בית הספר לפיזיקה ולאסטרונומיה
הפקולטה למדעים מדויקים
ע"ש ריימונד ובברלי סאקלר
אוניברסיטת תל אביב

School of Physics and Astronomy
The Raymond and Beverly Sackler
Faculty of Exact Sciences
Tel Aviv University

סימולציית קודים משטחיים קוונטים תחת שגיאות קוהרנטיות תלויות קיוביט

כתזה לתואר "מוסמך אוניברסיטה" בהנדסת חשמל
בית הספר לפיזיקה ואסטרונומיה
אוניברסיטת תל אביב

על ידי

רון כהן

מחקר זה בוצע בהנחייתו שלו

פרופ' חיים סוכובסקי

אדר תשפ"ג

School of Physics and Astronomy
The Raymond and Beverly Sackler
Faculty of Exact Sciences
Tel Aviv University

בית הספר לפיזיקה ולאסטרונומיה
הפקולטה למדעים מדויקים
ע"ש ריימונד ובברלי סאקלר
אוניברסיטת תל אביב

Quantum Surface Code Simulation under Qubit-Dependent Coherent Errors

M.Sc. Thesis
School of Physics and Astronomy
Tel Aviv University

by

Ron Cohen

This research is carried out under the supervision of

Prof. Haim Suchowski

March 23

Acknowledgments

I wish to thank and appreciate my supervisor, Prof. Haim Suchowski, for his guidance and support during the research.

Thanks to Prof. Yaron Oz and Prof. Moshe Goldshtein for their professional guidance at every step.

I special thank Dr. Andrew S. Darmawan for teaching me the best method to simulate quantum error correction he developed.

I want to thank my colleagues in the quantum group for cooperating, advising, thinking, and being so supportive. To Muhammad Erew, Yuval Reches, and Ido Kaplan. Special appreciation to M.Sc Yonatan Piazetzky for taking me under his professional care during the early stages of my Master's.

I want to thank the Femto-Nano group for their friendship, coffee times together, and support.

I wish to thank my family for being so supportive and warm whenever and without conditions. A special thanks to my brother and mentor, Alon Cohen, who guides me to make the best choices.

And enormous thanks to my best friend, my closest partner in life, who is always there for me, and my love - Karin, my wife.

0.1 Abstract

Fault-tolerant quantum information processing is essential for the realization of quantum computing devices [1]. A very popular quantum error correction architecture is the surface code, where every logical qubit is represented by several physical qubits. The physical qubits are arranged in a 2D array with stabilizers measured and errors detected and corrected to get a low logical error rate, which makes the device fault-tolerant [1]. Smaller physical errors could significantly reduce the error correction overhead, as they allow to use a smaller surface code per a single logical qubit, hence making the realization of the quantum processing device more achievable. In recent years, coherent errors have been recognized as a significant type of error to deal with, since they might sum up coherently. Reducing coherent errors is therefore one of the primary requirements to curtail the surface code's size, which would otherwise require thousands of physical qubits. In this work, for the first time, we investigated the effect of variation of different coherent errors over the different qubits of the surface code. The primary metric to assess the improvement is to measure the shrinking of the required size of surface code for several target logical error levels [1]. A promising simulation method to measure the logical error probability is using tensor networks, which reduces the numerical cost to scale exponentially with the square root of the array size rather than the size itself [2]. We have applied this methodology in the context of linear optics quantum computation, to evaluate the performance of a composite segmented design in a dual-rail realization [3], and show that it reduces the total amount of physical qubits by a factor of 6.5 in common realistic scenarios. We also showed that, as opposed to the case where a constant angle rotation error applies to all the qubits, when the sign of rotation is changing between qubits on the surface, the logical error is suppressed because of destructive interference. We believe our findings will serve as a helpful starting point for better surface code architecture in different technologies.

תקציר

כל קיוביט לוגי במחשב קוונטי, ממומש על ידי קיוביטים פיזיים בארכיטקטורה שמתקנת את השגיאות שלהם. ארכיטקטורת תיקון שגיאות קוונטית מאוד פופולרית היא קוד משטח. הוא מסדר את הקיוביטים הפיזיים במשטח דו-ממדי כדי לקבל שיעור שגיאות לוגי נמוך, מה שהופך את המחשב הקוונטי לחסין מפני שגיאות (עמיד בפני תקלות). שגיאות פיזיקליות קטנות יותר ידרשו משטח קטן יותר לכל קיוביט לוגי בודד, מה שהופך את היישום של המחשב הקוונטי לאפשרי יותר. בשנים האחרונות, שגיאות קוהרנטיות הפכו לסוג משמעותי של שגיאות שיש לטפל בהן, שכן הן מצטברות באופן קוהרנטי. הפחתת שגיאה קוהרנטית היא אחד המאמצים העיקריים שנעשו כדי להקטין את גודל קוד המשטח, שגודלו הוא $O(d^2)$ ויכול לכלול אלפי קיוביטים פיזיים. במחקר זה לראשונה אנו בודקים את השפעת השגיאה הקוהרנטית כאשר היא שונה בין כלל הקיוביטים במשטח. המדד העיקרי למדידת תוצאות השיפור הוא למדוד את התכווצות גודל קוד המשטח כפונקציה של השגיאה הלוגית הנדרשת. שיטת סימולציה מבטיחה למדידת השגיאה הלוגית היא שימוש ברשתות טנזוריות, המאפשרות לצמצם את מספר הפעולות בשורש ריבועי. באמצעות שיטה זו, מצאנו כי השיטה בה אנו משפרים שערים קוונטים אופטיים, המשתמשים בהרכבת מקטעי מוליכי גלים, מפחיתה את כמות הקיוביטים הכוללת בפקטור של כ-6.5 ליישומים נפוצים בשימוש מחשבים קוונטים. הראינו גם שביחס למקרה שבו שגיאת זווית הסיבוב קבועה לכל הקיוביטים, כאשר סימן הסיבוב משתנה בין קיוביטים על פני המשטח, השגיאה הלוגית קטנה משמעותית בגלל התאבכות הורסת. אנו סבורים שהממצאים שלנו יספקו נקודת התחלה מועילה לארכיטקטורות קודים משטחיים טובים יותר.

Table of Contents

Nomenclature

FBQC Fusion Based Quantum Computing

LOQC Linear Optic Quantum Computing

MBQC Measurement Based Quantum Computation

QC Quantum Computing

QEC Quantum Error Correction

qubits Quantum bits

RSA Rivest Shamir Adleman

SC Surface Code

1 Introduction

The development of quantum information processing and quantum computing are central parts of the current second quantum revolution [4]. Many research disciplines are involved in such revolution - physics, hardware, architectures, software, and advanced applications with algorithms [5]. Quantum Computer (QC) is suffering from infidelities and noise, which lead to errors, and can not exist without the core idea of Quantum Error Correction (QEC) [6, 7].

In classical information, noise is added to 0 / 1 bits and may lead to incorrect 0 / 1 identification. The solution to these errors is encoding. For example, we can replace every 0 with four zeros. In this way, even with some low chance of 0-1 flipping, the decoder will understand if the origin was 0 or 1 [8].

Quantum bits (qubits) are suffering from similar issues. Its state changes when noise is added to a qubit (whether from the environment or from the infidelity of the gate controlling it). This state is crucial for quantum computation [6, 7]. The interesting algorithms using those qubits rely on the exact states of the qubits [9]. Therefore we cannot allow errors to corrupt the state.

The noisier a quantum computer is, the more qubits it will need to compensate for its noise or infidelity [9]. Think of the classical repetition code mentioned earlier, where adding extra bits helped us mitigate the introduction of noise [8].

The resources overhead may reach up to thousands of times more than the silent case. E.g., in Shor's Algorithm, which is a quantum solver that can crack Rivest Shamir Adleman encryption (RSA encryption) encryption by decomposing numbers into primary factors. To decompose a 2000 bits number in 24 hours, Shor's algorithm will need a total amount of 300,000 silent qubits (logical qubits). For today's noise rate of 10^{-3}, each one of these 300,000 logical qubits will be replaced by 3,600 physical qubits, which will result in 10^9 **physical qubits** [1].

One of the dominant noises is coherent errors, which happen deterministically. To understand how this type of error affects the logical error, we need to simulate it. All literature so far examined the coherent error when it is the same for all the encoding qubits [10–15].

Our work examined, for the first time, the effect of coherent error in surface code when there is some distribution of different coherent errors over the different qubits of the surface.

Using our work, we examined how the method of segmented composite pulses [3, 16] reduces this overhead significantly by reducing the physical error. Our approach demands a factor of x6.5 fewer qubits for the same application than the legacy case. Given the limitation of algorithm time and physical qubits amount, this reduction opens the possibility to factorize numbers twice longer and simulate twice bigger molecules.

The Master thesis is organized as follow:

- In chapter 2, we describe the theoretical quantum computing background materials as well as describe the quantum error correction (QEC) theory. We describe in details Surface Code which is the most popular QEC technique, and their behavior under coherent errors.

- In chapter 3, we give the motivation to use tensor networks to simulate Surface Code under coherent error.

- In chapter 4, we give the method of simulation, using tensor networks. We also detail the physical noise model that we use in this work.

- In chapter 5, we show the results of simulation, and the improvement in logical error in different noise models.

- In chapter 6,we give a full prospective of the implementation of the Plus-Minus technique on linear cluster states. This chapter includes background, method and results.

- In chapter 7, we conclude our work, and present future work potential.

2 Background

2.1 Quantum Computing Basics

In this section, I will cover the basic definitions in quantum computation that are needed to understand the mechanism of quantum errors appearing in subsection 2.2 and quantum error corrections that are discussed in subsections 2.3 2.4. In this section, the definitions follow Refs [17] and [18].

Qubit

Qubit, **the quantum expansion of the classical bit**, is an object that stores a basic unit of 0/1 information. **Unlike a bit**, a qubit can hold a coherent superposition between 0 and 1. At the time of measuring, the qubit collapse to one of the state 0 or 1, with a projected value of the probability amplitude of the superposition state.

We represent those 2 probabilities as 2 numbers in a **vector**, or a sum (a superposition) of $|0\rangle$ state and $|1\rangle$ state at the same time:

$$\alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \tag{2.1}$$

Once we **measure** the qubit, it will not be in a superposition of 0 and 1 anymore, but its state will **collapse** into 0 or 1, in a probability that depends on those two probability amplitudes :

$$\begin{aligned} P(state\ will\ be\ 0\ after\ measurement) &= |\alpha|^2 \\ P(state\ will\ be\ 1\ after\ measurement) &= |\beta|^2 \end{aligned} \tag{2.2}$$

$$|\alpha|^2 + |\beta|^2 = 1 \tag{2.3}$$

Bloch Sphere

A very useful **visualization** of a qubit is the **Bloch Sphere**. It is a sphere, with a radius of 1. Where:

- A point on its **north pole** represents a state that is pure $|0\rangle$.
- A point on its **south pole** represents a state that is pure $|1\rangle$.
- Between the 2 poles are all the states that are **superpositions** of $|0\rangle$ and $|1\rangle$. The latitude is representing the ratio of probabilities. For example, in the equator, are all the states with probabilities of 50%/50%. Again, you might think that the qubit is in state 0 or 1 and we just do not know yet, but no, the **power** of the qubit, is that it is in **0 and 1 at the same time**.
- The **longitude** (going only around Z-axis) represents the **relative phase** between the $|1\rangle$ term to the $|0\rangle$ term of the superposition. The importance of the phase will be discussed later.

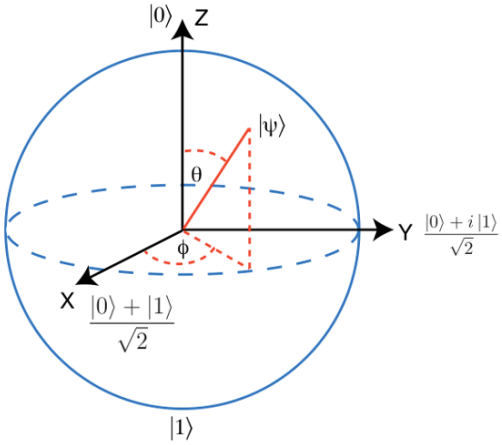


Figure 2.1: fig: The Bloch sphere representation of a qubit. θ tells the amplitudes of $|0\rangle$ and $|1\rangle$, and ϕ tells the phase between them.

1 Qubit Gate

Once we have a qubit, we can perform basic operations on it, and change it using **qubit gates**. An abstract representation of a qubit is commonly a 2x1 vector 2.1. The gate, which is a unitary operator that acts on the qubit is represented by a 2x2 matrix, which acts on this vector, and crates another 2x1 vector - the resulting qubit.

Eigenstates of Operator

Once we know what an operator/gate is, we can learn what is an **eigenstate** of it. Just as in **linear algebra**, a quantum eigenstate of operator A is a state that will not change itself up to a constant multiplication (eigenvalue) when A will be applied on that state. For example, the eigenstates of Z and X:

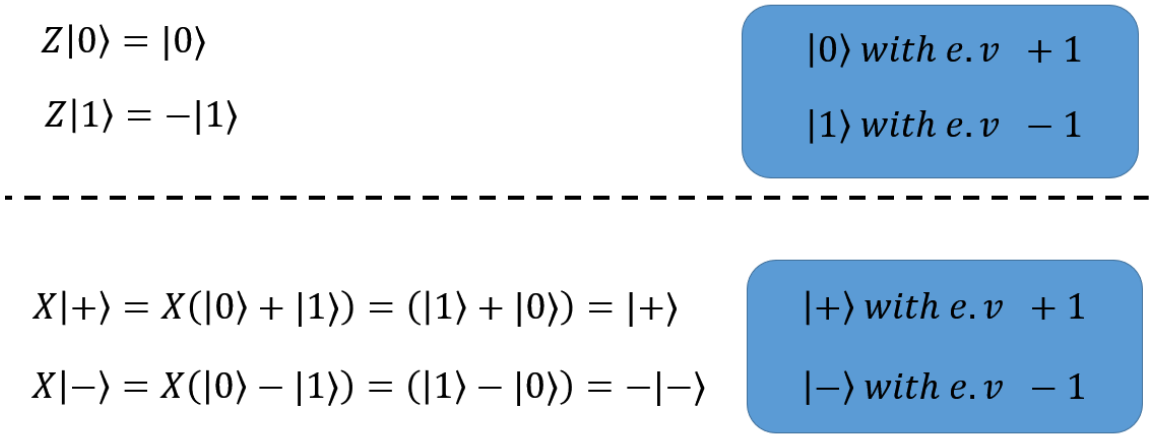


Figure 2.2: Eigenstates of a single Qubit operator

Measure in a different base

A measurement collapses the state to a 0 or 1 state, but can also collapse it to other states. The measurement value depends on the base that one uses, meaning the state collapses to neither 0 nor 1, with different probabilities. The change of basis will be a core concept to make error correction 2.3 and is used many times in QC. The way to do it is actually in a manner of collapse to 0 or 1, but, before measurement and after measurement, we change the state in the way that we choose. For example, if we want to collapse ourselves to an eigenstate of the X operator ($|+\rangle = |0\rangle + |1\rangle$ or $|-\rangle = |0\rangle - |1\rangle$), we can use the following

trick: H operator that will change $|+\rangle$ to $|0\rangle$ and $|-\rangle$ to $|1\rangle$. Measure and collapse the state to $|0\rangle$ or $|1\rangle$ Again H operator will change $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|-\rangle$ [19].

$$
\begin{aligned}
&|\psi\rangle = a|+\rangle + b|-\rangle \\
&\rightarrow^{\mathrm{H}} H|\psi\rangle = a|0\rangle + b|1\rangle \\
&\rightarrow^{MeasureZ} \begin{cases} P(|0\rangle) = |a|^2 \\ P(|1\rangle) = |b|^2 \end{cases} \\
&\rightarrow^{H} \begin{cases} P(|+\rangle) = |a|^2 \\ P(|-\rangle) = |b|^2 \end{cases}
\end{aligned}
$$

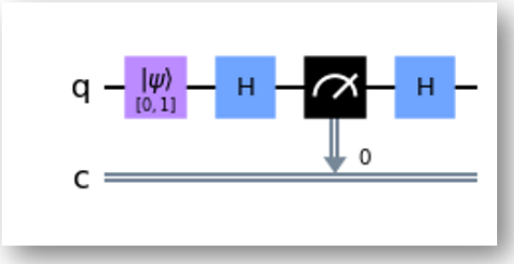
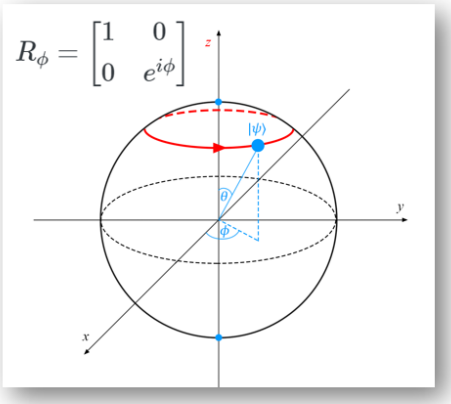


Figure 2.3: Measure in a base of X, differently than the common Z measurement

Any rotation

By using those 2x2 matrices, we can rotate our state on the Bloch Sphere, and change it to **any point that we like**. Now we can think of a small **coherent noise** that will be explained later 2.2, as a **small rotation**, with a small angle on the Bloch sphere 2.1. Here are the matrices that are needed to rotate the state around the x/y/z axis. See how for example, Ry or Rx rotation, will change the probabilities of the 0 and 1 of the state, and Rz will change only the relative phase between them:



$$
R_x(\theta) = \exp(-iX\theta/2) = \begin{bmatrix} \cos(\theta/2) & -i\sin(\theta/2) \\ -i\sin(\theta/2) & \cos(\theta/2) \end{bmatrix},
$$

$$
R_y(\theta) = \exp(-iY\theta/2) = \begin{bmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{bmatrix},
$$

$$
R_z(\theta) = \exp(-iZ\theta/2) = \begin{bmatrix} \exp(-i\theta/2) & 0 \\ 0 & \exp(i\theta/2) \end{bmatrix}.
$$

$$
U_3(\theta, \phi, \lambda) = \begin{bmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i\lambda+i\phi}\cos(\theta/2) \end{bmatrix}
$$

Figure 2.4: General rotation on a Bloch Sphere

More Qubits

Once we deal with more than 1 qubit, things get more interesting and complicated. For example, if we deal with two qubits, they can be in a superposition (meaning those states simultaneously) of $|00\rangle$,$|01\rangle$,$|10\rangle$, and $|11\rangle$ (4x1 vector). Each of them, with a probability of happening and collapsing after we measure those 2 qubits together.

$$|a\rangle = a_{00}|00\rangle + a_{01}|01\rangle + a_{10}|10\rangle + a_{11}|11\rangle = \begin{bmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{bmatrix}$$

$$p(|00\rangle) = |\langle 00|a\rangle|^2 = |a_{00}|^2$$

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

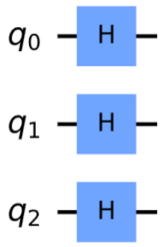
$$|a\rangle = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}, \quad |b\rangle = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$$

$$|ba\rangle = |b\rangle \otimes |a\rangle = \begin{bmatrix} b_0 \times \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \\ b_1 \times \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} b_0 a_0 \\ b_0 a_1 \\ b_1 a_0 \\ b_1 a_1 \end{bmatrix}$$

Figure 2.5: Two qubits state (top) vs a tensor product of two single-qubit states (bottom)

Here is an example of what happens to 3 qubits (8x1 vector), that start in $|000\rangle$ states, and change to an equal superposition of all the possible 8 states when H gate is applied to each one of them:

$$|q\rangle = |000\rangle$$



$$|cba\rangle = \begin{bmatrix} c_0 b_0 a_0 \\ c_0 b_0 a_1 \\ c_0 b_1 a_0 \\ c_0 b_1 a_1 \\ c_1 b_0 a_0 \\ c_1 b_0 a_1 \\ c_1 b_1 a_0 \\ c_1 b_1 a_1 \end{bmatrix} \quad |+++\rangle = \frac{1}{\sqrt{8}} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Figure 2.6: An example of three qubits state, each qubit initialized to $|+\rangle$. The final state is a tensor product of three single qubits state

Notice that when each one of them is in the $|0\rangle + |1\rangle$ state ($|0\rangle$ and $|1\rangle$ at the same time), dealing with the big picture of three qubits, the product states of all three qubits, tells that they are in a superposition of 8 combinations at the same time.

The Power of Quantum Computer

At this point, we can see how powerful quantum computer is. To store the information about 3 qubits in a classical computer, we need 8 **numbers**. And for N qubits, we need 2^N numbers. So, without knowing anything about quantum algorithms, we can already know that once we find a quantum algorithm solution to some problem, a classical computer will need **2 to the power of (The number of qubits that QC will need)** numbers to solve it in the same way that we found. Just to get some feeling, a problem that needs 100 qubits of QC, will need

$$2^{100} = 1,267,650,600,228,229,401,496,703,205,376 = 1.2E30$$

numbers to implement on a classical computer.

Two Qubits Gate

In order to create algorithms, and error correction, in quantum computers, we need operations that create **interactions between 2 qubits**, and operate on both of them together at the same time. The big power of those operators is that they are **acting simultaneously on all the terms of the superpositions**. We can also think of it as a $4x4$ matrix because it is acting on a 4x1 vector (2 qubits), and keeping it as a $4x1$ vector (2 qubits). For example, the controlled-NOT ($CNOT$) gate is a two-qubit (2Q) gate that is X flipping the 2nd qubit (target qubit) in every term where the 1st qubit (control qubit) is in $|1\rangle$ state. Notice this is happening on each of the terms of the state and keeping the superposition of the state.

