circuit are used

yet: errors. There are different kinds of errors and all of them differ from quantum computer. A few examples of errors are: coherence of the qubit, a flaw in a gate and a flaw in a measurement. Unfortunately the Qx-Simulator is not able to simulate these errors individually. Nevertheless it can implement noise, which we will use to imitate the errors made on a quantum computer. The rate which noise is added, is given in advance and could be seen as a chance of the noise added after one tick in the circuit. Therefore we know that by reducing the amount of ticks of an algorithm we will also reduce the amount of noise inserted in the circuit.

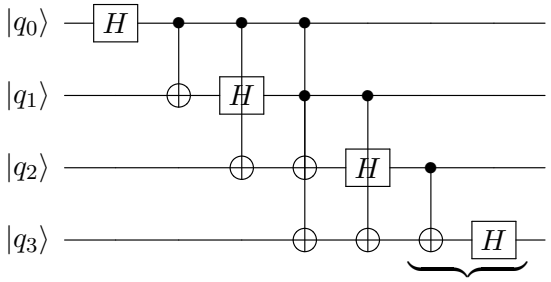
2. Bachmann & Landau notation

One often used equality regarding the analysis of a circuit is:

$$\sum_{m=1}^{M} m = \frac{1}{2}M(M+1)$$

A proof of this equality can be found in appendix A and it will be used without any further notice in the following chapters. In the previous section we have explained the importance to reduce the length of the algorithm. Recall that in figure 7 the amount of ticks reduced was only 1 and did not make a big difference. But in the example the algorithm was only used for 3 qubits, but how many ticks can be reduced if the algorithm was performed on more qubits? Since these algorithms are going to be performed on more qubits, they are analyzed by the length of the input. For example if the algorithm from figure 7 would be performed on 4 qubits instead of 3, it would be 4 ticks longer while with parallelization it would only be 2 ticks longer:[2]

FIGURE 10. Parallelization of extended basic circuit



To analyze the amount of ticks as the algorithm is applied on more qubits, the Big O notation will be introduced.

DEFINITION 2.1 (Big O). *Let $f, g : \mathbb{N} \to \mathbb{N}$. Define $f(n) = O(g(n))$ if and only if $\exists M \in \mathbb{R}_{\geq 0}, N \in \mathbb{N}$: $\forall n \in \mathbb{N}_{\geq N}$:*

$$f(n) \leq Mg(n)$$

Now we can analyze our algorithm in figure 7 and the parallelization by using the Big O notation. The input n of our function $f_1 : \mathbb{N} \to \mathbb{N}$ is the amount of qubits the algorithm is performed on. The output $f_1(n)$ will be the amount of ticks of the circuit. First we analyze example 7. Notice that by adding the m^{th} qubit, m gates are added to the algorithm. Since every gate is one tick, the total amount of ticks is:

$$f_1(n) = \sum_{k=1}^{n} k = \frac{1}{2}n(n+1) = \frac{1}{2}n^2 + \frac{1}{2}n$$

Let $g_1 : \mathbb{N} \to \mathbb{N}$ be given by $g_1(n) = n^2$. Let $M = 1$ and $N = 1$, then for all $n \geq N$ the following holds: $f_1(n) \leq Mg_1(n)$, so by definition $f_1(n) = O(n^2)$.

[2]Notice that the visualization of a parallel circuit is quite confusing, but the same gates as the original circuit are used

Now we will analyze the parallelization of the circuit. Notice that for every extra qubit (except the first one), two ticks are added. Thus $f_2 : \mathbb{N} \to \mathbb{N}$ is given by:

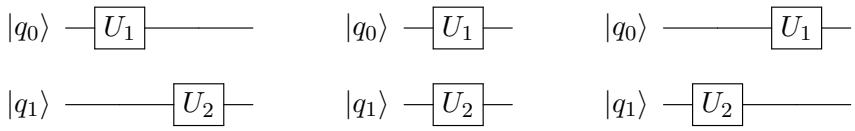
$$f_2(n) = 1 + \sum_{k=2}^{n} 2 = 1 + 2(n-1) = 2n + 1$$

Let $g_2 : \mathbb{N} \to \mathbb{N}$ given by $g_2(n) = n$. Let $M = 3$ and $N = 1$, then for all $n \geq N$ holds: $f_2(n) \leq M g_2(n)$ thus $f_2(n) = O(n)$.
The Big O notation exits to give an idea of the growth of a function. Since $f_1(n) \neq O(n)$ it can be concluded that f_1 grows faster then f_2. To illustrate this, let us consider the algorithm applied on $n = 100$ qubits: $f_1(100) = 5050$ while $f_2(100) = 201$ which is about 25 times less ticks.
Notice that if $f : \mathbb{N} \to \mathbb{N}$ is some polynomial with grade $k \in \mathbb{N}$ such that for some $a_k \in \mathbb{R}_{>0}$ and $a_{k-1}, \ldots, a_0 \in \mathbb{R}$ then holds $f(n) = a_k n^k + \cdots + a_1 n + a_0$. There can always be claimed that $f(n) = O(n^k)$. A proof can be found in appendix A. This will be used in further calculations.

3. Algorithm

Now that we have shown the parallelization of a circuit can significantly lower the amount of ticks needed, let us describe a method to make such a parallelization. The algorithm makes use of one important property: Gates acting on different (control and target) qubits can be performed in any order, even simultaneously. Of course the order of the gates that act on partially the same (control or target) qubit can not be changed. To illustrate this, let U_1 and U_2 be two single-qubit gates. the circuits in figure 11 are equivalent:

FIGURE 11. Order property illustrated



However the two circuit given in figure 12 are not equivalent in general:

FIGURE 12. Two not equivalent circuits

The algorithm is described step-by-step as it was performed by a computer. The input of the algorithm is a circuit without any parallel gates. This is important since the algorithm is going to sort the gates himself. Therefore unless two gates really need to be performed simultaneous, add them individually to a circuit:

(1) For every qubit make an empty queue.
(2) Start from the begin of the circuit (left). Number the gates in order they are executed.
(3) Start form the lowest number. Add the number of every gate to the end of the queue of all the qubits that the gate acts on.
(4) Take the lowest number at the start of the queues. Look if there are any other numbers which are at the start of a queue and not anywhere else in another queue.
(5) If so, these gate can be performed parallel with the gate from the lowest number.
(6) Remove the lowest number and the numbers that could be performed parallel from the queues.
(7) Unless all the queues are now empty, go to step 4.

Step 4 states: *'Look if there are any other numbers which are at the start of a queue and not anywhere else in another queue.'*. To do this we need to search every value at the front of a queue in the rest of all the other queues. This process is a lot of work and would take a long time to execute. However there is a way to get rid of this long search. In step 3 the queues were filled, so at this step the you know how many times (say t times) a specific gate number is added to all the queues. If an array is made with a size equal to the number of gates, at every gate number the value of t can be stored. Consequently, The search processes from step 4 could be replaced by checking if a gate is t times at the start of the queues.
In addition, this algorithm could be used to make a new circuit or by modifying the old one. This could be done by a different implementation of step 5. The code in appendix B uses the in-place version of the algorithm.

CHAPTER 5

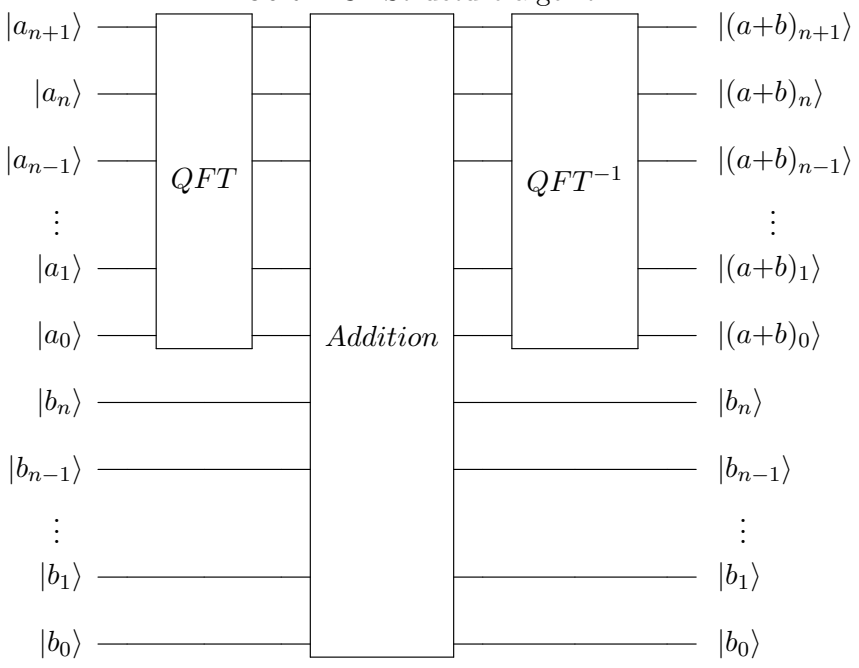
Addition algorithm

1. Structure

In this chapter we discuss the positive integer addition algorithm. In addition, the changes that were made in comparison to the paper from Thomas G. Draper are discussed. Which were necessary to make it possible to program on the Qx-simulator. The algorithm consists of three parts: the Quantum Fourier Transform by Schor (QFT), the Addition and the inverse of the QFT. They will be explained in order of execution. In section 6.3 the parallelization of the circuit is discussed. This is done by using the algorithm described in section 4.3. Before we explain the details of the different parts of the algorithm, we will show the structure of the algorithm.