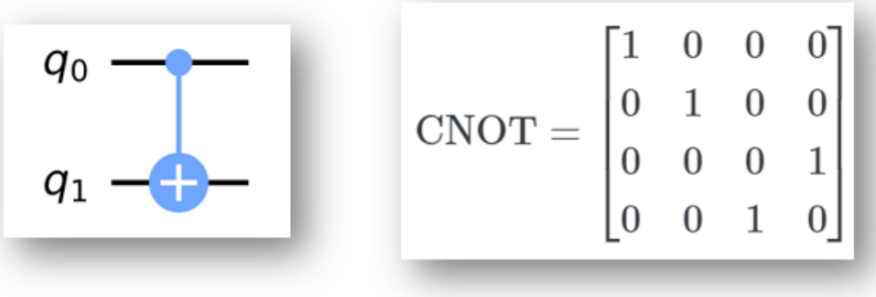


Figure 2.7: CNOT Gate - circuit drawing (left) and matrix representation (right)

Entanglement

Entanglement is the physical phenomenon that occurs when a group of particles are generated, interact, or share a proximity in a way such that the quantum state of each particle of the group cannot be described independently of the state of the others. For example, a two-qubits state such as $|00\rangle + |11\rangle$, is an entangled state. It is cannot be decomposed into a tensor product of two single-quantum states. As such, by measuring the first qubit to be $|0\rangle$, we know for sure that the second one must be measured as $|0\rangle$ too because the probability to $|01\rangle$ in this state is zero (this term does not exist).

CNOT is useful to create such an entanglement:

1. Start from $|00\rangle$ state

2. H on the first one will put it in a superposition of $|0\rangle + |1\rangle$, so the whole system is in state $|00\rangle + |10\rangle$

3. Then using *CNOT* when the first is controlling the second, will force that $|00\rangle$ will stay $|00\rangle$ (the first said 0 - meaning not to flip the second).

4. And $|10\rangle$ will force the second to flip since the first is 1.

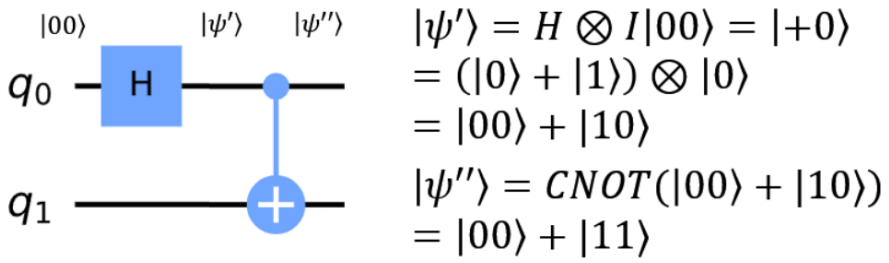


Figure 2.8: Bell pair, entanglement is created using H and CNOT gate

Notice the **second evidence of the power of QC**. The operator acted on the two terms of the superposition **at the same time**. A classical computer had to save the two terms separately, and act with the operator **separately one after another**.

Eigenstates of Two-qubits Operator

Also, two-qubit states' operators have eigenstates, which were found recently to be extremely useful in understanding error correction. In this section, we will introduce the

eigenstates of two-qubit operators and **their relation to error correction**. Let us start by describing the eigenstates of a ZZ operator (Z operation is performed on the first and on the second qubit). In this case, states that have an **even amount of 1s** (the eigenvalue will be $+1$) or states that have an **odd amount of 1s** (the eigenvalue will be -1). This is because the Z operator is multiplying the state by -1 every time it meets a $|1\rangle$ state.

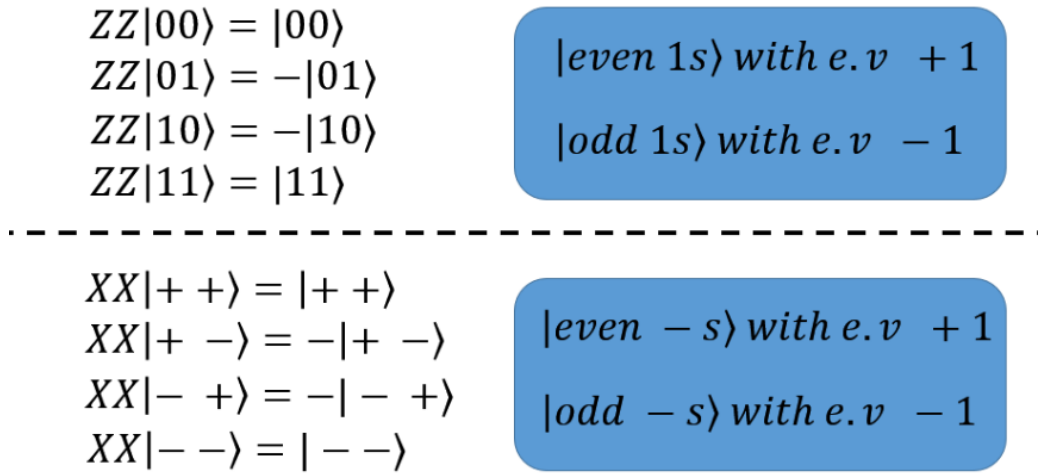


Figure 2.9: Eigenstates of 2Q gates - ZZ and XX

Eigenstates of XX are **equal superposition** of the options. It makes sense since each of the Xs is flipping its qubit:

- On the state $|++\rangle = (|0\rangle + |1\rangle)(|0\rangle + |1\rangle) = (|00\rangle + |01\rangle + |10\rangle + |11\rangle)$, **nothing will change.** So it is an eigenstate with eigenvalue $+1$
- On the state $|+-\rangle = (|0\rangle + |1\rangle)(|0\rangle - |1\rangle) = (|00\rangle - |01\rangle + |10\rangle - |11\rangle)$, we will get the same up to **minus**: $-|+-\rangle$. So it is an eigenstate with eigenvalue -1.

Stabilizer

The last and most important building block of Quantum Error Correction is the **Stabilizer**. What the stabilizer is doing, is :

1. The stabilizer's input can be any general quantum state.
2. **Collapsing** this state to a chosen **eigenstate** of an operator that we choose.

3. The output of the stabilizer **tells us** if the collapsing happened to the -1 eigenstate or to the $+1$ eigenstate.

Formally, the stabilizer defines a subspace where all states $|\Psi\rangle$ which satisfy $s|\Psi\rangle = |\Psi\rangle$ for all stabilizer elements s in the stabilizer group that defines the code.

Let's see an example of a stabilizer that is collapsing 2.1 the general 4 qubits state, to an **eigenstate of the** $ZZZZ$ operator (Z on each of the four qubits). From the last example, we know that the two possible eigenstates are those with an even number of 1s (with $+1$ eigenvalue) or those with an odd number of 1s (with -1 eigenvalue). It can be implemented using four CNOT gates where the target is an ancillary qubit:

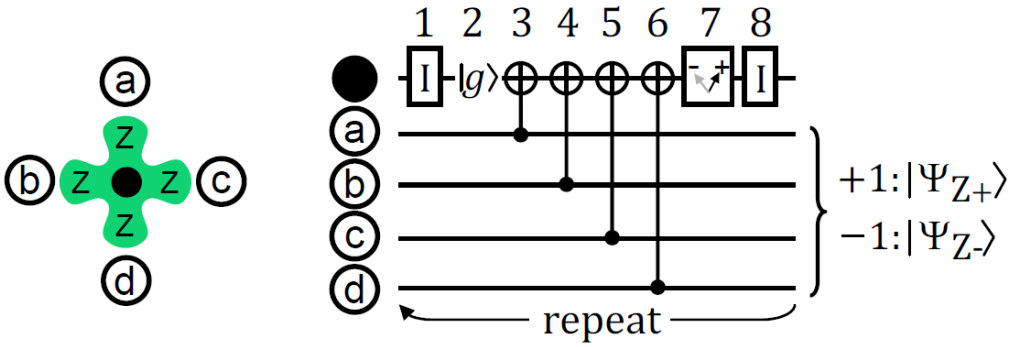


Figure 2.10: Implementation of a ZZZZ stabilizer - visual representation (left) circuit (right)

Notice that in each of the terms of the state which might be in a superposition of $2^4 = 16$ possible states ($|0000\rangle, ..., |1111\rangle$):

- **Term with an odd amount of** 1**s - in** $|abcd\rangle$, will cause the state of the first (black dot) qubit, which is initialized to $|0\rangle$, to be flipped odd times, which will result in being $|1\rangle$.

- **Term with an even amount of** 1**s - in** $|abcd\rangle$, will cause the state of the first qubit, which is initialized to $|0\rangle$, to be flipped even times, which will result in being $|0\rangle$.

Once we **measure the first qubit** since the measurement must give a result of 0 or 1, in some probability, one of these 2 cases will happen:

- **A measurement result of 1 -** All the terms with the even times of 1s in $|abcd\rangle$ will vanish, leaving us only with terms with an odd amount of 1s.

• **A measurement result of 0 -** All the terms with the odd times of 1s in $|abcd\rangle$ will vanish, leaving us only with terms with an even amount of 1s.

As will be seen later - this vanishing of terms will help us to remove errors.

For simplicity, we demonstrate here the stabilizing process in a case of two qubits (instead of four) where our arbitrary state is represented by qubits a and b and the ancillary qubits (the one that helps us to measure the parity) called M and initialized to $|0\rangle$. The stabilizing operation goes as follows:

$$
\begin{aligned}
|M\rangle|\psi_A\rangle|\psi_B\rangle = & \\
& = |0\rangle \otimes (a|0\rangle + b|1\rangle) \otimes (c|0\rangle + d|1\rangle) \\
& = ac|000\rangle + ad|001\rangle + bc|010\rangle + bd|011\rangle \\
& \xrightarrow{CNOT_{aM}} \\
& ac|000\rangle + ad|001\rangle + bc|\mathbf{1}10\rangle + bd|\mathbf{1}11\rangle \\
& \xrightarrow{CNOT_{bM}} \\
& ac|000\rangle + ad|\mathbf{1}01\rangle + bc|110\rangle + bd|\mathbf{0}11\rangle \\
& \xrightarrow{MeasureMinZbasis} \\
& |\psi_A\rangle|\psi_B\rangle = \begin{cases} |00\rangle + |11\rangle & \text{if } M_Z = 0 \\ |01\rangle + |10\rangle & \text{if } M_Z = 1 \end{cases}
\end{aligned}
\tag{2.4}
$$

The same process can be done using XXXX stabilizer, which is collapsing to states with an odd/even number of $|-\rangle$ state:

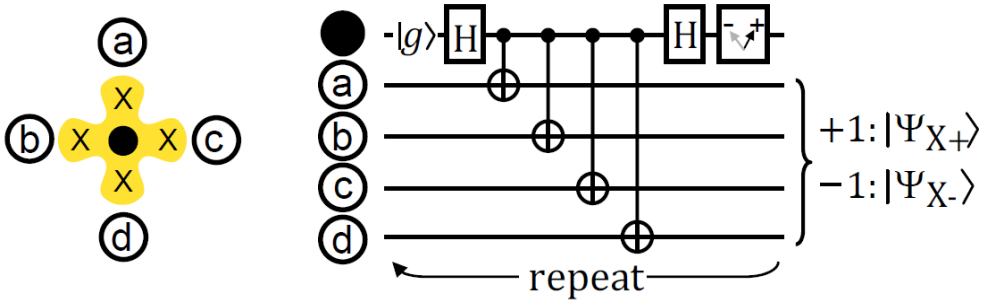


Figure 2.11: XXXX stabilizer

Eigenstates of XXXX also mean that if we try to apply XXXX on the state (flip each

of the four qubits), the state will remain **unchanged**. (up to -1 multiplied in the case of -1 eigenvalue). So a state like $|0011\rangle + |1100\rangle$ will remain the same after applying XXXX:

$$XXXX(|0011\rangle + |1100\rangle) = |1100\rangle + |0011\rangle = |0011\rangle + |1100\rangle$$

At this point, we have all background needed to know how quantum error correction is done.

Pauli Decomposition

Pauli matrices are the bases of any general matrix. Any $2x2$ matrix can be represented as a linear combination of the 4 Pauli matrices. The decomposition is given by:

$$A = a_I I + a_X X + a_Y Y + a_Z Z \tag{2.5}$$

When a_P is the amplitude of the P Pauli matrix:

$$a_P = \frac{Tr(AP)}{2} \tag{2.6}$$

For $2^n x 2^n$ matrices, we can decompose into bases of tensor multiplication of Pauli matrices:

$$A = \sum_J a_J P_J \tag{2.7}$$

Where we sum over each J combination of Pauli tensor multiplication P_J, and the a_J coefficients are:

$$a_J = \frac{1}{2^n} Tr(AP_J) \tag{2.8}$$

Decompose to Rotation

Another way to decompose a unitary operation is into parameters that describe its rotation on the Bloch sphere - the angle of total rotation, the three parameters of the rotation direction, and the global phase. When comparing to the sum of Pauli decomposition in 2.1, we can find the relation between the operator of rotation around an axis to the amplitudes of the Pauli operators [18]:

$$u(\vec{n}, \theta) = exp\left(i\frac{1}{2}\theta\vec{n}\cdot\vec{\sigma}\right) = \cos\left(\frac{1}{2}\theta\right) + i(\vec{n}\cdot\vec{\sigma})\sin\left(\frac{1}{2}\theta\right) \tag{2.9}$$

Where $\vec{n}$ is the axis of rotation, θ is the total rotation angle between the input state and the resulted state, and σ is the 3 Pauli matrices. Notice that when comparing to the composition of 2.1, we can now find θ. The total decomposition, including extracting the global phase, is:

```
# Decompose to Pauli sum
(aI,aX,aY,aZ) = decompose_2x2mat_to_pauli(A)
# global phase extraction
alpha = -1.j*np.log(np.sqrt(np.linalg.det(A)))
# Total rotation according to formula, after vanishing global phase
theta_total_rot_rad = 2*np.arccos(aI*np.exp(-1.j*alpha))
```

2.2 Noise

In this section, I follow Ref. [17]. Errors are the factor that is affecting the information and changing the state of our qubits in a way that is changing the final result of any algorithm. As long as the qubit is not interacting with anything, the state will remain the same and will not be changed. Algorithm errors arise when an interaction is happening.

In order to quantify the accuracy of a unitary operator we will use the following formulation. Any interaction can be described as a unitary operator that is acting on the whole system. Any unitary operation E is reversible, meaning that it is possible to operate with the reversed operation $E^\dagger$, and to return to the original state:

$$E^\dagger E = I \tag{2.10}$$

Where I is the identity matrix. It sounds very easy thing to do, but there are a few problems with this idea. First, we don't know which E happened. This E came from a system that is bigger and more complicated than the single qubit, so there might be a piece of information that leaked from the qubit to the outer environment. This E might also be probabilistic, from the chaotic behavior of the environment, so again, we can't know which E happened. Even if we did know which E happened, we can't engineer a system that can't operate with exactly the same $E^\dagger$ that we want, it will always be a little bit different.

At this point, we divide into 2 types of errors :

Coherent Errors

Those errors refer to errors that can be described as unitary operations, without a probabilistic nature. The source of such errors is usually an inaccurate control of the qubit.

Mathematically can be described as a unitary operation on the state:

$$|\psi\rangle \rightarrow E|\psi\rangle \tag{2.11}$$

Or on the density matrix:

$$\rho \rightarrow E\rho E^{\dagger} \tag{2.12}$$

For example, in Linear Optic Quantum Computing (LOQC) [20] that is based on linear optics, gates are implemented with couplers between two waveguides. The parameters of the waveguides and the coupling determine a gate U that we want to make. Since fabrication always comes with artifacts, the U planned will be slightly different:

$$U_{with-error}|\psi\rangle = U_{with-error}U_{ideal}^{\dagger}U_{ideal}|\psi\rangle = EU_{ideal}|\psi\rangle \tag{2.13}$$

So the error operator is:

$$E = U_{with-error}U_{ideal}^{\dagger} \tag{2.14}$$

A graphical way to describe the coherent error is as a rotation on the Bloch sphere (see subsection 2.1). Since any unitary operation can be described as a rotation matrix. Since the coherent errors are usually minor, meaning nearly Identity operator, we can also assume that their rotation angle is usually small. Finally, we can assume that coherent error is a relatively small rotation on the Bloch sphere.

Incoherent Errors

Incoherent errors cause the quantum system to lose information to the environment and, therefore, can't be corrected using an inverse unitary operation which is unknown to us [21]. Usually, we describe such errors as a set of unitary operations that might happen in some according to a set of probabilities. Mathematically we describe it as an operation on the density matrix that is summing unitary operations with different probabilities that sum to 1.

For example, a bit-flip error is causing X gate to happen in probability p (and I other-

wise):

$$\rho \to (1-p)\rho + pX\rho X \tag{2.15}$$

The new density matrix is a mixed state of the flip case and the non-flip one. The same can be done with Phase Flip error, where Z happens in some probability.

Generally speaking, we can describe in-coherent error with Kraus representation, which is a set of K matrices $[A_0,...,A_{K-1}]$ such that the evolution of a density matrix is given by: [22]

$$\varepsilon(\rho) = \sum_{i=0}^{K-1} A_i \rho A_i^\dagger \tag{2.16}$$

Notice that coherent errors can be described as a Kraus set of one unitary operator. Also, notice that it is not possible to write in-coherent as a representation of coherent error:

$$\forall K > 1 \implies \sum_{i=0}^{K-1} A_i \rho A_i^\dagger \neq E\rho E^\dagger \tag{2.17}$$

2.3 Error Correction

In this section, I follow Ref. [17]. After understanding the mechanism of errors in QC in section2.2, in this section, we will understand the most basic concepts of correcting them. We will see how it corrects X errors starting from the basic Repetition Code. We will use this code to understand the definition of the distance of any QEC code, which tells us the maximum amount of correctable errors. Then we will see how stabilizers as explained in subsection 2.1 are defined in the case of Repetition Code. Finally, I will present how we can compare the two types of coherent as explained in subsection2.2 and non-coherent errors.

Repetition Code

This code is explained to demonstrate basic concepts of error correction. This type of encoding duplicates the information just as in the classical case from Ref. [23], e.g., from a single qubit to three qubits. It can identify and correct the state in case of an X/Z error (depending on the type of the code). For example, the following technique is used to correct X-errors:

1. We start with a qubit that contains the **information**, which is the 2 probability coefficients of $|0\rangle$ and $|1\rangle$.

2. **Entangle/share/encode** this information, using a **CNOT** gate, with two more qubits.

3. Use two more ancilla qubits initialized to $|0\rangle$, and **will be measured to look for an error at the end.**

4. Entangle/share the information of qubits **1,2 with the first ancillary.**

5. Entangle/share the information of qubits **2,3 with the second ancillary**.

6. **Measure** the state of the two ancillaries.

In case there is **no error**- the measurement will return $(0,0)$ since, in every term of the superposition, each ancillary **flipped itself an even amount** of times (zero times in the $|000\rangle$ term and two times in the $|111\rangle$ term):

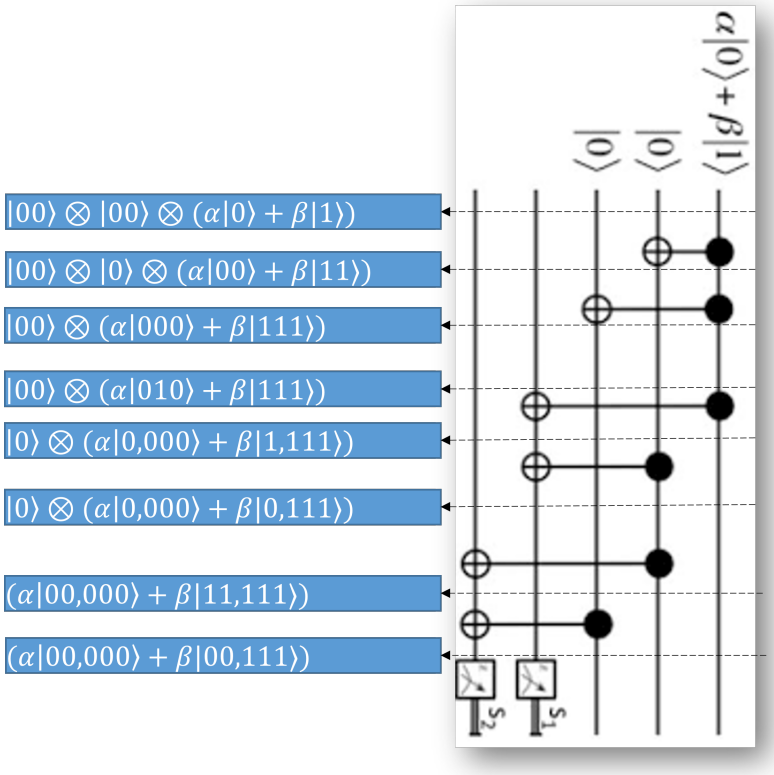


Figure 2.12: Repetition code in the case without error. Every 2 (cyclic) neighbors data qubits are measured on a ZZ basis into ancilla qubits. Without error, the ancilas will be measured as $|00\rangle$

.

We can treat the encoded state as a new base for our computation, and we can call $|000\rangle = |0_L\rangle$ (**Logical zero**) and $|1\rangle = |1_L\rangle$ (**Logical One**).