Suppose we would like to add two positive integers a and b. Then there exist a $n \in \mathbb{N}$ such that $a < 2^{n+1}$ and $b < 2^{n+1}$. Since a and b are positive integers there exist $a_0, \ldots, a_n \in \{0, 1\}$ such that $a = a_n 2^n + \cdots + a_1 2^0$ and there exists $b_0, \ldots, b_n \in \{0, 1\}$ such that $b = b_n 2^n + \cdots + b_1 2^0$. Let $a_0, \ldots, a_n$ and $b_0, \ldots, b_n$ be qubits and their states being $|0\rangle$ or $|1\rangle$ corresponding to the original value. Since the answer of $a + b$ could take up $n + 2$ bits, define a qubit a_{n+1}. This qubit will be in a state $|0\rangle$ at the beginning of the algorithm. The structure used for the algorithm is shown below in figure 13.

FIGURE 13. Structure algorithm



Notice that the total amount of qubits used in this algorithm will be only $2n + 3$, which means their are no 'carry bits' used. While the addition part will perform on all the $2n + 3$ qubits, the QFT and QFT inverse parts of the algorithm will only be performed on the first $n + 2$ qubits from a.

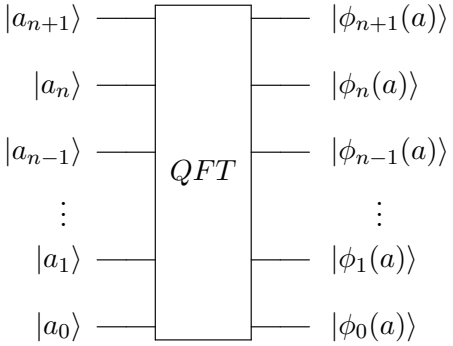
If no errors are implemented, the final states form figure 13 are always equal to a of the classical states. At the end, every qubit would be measured using a measurement gate from chapter 2.

When errors are inserted, the measurement gate is necessary in the visualization since the qubits could be in any state.
In the following sections, the restrictions of the Qx-simulator are discussed. If someone wants to use this algorithm using a different simulator or a real quantum computer, different changes may need to be made.

2. QFT

Let $k \in \mathbb{N}$, the QFT uses the H and c-R_k gates from chapter 2. Define the state $|\phi_k(a)\rangle \in \mathbb{C}^2$ by $|\phi_k(a)\rangle = \frac{1}{2}\sqrt{2}(|0\rangle + e^{i\pi a/2^k}|1\rangle)$. The purpose of the QFT is to change the state $|a_i\rangle$ into $|\phi_i(a)\rangle$ for all $i \in [0, n+1]$. This is visualized in figure 14. Since the individual states of the qubits can be obtained after the QFT, we imply that the QFT leaves the states untangled. To confirm that this claim, one can look at a more detailed QFT from figure 19. here you can see that the control qubits of all the gates are always in a classical state $|0\rangle$ or $|1\rangle$. Therefore the states are untangled at all times.

FIGURE 14. Structure QFT



Notice that $e^{i\pi a/2^k}$ is hard to read. Therefore we will introduce some notation. Let $e : \mathbb{R} \to \mathbb{C}$ given by $e(x) = e^{2i\pi x}$. Notice $e^i = e^{i2\pi k}$ for all $k \in \mathbb{N}$, therefore for all $x, y \in \mathbb{N}$ the following holds: $e(x) = e(y)$. Let $k \in [1, n+1]$, we can conclude the following:

$$\begin{aligned} a &= a_{n+1}2^{n+1} + \cdots + a_k 2^k + \cdots + a_0 2^0 \\ \frac{a}{2^k} &= a_{n+1}2^{n+1-k} + \cdots + a_k 2^0 + a_{k-1}2^{-1} + \cdots + a_0 2^{-k} \\ e(\frac{a}{2^k}) &= e(a_{k-1}2^{-1} + \cdots + a_0 2^{-k}) \\ &= e(0.a_{k-1} \dots a_0) \end{aligned}$$

Where $0.a_{k-1} \dots a_0$ is the binary representation of the number $a_{k-1}2^{-1} + \cdots + a_0 2^{-k}$. Since $a \in \mathbb{N}$ we know that for $k = 0$ we have $e(\frac{a}{2^k} = e(a) = e(0)$. Using this notation, the state $|\phi_k(a)\rangle$ becomes:

$$|\phi_k(a)\rangle = \frac{1}{2}\sqrt{2}(|0\rangle + e(a/2^{k+1})|1\rangle) = \frac{1}{2}\sqrt{2}(|0\rangle + e(0.a_k \dots a_0)|1\rangle) \tag{19}$$

We will show how the states $|\phi_k(a)\rangle$ can be derived. Let $k \in [0, n+1]$, we will consider the state $|a_k\rangle$. Every step we make in the calculation is related to the algorithm of the QFT which is given in detail in figure 19. Note that $|a_k\rangle$ is in a state $|0\rangle$ or $|1\rangle$ at the start of the algorithm. In the beginning, the qubit may be used as a control qubit for multiple gates, but since the qubit is in a classical state, this will not change the state $|a_k\rangle$.
First the the H gate will be applied to the state $|a_k\rangle$. Notice that if $a_k = 0$, then

$$e(a_k 2^{-1}) = e(0) = e^{2i\pi 0} = 1$$

and if $a_k = 1$, then

$$e(a_k 2^{-1}) = e(2^{-1}) = e^{2i\pi 2^{-1}} = e^{i\pi} = -1$$

With this in mind, applying the H gate results in the following:

$$\begin{aligned}|a_k\rangle \implies &\begin{cases}\frac{1}{2}\sqrt{2}(|0\rangle+|1\rangle) & \text{If } a_k=0\\ \frac{1}{2}\sqrt{2}(|0\rangle-|1\rangle) & \text{If } a_k=1\end{cases}\\ &=\frac{1}{2}\sqrt{2}(|0\rangle+e(a_k2^{-1})|1\rangle)\\ &=\frac{1}{2}\sqrt{2}(|0\rangle+e(0.a_k)|1\rangle)\end{aligned}$$

Afterwards, there will be k times a c-$R_{k'}$ gate applied to the qubit for some $k'\in[2,k+1]$. The control qubits are still in a state $|0\rangle$ or $|1\rangle$ when the c-$R_{k'}$ gates are applied. Therefore we will first consider these two cases separately and later obtain a general expression. The first c-$R_{k'}$ gate that is applied is a c-R_2 gate with a_{k-1} as the control qubit. The state will change in the following way:

$$\begin{aligned}\frac{1}{2}\sqrt{2}(|0\rangle+e(0.a_k)|1\rangle) \implies &\begin{cases}\frac{1}{2}\sqrt{2}(|0\rangle+e(0.a_k)|1\rangle) & \text{If } a_{k-1}=0\\ \frac{1}{2}\sqrt{2}(|0\rangle+e(0.a_k)e(2^{-2})|1\rangle) & \text{If } a_{k-1}=1\end{cases}\\ =&\frac{1}{2}\sqrt{2}(|0\rangle+e(0.a_k)e(a_{k-1}2^{-2})|1\rangle)\\ =&\frac{1}{2}\sqrt{2}(|0\rangle+e(0.a_k)e(0.0a_{k-1})|1\rangle)\\ =&\frac{1}{2}\sqrt{2}(|0\rangle+e(0.a_k+0.0a_{k-1})|1\rangle)\\ =&\frac{1}{2}\sqrt{2}(|0\rangle+e(0.a_ka_{k-1})|1\rangle)\end{aligned} \tag{20}$$

Next a c-R_3 gate will be applied with a_{k-2} as the control qubit. To calculate the new state, a calculation similar to equation 20 is made and will result in the following:

$$\frac{1}{2}\sqrt{2}(|0\rangle+e(0.a_ka_{k-1})|1\rangle) \implies \frac{1}{2}\sqrt{2}(|0\rangle+e(0.a_ka_{k-1}a_{k-2})|1\rangle)$$

This process will be done for all k'. Due to equation 19, the state obtained after this process (i.e. the QFT) will be:

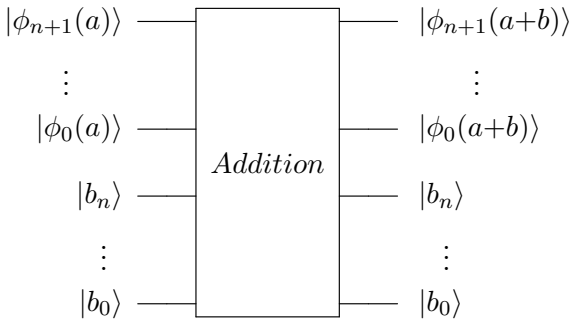
$$|a_k\rangle \implies \frac{1}{2}\sqrt{2}(|0\rangle+e(0.a_ka_{k-1}\dots a_0)|1\rangle)=|\phi_k(a)\rangle \tag{21}$$

Indeed, this process results in the correct states after the QFT is performed.

3. Addition

The addition part of the algorithm needs to change the states $|\phi_k(a)\rangle$ to $|\phi_k(a+b)\rangle$ for all $k\in[0,n+1]$. This is visualized in figure 15. Notice that the states from b will be unchanged during this process using the same reasoning as described in the previous section.

FIGURE 15. Structure of the Addition



Notice that the addition is only making use of c-$R_{k'}$ gates for $k'\in[1,n+2]$ where the control qubit is one of the states of b. To calculate the new state, the same calculations are made during

equation 20. Let $k \in [0, n+1]$ when the c-R_1 gate is applied with b_k as the control qubit, the state will change in following way:

$$|\phi_k(a)\rangle = \frac{1}{2}\sqrt{2}(|0\rangle + e(0.a_k a_{k-1} \dots a_0)|1\rangle)$$
$$\implies \frac{1}{2}\sqrt{2}(|0\rangle + e(0.a_k a_{k-1} \dots a_0 + 0.b_k)|1\rangle)$$

This is done until b_0 and results in:

$$|\phi_k(a)\rangle \implies \frac{1}{2}\sqrt{2}(|0\rangle + e(0.a_k a_{k-1} \dots a_0 + 0.b_k b_{k-1} \dots b_0)|1\rangle) = |\phi_k(a+b)\rangle \tag{22}$$

Unfortunately the c-R_1 gate can not be implemented in the Qx-simulator. Nevertheless, notice that the c-Z gate can be used instead. Recall from chapter 2 that those two gates are the same. Notice that b_{n+1} does not exist and will be treated as 0. In the Qx-simulator, the c-R_k gate is implemented in such a way that it is easy to program the QFT. Therefore the number k is related to 'distance' of the control and target qubit. In order to still use these gates for the addition, we need to implement SWAP gates to correct the distance between the qubits. This is visualized in figure 15. Notice that the first c-Z gate is not implemented since b_{n+1} would always be zero and therefore would never be executed.

4. Inverse of the QFT

The QFT^{-1} is the inverse of the QFT explained in section 5.2. Since every gate is an unitary matrix multiplication with the state of the system, the multiplication can be undone by the inverse matrix.
For example, let $|\psi\rangle$ be a quantum register of $m \in \mathbb{N}$ qubits. On this system an algorithm A is applied. This algorithm first applies the gate A_1 to the system, then the gate A_2 and so on until the gate A_p for some $p \in \mathbb{N}$. Since every gate is an unitary matrix, the inverse of A_i will be $A_i^\dagger$. The inverse of the algorithm A will be A^{-1}, which means applying $A_p^\dagger$ to the system, then the gate $A_{p-1}^\dagger$ and so on until $A_1^\dagger$ is applied. So applying A to the state $|\psi\rangle$ results in:

$$A_p A_{p-1} \dots A_2 A_1 |\psi\rangle$$

By applying the A^{-1} to this state the following holds

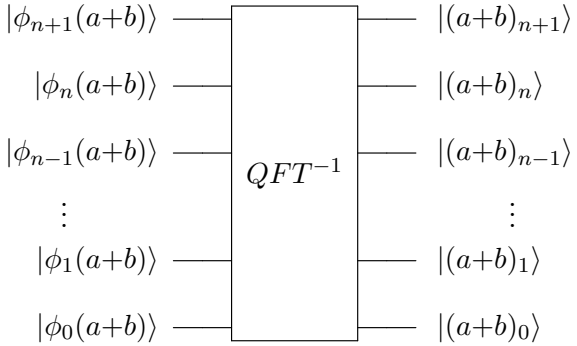
$$\begin{aligned} A^{-1}(A|\psi\rangle) &= A_1^\dagger A_2^\dagger \dots A_{p-1}^\dagger A_p^\dagger (A_p A_{p-1} \dots A_2 A_1 |\psi\rangle) \\ &= A_1^\dagger A_2^\dagger \dots A_{p-1}^\dagger A_{p-1} \dots A_2 A_1 |\psi\rangle \\ &\vdots \\ &= A_1^\dagger A_2^\dagger A_2 A_1 |\psi\rangle \\ &= A_1^\dagger A_1 |\psi\rangle \\ &= |\psi\rangle \end{aligned}$$

This can applied to the QFT and the QFT^{-1}. The structure of the QFT^{-1} is given on the next page in figure 16.
Unfortunately, the c-$R_k^\dagger$ gate is not implemented in the Qx-simulator for $k \geq 2$. Recall from chapter 3 that we can decompose every controlled gate using the c-X, $R_z(\theta)$ and $R_y(\theta)$ gates. Since $R_k = Z(\frac{2\pi}{2^k}) = \begin{bmatrix} 1 & 0 \\ 0 & e^{2i\pi 2^{-k}} \end{bmatrix}$, we know that for some $a, b \in \mathbb{C}$ and $\varphi \in \mathbb{R}$ the following holds

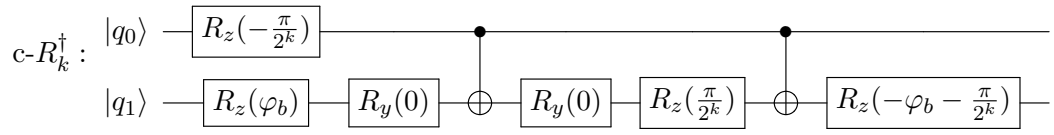
$$R_k^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & e^{-2i\pi 2^{-k}} \end{bmatrix} = \begin{bmatrix} a & -\overline{b}e^{i\varphi} \\ b & \overline{a}e^{i\varphi} \end{bmatrix}$$

So $a = 1$, $b = 0$ and $\varphi = -2\pi 2^{-k}$. Since $a, b \in \mathbb{C}$ let $r_a, r_b \in \mathbb{R}_{\geq 0}$ and $\varphi_a, \varphi_b \in \mathbb{R}$ such that $a = r_a e^{i\varphi_a}$ and $b = r_b e^{i\varphi_b}$. Notice since $a = 1$ that $r_a = 1$ and $\varphi_a = 0$. Also notice that $r_b = 0$ and φ_b is unknown. Define $\theta = \tan^{-1}(\frac{r_b}{r_a}) = \tan^{-1}(0) = 0$, $\alpha = \varphi_b - \varphi_a = \varphi_b$ and

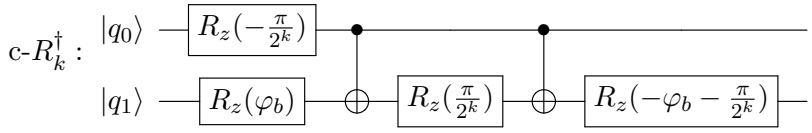
FIGURE 16. Structure QFT^{-1}



$\beta = -\varphi_a - \varphi_b = -\varphi_b$. The general decomposition is given by figure 6. Using the values of $R_k^\dagger$ we obtain the following



Using theorem 2.2 this decomposition can be reduced to

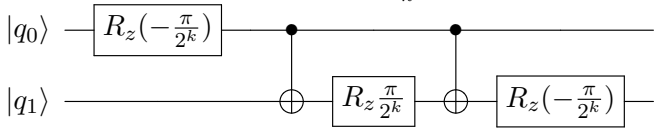


Because of theorem 2.1, the order of two gates $R_z(\theta)$ and $R_z(\phi)$ can be interchanged when they act on the same qubit. Since the value of φ_b can not be known, it would make sense to isolate the variable from the decomposition if possible. As usual, we have to consider the two cases where the control qubit turns out to be 0 or 1 after a measurement. So it is sufficient to check every possibility for the cases where $|q_0\rangle$ is in a state $|0\rangle$ or $|1\rangle$ at the moment of the c-X gates. Notice that the $R_z(\theta)$ gate is a rotation around the z axis of the Bloch sphere representation of a qubit. If a qubit is in the states $|0\rangle$ or $|1\rangle$ it is invariance under such a rotation since the qubit is located on the z axis. If q_0 is in a state $|0\rangle$ the only gates that are performed on the state $|q_1\rangle$ are $R_z(\theta)$ gates and thus the order can be interchanged. If q_0 is in a state $|1\rangle$, the following gates are performed:

$$\begin{aligned} R_z(\varphi_b)XR_z(\frac{\pi}{2^k})XR_z(-\varphi_b - \frac{\pi}{2^k}) &= XR_z(-\varphi_b)R_z(\frac{\pi}{2^k})XR_z(-\varphi_b - \frac{\pi}{2^k}) \\ &= XR_z(\frac{\pi}{2^k})R_z(-\varphi_b)XR_z(-\varphi_b - \frac{\pi}{2^k}) \\ &= XR_z(\frac{\pi}{2^k})XR_z(\varphi_b)R_z(-\varphi_b - \frac{\pi}{2^k}) \end{aligned}$$

So the $R_z(\varphi_b)$ gate can in both cases be executed behind the two c-X gates. Due to theorems 2.1 and 2.2, the variable φ_b can be left out of the decomposition, this is visualized in figure 17.

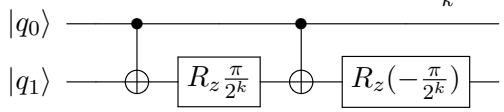
FIGURE 17. c-$R_k^\dagger$ in detail



This is the right way to implement the c-$R_k^\dagger$ gate. However, one little reduction can be made due to the knowledge we have about the algorithm. At the start of the algorithm all the qubits

are in the states $|1\rangle$ or $|0\rangle$. Notice this is also the case at the end of the algorithm. This means that after applying several $R_z(-\frac{\pi}{2^k})$ gates on a control qubit, this qubit is in a state on the z axis of the Bloch sphere representation. Since this gate is a rotation around the z axis, it will not change the state of the control qubit. Notice that the state of the target qubit will be unchanged as well because it will be a global phase shift seen from $|q_1\rangle$. Therefore, the $R_z(-\frac{\pi}{2^k})$ gate of the control qubit can be removed from the decomposition. The new decomposition is visualized in figure 18 and the QFT^{-}1 is complete.

FIGURE 18. Reduced c-$R_k^\dagger$



An example of the addition algorithm for $n = 3$ is given in quick by the following url: `https://tinyurl.com/y8eqapk4`. This url can be used as a visualization and experimentation of the algorithm. An general visualization is given in the figures 19 and 20 on the next page.