In case there is **single X error**- for example, in the 2nd qubit, the measurement will return $(1,1)$ since, in every term of the superposition, each ancillary **flipped itself an odd amount of times** (one time in each term):

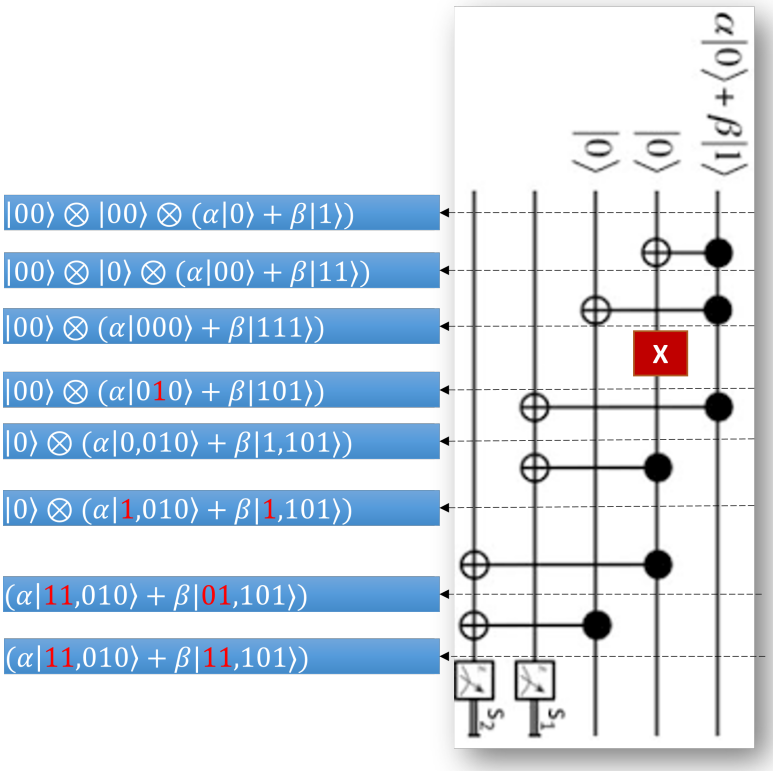


Figure 2.13: Repetition code in the case with one error

Distance of Code

The maximum amount of errors that this code can handle is **only one X error**. There is a very intuitive way to understand why. Look at this example of **two X errors**:

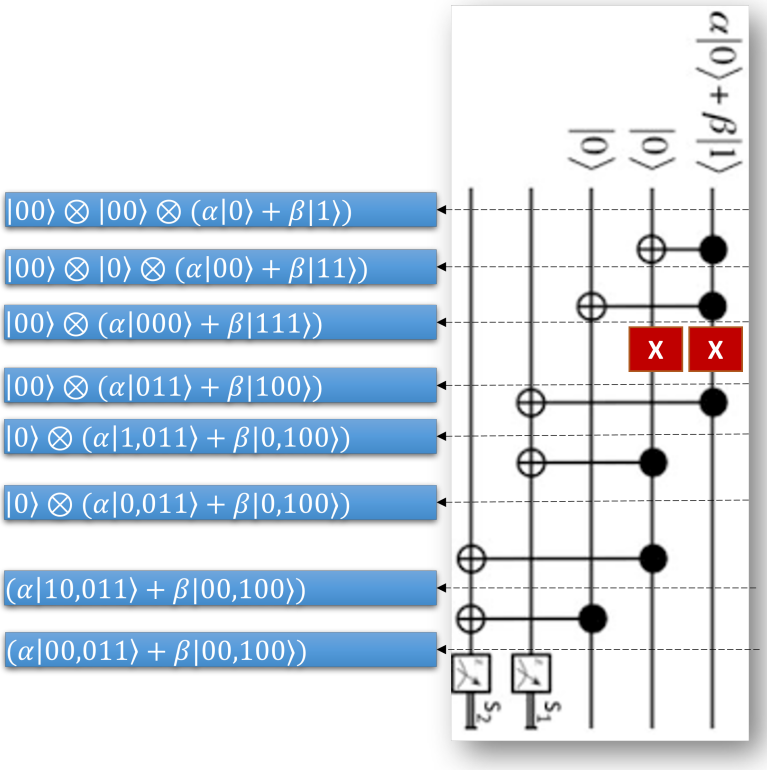


Figure 2.14: Repetition code in the case with two errors

Since both qubits 1 and 2 flipped with the X error, the first ancillary flipped itself an even number of times.

1. Flipped 2 times from the $|000\rangle$ term that became $|011\rangle$ because of the errors.
2. And zero times from the $|111\rangle$ term that became $|100\rangle$ because of the errors.

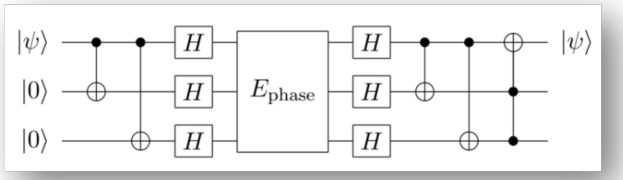
The **Distance of Quantum Code** (and also for classical codes) is defined as the number of bits you have to flip to switch between logical states. In our case, flipping three qubits is needed to switch between the two logical states $|0_L\rangle = |000\rangle$ and $|1_L\rangle = |111\rangle$. Therefore, in this case, the **distance of the code distance is 3**.

Since the codes assume that the majority is the 'real' information in the state (the value that is shown a higher amount of times in the term), we can say that the maximum errors that are allowed with a logical error to happen are **less than half**: $\lfloor \frac{d}{2} \rfloor = \lfloor \frac{3}{2} \rfloor = 1$.

In case of more than one error, we will have a logical error.

Correction of Z errors

Dealing with Z errors (phase flip errors) can be done in a very similar code that change's its base from eigenstates of Z ($|0\rangle$ and $|1\rangle$) into eigenstates of X ($|+\rangle$ and $|-\rangle$):



$$|000\rangle \rightarrow |{+}{+}{+}\rangle \rightarrow^{Z_2\text{-}error} |{+}\ {-}\ {+}\rangle \rightarrow |010\rangle$$

Figure 2.15: Repetition code to correct Z errors

The Stabilizers of the Repetition Code

For the repetition code that we analyzed, its stabilizer (as explained in subsection 2.1) group is the group of three operators: Z_1Z_2, Z_2Z_3, and Z_3Z_1. The uniqueness of Z_aZ_b measurement is this is an **operation that collapses** (like any quantum measurement) **the state that it measures to one of two possible cases** that are the eigenstate of Z_aZ_b. The eigenstates of such an operator :

- **Case A -** quantum terms that have an **even** number of 1s in their states - $|00\rangle$ and $|11\rangle$. Those will have a **positive** eigenvalue since each Z will take **even** times of minus sign out of the term.

- **Case B -** quantum terms that have an **odd** number of 1s in their states - $|01\rangle$ and $|10\rangle$. Those will have a **negative** eigenvalue since each Z will take **odd** times of minus sign out of the term.

This is exactly what the repetition code is doing. Notice how the CNOT gate will flip twice the ancillary in terms that have even parity and vice versa. Notice also how the ancillary measurement ensures that **only one of the cases will survive** as can be shown in section 2.3. So now we understand what is the meaning when we say Z_1Z_2, Z_2Z_3, and Z_3Z_1 are the stabilizers that is defined in subsection 2.1 of the repetition code, and we are ready to understand the equivalence between coherent and in-coherent errors.

The Equivalence Between Coherent and Incoherent Errors

We might think that the only error that can happen to a qubit is some Pauli X / Y / Z in some probability p. Actually, quantum states are very **fragile** and can have errors in **100% probability**, most of the time those errors will be a **small unitary rotation on the Bloch Sphere** as defined in subsection 2.1.

This type of error, we call **Coherent Errors** as mentioned in section 2.2. Most of the time, we will describe them as an operator that is a **sum of 4 operators**, that creates a **superposition of 4 things that happened to the state**: I, X, Y, Z.

In the following example, we will understand why that is possible. Consider a general unitary X rotation error operation that happened to the state:

$$R_x(\theta) = exp(-iX\theta/2) = \begin{bmatrix} cos(\theta/2) & -isin(\theta/2) \\ -isin(\theta/2) & cos(\theta/2) \end{bmatrix} \tag{2.18}$$

Which can be described as:

$$R_x(\theta) = exp(-iX\theta/2) = cos(\theta/2)I - isin(\theta/2)X \tag{2.19}$$

Notice the R_x rotation is splitting the original state (that didn't have any error) to a superposition of 2 cases - **nothing** happened to it (with **big** amplitude) and **X** happened to it (with **small** amplitude):

$$R_x^{Q_2}(\theta)\,|0_L\rangle = cos(\theta/2)\,|000\rangle - isin(\theta/2)\,|010\rangle \tag{2.20}$$

When trying to make an error correction on such an error, we know how to solve each of the I / X cases separately, but we still don't know what will happen if the state is in a superposition of **I and X at the same time**.

The state **collapses to one of the 2 cases of I/X**, in a probability that depends on the rotation of the angle! + **We will know if X happened** and we will know to correct it! Let's see it in the math:

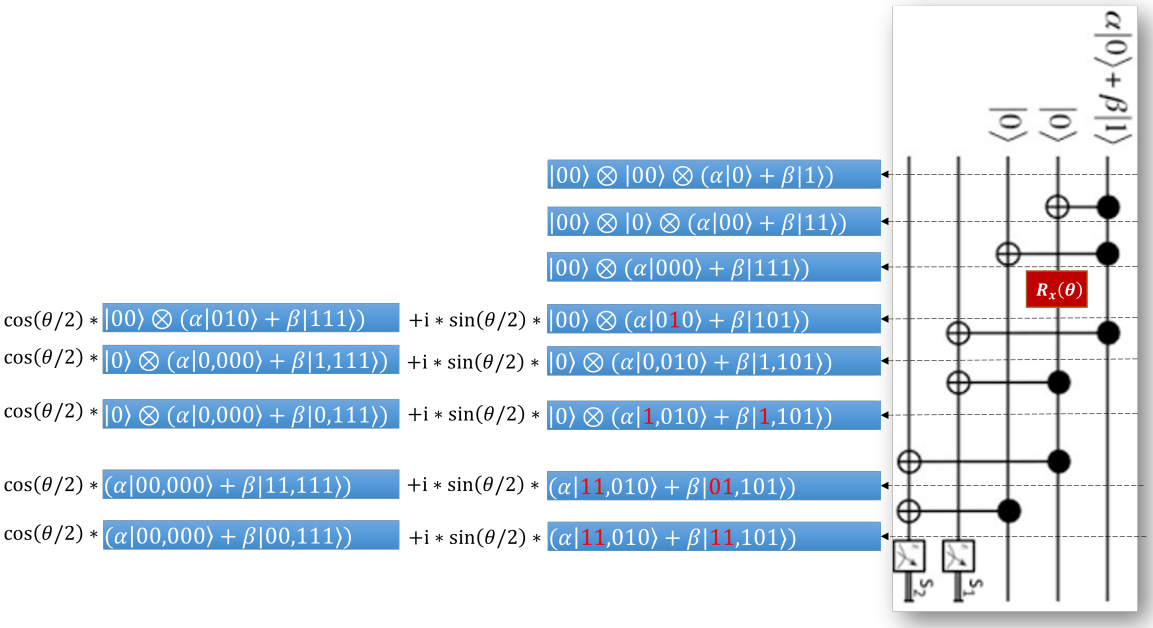


Figure 2.16: Repetition code with Rx Error

So if we examine the final state just before measurement, we know that each one of the stabilizers of the code **will allow only one of its two eigenstates**, so the only terms that will **survive are terms that have matching eigenvalues** of the stabilizers:

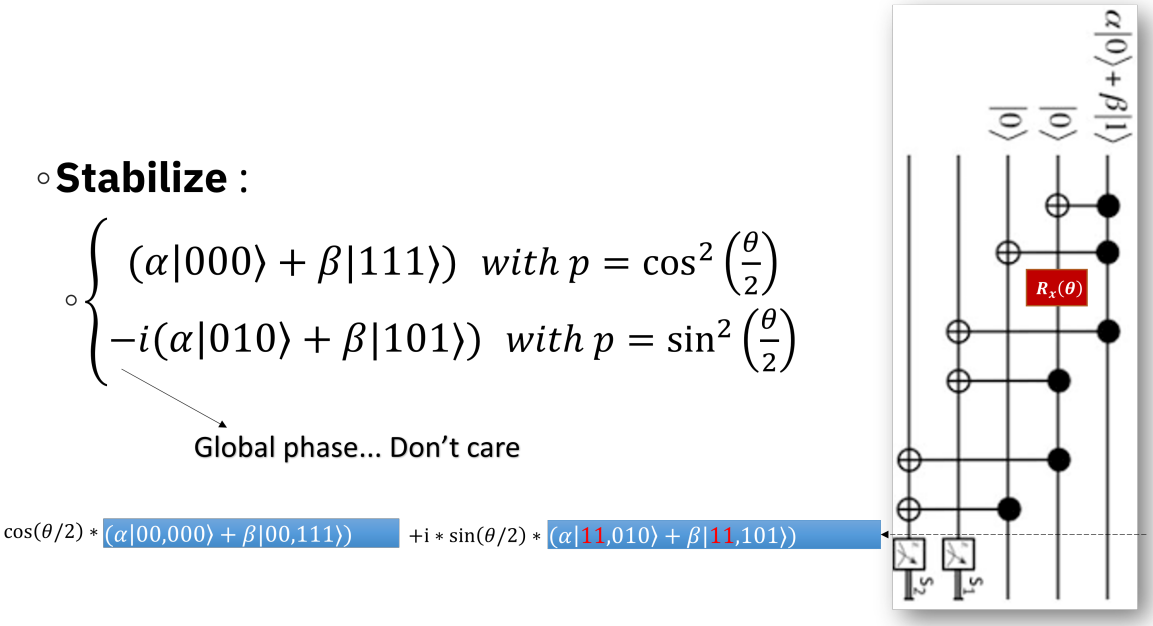


Figure 2.17: Repetition code with Rx Error after measurement

This process can be generalized to any general unitary rotation around the Bloch Sphere, which can be described as a linear combination of I, X, Y, and Z:

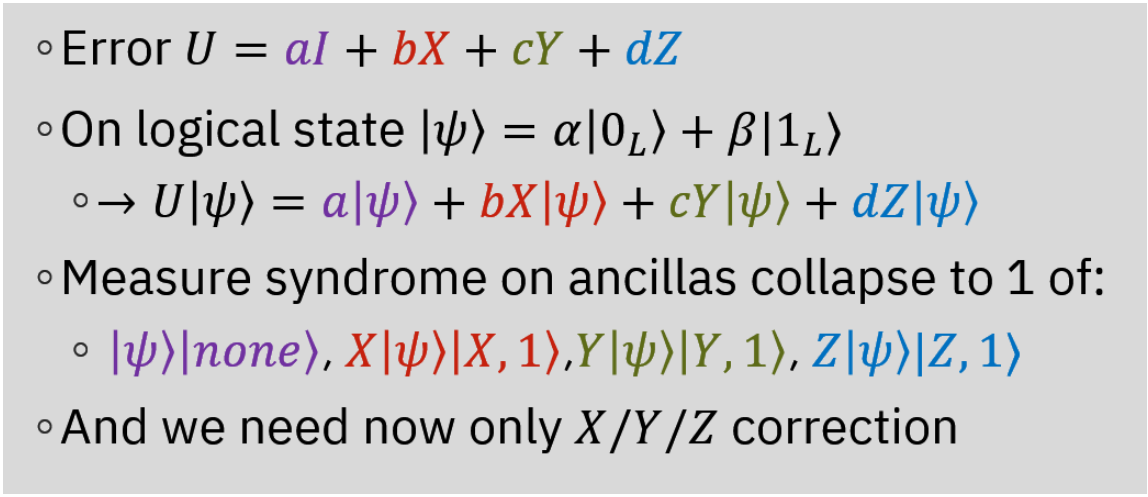


Figure 2.18: Coherent error as a superposition of Pauli

The Non-Equivalence Between Coherent and Incoherent Errors

Following the discussion of the equivalence between coherent and non-coherent errors, we will see now that in the general case, it is a very heavy challenge to find a non-coherent error that is equivalent to the given parameters of the coherent error.

Let's start with an example, a simple stabilizer as defined in subsection 2.1, that collapses to eigenstates of ZZ measurement and deals with X errors in only two data qubits.

After first stabilization, we will have $|00\rangle + |11\rangle$ on the two data qubits (if ancillary is measured as zero).

After coherent error of the form $E = (\sqrt{1-e_1^2}I + e_1X)(\sqrt{1-e_2^2}I + e_2X)$ the state will be :

$$
\begin{aligned}
|\psi\rangle = \\
&= |0\rangle E(|00\rangle + |11\rangle) \\
&= |0\rangle(E|00\rangle + E|11\rangle) \\
&= |0\rangle((\sqrt{1-e_1^2}I + e_1X)(\sqrt{1-e_2^2}I + e_2X)|00\rangle + (\sqrt{1-e_1^2}I + e_1X)(\sqrt{1-e_2^2}I + e_2X)|11\rangle) \\
&= |0\rangle((\sqrt{1-e_1^2}|0\rangle + e_1|1\rangle)(\sqrt{1-e_2^2}|0\rangle + e_2|1\rangle) + (\sqrt{1-e_1^2}|1\rangle + e_1|0\rangle)(\sqrt{1-e_2^2}|1\rangle + e_2|0\rangle)) \\
&= |0\rangle((\sqrt{1-e_1^2}e_2 + e_1\sqrt{1-e_2^2})|01\rangle + (e_1\sqrt{1-e_2^2} + \sqrt{1-e_1^2}e_2)|10\rangle + \\
&+ (\sqrt{1-e_1^2}\sqrt{1-e_2^2} + e_1e_2)|00\rangle + + (e_1e_2 + \sqrt{1-e_1^2}\sqrt{1-e_2})|11\rangle) \\
&= |0\rangle[(e_1\sqrt{1-e_2} + \sqrt{1-e_1^2}e_2)(|01\rangle + |10\rangle) \\
&+ (e_1e_2 + \sqrt{1-e_1^2}\sqrt{1-e_2^2})(|00\rangle + |11\rangle)]
\end{aligned}
\tag{2.21}
$$

After one more stabilization, we will collapse:

$$
\begin{aligned}
p &= (e_1\sqrt{1-e_2^2} + \sqrt{1-e_1^2}e_2)^2, to - |01\rangle + |10\rangle \\
p &= (e_1e_2 + \sqrt{1-e_1^2}\sqrt{1-e_2^2})^2, to - |00\rangle + |11\rangle
\end{aligned}
\tag{2.22}
$$

which is **not** like saying that X_1 will happen in $p = e_1^2$ and X_2 in $p = e_2^2$ which looks like:

$$
\begin{aligned}
p &= (e_1\sqrt{1-e_2^2})^2 + (\sqrt{1-e_1^2}e_2)^2, to - |01\rangle + |10\rangle \\
p &= (e_1e_2)^2 + (\sqrt{1-e_1^2}\sqrt{1-e_2^2})^2, to - |00\rangle + |11\rangle
\end{aligned}
\tag{2.23}
$$

The coherent case has a term of $+2e_1e_2\sqrt{1-e_2^2}\sqrt{1-e_1^2}$ in both flip and non-flip. This term does not exist in the non-coherent case.

Where the coherent in the worst case is twice bigger than the twirl in this case:

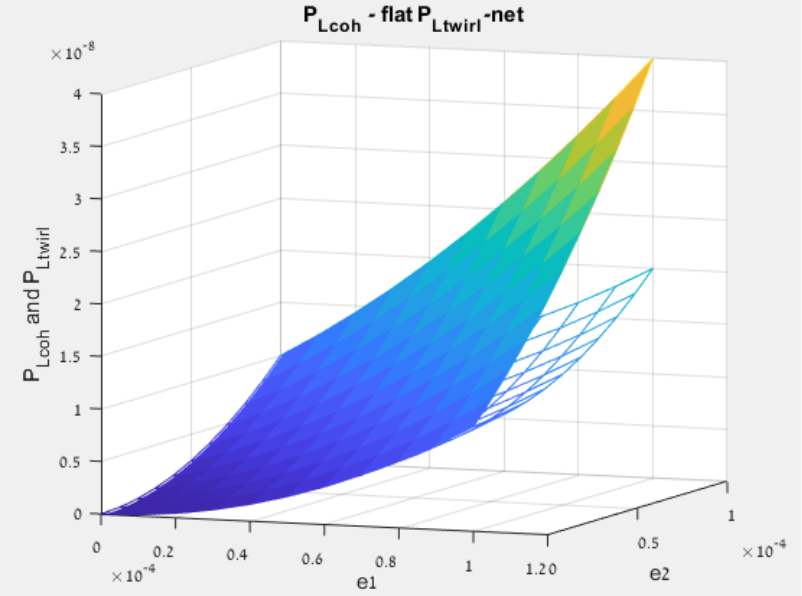


Figure 2.19: Twirl (Non-Coherent) and Coherent Error vs e_1 and e_2 - size of errors

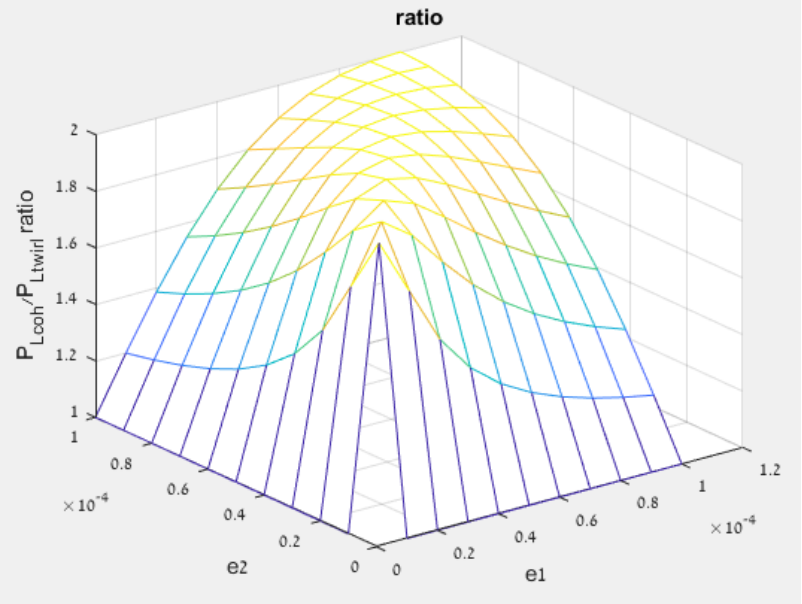


Figure 2.20: Ratio of Coherent Error / Twirl (Non-Coherent) vs e_1 and e_2 - size of errors

The analytical way to convert a coherent error into an incoherent error is as follows:

1. To calculate the matrix E

2. Decompose is into terms like $a_{ijk}XXZ$

3. Group the terms that give the same eigenstates of the stabilizer 2.1 (e.g. XI and IX, also II and XX)

4. Use the terms of each group, to operate with probabilities, in only one operator (e.g. use the group terms of XI and IX to make only XI

This way is very complicated in more than a few qubits, since we need to gather together all the terms with equivalent errors. In the next chapters, we will see how to solve this numerically with simulations.