FIGURE 19. QFT in detail

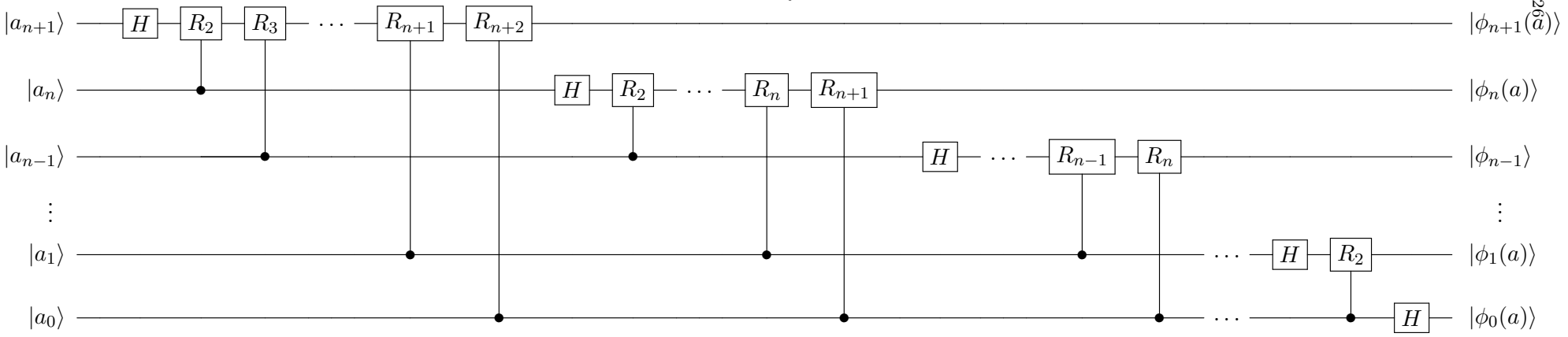
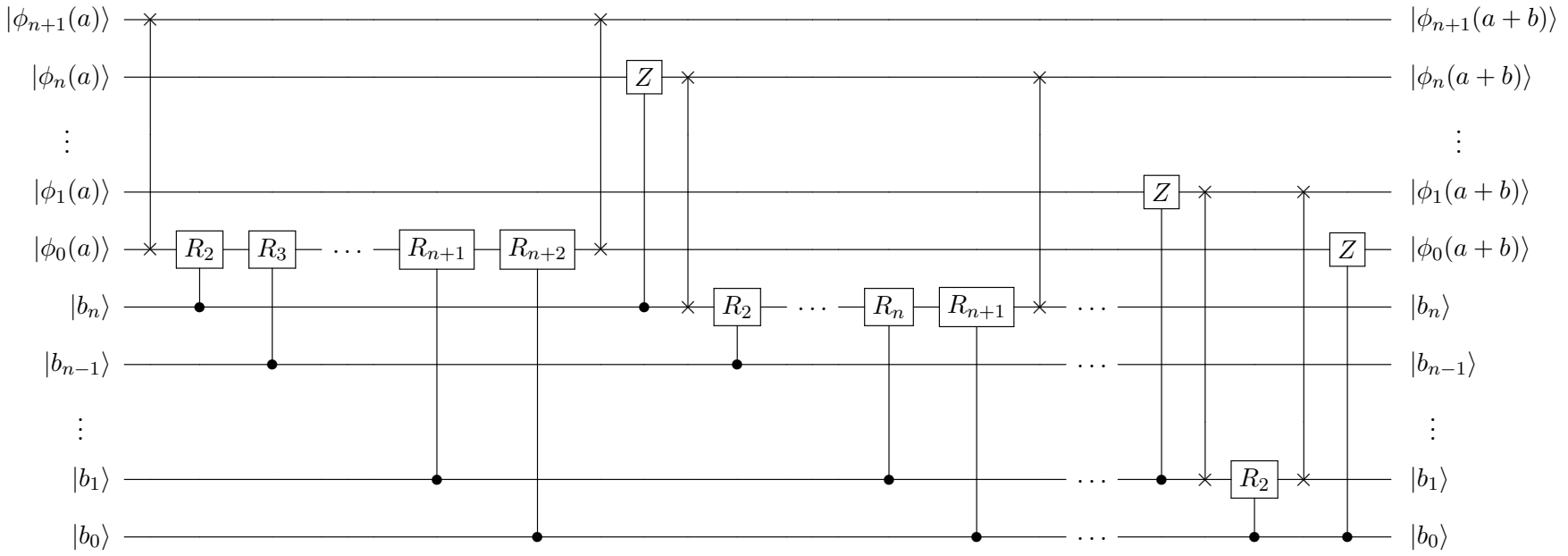


FIGURE 20. Addition in detail



5. Error model

An error model is useful since we want to implement a specific amount of noise into the circuit. The error model from the Qx-simulator is called a 'depolarization channel'. The error model depends on a predetermined probability $p \in [0, 1]$. This probability is the chance an error occurs per qubit per tick. An error is visualized in the following way: —[E]—
An error gate can be one of three things:

- A x-rotation: the X gate
- A y-rotation: the $R_y(\frac{1}{2}\pi)$ gate
- A z-rotation: the $R_z(\pi)$ gate ($\equiv Z(\pi)$ gate)

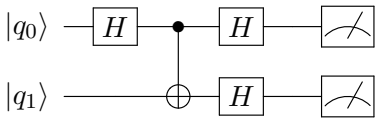
Notice that an error gate is always a π rotation around one of the axis of the Bloch sphere representation. Therefore, an error gate is involutory. Also notice that a z-rotation is the same as a 'x and y' rotation:

$$XR_y(\frac{1}{2}\pi) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = Z(\pi) \equiv R_z(\pi)$$

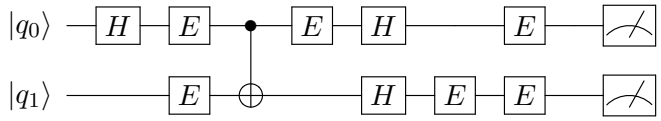
Note that the same argument holds in different cases: a y-rotation is a 'x and z'-rotation and a x-rotation is a 'y and z'-rotation. Which means that multiple error gates in a row can always be reduced to at most one single error gate. The x-rotation error is often called a 'bit flip' and the z-rotation is a 'phase flip'.
Suppose we have a circuit of N qubits with a total of t ticks. Note that the circuit can have parallel gates. Therefore the amount of gates is always more or equal to the number of ticks t. Let p be the predetermined probability addressed earlier. Define $E : \{0, \ldots, N\} \to [0, 1]$ by:

$$E(i) = \sum_{j=0}^{i} \frac{N!}{j!(N-j)!} p^j (1-p)^{N-j}$$

Then $E(i)$ is the probability that i or less errors occur at the same tick.
For every tick, a value $x \in [0, 1]$ is uniformly distributed. If $x \leq E(0)$, no errors will occur in this tick. Else, there will be an $i \in \{1, \ldots, N\}$ such that: $E(i-1) < x \leq E(i)$. Which means that i error gates will be inserted directly after this tick. What kind of error gate will be implemented is uniformly distributed. Therefore they all have the same probability to occur. Note that the error from a measurement is added before the measurement gate instead of after the gate. This will have some consequences since we want to analyze our algorithm without measuring the qubits. In this way the complete states can be obtained and we only have to do multiple runs to implement different errors.
Consider the following circuit:



If the probability an error occurs per qubit per tick is $0 < p < 1$. The circuit with implemented noise could look like:



Where each E gate is one of the error gates. Recall that two error gates in a row can always be described with a at most one single error gate. The probability for an to error at one qubit is p without measurement gates. However with measurement gates, the probability for an error at this tick would be $2p - \frac{4}{3}p^2$. A proof can be found in appendix A. In the analysis we choose $p \in (0, 0.1]$, for p close to 0 we know that $2p - \frac{4}{3}p^2 \approx 2p$. Therefore the implemented errors from the last tick in the Qx-simulator will be around half the amount of errors we would like to be implemented. This is solved by adding an identity gate to the end of the circuit. In this way the error from the measurement gate is implemented on purpose.

CHAPTER 6

Analysis

1. Structure

We defined our algorithm, but we still want to know how it preforms. Therefore we will do a theoretical analysis and multiple simulations. This is done for both the normal and the parallelization of the circuit. In this way we can compare both circuits and give some meaning to the results. In the sections regarding the theoretical analysis, the amount of ticks is calculated. Therefore, we are using the knowledge obtained from section 4.2 to analyze our algorithm and his parallelization. In section 6.4 the results from the Qx-simulator are shown. This is done for different input sizes and error rates. The analysis of the algorithm will be separated in three different parts: the QFT, Addition and QFT^{-1}. These parts are defined in chapter 5 as well as a predetermined number $n \in \mathbb{N}$ such that $a < 2^{n+1}$ and $b < 2^{n+1}$, which implies that a and b can be written in binary using $n+1$ qubits. To make the calculations more readable, define $N = n + 1$. Since the algorithm scales with the input size, the algorithm can also be analyzed with the number N. Recall that the QFT and QFT^{-1} are acting on the first $N+1$ qubits while the addition is acting on all the $2N+1$ qubits. Note that the formulas for calculating the amount of ticks of both the normal circuit and the parallelization are checked up to $N = 30$. Only at $N = 1$ there is a difference of 1 in the parallelization, but this will be explained in section 6.3 at the moment the 'mistake' is made.

2. Theoretical Analysis

All the parts are discussed separately. The figures 18, 19 and 20 will be used as reference points. Since we look at the original algorithm, the amount of ticks is equal to the number of gates.

QFT:

On all the $N+1$ qubits a H gate is performed. Also for all $k \in [0, N+1]$, the qubit a_k is a target qubit of k c-$R_{k'}$ gates for some k'. Therefore the number of gates is equal to

$$\begin{aligned} N + 1 + \sum_{j=0}^{N} j &= N + 1 + \frac{1}{2}(N+1)N \\ &= N + 1 + \frac{1}{2}N^2 + \frac{1}{2}N \\ &= \frac{1}{2}N^2 + \frac{3}{2}N + 1 \end{aligned} \tag{23}$$

Addition:

The addition part is a bit more difficult. In general, a qubit a_k is the target of a c-Z gate, two SWAP gates and k c-$R_{k'}$ gates for some k'. There are two exceptions: on a_0 there are none SWAP gates executed and on a_{N+1} there is not

a c-Z gate performed. The general form is:

$$\begin{aligned} -2-1+\sum_{j=0}^{N}(3+j) &= -3+3(N+1)+\sum_{j=0}^{N} j \\ &= 3N+\frac{1}{2}N(N+1) \\ &= 3N+\frac{1}{2}N^2+\frac{1}{2}N \\ &= \frac{1}{2}N^2+\frac{7}{2}N \end{aligned} \tag{24}$$

QFT^{-1}:

As stated before, the algorithm of the QFT^{-1} normally uses the same amount of gates as the QFT. Since we are using the decomposition shown in figure 18, every c-$R_k^\dagger$ takes up 4 gates. Therefore, the amount of gates of the QFT^{-1} would be:

$$\begin{aligned} N+1+\sum_{j=0}^{N}4j &= N+1+4\sum_{j=0}^{N} j \\ &= N+1+2N(N+1) \\ &= N+1+2N^2+2N \\ &= 2N^2+3N+1 \end{aligned} \tag{25}$$

The total amount of gates (and thus the amount of ticks) of the algorithm can be obtained by adding equations 23, 24 and 25 together:

$$\frac{1}{2}N^2+\frac{3}{2}N+1+\frac{1}{2}N^2+\frac{7}{2}N+2N^2+3N+1 = 3N^2+8N+2 \tag{26}$$

In chapter 4 we have shown that the expression from equation 26 is equal to $O(N^2)$. Since $N = n+1$ we can conclude that the algorithm will be executed in $O(n^2)$ ticks.

3. Theoretical Analysis after Parallelization

Since the parallelization algorithm is written for a general circuit, we can only obtain the parallel circuit by following the rules of the algorithm. Recall that the parallelization algorithm only uses the property shown in figure 11 and will try to execute every gate as soon as possible. Following this rule, we notice that the three different parts (QFT, Addition & QFT^{-1}) can not be parallelized between one another. This is important, because now the different parts can independently be reviewed.

QFT:

Since the k from a c-R_k gate is related to the distance of the control and target qubit from the gate, it is possible to make some statements about them. Notice that a H gate can be performed simultaneously with a c-R_k gate for $k \geq 3$ when the H gate is one qubit below the target qubit of the c-R_k gate. In a similar matter, for $k \geq 2$ notice that a c-R_k gate can be performed simultaneously with a c-$R_{k'}$ gate for $k' \geq k+2$, but only when the c-R_k gate is one qubit below the target qubit of the c-$R_{k'}$ gate.

Let $j \in [0, N]$ we will examine the qubit a_j. The qubit a_j is the target qubit of one H gate and j times a c-R_k gate for some $k \in [2, j+1]$. The qubit below a_j is a_{j-1}. Because of the previous statement, all of the gates with target qubit a_{j-1} can be performed simultaneously with a_j from the c-R_3 gate. This is the third gate from applied on a_j. Since there is one less gate with target qubit a_{j-1}, only

one gate can not be performed together with a_j. The QFT is applied on $N+1$ qubits. Therefore the amount of ticks of the QFT will be:

$$1 + N + \sum_{j=0}^{N-1} 1 = 1 + N + N = 2N + 1 \tag{27}$$

Addition:

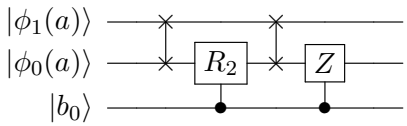
The addition part is applied on all the qubits. Notice that all the gates from the addition only have target qubits of a. We are using the circuit representation shown in figure 20. Suppose a consist of $N+1$ qubits. The addition consist of:

- N c-Z gates
- $2N$ SWAP gates
- $k-1$ c-$R_{k'}$ gates for every qubit a_k

We can distinguish $N+1$ different pieces of the addition algorithm. One piece consist of one c-Z gate, 2 SWAP gates with $k-1$ times a c-$R_{k'}$ gate in between. The exceptions are the first piece (which does not have a c-Z gate) and the last piece (which does not have SWAP gates). Therefore the first piece consist of a total of $N+2$ gates. Notice that the individual pieces can not be reduced in amount of ticks after parallelization since all the gates are acting on the same qubit. The next piece also consist of $N+2$ gates. This piece can be performed completely parallel with the first piece accept two gates, because the c-Z gate can not be performed simultaneously with the c-R_2 gate from the previous piece. This holds for all the next pieces. Therefore the amount of ticks can be described with:

$$\begin{aligned} N + 2 + \sum_{j=0}^{N-2} j &= 2 + N + 2(N-1) \\ &= 2 + N + 2N - 2 \\ &= 3N \end{aligned} \tag{28}$$

Note if $N = 1$, the unlucky situation happens that the swap gate and the c-Z gate are performed on the same qubit. This situation is illustrated below:



In this case, the amount of ticks can not be reduced to $3N = 3$, but only to $3N+1$. Therefore, our final formula will not hold for the case $N = 1$, but only for $N \geq 2$.

QFT^{-1}:

The QFT^{-1} part can be distinguished in $N+1$ different parts, each part consists of all the gates with target qubit a_k. Notice that every part $k \in [0, N]$ consist of $4k+1$ gates. Also notice that the first two parts ($k = 0$ and $k = 1$) of the QFT^{-1} will always be performed for $N \geq 1$. These two parts will have a total of 6 gates which can not be performed simultaneous. For every other part, it holds that $4(k-1)-2$ gates can be performed simultaneous with the previous part. The reason for this is that the first c-X gate of a part will be held back by the second c-X gate from the previous part. The amount of gates form a part that can not be performed simultaneously with the previous part are: $4k+1-(4(k-1)-2) = 7$. Therefore the total amount of ticks will be:

$$6 + \sum_{j=2}^{N} 7 = 6 + 7(N-1) = 7N - 1 \tag{29}$$

The total amount of ticks of the algorithm after parallelization can be obtained if the three equations are added together:

$$2N + 1 + 3N + 7N - 1 = 12N \tag{30}$$

Which means that the order of ticks is $O(N)$. Since $N = n + 1$ we can conclude that the parallelization of the algorithm will execute in $O(n)$ time. Notice that this is significantly lower than the $O(n^2)$ time for the normal circuit to execute.

4. Results

Both the normal circuit and the parallelization are ran with the same input values and error rates. The input sizes variate between 1 qubit up to 8 qubits per number ($N \in [1, 8]$). There are two reasons why the program is not executed for $N > 8$.

- the runtime of 1000 runs for 8 qubits per number is around the 70 seconds and is increasing fast. For example, the runtime of 1000 runs for 7 qubits per number is around the 15 seconds.
- My computer can run inputs up to $N = 11$, because he needs to make a register that can store 2^{2*N+1} states. If $N = 12$ this would be 2^25 states, which is around 33.6 million.

The results up to $N = 8$ already give a great impression of the benefits. So the cases where $N \in [9, 11]$ are not of added value. The errors are chosen to be $10^{-\frac{1}{2}i}$ with $i \in [2, 5]$. In this way, the error is decreasing every time with the same factor $10^{-\frac{1}{2}}$. The reason why $i \in [2, 5]$ is because $i \leq 2$ will give really bad results and $i \geq 5$ will give really good results. Therefore the interesting range will be $i \in [2, 5]$.
To do these simulations, the code from appendix B with the following settings were used:

- push: 1000
- measurement: 0
- show_circuit: 0
- error_testing:1
- parallel_optimize: 1 or 0
- error: $10^{-\frac{1}{2}i}$ with $i \in [2, 5]$
- analyze_gates: 0

A simulation can give a lot of different information. Using the settings above, it will give an approximation on the probability of measuring every possible number. This is done with or without parallelization, different error rates and different input sizes. When there is chosen to add two numbers of size N, the program decides to add the two number a and b with $a = b = 2^N - 1$. This is done because the difference in input should not make a difference on the error as long the input uses the same amount of qubits.
In binary notation, we are adding $a = 11 \ldots 11$ and $b = 11 \ldots 11$ together. Both numbers consist of the same amount of N qubits. The answer the algorithm should give without errors is $a + b = 11 \ldots 110$, which has a length of $N + 1$ qubits. Since we are adding errors into the algorithm, the program may give a different answer then the original correct solution. In fact, if the algorithm is not measured at the end, we can derive all the probabilities of measuring a specific quantum state since we are working in a simulator. By running the program again, different errors will be inserted in the algorithm and thus a different distribution will be found. Our final approximation on the probability of measuring a specific state with random errors is obtained by running the program a lot of times. We have chosen to do this 1000 times. For a quantum algorithm, this does not sound as many runs. Nevertheless it is sufficient to do since the only reason we run the program multiple times is to insert different errors and not to obtain one single distribution, because the single distribution is obtained after just one run of the algorithm. Still for the relevant values, another calculation is made with 10000 runs. In the final tables this value is used instead of the one obtained with 1000 runs.

In the tables 3 and 4, the error rate is given in the upper row and the input size N is given in the left column. In every cell, there are two values: the left value is the probability of the correct answer; the right value is the biggest probability of every value except the correct value. These values are imported, because if the first value is smaller than the second one, the algorithm is unable to preform any calculation corresponding to the given input size N and error rate. Those cells are marked red and the others are marked green. Notice that the values of some cells are relatively close to each other. For example look at table 3 with input size $N = 8$ and error rate $10^{-\frac{5}{2}}$. The probability that the correct value is measured is: 0.00834128 and the biggest probability a specific wrong answer is measured is 0.00823629. Someone could say that with enough measurements you can distinguish the correct from the wrong answer because $0.00834128 > 0.00823629$. Notice that we have two problems. Since the values are very close together you would need a lot of measurements before you could say with a given certainty what the correct answer is. In theory this is not a limit so it should be possible. The second and more serious problem is that another 10000 runs may give slightly different values, because we only use a finite amount of runs and we are trying to approximate the true value[1]. It is possible that for the true values the opposite is true. For this reason, this cell is marked red instead of green. Notice we have some white spots in the tables. These cells are not interesting since the algorithm is already not able to give the right answer for a lower input size N. For unity reasons we have chosen to only leave out the same values in both the tables.