Further discussions about equivalence and non-equivalence between coherent and incoherent errors, can be found in Ref. [24–26]

2.4 Surface Code

In this section, I follow Ref. [1].

Introduction to Surface Code

One of the most common methods to create a fault-tolerant quantum computing architecture is called Surface Code (SC). Its fundamentals are very similar to the more simple repetition code, but its topological behavior is making it a great code that deals greatly with local errors.

Let's not restrict ourselves to looking at a 1-dimensional string of qubits with stabilizers limited to the neighbor on the right and left. Perskil said [27, 28] that Topological Quantum Computation can be more fault tolerant than conventional approaches. Once our information is encoded on some large structure, it is less sensitive to local errors, and the whole structure does not broke. Think of a string with a loop. Minor changes will not change the fact that it is only one loop. It will be used to create a new kind of Quantum Error Correction scheme.

So, surface code deals with tying strings from one side to another. Those strings are made of a $|1\rangle$ of a qubit. The basic idea is to tie strings from one surface side to another. We encode the information in the place where they are linked. There is a vast difference

between the topology of a string that is tied from one side to another, then tying the string from both ends to the same side.

Surface Codes use this fact to encode fault tolerant $|0\rangle$ and $|1\rangle$. For example, if we consider pink dots as $|1\rangle$ and white dots as $|0\rangle$, a term in the logical $|0_L\rangle$ state can look like :

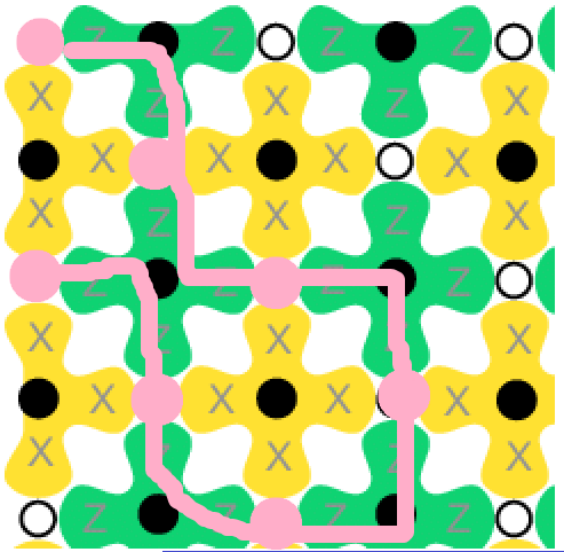


Figure 2.21: A possible term in the superposition that is composing logical $|0_L\rangle$

While its twin term in the logical $|1_L\rangle$ state will look like this:

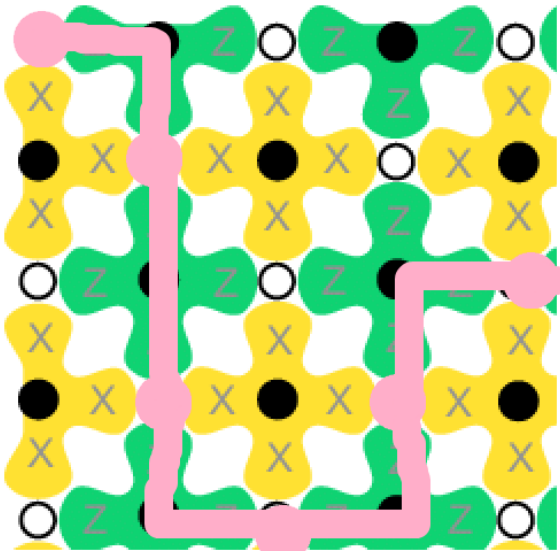


Figure 2.22: A possible twin term (to the last figure) in the superposition that is composing logical $|1_L\rangle$

Notice that the second one was created by tearing the string and reattaching it on both ends.

Let's see another example. When a term in the $|1_L\rangle$ that looks like this string -

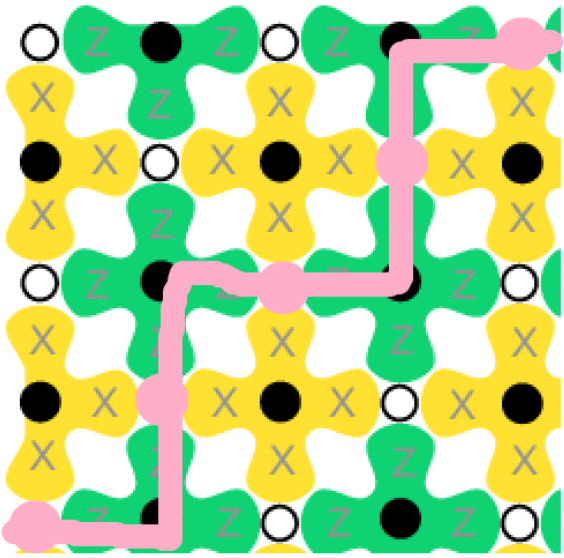


Figure 2.23: A possible term in the superposition that is composing logical $|0_L\rangle$

Its twin term in the $|0_L\rangle$ state will look like this -

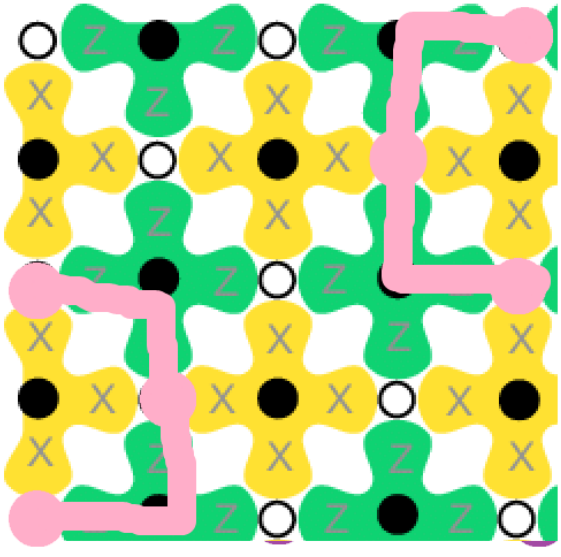


Figure 2.24: A possible twin term (to the last figure) in the superposition that is composing logical $|1_L\rangle$

The big picture is that the whole state of $|0_L\rangle$ and $|1_L\rangle$ will look like a superposition of

all possible strings between the surface's left and right walls. On the other side, the two states are very well distinguished because we need a long chain of X flips (logical X gate) to change the $|0_L\rangle$ into the $|1_L\rangle$ state (and vice versa). This unique property promises that random unwanted X flips are unlikely to create such a long chain of X flips because of a very low probability of happening. This is what makes the surface code so immune to faults.

In the following sections, we will see more examples and explain how surface codes create those states of superposition of strings using stabilizers that we already discussed.

Stabilizers of Surface Code

We saw in the last section 2.4 that surface code is made of strings of $|1\rangle$ between the right and left walls. This time we will learn how stabilizers create those strings. Please note that we are assuming a quiescent state - meaning there are no faults in the qubits.

ZZZZ Stabilizers

As in the explanation about the ZZZZ stabilizer, it tells us the amount of one in the four qubits that it measures - if it is odd or even. +1 measurement tells us that it is even. Now think of strings passing through that cross shape of the stabilizer. Notice that it will always go through an even number of vertices- two times in case of one string, zero times in case of no string, and four times in case of two strings.

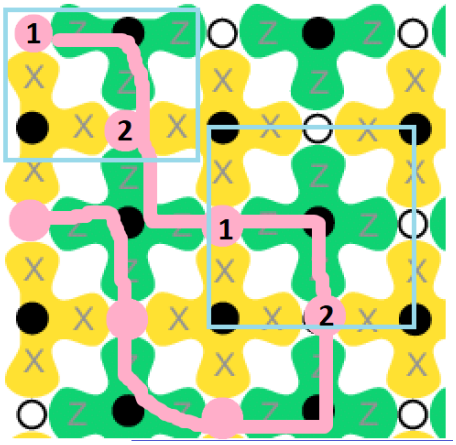


Figure 2.25: In case where one string is passing through, two vertices will be 1

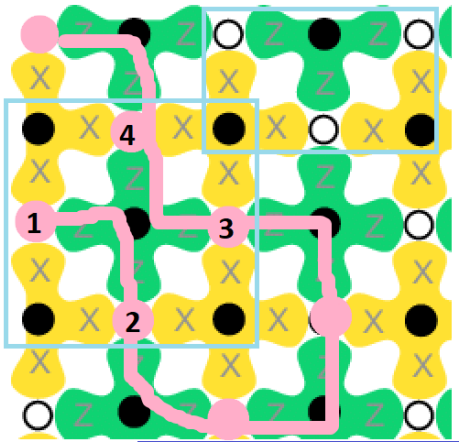


Figure 2.26: In case where two strings are passing through, four vertices will be 1. In case of zero strings, zero vertices will be 1

Now we see that once we perform all the ZZZZ stabilizers, one after another, if they all return a +1 result - the resulting case is a superposition (we are in quantum physics) of strings that are tied between the left and right walls. Strings between the top and bottom are illegal because they will enforce the ZZZZ stabilizer result to be -1.

XXXX Stabilizers

Let's have a look at the XXXX yellow stabilizers, and what they are forcing. If their result is +1, it means that they are forcing eigenstates on the measured qubits. For the eigenstates of XXXX measurement, we need to find states that left the same after performing the four X operators. That might be impossible since Xs are flipping the qubit. In the magical world of quantum, everything is possible. The XXXX stabilizer will allow a superposition of 2 terms with XXXX between them. For example, GHZ state -

$$XXXX(|0000\rangle + |1111\rangle) = |1111\rangle + |0000\rangle = |0000\rangle + |1111\rangle$$

Notice it forces the amplitude of the 2 terms to be equal.

So once all XXXX stabilizers act on the surface, they force each string to have a twin, one more term in the superposition that allows the XXXX stabilizer to return a +1 result. If we assume that the surface is in a superposition of some strings -

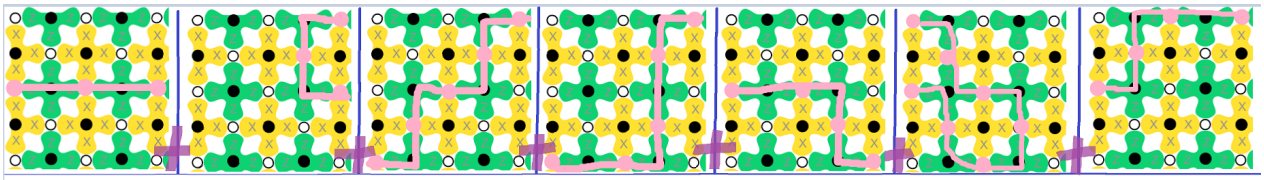


Figure 2.27: A superposition of possible strings on surface

And here is what the twin look like. First, notice that the qubits (white dots) on the top and bottom rows experience XXXX stabilizer only one time, so in the twin, they must have the X flip of the original -

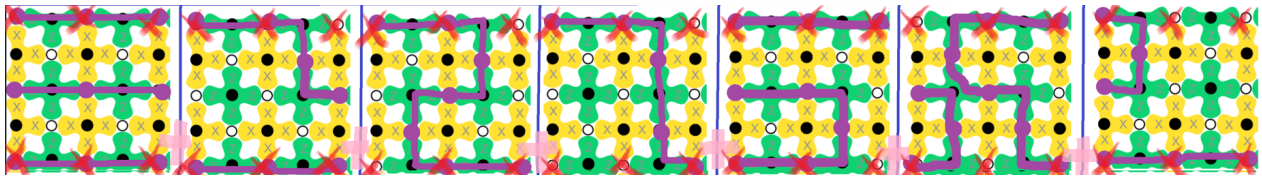


Figure 2.28: A superposition of the twins of the original strings from figure 2.27 - Top and Bottom row are marked as the qubits that must be flipped

And in the other qubits, since they have two XXXX stabilizers from two sides, they must remain the same as the original because they flip twice -

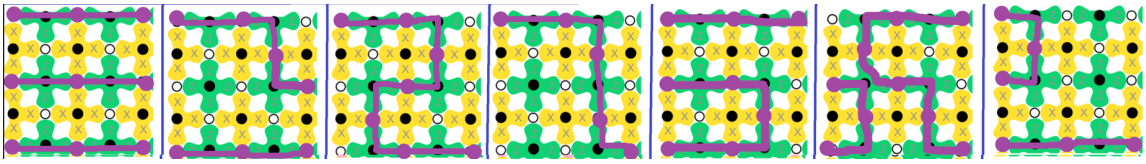


Figure 2.29: A superposition of the twins of the original strings from 2.27

And the total state, in this case, will be a superposition of all the strings and their XXXX stabilizers twin -

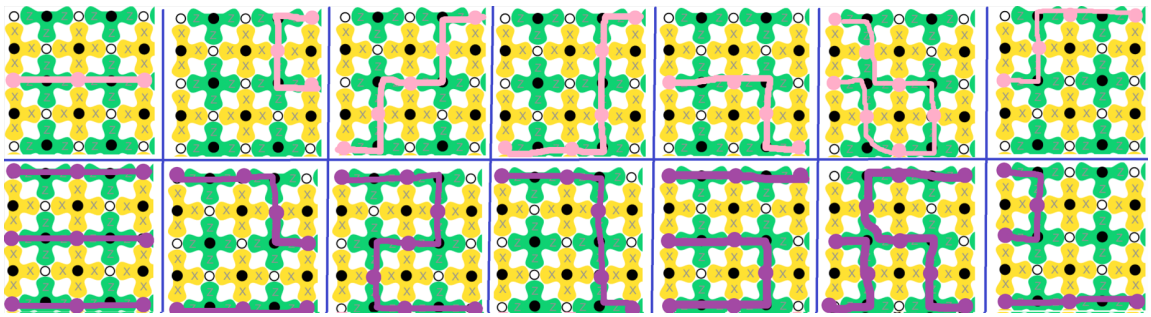


Figure 2.30: A superposition of original and twins

Error Corrections in Surface Code

One Error Case

In the following section, we will see how the state is corrected in the **case of a physical error that happens to the qubits on the surface, only one error in a time frame**.

In some probability ("the physical error rate"), an X/Y/Z flip can happen to one of the qubits. In this case, the stabilizers (measurements) next to the error are doing two things:

1. In case of coherent error which is a sum of Pauli operators, **the error vanishes** in some big probability. Or convert the error to a simple Pauli X/Y/Z error in a small probability.

2. The stabilizer **will indicate** whether the Pauli error is still there, where it is, and what its type is (this information is usually enough to correct it manually!)

Let's see it using a graphic example. Suppose we start from a quiescent state, which is a superposition of an even amount of strings from the right edge to the left edge of the surface:

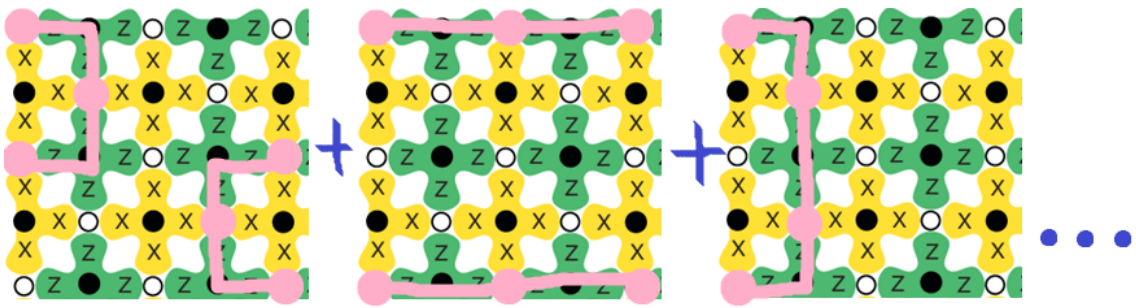


Figure 2.31: State without errors, a superposition of all cases of even amount of strings of $|1\rangle$ from left to right edge

Notice that each stabilizer measurement (on the black dots ancillary qubits) will return $+1$ result since there are even amounts of $|1\rangle$ to each stabilizer.

In a case of an error, let's assume X error:

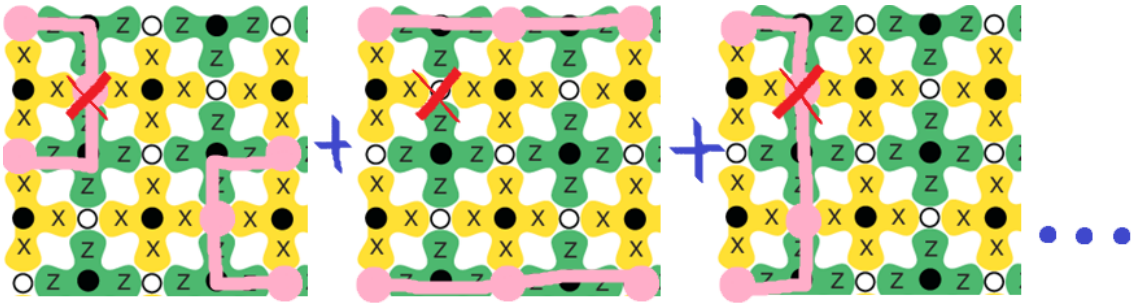


Figure 2.32: Same state with one X error on one of the qubits, and affecting all terms of superposition

Which flips $|0\rangle$ to $|1\rangle$ and flips$|1\rangle$ to $|0\rangle$:

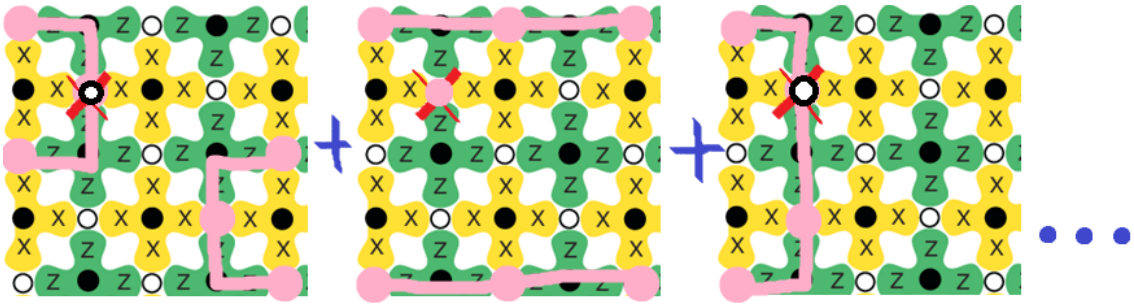


Figure 2.33: The X error is "tearing the string", and flipping the state of the qubits in each term of the superposition

Two stabilizers will prompt that they found the error because a measurement of $ZZZZ$ **stabilizer (the syndrome) that finds if the amount of** $|1\rangle$**s is even or odd** will see an odd amount of $|1\rangle$s:

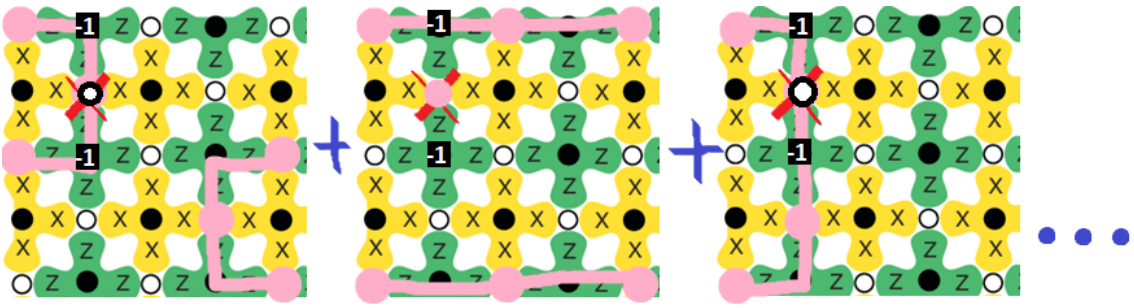


Figure 2.34: The effect of the one X error on the syndrome is a -1 result to each of the ZZZZ stabilizers that touch the error

Now, imagine that you do not know the error happened. All you know is the measurements that are going to the classical computer that controls the error correction: **the -1 results of the ZZZZ stabilizers** (also XXXX, but we do not look at them for now), so **all you see is**:

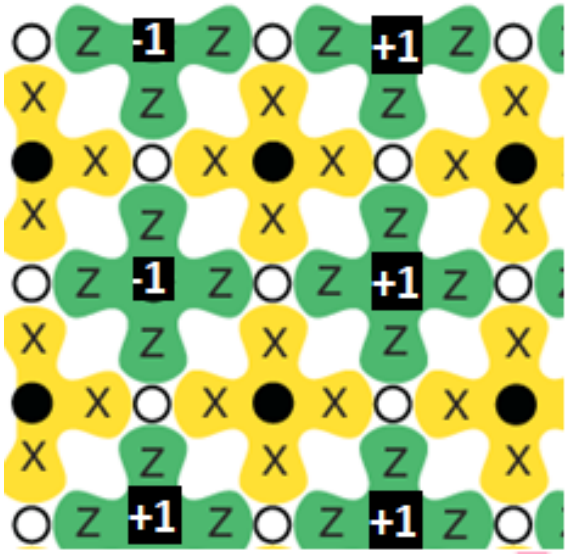


Figure 2.35: The syndrome is a -1 result to each of the ZZZZ stabilizers that touch the error which is yet unknown

It is easy to see that probably what happened is a **Pauli X operation between the two -1 results!** (it could also occur from more than one error case, but we keep it to the next chapter).

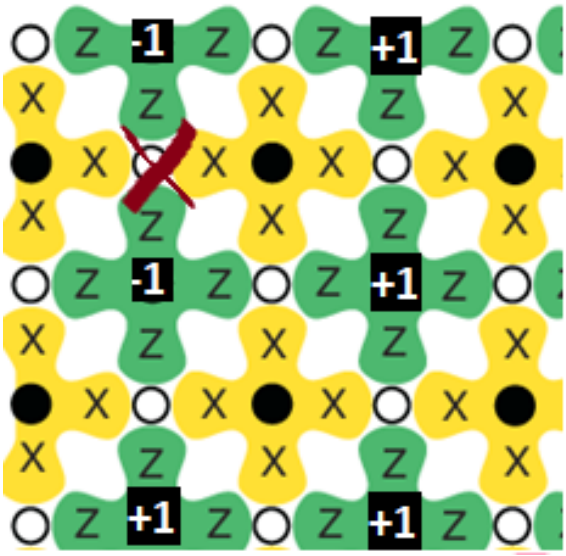


Figure 2.36: Assuming only one error happened, the error is an X error that happened between the two -1 results from the syndrome of the ZZZZ stabilizers

Since XX=I, we can operate with X again on this errored qubit and fix that error.

Trivial Errors

This section explains a critical term in error correction - **trivial errors**. Which is essential to understand other issues later. A trivial error is an error that happens to the encoded state but actually does not change it at all. It remains the same thing.

For example, we can see it through the topological properties of surface code. Let's look at a quiescent state of surface code which is a superposition of all possible cases with an even string amount from the left edge to the right edge (internal loops do not affect the oddity of the state):

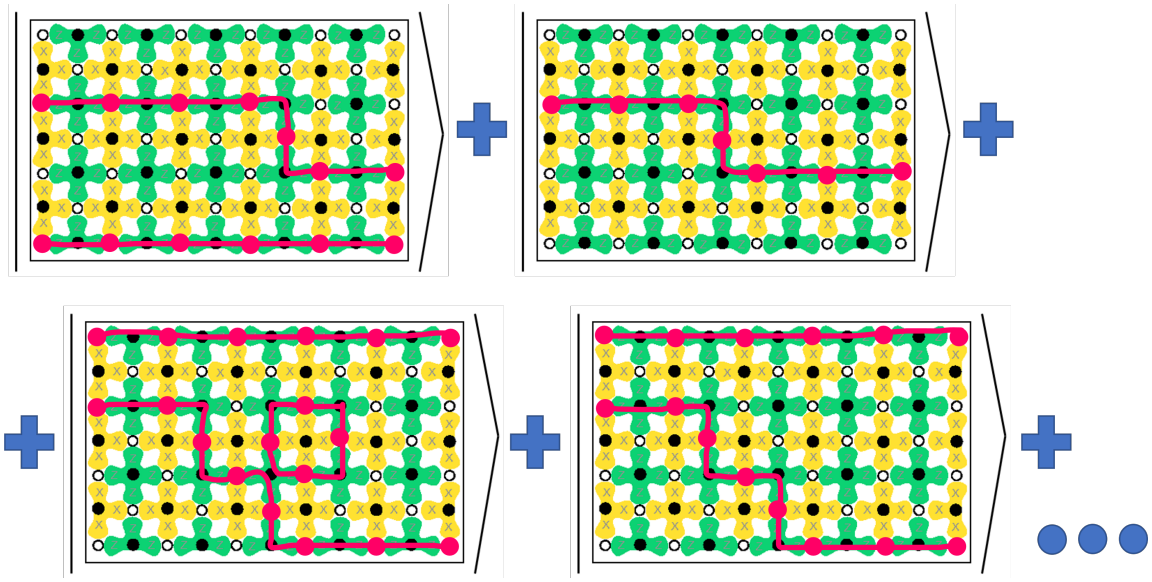


Figure 2.37: $|0_L\rangle$ state is a superposition of all possible even amount of strings of $|1\rangle$ from left to right. The addition of internal loops does not change the oddity, therefore for any term, exists a twin with an additional internal loop. Here we see two of the terms with their twin with a small square loop

Notice that the difference between the right and the left picture is just an internal loop which is legal because it keeps an even amount of strings from right to left.

Now we can quickly see what a trivial error is. Let's assume that a loop of four X flips happened in the following location:

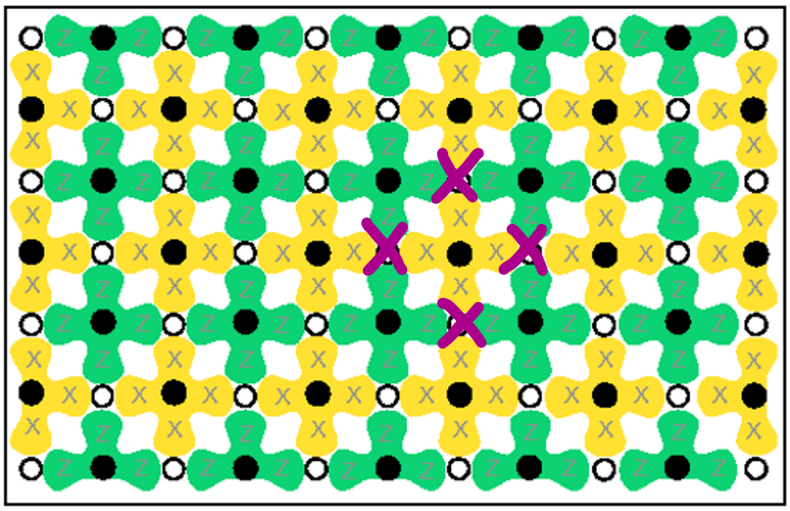


Figure 2.38: An example of an error which is an internal loop chain of X flips, matches the loop difference of Fig. 2.37

Since terms with and without this XXXX loop are anyway contained in the quiescent no errors state, the state will remain **the same**. The left will become the right, and the right will become the left, which is the same since they are in a superposition anyway:

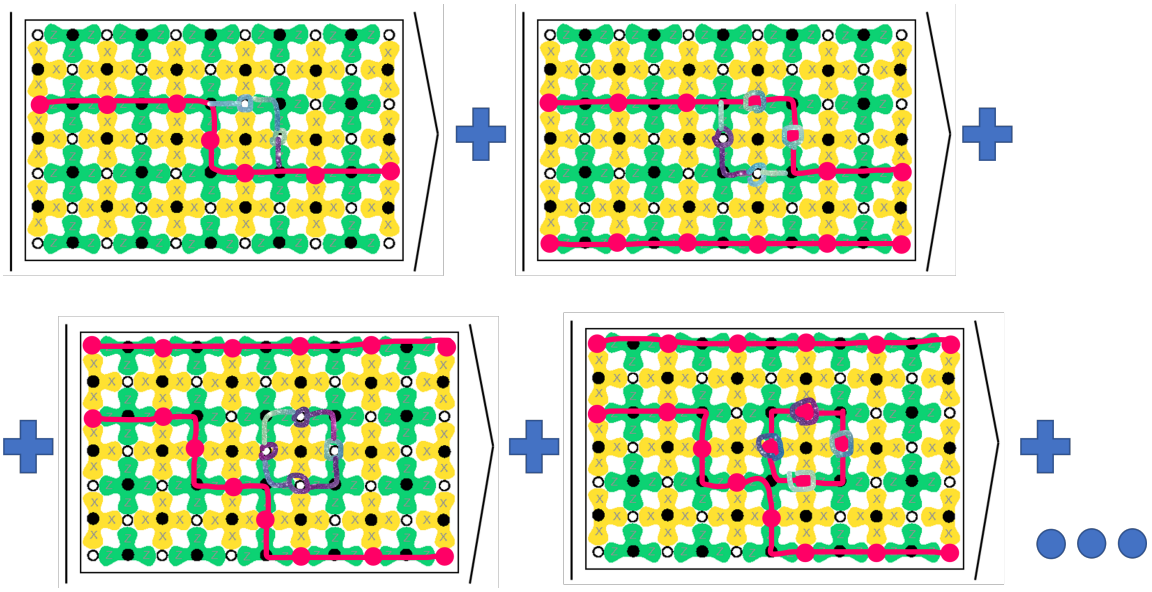


Figure 2.39: The state did not change after the error from Fig. 2.38. The right terms from Fig. 2.38 became the left terms, and vice versa.

So, when dealing with trivial errors in surface code, this is actually the internal loops. Any closed loop will not change the state anyway because there will always be a term that is in the superposition anyway.

Trivial Error and Error Correction

Trivial errors have an interesting effect on error correction. Sometimes we create a loop, and it doesn't happen by errors. For example, if there is a syndrome that looks like a chain of two X errors:

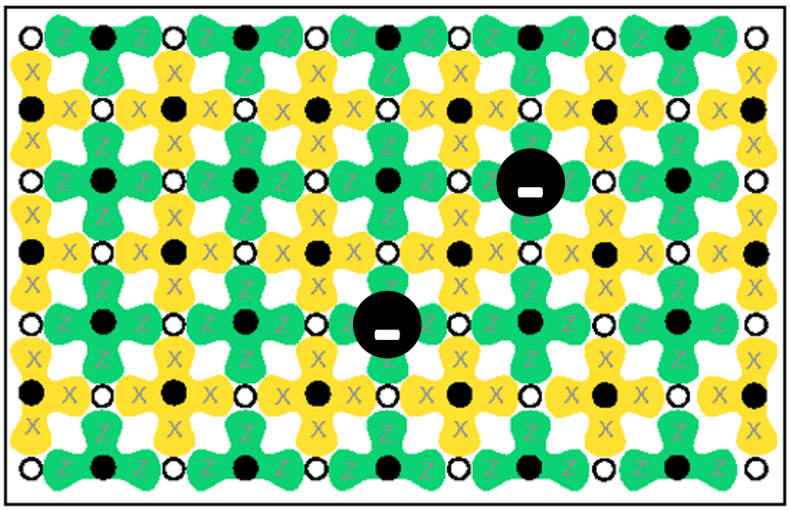


Figure 2.40: A measurement of the syndrome, raised these results which imply a chain of errors that connects them

This can be caused by these two X errors (Notice that +1 syndrome seems like no error because it has two X - even amount of X errors, which affect it as a $-1 * -1 = +1$):

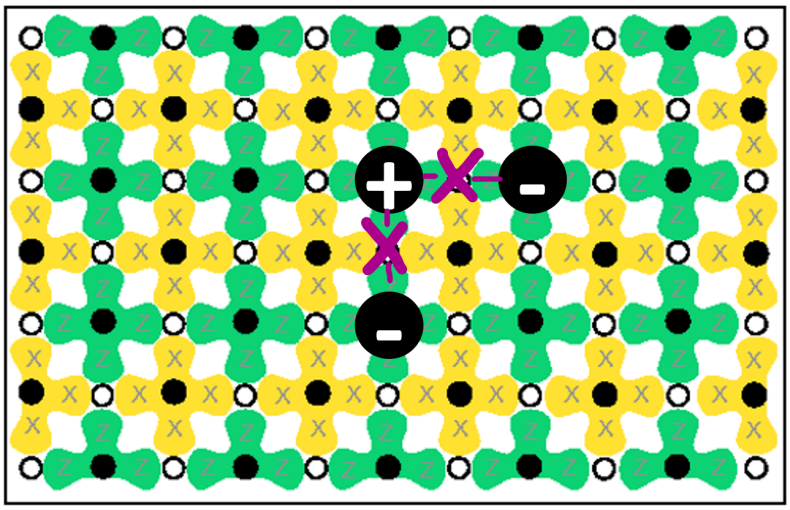


Figure 2.41: Chain of top two X flips might cause that syndrome

Or by those two X flips:

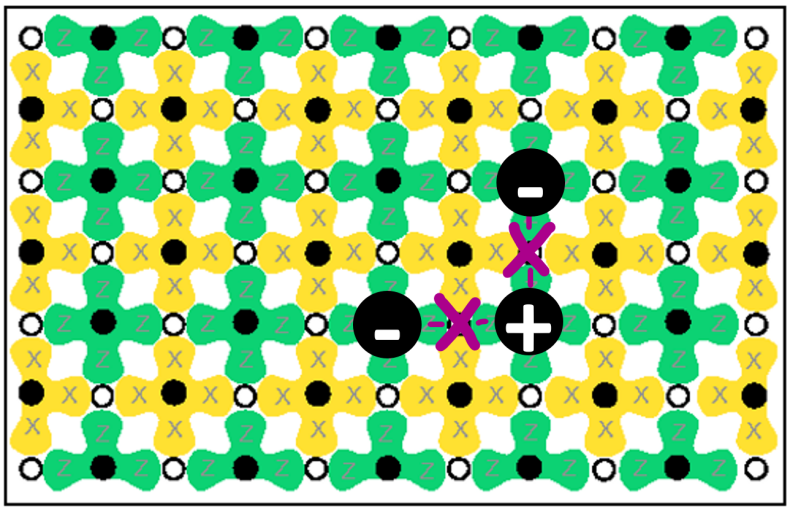


Figure 2.42: Chain of bottom two X flips might cause that syndrome

Or even by a longer chain of flips:

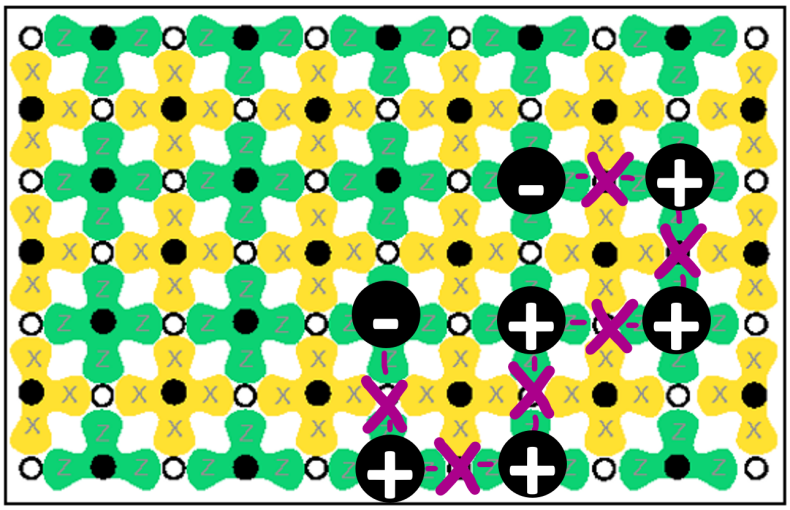


Figure 2.43: Long chain of X flips might cause that syndrome too

So now the computer should decide which error happened. Actually, we can fix which error we like. As long it matches the syndrome, it will close an internal loop, the error will become **trivial** (because it will be a **loop** of X), and it will be corrected. We will pick the

shortest path because it is the most probable, and as we will see in the next section, the correction might create an error that is not trivial and not an internal loop (it might make an external loop which is BAD!) Logical Error! So, for now, in the example of the long X chain, fix some short path connecting the two minuses of the syndrome, and we will be fine. In any error that happens, the loop will be closed in any way.

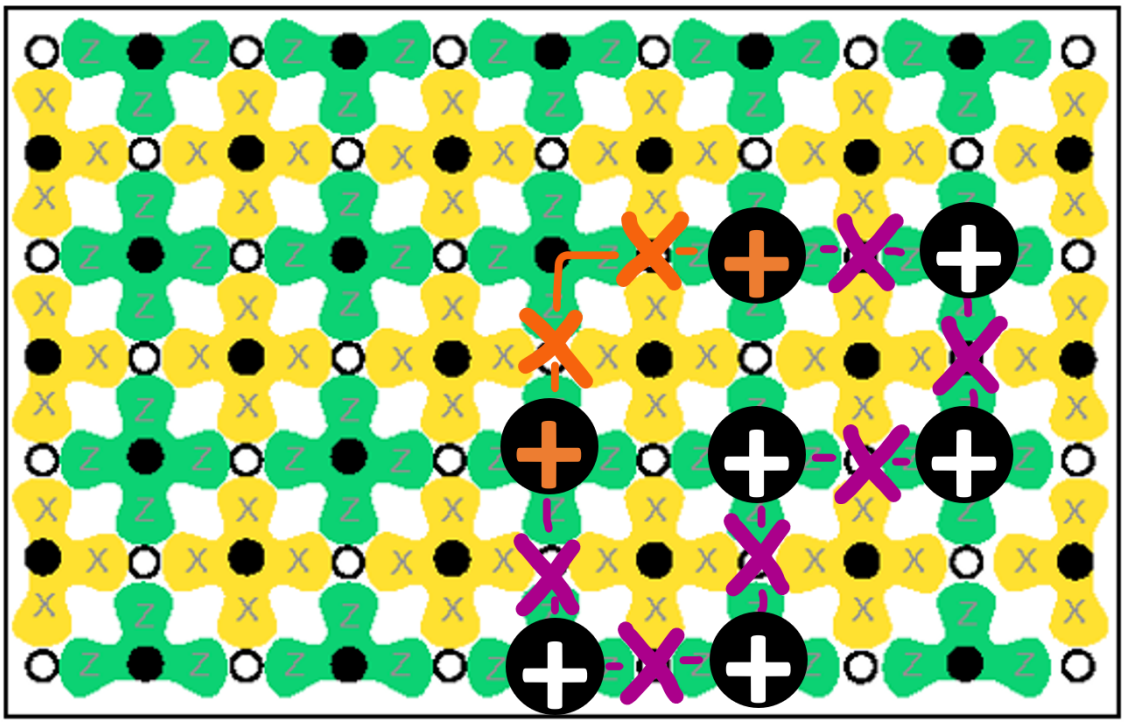


Figure 2.44: In any unknown chain of errors that happened, closing an internal loop will correct the state and will return the syndrome to be corrected

Logical Error

In the previous section 2.4, we saw that error correction detects errors and vanishes them. But still, there will be errors in case of too many errors, the error correction procedure detects wrong and even makes more errors than before. When such a thing happens, we call it a Logical Error.