	0,1		$10^{-\frac{3}{2}}$		0,01		$10^{-\frac{5}{2}}$	
1	0.27045	0.3793	0.50545	0.2752	0.78965	0.1233	0.92285	0.0463
2	0.134061	0.221523	0.165182	0.209176	0.451353	0.134284	0.762621	0.0570876
3	0.0601436	0.112097	0.0683512	0.116162	0.191802	0.105916	0.540766	0.0754021
4	0.0336509	0.0611537	0.0351125	0.0589036	0.064375	0.0645881	0.306778	0.0802711
5					0.0224336	0.031892	0.154869	0.0575671
6					0.00798384	0.0176539	0.0654961	0.033179
7					0.00398747	0.0076473	0.0252142	0.0167067
8					0.00254026	0.00363275	0.00834128	0.00823629

TABLE 3. Results: Normal circuit

	0,1		$10^{-\frac{3}{2}}$		0,01		$10^{-\frac{5}{2}}$	
1	0.29475	0.3695	0.54555	0.27185	0.8158	0.11735	0.93645	0.04195
2	0.110416	0.230068	0.239152	0.203304	0.569495	0.115691	0.837026	0.0445888
3	0.0581316	0.114572	0.096711	0.122477	0.341537	0.102147	0.697436	0.0509187
4	0.0259028	0.0583002	0.0382769	0.0672328	0.183066	0.0726129	0.543162	0.0579935
5					0.0839273	0.0450361	0.407117	0.0574072
6					0.0412412	0.0270095	0.283642	0.049151
7					0.0177059	0.0131818	0.191996	0.0404665
8					0.00647699	0.00675828	0.116269	0.0290022

TABLE 4. Results: Parallelization of the circuit

[1]Note that their is not really a true value since we are using random noise, our true value is an average of all possible conditions

With the knowledge from sections 6.2 and 6.3, these results are not surprising. In the tables we can see that the parallelization preforms much better than the original circuit for bigger input sizes N. This is due the number of ticks which is of order $O(N^2)$ for the original circuit and $O(N)$ for the parallelization. Since the amount of ticks is lower as the input N gets larger, there is less time for errors to be inserted and thus the algorithm preforms better. All the green cells in the tables are calculations that can be made if the amount of runs is big enough. Recall that this can be done without using any quantum error correction code or carry bits. Therefore we are just using $2N + 1$ qubits for the calculation.

CHAPTER 7

Conclusion and Outlook

In this paper we have shown a way to implement the + operator for positive integers on the Qx-simulator. Depending on the quantum computer or simulator, different changes need to be made regarding the algorithm. We have given an exact calculation on the amount of ticks the parallelization and the normal circuit needs to execute the algorithm. From this exact calculation we could derive that the normal circuit executes in $O(n^2)$ time and the parallelization executes in $O(n)$ time. Which means that the parallelization will execute faster for larger input values. Therefore, the amount of time for errors to occur is lower and thus less errors are made. This result could also be seen in the simulations made using the Qx-simulator.
There is still one important questions that needs an answer: 'given the input size, error rate and a certainty α. How many runs do we need such that we can measure the correct value with a certainty of α?' When the answer is known, we can prefix the amount of runs given the input size, error rate and the value α. Which will be more reliable than using a random amount of runs.
In the paper from Thomas G. Draper, he refers to a AQFT. He claims that this could be used instead of a QFT. This could reduce the amount of ticks needed to $log_2(n)$. However, experimentations with the AQFT did not give reliable results at all. This may be a solution if someone wants to use this addition algorithm for way bigger input sizes than used in this paper.
Now we are able to add two positive integers together, we can use the same technique to make other calculations. In fact, when we are able to add two positive integers of size $N \in \mathbb{N}$ given the error rate, we are also able to add two positive integers of size $m \in \mathbb{N}$ with $m > N$. This can be done by dividing the two numbers in parts of length N. Notice that we will need carry bits since this will be done in a classical way.[1]
Since we can add two finite numbers together, this strategy can be performed on numbers $x \in (0, 1)$ with a finite binary expression. Suppose a number $x \in (0, 1)$ could be described using $n+1$ bits such that: $x = 0.x_n x_{n-1} \dots x_0$ with $x_0, \dots x_n \in \{0, 1\}$. If a number $y \in (0, 1)$ could be described in the same way, then $x + y$ can be calculated. This is done by adding $x_n x_{n-1} \dots x_0$ and $y_n y_{n-1} \dots y_0$ together. The final answer will be: $(x + y)_{n+1}.(x + y)_n(x + y)_{n-1} \dots (x + y)_0$.
Any real numbers $x, y \in \mathbb{R}$ with a finite binary expression can now be added together. This is done by dividing x and y in two parts: $x = x' + x''$, $y = y' + y''$ with $x', y' \in N$ and $x'', y'' \in (0, 1)$. These two parts can be calculated using the just described strategies. Therefore, we obtain $(x' + y')$ and $(x'' + y'')$. Notice that we need one carry bit for adding $(x' + y')$ and $(x'' + y'')$ together to obtain the final answer $x + y$.

[1] This may could be done in a smarter way, but we will not discuss this in further detail.

Acknowledgments

I thank by mentor Dr. M. Möller for his open minded view and the meetings. I thank Dr. N. Khammassi from the QuTech team for being very helpful answering questions and the use of the Qx-simulator.

References

[1] Draper, T.G. 2000. "Addition on a quantum computer"
http://arxiv.org/abs/quant-ph/0008033v1

[2] Michael A. Nielsen & Isaac L. Chuang, 2010, "Quantum Computation and Quantum Information", 10th edition, CAMBRIDGE UNIVERSITY PRESS
www.cambridge.org/9781107002173

[3] David P. DiVincenzo, 2000, "The Physical Implementation of Quantum Computation"
http://arxiv.org/abs/quant-ph/0002077v3

[4] Richard P. Feynmann, 1959, "There's Plenty of Room at the Bottom"
http://www.zyvex.com/nanotech/feynman.html

[5] Craig Gidney, 2016, "Quirk - Drag-and-Drop Quantum Circuit Simulator"
http://algassert.com/quirk, License: https://creativecommons.org/licenses/by/4.0/

[6] Nader Khammassi, 2016, "Qx-Simulator"
http://quantum-studio.net/

APPENDIX A

Proofs of essential mathematical statements

Proof of: $A^\dagger = \overline{A}^T$

PROOF. Let $n \in \mathbb{N}$, let $x, y \in \mathbb{C}^n$ and A be a complex $n \times n$ matrix. By definition we know that $\langle x|Ay\rangle = \langle A^\dagger x|y\rangle$. If we take the transpose and the complex conjugate:

$$\begin{aligned}
\langle x|Ay\rangle &= \langle A^\dagger x|y\rangle \\
\overline{x}^T Ay &= \overline{A^\dagger x}^T y \\
y^T A^T \overline{x} &= y^T \overline{A^\dagger x} \\
\overline{y}^T \overline{A}^T x &= \overline{y}^T A^\dagger x
\end{aligned}$$

Since x, y are chosen randomly, this must hold for all $x, y \in \mathbb{C}^n$. Thus $A^\dagger = \overline{A}^T$. □

Proof of: $U^\dagger U = I \implies UU^\dagger = I$

PROOF. Let $n \in \mathbb{N}$, U be a complex $n \times n$ matrix such that $U^\dagger U = I$. Notice that $det(U^\dagger U) = det(I) = 1$, but also $det(U^\dagger U) = det(U^\dagger) det(U)$. So $det(U) \neq 0$. Thus U is invertible and has a left inverse $U^\dagger$. Since it is invertible it has some right inverse U^{-1}, such that $UU^{-1} = I$.

$$\begin{aligned}
U^\dagger U &= I \\
UU^\dagger U &= U \\
UU^\dagger UU^{-1} &= UU^{-1} \\
UU^\dagger &= I
\end{aligned}$$

□

Proof that $R_z(\phi)$ equivalent to another matrix

PROOF. Let q_0 be a qubit. Let $\alpha, \beta \in \mathbb{C}$ such that $|q_0\rangle = \alpha|0\rangle + \beta|1\rangle$. Evaluate $R_z(\phi)|q_0\rangle$:

$$\begin{aligned}
R_z(\phi)|q_0\rangle &= \begin{bmatrix} e^{-i\frac{1}{2}\phi} & 0 \\ 0 & e^{i\frac{1}{2}\phi} \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \\
&= \begin{bmatrix} \alpha e^{-i\frac{1}{2}\phi} \\ \beta e^{i\frac{1}{2}\phi} \end{bmatrix} \equiv e^{i\frac{1}{2}\phi} \begin{bmatrix} \alpha e^{-i\frac{1}{2}\phi} \\ \beta e^{i\frac{1}{2}\phi} \end{bmatrix} \\
&= \begin{bmatrix} \alpha \\ \beta e^{i\phi} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix} |q_0\rangle
\end{aligned}$$

□

Proof of: $\exists\theta \in [0, \frac{1}{2}\pi] : r_a = cos(\theta), r_b = sin(\theta)$

PROOF. It is known that $r_a = |a| \geq 0$ and that $|a|^2 + |b|^2 = 1$. Since $|a|^2$ and $|b|^2$ are greater then zero, we get: $0 \leq r_a \leq 1$. The same applies to r_b. Because of the fact hat $r_a \in [0, 1]$ there exist a $\theta \in [0, \frac{1}{2}\pi]$ such that $r_a = cos(\theta)$. Substituting this in the restriction $|a|^2 + |b|^2 = 1$ gives:

$$\begin{aligned} \cos^2(\theta) + |b|^2 &= 1 \\ |b|^2 &= 1 - \cos^2(\theta) = \sin^2(\theta) \\ (r_b =) \quad |b| &= \pm\sin(\theta) \end{aligned}$$