As we saw, some classical process is running, examining the resulting syndrome, and guessing the errors that happened. The assumed path is corrected as long as it closes an

internal loop (as explained in subsection 2.4). But not any path that will be chosen be successful. Let's assume that the decoder saw the following Z syndrome:

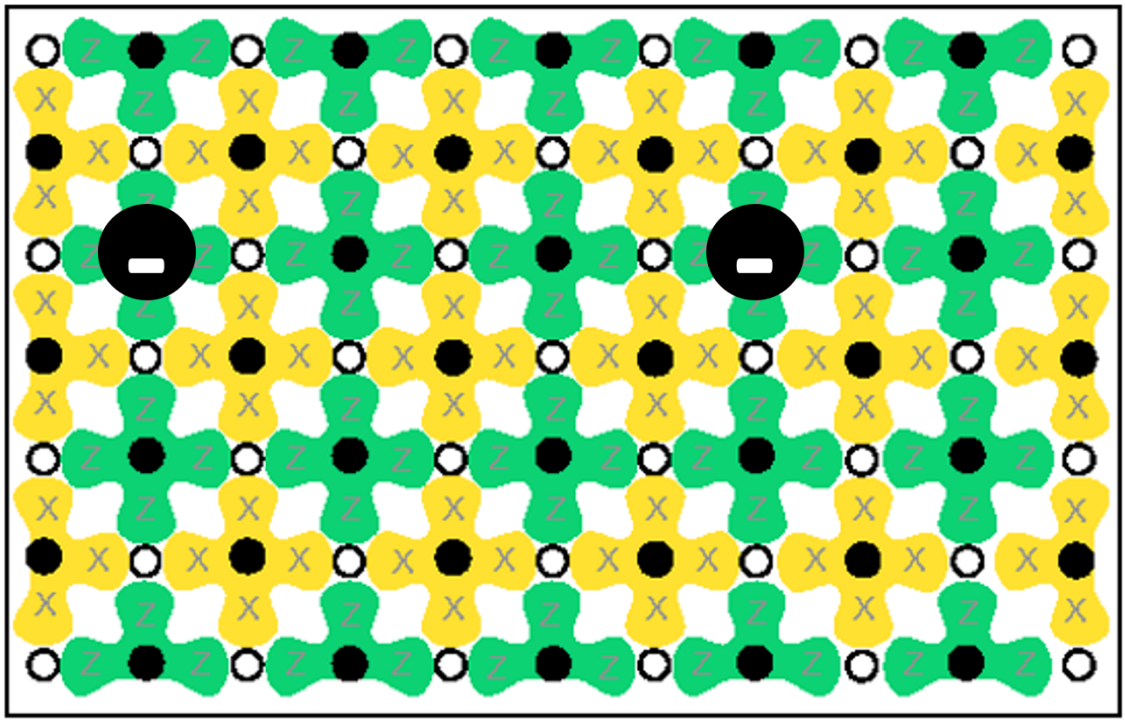


Figure 2.45: A Z syndrome that indicates that X error happened somewhere

Now it has to guess which of the following errors caused this syndrome and correct it accordingly:

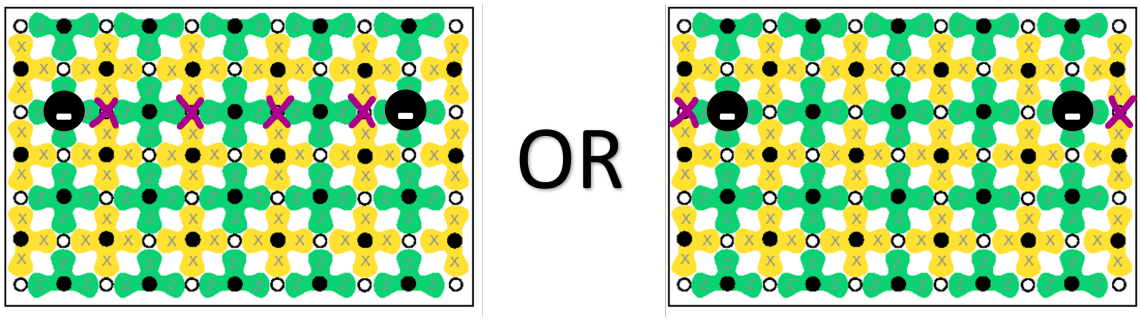


Figure 2.46: Two options that could cause the Z syndrome from 2.45 - A chain that connects the error indicators, or two different chains of X flips

Since errors tend not to happen (a chance of less than 0.5), it is much more probable that the case of two flips happened with a chance of P_X^2, which is bigger than the four X

flips, which have a chance of P_X^4.

If it guesses right, everything is OK; the error is corrected. If it guesses wrong, the new state will be a logical state with a chain of X applied to it:

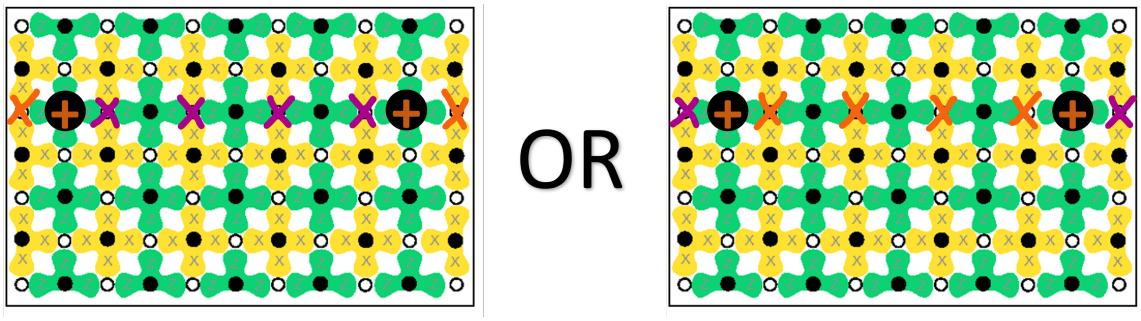


Figure 2.47: In the case of the wrong guess, the correction will cause a chain of X flips from left to right, which will flip the logical state

Differently from trivial errors, internal loops, this string is like an external loop going around the surface. It is bad because it **changes the oddity of the state from even to odd or vice versa**. It adds a string of $|1\rangle$s and, therefore, can flip $|0_L\rangle$ to $|1_L\rangle$ or vice versa. It is because it operates with a logical X operator - a chain of X's from left to right.

2.5 Coherent Errors in Surface Codes

In this section, we will apply our knowledge of coherent errors (described in subsection 2.2) and Surface Code (described in section 2.4), to understand their behavior of them together.

Let us assume a surface code in an ideal state of surface code, without errors, a logical zero $|0_L\rangle$ which is a superposition of an even amount of all possible strings from left to right, and as explained in section 2.4 and shown graphically at Fig. 2.37. So at this point the state:

$$|\psi\rangle = |0_L\rangle \tag{2.24}$$

Now, let us assume that a small coherent error happened only to the first qubit. This error splits the state $|\psi\rangle$ into a superposition of Pauli operations. As explained in 2.1, and assuming this rotation is small angle θ, the state will split into a big amplitude of I operator, and small amplitude $\varepsilon_X^1, \varepsilon_Y^1, \varepsilon_Z^1$ of X, Y, Z accordingly.

$$|\psi\rangle_1 = (a_1 I_1 |0_L\rangle + \varepsilon_1^X X_1 |0_L\rangle + \varepsilon_1^Y Y_1 |0_L\rangle + \varepsilon_1^Z Z_1 |0_L\rangle) \tag{2.25}$$

Since errors are operating on all the qubits, checking the effect of the error of the next qubit, the new state before any measurement will be:

$$\begin{aligned}
|\psi\rangle_2 = & \\
& (a_2 I_2 |\psi\rangle_1 + \varepsilon_2^X X_2 |\psi\rangle_1 + \varepsilon_2^Y Y_2 |\psi\rangle_1 + \varepsilon_2^Z Z_2 |\psi\rangle_1) = \\
& (a_2 I_2 (a_1 I_1 |0_L\rangle + \varepsilon_1^X X_1 |0_L\rangle + \varepsilon_1^Y Y_1 |0_L\rangle + \varepsilon_1^Z Z_1 |0_L\rangle) + \\
& \varepsilon_2^X X_2 (a_1 I_1 |0_L\rangle + \varepsilon_1^X X_1 |0_L\rangle + \varepsilon_1^Y Y_1 |0_L\rangle + \varepsilon_1^Z Z_1 |0_L\rangle) + \\
& \varepsilon_2^Y Y_2 (a_1 I_1 |0_L\rangle + \varepsilon_1^X X_1 |0_L\rangle + \varepsilon_1^Y Y_1 |0_L\rangle + \varepsilon_1^Z Z_1 |0_L\rangle) + \\
& \varepsilon_2^Z Z_2 (a_1 I_1 |0_L\rangle + \varepsilon_1^X X_1 |0_L\rangle + \varepsilon_1^Y Y_1 |0_L\rangle + \varepsilon_1^Z Z_1 |0_L\rangle))
\end{aligned} \tag{2.26}$$

Graphically, we can look at the exponential growth of the terms as shown in Fig. 2.48.

And so on, we get a superposition of all 4^N (where N is the number of qubits in the surface) possible error combinations, with different amplitudes that depend on the ε.

Once stabilizers are measured, the state collapses to one of the combinations, in a probability that depends on the εs. Calculating this probability (which is critical for calculating the resulting logical error) demands a very complicated calculation since we have to sum up equivalent terms, as shortly explained in subsection 2.3. You can also think of two terms of two different trivial errors, which result exactly in the same state.

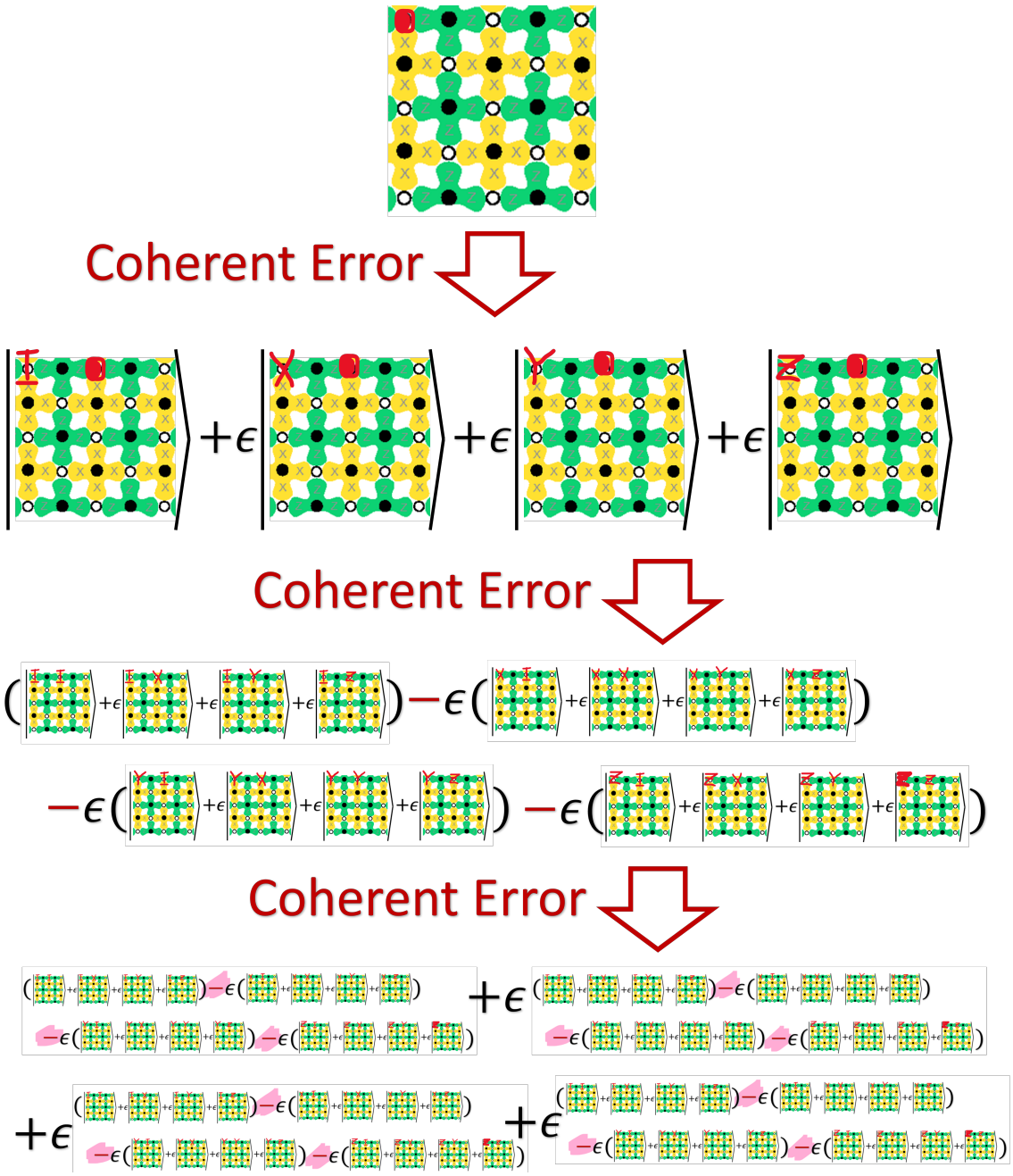


Figure 2.48: Starting from a quiescent state which is any no-errors state, this state splits into 4 for every qubit with a coherent error in the surface

3 Motivation to Simulation Surface Code with Tensor Networks

So far, we have discussed the theory of quantum errors, quantum error correction, and the surface code approach. To measure the resulting logical error as a function of the surface properties and the error model, we need to simulate it somehow.

The surface code contains $O(d^2)$ physical qubits (data and ancillary qubits), which demand naively an exponential amount of classical information to be simulated. This behavior limits us naively to simulate up to $d = 5$ surface code and does not let us examine the behavior for small logical errors that are needed for practical applications (their d is much bigger than $d = 5$) for arbitrary error models.

When errors are not coherent errors, some methods as we can find in Ref. [29, 30] suggest ways to optimize the simulator running time drastically.

Since our errors are coherent, a naive but accurate approach is to analytically write the formulas that represent the full state after the error. Its drawback is that when an arbitrary error happens to the state, it is "splitting" the world - creating many terms in the superposition. Some of them might cancel each other, and it will be hard to calculate those destructive superpositions in surfaces bigger than $d = 4$ [10, 12, 15].

Another approach is converting the surface to Majorana particles, which allows us to simulate very large surfaces due to a reduction polynomial and not exponential time. The drawback of this approach is that it limits us to a specific type of error model and not an arbitrary error model - only Z or only X rotations error [11, 13].

An approach that lets a sweet spot trade-off is a simulation of surface code on tensor networks [2]. The complexity of this approach is $O(2^d)$ instead of the naive $O(d * 2^{d^2})$. The second benefit is that we can simulate any arbitrary error model. This approach will be described in detail in the next chapter.

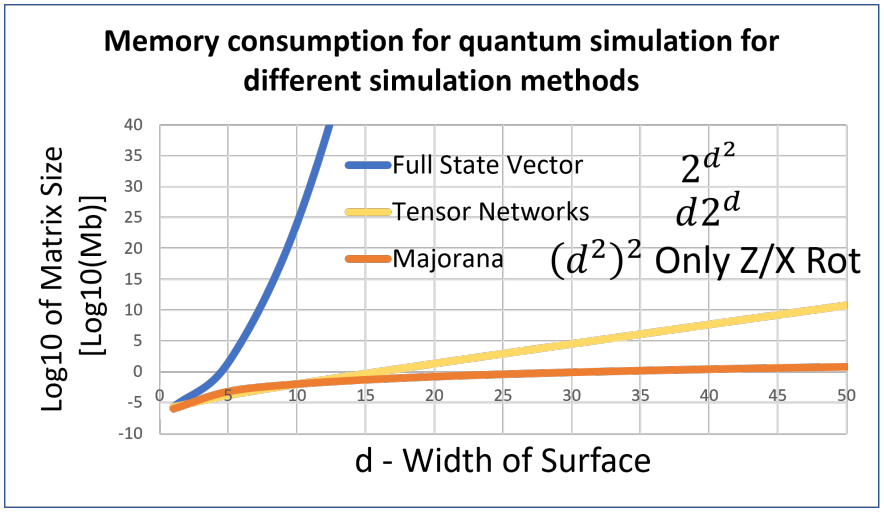


Figure 3.1: A comparison of the full state vector, Majorana, and tensor networks simulation approaches to simulate QEC. The complexity is measured as the amount of memory needed to store the simulation information. Measured as a function of the surface width

4 Simulation

Generally speaking, the goal of the simulation is to find the logical error as a function of the surface code size in different models of coherent error. The idea is to take the simulation developed by Ref. [2], make some changes to get results faster, and get the logical errors at a few small widths of the surface. Then, since we want to know the sizes needed for even more minor logical errors, we extrapolate those results to see the relation between them on larger surfaces.

4.1 Surface Code Architecture and the Conversion to Tensor Networks

The architecture here is based on the circuit model rotated surface code, where qubits are arranged in a 2D array, using the simulation described in Ref.[2]. The use of tensor network to simulate quantum systems is very wide and can be found also for example in Ref. [31–33]. The conversion of the surface code to tensor networks is based on the following stages, which are explained in detail at Ref. [2]:

1. $d * d$ qubits are represented each one with $2 * 1$ vector, and are arranged in a 2D lattice

2. Each of them is initialized to $|0\rangle$ and then stabilized to the quiescent state of surface code using activation of all the XXXX stabilizers. The XXXX stabilizers are represented as a U shape network projecting the four qubits into a +1 eigenstate of XXXX. Now the network is represented with d^2 edges, which describe a quantum state in a Hilbert space of d^2 qubits.

3. Then, the whole state is converted into a density matrix ρ_0 to enable it to operate with the noisy channel in the next state. This is done by creating a dagger copy of the networks, which makes a new network with $2d^2$ open edges corresponding to this density matrix. This is graphically shown in the following figure 4.1:

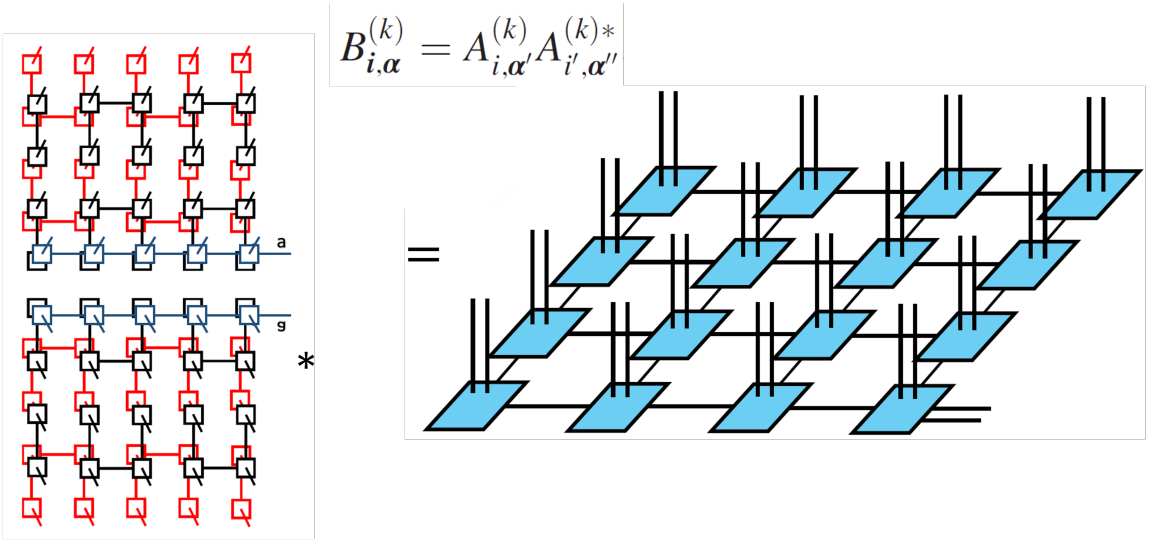


Figure 4.1: Adding a dagger copy of the state network, makes a new network with $2d^2$ open edges corresponding to its density matrix

4. At this point, we are operating on each site with a tensor E that describes the coherent error. This is done by a tensor which corresponds to this unitary operation. $\rho_{noisy} = E\rho_0 E^\dagger = E^\dagger E\rho_0$. Since we finish by tracing anyway, the trace is cyclic, and therefore, we multiply by $E^\dagger E$. This is graphically shown in the following figure 4.2:

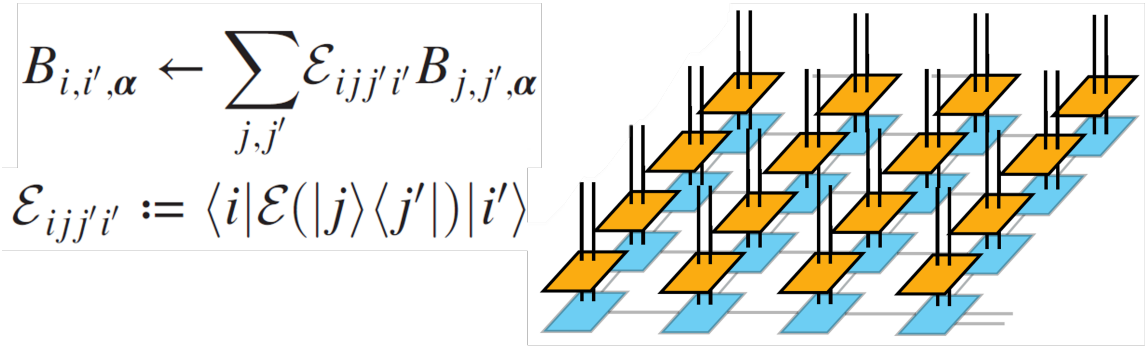


Figure 4.2: A noise is added using multiplication in a tensor that represents the noise

5. Next, we need to extract the syndrome of the stabilizers measurements, which is a probabilistic outcome. So we need to find each probability p_k to get m_k bit syndrome results for the k'th stabilizers. This is done by performing the k'th stabilizer again to the network, and a trace of it will give us a scalar which is the p_k we are looking for. This process is demonstrated in figure 4.3:

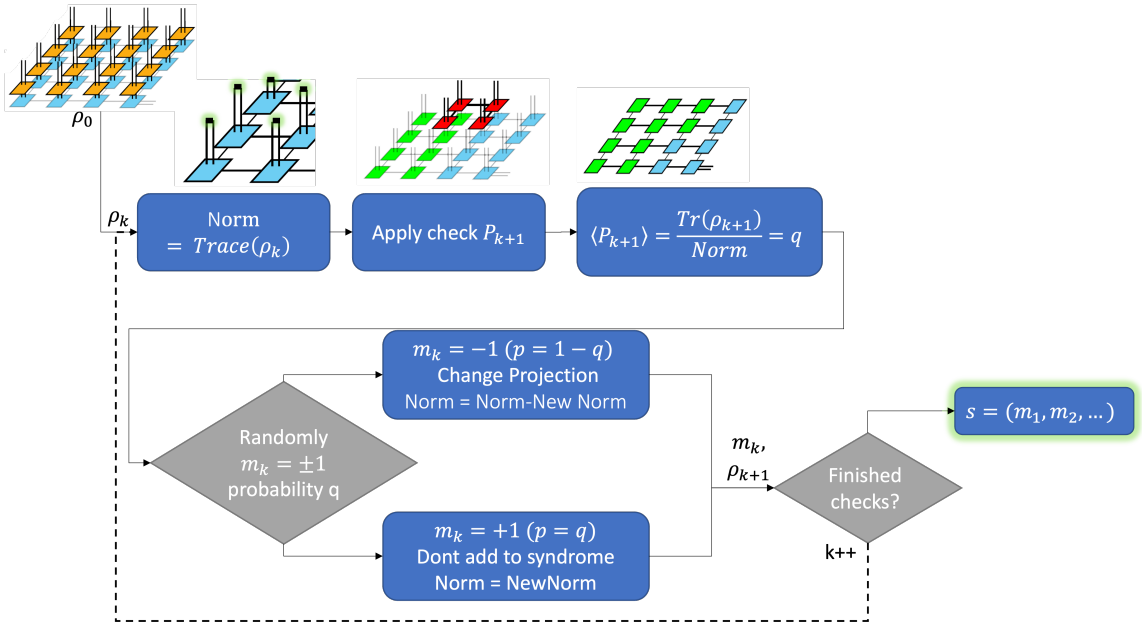


Figure 4.3: For every stabilizer check, we measure the probability to get the stabilizer result, using the trace of the density matrix which is the density matrix after applying all previous stabilizers, normalized by the trace of the density matrix before stabilizing. Collecting all the results will give the syndrome for the current stabilizers round.

6. Now, when we have the probabilities for all the stabilizers, we are non-deterministically getting some syndrome resulting from the possibilities.

7. A classical decoding algorithm examines the syndrome and performs error corrections accordingly.

8. To extract the logical error from the final state, use the diamond norm of the state relative to the reference of the initial state. This is done based on the Choi matrix of the state, which describes how much logical X/Y/Z happened to the state using tracing tricks, which by turn converted to a density matrix.

Since the process is probabilistic, one cycle of the stages is not enough to calculate the logical error, but an averaging has to be done on $O(1/P_L)$ samples.

4.2 Error Model - Linear Optic Coupler Gate

Two different error models were considered in this work. The first, described in this section, examines the effect of fabrication errors in silicon photonic waveguides, against the improved segmented composite approach suggested by Ref. [3, 16]. When designing a linear optical gate using couplers, as detailed in Ref. [34, 35], the achieved gate is slightly

different from the wanted gate in a manner of a slight unitary rotation around the Bloch sphere which is actually a coherent error. An approach to reduce this systematic error is to design the coupler in an architecture that makes it more robust to the fabrication error. This is done by dividing the coupler into composite segments, which gives more freedom to choose geometry parameters that reduce the sensitivity of the resulted gate to geometry changes [3, 16].

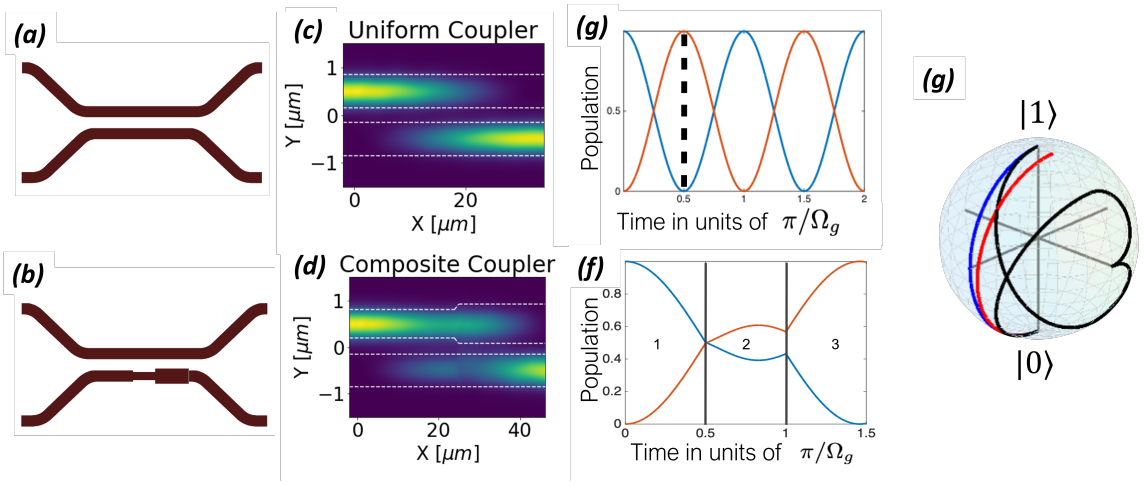


Figure 4.4: **(a-f)** Uniform coupler (top) vs segmented composite coupler (bottom) **(a,b)** Coupler geometry design, view from top **(c,d)** Electric field intensity heat map inside the couplers **(e,f)** Population of $|0\rangle$ and $|1\rangle$ state has smaller slopes in the composite scheme. Adapted from [3]. **(g)** Bloch sphere path of X gate ideal (blue), using uniform coupler results in bigger error (red) than segmented composite (black).

The performance of the improved approach can be shown by the distribution of the angle of the rotation error it is creating.

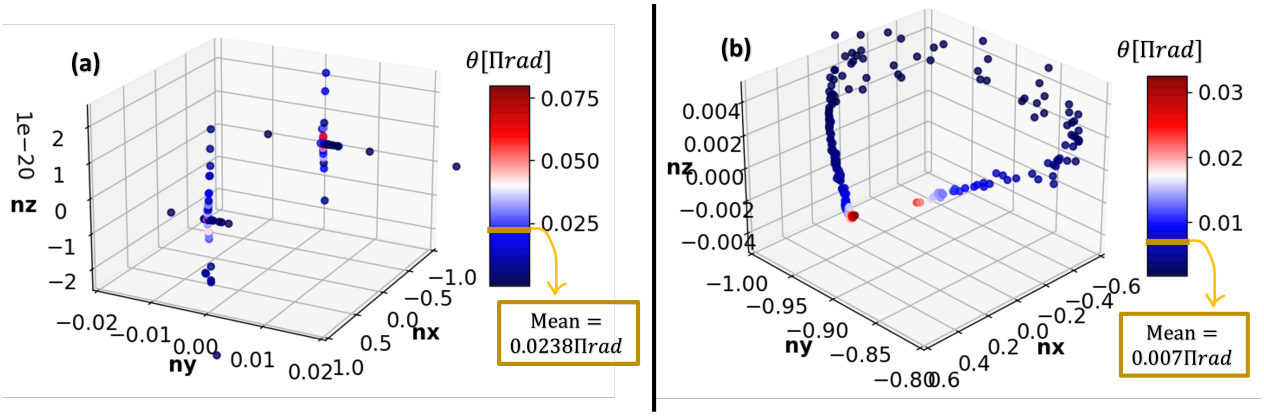


Figure 4.5: Distribution of the error in **(a)** a uniform design and **(b)** a composite design . nx,ny,nz is the axis of rotation of the error, and θ is the angle of rotation. We can see that uniform design has an error angle mean value of $23.8m\Pi Rad$, and composite design has an error angle mean value of $7m\Pi Rad$, more than three times smaller.

Another metric to measure how much the composite approach is better than the uniform one is to measure the number of physical qubits needed per one logical qubit in surface code for some fixed destination of logical error (or for a fixed size of surface code, to measure the improvement of the logical error).

The way to measure the logical error is by using the tensor network simulation. The way we insert this physical error is made the following way:

1. An optimization algorithm from Ref. [3] is finding the segments geometry parameters with the best average fidelity for an X gate, given a noise model which is assuming a Gaussian distribution of the width of the waveguide, with a standard deviation of $15nm$, and a complete correlation between the pair of the waveguides.

2. Those constant ideal parameters are used by a probabilistic function, which returns a probabilistic non-ideal X gate, which depends on the chosen parameters and on the width noise model.

3. The coherent error E of the X gate is extracted using Eq. 2.14. This coherent error is a unitary operation that is equivalent to a small rotation on the Bloch sphere, nearly an identity gate.

4. Stages 2-3 are repeated separately to each of the qubits on the surface, which creates a distribution of coherent errors on all surface.

5. Each error E is converted to a tensor that is equivalent to a quantum channel that the density matrix goes through. The tensor is $E^{\dagger} \bigotimes E$. This tensor is multiplied in its

corresponding location in the tensor network $\rho_{noisy} = E\rho_0 E^\dagger = E^\dagger E\rho_0$.

The rest of the simulation is continuing as explained in section 4.1.

4.3 Error Model of constant Rotation with Inter Sign Flips

Another error model we observed was chosen when we noticed the fact the coherent error sum up as explained in section2.5, they do not necessarily worse than their non-coherent equivalent. **They might also be better if the superposition of the terms that take the state out of the code space, cancel each other in a destructive interference.**

We saw that the literature about the effect of coherent error in quantum error correction, chose a constant rotation to all the qubits, and did not examine the effect of different errors between the qubits [10–15].

Ref. [36] suggested slicing stabilizers into two equally weighted Pauli operators and then applying them by rotating in opposite directions, causing their overrotations to interfere destructively on the logical subspace. Refs. [36] and [37] suggested changing the signs of the stabilizer generators to change how the coherent errors interfere, leading to a QEC improvement.

None of them examined what happens when the sign of the error is different for the error on the data qubits themself or in surface code.

As was explained in subsection 2.3, the amplitude of coherent error that operated on different qubits, sum up, and therefore with the same sign of summing will increase the error. But if the summing up had different sign, it was actually reducing the total error and not increasing it, since it is destructively cancels terms that are taking the state out of the code space.

We can see it through a graphical example. As explained in subsection 2.4, a state without any coherent error, can look like a superposition of strings from left to right. When a coherent error happens, it splits the terms as it is composed of the sum of different Pauli operators.

Let's see a simple example where two error terms cancel each other.

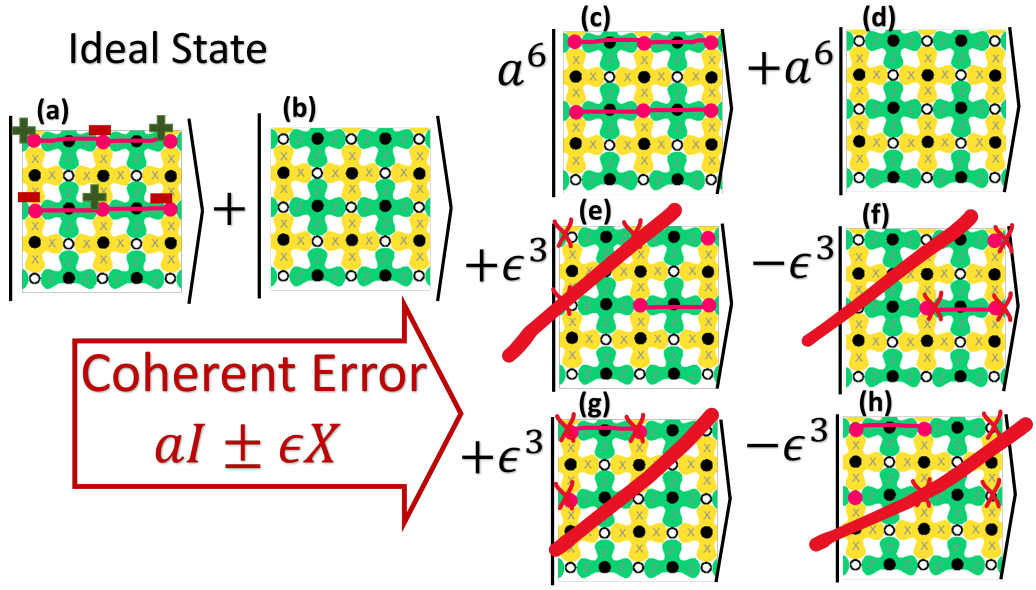


Figure 4.6: Examining superposition of only two non-error terms: Two strings of $|1\rangle$ from left to right (a), all $|0\rangle$ (b). After applying this error model, error terms are created from both terms. (e) is created from (a) with a plus sign, and (f) is created from (b) with a minus sign with a minus amplitude. The same happens for (g) and (h). This behavior results in a cancellation of those two error terms.

Therefore, we tested an error model where the same rotation angle was taken to all the qubits, but with antifermionic-like flips of the sign of rotation between nearest neighbors.

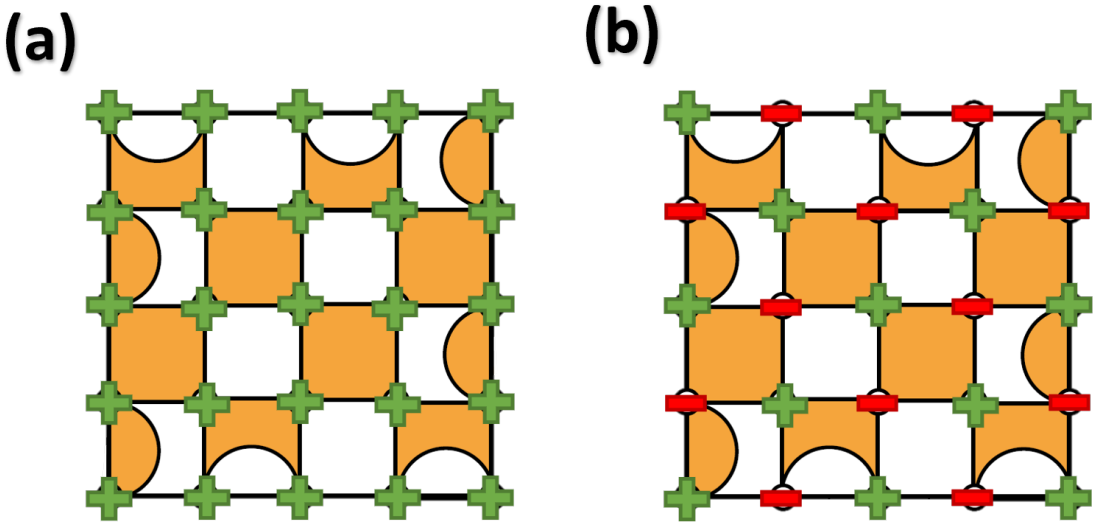


Figure 4.7: **(a)** A simple error model where all data qubits are rotated in the same axis, sign, and angle size. **(b)** Same angle for all data qubits, but a different sign of angle between nearest qubits

4.4 Accelerating Time of Computation using Convergence Test

As explained in section 4.1, since one run of the initialize-noise-stabilize round is probabilistic, the output diamond norm is a random variable. It can have error rates that depend on the size of the surface, and on the noise parameters, extracting the logical error from the diamond norm of the simulation demands averaging of the logical error on the number of samples as $O(1/p_{error})$ which will satisfy a steady average that describes the real logical error.

Since we are just looking for the logical error up to one order of magnitude error from the resulting one, we can shorten the number of runs, and reduce time.

The chosen distribution for the diamond norm over the samples is exponential distribution, which is suitable for processes with a constant average rate, which is the situation in the physical meaning of accidental errors on the surface. We can impress that the diamond norm's histogram looks like Poisson distribution and fits.

We can use the confidence interval technique to examine the data of the diamond norm. We stop the simulation when we know that in a sufficient probability, the logical error, is probably close enough to the mean of the samples so far.

We choose the maximum samples per run $N_{samples}$ by approximation of the resulted logical error P_L by $N_{samples} = O(1/P_L)$. For more than a few days, we try to run with fewer samples, in order to complete the simulation. After every $0.1 * N_{samples}$ we stop and check if the mean of the diamond norms so far converged. It is done by:

```
# Examining the 30% last tail of the means and std vector
diamond_lengths = [int(samples_so_far * n / 100) - 1 for n in range(70, 101)]
L = len(diamond_lengths)
# Vector of the mean from the start the the diamond_lengths[i] sample
diamond_means = [mean(diamond_norms[:L) for L in diamond_lengths]
# Vector of Standard Deviations
diamond_stds = [mean(diamond_norms[:L) for L in diamond_lengths]
# Vector of Standard Errors = STD/sqrt(N)
diamond_stes = [diamond_stds[i] / sqrt(diamond_lengths[i])
    for i in range(L)]
# Upper and lower limit in the margin of error of 100%-99.9% = 0.1%
# in Gaussian distribution is 3.29*STE above and below the mean
upper_limit_of_mean = [diamond_means[i] + 3.29 * diamond_stes[i]
```

```
    for i in range(L)]
lower_limit_of_mean = [diamond_means[i] - 3.29 * diamond_stes[i]
    for i in range(L)]
# Cutting the lower limit to only positive values possible
lower_limit_of_mean = [max([1E-30, x]) for x in lower_limit_of_mean]
# The error in terms of order of magnitude is determined by
# the higher ratio between the upper/mean to mean/lower
orders_of_error = [log10(max([upper_limit_of_mean[i] / diamond_means[i],
    diamond_means[i] / lower_limit_of_mean[i]])) for i in range(L)]
# When mean converges, the ratio between the margin of error to the mean
# is smaller than 10^0.2. For a long tail that satisfies it, we can be sure
# that larger future of diamond norms, will not change the current
# mean that much.
if all([o < 0.2 for o in orders_of_error]):
    # We can break and finish the simulation
    break
```

When looking at the diamond norms mean as a function of the number of samples, we can notice this convergence.

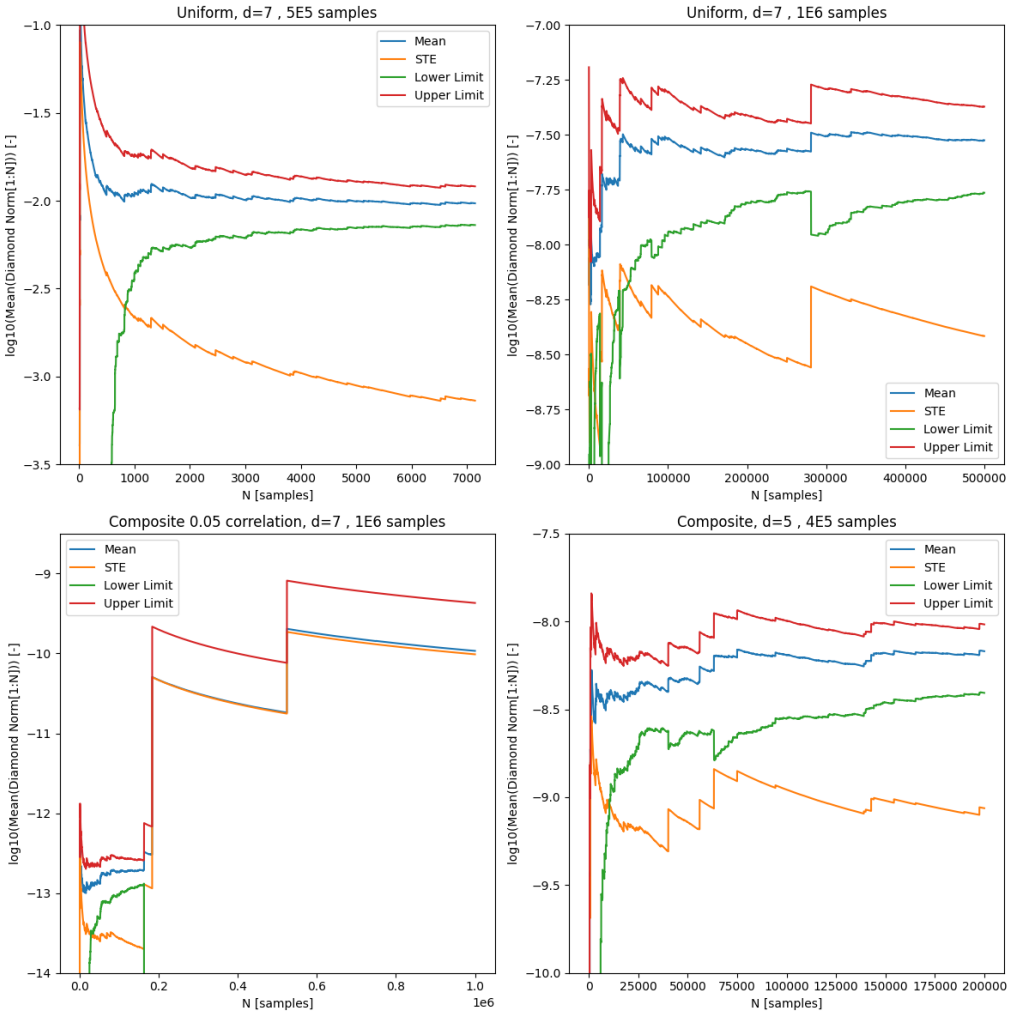


Figure 4.8: Convergence of the mean of the diamond norm as a function of the number of samples. Notice that in $N = O(1/P_L)$ - mean converge to an asymptotic line happen, STE is becoming below $STE < MEAN/10$, and upper and lower limits are tighter around the mean. Also, notice the case of insufficient samples, those conditions do not apply. Once the log10 of the difference between the tail of limits and the mean is smaller than 0.2, we know that the limit is tight enough and we can stop the simulation while we know that the error is in 0.999 probability not worse than a ratio of $10^{0.2} = 1.5$

5 Results

5.1 The Big Picture

Parameters of the Simulation

We run the simulation and examine the logical error as a function of -

1. The noise model. Whether it is a composite coupler or a uniform coupler.

2. The size of the surface. We run on 3x3, 5x5, and 7x7 sizes, which are possible to simulate in a few days. The reason for only odd widths is because of the tensor network structure, which should be modified for even widths. Widths higher than 7 are possible but take a long time, which also depends on the error's size as explained in Fig. 4.8.

3. Geometry parameter as it is expected in practical fabrication today. The parameter σ_W is the standard deviation of the width of the waveguide. We run also $2 * \sigma_W$ and $0.5 * \sigma_W$.