Since $r_b \geq 0$ we get that $r_b = sin(\theta)$. □

Proof of: $a_k n^k + \cdots + a_1 n + a_0 = O(n^k)$

PROOF. Let $f : \mathbb{N} \to \mathbb{N}$ be a polynomial of grade $k \in \mathbb{N}$ given by $f(n) = a_k n^k + \cdots + a_1 n + a_0$ with $a_k \in \mathbb{R}_{>0}$ and $a_{k-1}, \ldots, a_0 \in \mathbb{R}$. Let $N = 1$ and $M = \sum_{j=0}^{k} |a_j|$ then for all $n \geq N$:

$$\begin{aligned} f(n) &= a_k n^k + \cdots + a_1 n + a_0 \\ &\leq |a_k| n^k + \cdots + |a_1| n + |a_0| \\ &\leq |a_k| n^k + \cdots + |a_1| n^k + |a_0| n^k \\ &= (|a_k| + \cdots + |a_0|) n^k \\ &= M n^k \end{aligned}$$

Thus for all $n \geq N$: $f(n) \leq M n^k$, so by definition $f(n) = O(n^k)$. □

Proof of: $\sum_{m=1}^{M} m = \frac{1}{2}M(M+1)$

PROOF. Let $M \in \mathbb{N}$, we evaluate $\sum_{m=1}^{M} m$:

$$\begin{aligned} \sum_{m=1}^{M} m &= \frac{1}{2}\left(\sum_{m=1}^{M} m + \sum_{m=1}^{M} m\right) \\ &= \frac{1}{2}\left(\sum_{m=1}^{M} m + \sum_{m=1}^{M} M + 1 - m\right) \\ &= \frac{1}{2}\left(\sum_{m=1}^{M} m + M + 1 - m\right) \\ &= \frac{1}{2}\left(\sum_{m=1}^{M} M + 1\right) \\ &= \frac{1}{2}(M+1)\sum_{m=1}^{M} 1 \\ &= \frac{1}{2}(M+1)M \end{aligned}$$

At step two, we used: $\sum_{m=1}^{M} m = \sum_{m=1}^{M} M + 1 - m$. Notice that $\sum_{m=1}^{M} M + 1 - m$ adds the same values together, but in a reversed order relative to $\sum_{m=1}^{M} m$. □

Proof of: Double error gates

PROOF. The probability that a single error gate occurs at one qubit is p. Suppose we have two spots, each of them have a probability p that an error gate occurs. What is the overall probability that an error occurs?
To answer this question, we have to separate two different cases. The obvious one being when an error occurs in only one of the two spots. The probability that this will happen is:

$$2p(1-p) = 2p - 2p^2$$

The other case is when an error occurs in both spots. When these errors are of the same kind, the original state is obtained since all error gates are Involutory. An error after the two error gates will only occur when they are of different kind. The probability that this will happen is:

$$p\frac{2}{3}p = \frac{2}{3}p^2$$

These two probabilities are added together: $2p - 2p^2 + \frac{2}{3}p^2 = 2p - \frac{4}{3}p^2$. Notice that this is actual different than the original value p. □

APPENDIX B

Code

Listing B.1. C++ Code

```cpp
#include <iostream>
#include <cstdint>
#include <vector>
#include <fstream>
#include <string>
#include <math.h>

#include <xpu.h>
#include <xpu/runtime>

#include <core/circuit.h>
#include <qcode/quantum_code_loader.h>
#include <core/error_model.h>

int main(int argc, char **argv){
    //variables
    std::ifstream file;
    file.open("settings.txt");
    int a, a1, a2;
    double boundary = 0.001;
    int push, analyse_gates_i, measure_i, show_circuit_i, error_testing_i,
        ↪ parallel_optimize_i;
    double error, chance;
    char set;
    xpu::timer timer_make;
    xpu::timer timer_execute;
    std::string l1, l2, l3, l4, l5, l6, l7;
    std::string i1, i2, i3, i4, i5, i6, i7;
    //input settings
    while(set!='y' && set!='n'){
        std::cout << "change settings: [y/n] ";
        std::cin >> set;
    }
    if(set == 'y'){
        std::getline(file,l1); //push
        std::getline(std::cin,i1);
        if(l1.empty()){push =0;}
        else{push = std::stoi(l1);}
        do{std::cout << "push is now: " << push << " insert new value: ";
            std::getline(std::cin,i1);
            if(!i1.empty()){push = std::stoi(i1);}
        }while( push < 1);
```

```
42        std::getline(file,l2); //measurement
43        if(l2.empty()){measure_i =0;}
44        else{measure_i = std::stoi(l2);}
45        do{std::cout << "measure is now: " << measure_i << " insert new
              ↪ value: ";
46            std::getline(std::cin,i2);
47            if(!i2.empty()){measure_i = std::stoi(i2);}
48        }while(measure_i!=1 && measure_i!=0);
49        std::getline(file,l3); //show_circuit
50        if(l3.empty()){show_circuit_i =0;}
51        else{show_circuit_i = std::stoi(l3);}
52        do{std::cout << "show_circuit is now: " << show_circuit_i << "
              ↪ insert new value: ";
53            std::getline(std::cin,i3);
54            if(!i3.empty()){show_circuit_i = std::stoi(i3);}
55        }while(show_circuit_i!=1 && show_circuit_i!=0);
56        std::getline(file,l4); //error_testing
57        if(l4.empty()){error_testing_i =0;}
58        else{error_testing_i = std::stoi(l4);}
59        do{std::cout << "error_testing is now: " << error_testing_i << "
              ↪ insert new value: ";
60            std::getline(std::cin,i4);
61            if(!i4.empty()){error_testing_i = std::stoi(i4);}
62        }while(error_testing_i!=1 && error_testing_i!=0);
63        std::getline(file,l5); //parallel_optimize
64        if(l5.empty()){parallel_optimize_i =0;}
65        else{parallel_optimize_i = std::stoi(l5);}
66        do{std::cout << "parallel_optimize is now: " << parallel_optimize_i
              ↪  << " insert new value: ";
67            std::getline(std::cin,i5);
68            if(!i5.empty()){parallel_optimize_i = std::stoi(i5);}
69        }while(parallel_optimize_i!=1 && parallel_optimize_i!=0);
70        std::getline(file,l6); //analyse_gates
71        if(l6.empty()){analyse_gates_i =0;}
72        else{analyse_gates_i = std::stod(l6);}
73        do{std::cout << "analyse_gates is now: " << analyse_gates_i << "
              ↪ insert new value: ";
74            std::getline(std::cin,i6);
75            if(!i6.empty()){analyse_gates_i = std::stod(i6);}
76        }while(analyse_gates_i!=0 && analyse_gates_i!=1);
77        std::getline(file,l7); //error
78        if(l7.empty()){error =0;}
79        else{error = std::stod(l7);}
80        do{std::cout << "error is now: " << error << " insert new value: ";
81            std::getline(std::cin,i7);
82            if(!i7.empty()){error = std::stod(i7);}
83        }while(error<0 || error>1);
84        std::cout << "-----------------------------------------" << std::endl;
85        file.close();
86        std::ofstream file;
87        file.open("settings.txt");
88        file << push << '\n';
89        file << measure_i << '\n';
```

```
90          file << show_circuit_i << '\n';
91          file << error_testing_i << '\n';
92          file << parallel_optimize_i << '\n';
93          file << calculate_push_i << '\n';
94          file << error << '\n';
95          file << analyse_gates_i << '\n';
96          file.close();
97      }
98      else{
99          std::getline(file,l1); push = std::stoi(l1);
100         std::getline(file,l2); measure_i = std::stoi(l2);
101         std::getline(file,l3); show_circuit_i = std::stoi(l3);
102         std::getline(file,l4); error_testing_i = std::stoi(l4);
103         std::getline(file,l5); parallel_optimize_i = std::stoi(l5);
104         std::getline(file,l6); calculate_push_i = std::stoi(l6);
105         std::getline(file,l7); error = std::stod(l7);
106         std::getline(file,l6); analyse_gates_i = std::stoi(l6);
107         file.close();
108     }
109     //input error testing
110     if(error_testing_i==1){
111         do{std::cout << "how many bits for one int: ";
112             std::cin >> a;}while(a<1);
113         a1 = pow(2,a)-1;
114         a2 = a1;
115     }
116     //input numbers
117     else{
118         do{std::cout << "int 1: ";
119             std::cin >> a1;}while(a1<0);
120         do{std::cout << "int 2: ";
121             std::cin >> a2;}while(a2<0);
122     }
123     //general
124     int ans = a1+a2;
125     int maxint = (a1>a2) ? a1 : a2;
126     int size =0; //amount of qubits for one number: N
127     while (maxint >= pow(2,size)){size++;}
128     //input to binary
129     int q1[size], q2[size]; // q1=x0 x1 x2 .. a1=2^0*x0+2^1*x1+2^2*x2..
130     for(int j=size-1;a1>0;j--){q1[j]=a1%2; a1=(a1-(a1%2))/2;}
131     for(int j=size-1;a2>0;j--){q2[j]=a2%2; a2=(a2-(a2%2))/2;}
132     //building circuit
133         uint32_t n=2*size+1; //totaal amount of qubits
134         qx::circuit c(n);
135         timer_make.start();
136     //adding gates
137         //input
138         for(int j=0;j<size;j++){if(q1[j]==1){c.add(new qx::pauli_x(j+1));}}
139         for(int j=0;j<size;j++){if(q2[j]==1){c.add(new qx::pauli_x(j+size
            ↪ +1));}}
140         if(analyse_gates_i==1){
141             std::cout << "input: " << c.size() << std::endl;
```

```
142         }
143         //qft
144         for(int j=0;j<=size;j++){
145             c.add(new qx::hadamard(j));
146             for(int k=j;k<size;k++){c.add(new qx::ctrl_phase_shift(k+1,j));}
147         }
148         if(analyse_gates_i==1){
149             std::cout << "qft: " << c.size() << std::endl;
150         }
151         //addition
152         for(int j=0;j<size;j++){
153             c.add(new qx::swap(j, j+size));
154             for(int k=j;k<size;k++){c.add(new qx::ctrl_phase_shift(k+size+1,
                    ↪ j+size));}
155             c.add(new qx::swap(j, j+size));
156             c.add(new qx::cphase(j+size+1,j+1));
157         }
158         if(analyse_gates_i==1){
159             std::cout << "addition: " << c.size() << std::endl;
160         }
161         //qft invers
162         for(int j=size;j>=0;j--){
163             for(int k=size;k>j;k--){
164                 double phi = -1 * M_PI / (pow(2,k-j));
165                 c.add(new qx::cnot(k, j));//c-not
166                 c.add(new qx::rz(j,-1*phi/2));//Rz(-phi/2) op targ
167                 c.add(new qx::cnot(k, j));//c-not
168                 c.add(new qx::rz(j, phi/2));//Rz(phi/2) op targ
169             }
170             c.add(new qx::hadamard(j));
171         }
172         if(analyse_gates_i==1){
173             std::cout << "qft    : " << c.size() << std::endl;
174         }
175     //parallelization
176     if(parallel_optimize_i==1){
177         std::vector< qx::gate * > para_verz ={}; //gates to add
178         std::vector< std::vector<int> > pos2 = {}; //position gates
179         std::vector< std::vector<int> > q;//circuit in queue's
180         std::vector<int> position = {};//the other values
181         std::vector<int> dept (c.size(),0);
182         int loop =n; int min;
183         for(int j = 0; j<n; j++){q.push_back(std::vector<int>());}
184         for(int gate_it =0; gate_it < c.size(); gate_it ++){
185             qx::gate * gate = c.get(gate_it);//filling q
186             std::vector<uint32_t> control = gate -> control_qubits();
187             std::vector<uint32_t> target = gate -> target_qubits();
188             for(int j =0; j<control.size(); j++){
189                 q[control[j]].push_back(gate_it);dept[gate_it]++;
190             }
191             for(int j =0; j<target.size(); j++){
192                 q[target[j]].push_back(gate_it);dept[gate_it]++;
193             }
```

```
194             if(!target.empty()){loop = target[0];}
195         }
196         while(loop<n){
197             min = q[loop][0];
198             position.clear();
199             for(int j = 0; j < n; j++){
200                 if(!q[j].empty() && q[j][0]<min){min = q[j][0];}
201             }//searching lowest value
202             for(int j = 0; j < n; j++){
203                 if(!q[j].empty() && q[j][0]!=min){position.push_back(q[j][0])
                        ↪ ;}
204             }//searching all other values
205             std::vector<int> flaw = {};
206             for(int i =0; i<position.size();i++){//wrong gates
207                 int count = 0;
208                 for(int j = 0; j<position.size();j++){
209                     if(position[j]==position[i]){count++;}
210                 }
211                 if(count < dept[position[i]]){
212                     flaw.push_back(position[i]);
213                 }
214             }
215             for(int i =0; i<flaw.size();i++){//del gates
216                 for(int j =position.size()-1; j>=0;j--){
217                     if(flaw[i]==position[j]){
218                         position.erase(position.begin()+j);
219                     }
220                 }
221             }
222             for(int j = 0; j<position.size();j++){//unique
223                     for(int k = j+1; k<position.size();k++){
224                     if(position[j]==position[k]){
225                         position.erase(position.begin()+k);
226                         k--;
227                     }
228                 }
229             }
230             if(position.empty()){
231                 for(int j=0;j<n;j++){
232                     if(!q[j].empty() && q[j][0]==min){
233                         q[j].erase(q[j].begin());
234                     }
235                 }
236             }//del lowest value
237             else{//building parallel gate& adding gates
238                 pos2.push_back(position);
239                 int g = pos2.size()-1;
240                 pos2[g].insert(pos2[g].begin(), min);
241                 qx::parallel_gates * parallel = new qx::parallel_gates();
242                 parallel -> add(c.get(min));
243                 for(int k =0 ;k<n;k++){
244                     if(!q[k].empty() && q[k][0]==min){
245                         q[k].erase(q[k].begin());
```

```
246                 }
247             }
248             for(int j =0; j<position.size(); j++){
249                 parallel -> add(c.get(position[j]));
250                 for(int k =0 ;k<n;k++){
251                     if(!q[k].empty() && q[k][0]==position[j]){
252                         q[k].erase(q[k].begin());
253                     }
254                 }
255             }
256             para_verz.push_back(parallel);
257         }
258         loop =0;//checking if done
259         while(q[loop].empty()){loop++;}
260     }
261     //insert parallel gate& del original gate
262     for(int j =0; j<para_verz.size(); j++){
263         c.erase(pos2[j][0]);
264         c.insert(pos2[j][0],para_verz[j]);
265         pos2[j].erase(pos2[j].begin());
266     }//del other gates
267     std::vector<int> pos3 = {};
268     for(int j =0; j<pos2.size(); j++){
269         for(int k = 0; k<pos2[j].size(); k++){
270             if(pos3.empty()){pos3.push_back(pos2[0][0]);}
271             else{
272                 int it =0;//biggest value at the back
273                 while(pos2[j][k]>pos3[it] && it<pos3.size()){it++;}
274                 pos3.insert(pos3.begin()+it,pos2[j][k]);
275             }
276         }
277     }
278     while(!pos3.empty()){
279         c.erase(pos3.back());
280         pos3.pop_back();
281     }
282 }
283 //measurement
284 if(measure_i==1){c.add(new qx::measure());}
285 else{c.add(new qx::identity(0));}
286 timer_make.stop();
287 timer_execute.start();
288 //gates output
289 if(show_circuit_i==1){c.dump();}
290 if(analyse_gates_i==1){
291     std::cout << "parallel: " << c.size() << std::endl;
292     return 0;
293 }
294 //execute
295 qx::qu_register reg(n);
296 xpu::complex_d null;
297 std::vector<int> state;
298 std::vector<int> values;
```

```
299     std::vector<double> prob;
300     if(error==0){//perfect circuit
301         c.execute(reg);
302         for(int i=0;i<pow(2,n);i++){
303             if(reg[i]!=null){
304                 double chance = pow(reg[i].re,2)+powf(reg[i].im,2);
305                 if(chance>boundary){
306                     state.push_back(i);
307                     prob.push_back(chance);
308                 }
309             }
310         }
311     }
312     else{//error circuit
313         qx::depolarizing_channel dep_ch(&c, n, error);
314         for (int j=0; j<push;j++){
315             reg.reset();
316             qx::circuit * noisy_c = dep_ch.inject(false);
317             noisy_c -> execute(reg);
318             for (int i=0; i<pow(2,n); i++){
319                 if(reg[i]!=null){
320                     chance = pow(reg[i].re,2)+powf(reg[i].im,2);
321                     if (chance > boundary){
322                         if(state.empty()){//new state
323                             state.push_back(i);
324                             prob.push_back(chance/push);
325                         }
326                         else{//adding probability
327                             int k = 0;
328                             while (state[k] != i && k<state.size()){k++;}
329                             if (state[k] == i){prob[k] += chance/push;}
330                             else{
331                                 state.push_back(i);
332                                 prob.push_back(chance/push);
333                             }
334                         }
335                     }
336                 }
337             }
338         }
339     }
340     timer_execute.stop();
341     //states to values
342     for(int k =0; k<state.size(); k++){
343         int m = state[k];
344         for (int nn = n; nn>size; nn--){//del int b from front
345             if(m>pow(2, nn)){m-=pow(2,nn);}
346         }
347         int mm =0;//the value
348         for(int k =0;k<=size;k++){//reverse bit order
349             if(m>=pow(2,size-k)){m-=pow(2,size-k);mm+=pow(2,k);}
350         }
351         values.push_back(mm);
```

```
352     }
353     //adding same values
354     for(int k = 0; k+1 < values.size(); k++){
355         for(int j = k+1; j<values.size();j++){
356             if(values[k]==values[j]){
357                 prob[k]+=prob[j];
358                 values.erase(values.begin()+j);
359                 prob.erase(prob.begin()+j);
360                 j--;
361             }
362         }
363     }
364     //output
365     std::cout << "[size: " << size << "  qubits: " << n << "  error: " <<
        ↪ error << "  gates: " << c.size() << "  push: " << push << "
        ↪ measure: " << measure_i << "  parallel: " << parallel_optimize_i
        ↪  << "]" << std::endl;
366     std::cout << "final probability: " << std::endl;
367     double total =0.0;
368     double ans_prob;
369     double best_prob =0.0;
370     int best =0;
371     std::string push2_string;
372     for(int i = values.size()-1;i>=0;i--){
373         if(prob[i] > boundary/10){
374             std::cout << "Value:  " << values[i] <<  " with prob: " << prob[
                ↪ i] << std::endl;
375             total += prob[i];
376         }
377         if(values[i] == ans){ans_prob = prob[i];}
378         if(prob[i]>best_prob && values[i]!=ans){
379             best = values[i];
380             best_prob = prob[i];
381         }
382     }
383     std::cout << "                    ---------- +" << std::endl;
384     std::cout << "                     " << total << std::endl;
385     std::cout << "answ: " << ans << ", prob: " << ans_prob << std::endl;
386     std::cout << "best: " << best << ", prob: " << best_prob << std::endl;
387     std::cout << "time to build: " << timer_make.elapsed()<< ", time to
        ↪ run: " << timer_execute.elapsed() << std::endl;
388     return 0;
389 }
390 /*  changes made in qx:
391 gate.h -> Ln 837 - 894 (class identity)
392 gate.h -> Ln 2346 & 2347 (+1 to cr gate)
393 curcuit.h -> Ln 132 & 155 (//println)
394 curcuit.h -> Ln 176 - 179 (erase functie)
395
396 TU Delft, The Netherlands
397 author: Menno Looman
398 date: 2018-07-02
399 */
```