4. The correlation factor ρ between the widths of the different qubits.

The coherent error for each qubit is random, and we are running different errors between different qubits on the same surface, but with the same error for every qubit on all the samples.

Than, for every working point, we run a few times and examine the mean of the logical error in the same working point. It is expected to have a small deviation on bigger surfaces since there are more different qubits on the same surface.

Extrapolation of the Data

The logical errors for every working point are collected. The mean of it is considered the final result of the logical error in this working point.

Since we are running on small surfaces, and want to know the error for bigger surfaces, we extrapolate the data. The extrapolation is based on models that describe the logical error as exponential in the surface width, and, therefore, linear in the log of it [38].

Practically, the fit is done using the two parameters of exponential fit:

$$P_L = A\exp(-Bd) \tag{5.1}$$

Where d is the width of the surface and A, B are parameters that depend only on the noise model.

Threshold Theory

We use a model that is used to make sure our data fits the threshold theory as explained in Ref.[1, 39–41]. In this model, the threshold parameter is only a function of the encoding, and will not change for different d and physical error rate $p_{physical}$. Since here $p_{physical}$ does not exist, we examine an approximate equivalent of it. Returning to the model explained in subsection 2.3, we need an angle θ of rotation that describes the error. The rotation angle is extracted using Eq. 2.9. Since θ is a random variable, we take the mean of its absolute value:

$$\theta_0 = MEAN(|\theta|) \tag{5.2}$$

The equivalent $p_{physical}$ using subsection 2.3 is:

$$p_{physical} = \sin^2(\theta_0) \tag{5.3}$$

Then, we compare the thresholds between the composite and the uniform using the following:

$$P_L \propto \exp(-Bd) \propto (\frac{p_{physical}}{p_{th}})^d \tag{5.4}$$

Therefore, extracting the threshold from two different physical errors should be the same, and does not depend on d and $p_{physical}$

$$p_{th} \propto \frac{p_{physical}}{(P_L)^{1/d}} \tag{5.5}$$

Or, between two physical errors with the same d,

$$\frac{P_L^1}{P_L^2} = (\frac{P_{physical}^1}{P_{physical}^2})^d \tag{5.6}$$

5.2 Results of Uniform vs Composite

Results for Sigma Width = 16.6*nm*

Here are the results of the logical error when taking the noise model as explained in 4.2 with a normal fabrication error of :

$$\sigma_0^W = 16.6nm \tag{5.7}$$

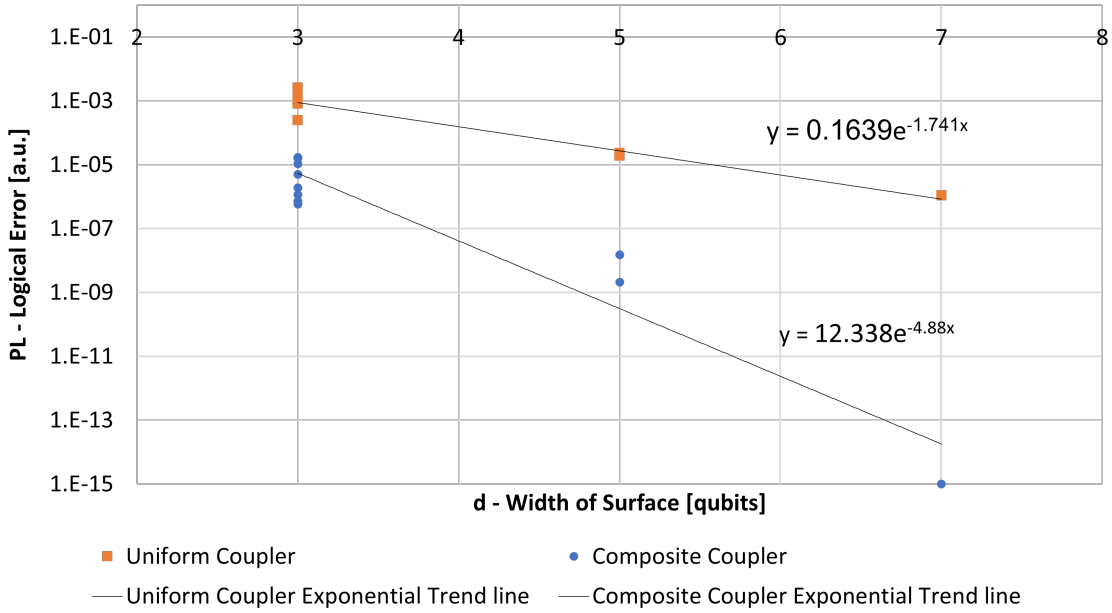


Figure 5.1: The Logical Error vs the width of the simulated surface width $d = 3,5,7$ qubits. Notice that the decay rate of the composite is bigger than the decay rate of the uniform design. It means that we can reach smaller logical errors with smaller d. Fabrication error is $\sigma_0^W = 16.6nm$.

When extrapolating using the approach form subsection 5.1 the linear decrease of the log of the logical error, we see the following results:

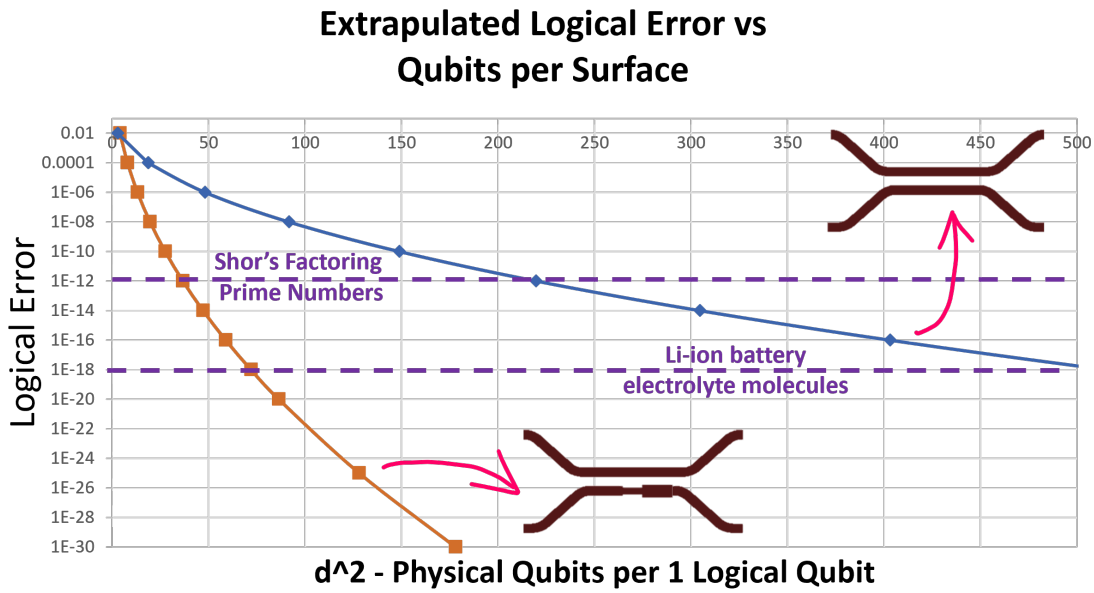


Figure 5.2: The logarithmic extrapolated logical error vs physical qubits amount in 1 logical qubit d^2, that are bigger than the simulated d. Composite design **(orange)** vs uniform design **(blue)**. For the target logical error of Shor's Factoring algorithm ($P_L = 1E-12$) and Li-Ion battery electrolyte molecules simulation ($P_L = 1E-18$), we reduce the amount of qubit in a factor of 5.9-6.4. Fabrication error is $\sigma_0^W = 16.6nm$.

Bigger or Smaller Sigma Width

To see the results in a wider parameter space of the fabrication error, we examined the results also for $\sigma^W = 2\sigma_0^W$ and for $\sigma^W = 0.5\sigma_0^W$:

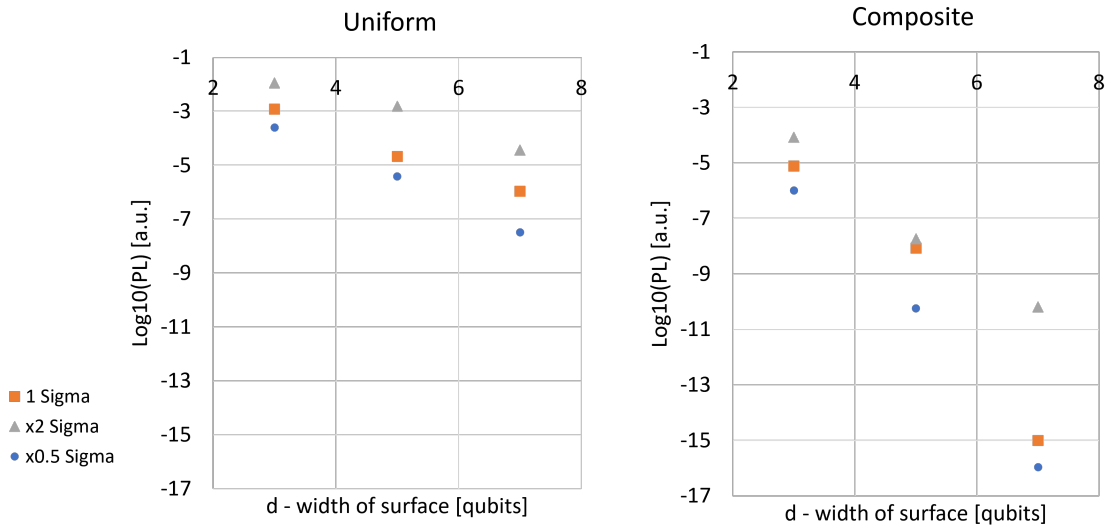


Figure 5.3: The $log_{10}(LogicalError)$ vs the width of the simulated surface width $d = 3,5,7$. Still, for all fabrication errors, the slope of composite **(right)** is bigger than the slope of the uniform design **(left)**. Fabrication errors are $1\sigma_0^W = 16.6nm$ **(orange)**, $2\sigma_0^W = 33.2nm$ **(gray)**, $0.5\sigma_0^W = 8.3nm$ **(blue)**.

And with extrapolation to bigger surfaces:

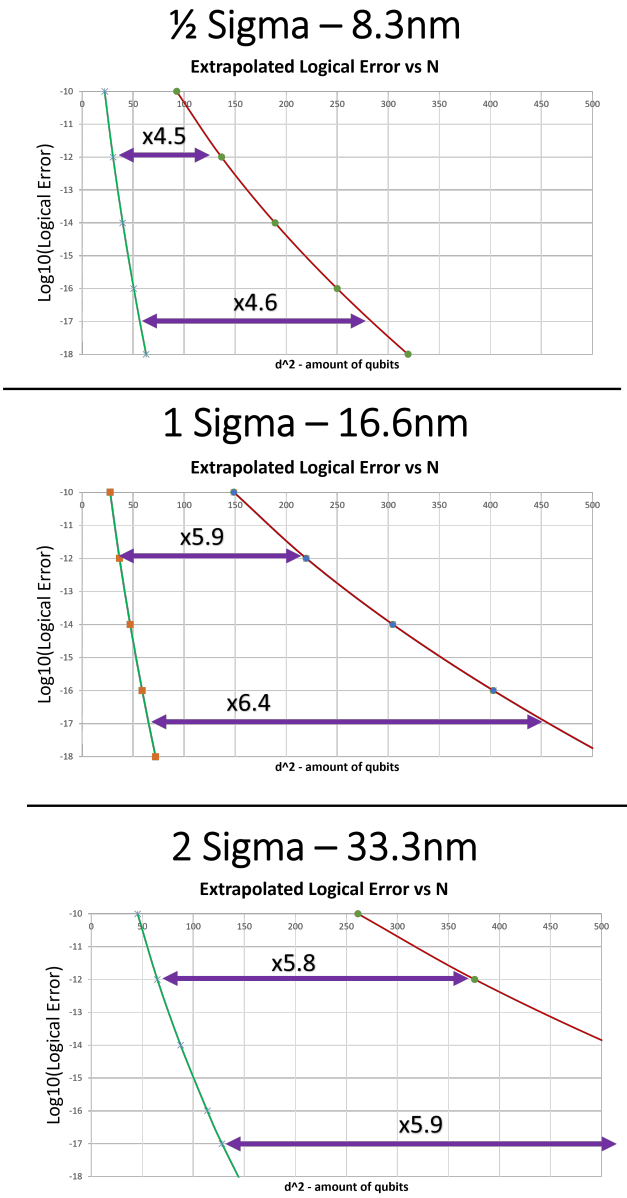


Figure 5.4: The logarithmic extrapolated logical error vs physical qubits amount in 1 logical qubit d^2, that are bigger than the simulated d. Composite design **(green)** vs uniform design **(red)**. For the target logical error of Shor's Factoring algorithm ($P_L = 1E-12$) and Li-Ion battery electrolyte molecules simulation ($P_L = 1E-17$), we reduce the amount of qubit in a factor of 4.5-6.4. Fabrication errors are $1\sigma_0^W = 16.6nm$ **(middle)**, $2\sigma_0^W = 33.2nm$ **(bottom)**, $0.5\sigma_0^W = 8.3nm$ **(top)**.

5.3 Results of constant Rotation with Inter Sign Flip

Here are the results of the logical error when taking the noise model as explained in section 4.3 for a Z rotation angle of $\theta = \pm 0.1\pi$:

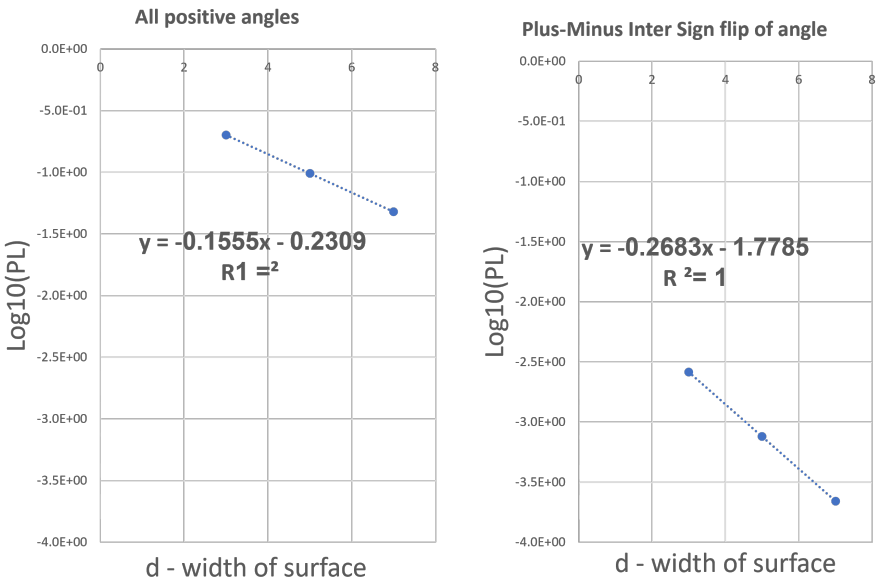


Figure 5.5: The $log_{10}(LogicalError)$ vs the width of the simulated surface width $d = 3,5,7$. Notice that the slope of inter-flip case **(right)** is twice bigger than the slope of the case of the all positive angle **(left)**. It means that we can reach smaller logical errors with smaller d. Taken with a Z rotation angle of $\theta = \pm 0.1\pi$.

When extrapolating the results from subsection 5.1 the linear decrease of the log of the logical error, we see the following results:

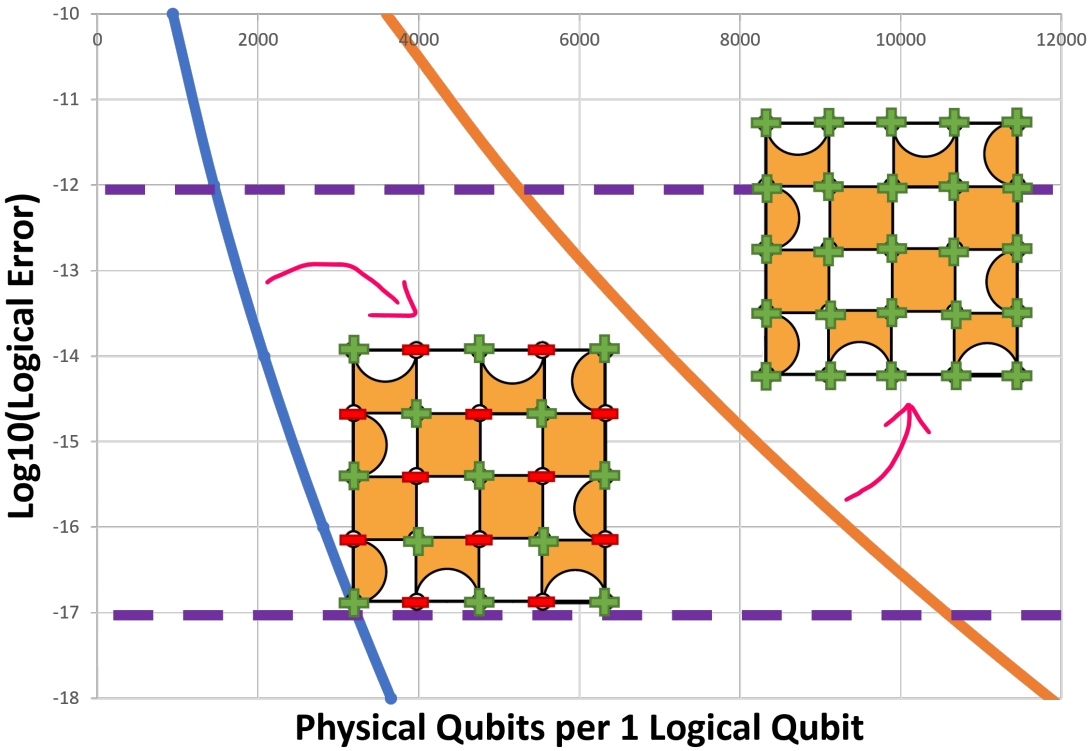


Figure 5.6: The logarithmic extrapolated logical error vs physical qubits amount in 1 logical qubit d^2, that are bigger than the simulated d. All positive angles case **(orange)** vs inter-sign flip case **(blue)**. For the target logical error of Shor's Factoring algorithm ($P_L = 1E-12$) and Li-Ion battery electrolyte molecules simulation ($P_L = 1E-18$), we reduce the amount of qubit in a factor of 3.5. Taken with a Z rotation angle of $\theta = \pm 0.1\pi$.

6 Fusion of Small Clusters using Plus-Minus Technique

6.1 Background for Fusion of Clusters

This section will cover the background needed for the theory behind quantum computation based on cluster states and fusion gates. Using this background, we will analyze the effect of coherent errors on such systems.

Measurement Based Quantum Computing (MBQC)

One of the promising methods to implement a quantum computer is MBQC. In this approach, computing is implemented in two main steps.

The first one is to create a massive cluster of qubits that is not dependent on the algorithm up to the fact it has to be big enough. The cluster is actually a network of entangled qubits.

The creation of this big cluster is done using a fusion of small clusters together. This fusion is done by common measurement of two qubits from two different clusters, which in some probability implements two qubits gate that entangle qubits from different small clusters.

The second step is to compile operators from the algorithm into measurement, which vanish some of the qubits by measurement and transfer the quantum information into the entangled qubits that were not measured yet.

For further reading, please follow Ref. [35, 42–47].

Fusion Based Quantum Computing (FBQC)

A new approach that MBQC inspires is Fusion Based Quantum Computing (FBQC) which performs the fusion of the small clusters while running the algorithm. So instead of building a very big cluster from ahead, it dynamically builds it while the algorithm runs. Further details can be found in Ref. [48–50].

Fusion Gate

As mentioned before, a fusion gate is a probabilistic way to create a two-qubit gate when it is not possible in a deterministic way. This is precisely the case when we talk about linear optics. So in this section, we will focus on one of the possible implementations of a fusion gate - a type-2 fusion gate, which probabilistic makes a ZZ measurement of two qubits. In this section, I follow Ref. [35, 46].

Assuming the following circuit, which is a fusion gate type-2 :

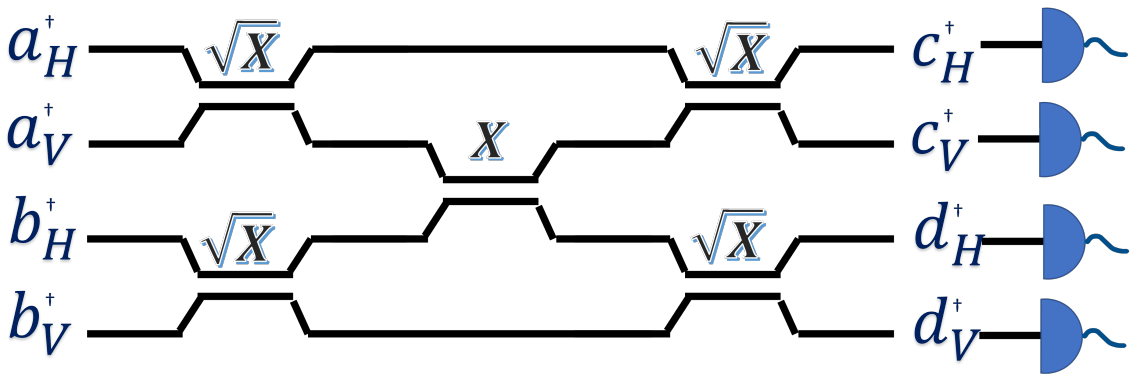


Figure 6.1: The linear optic operation is composed from 5 $\sqrt{X}$ beam splitters. It ends with the detection of all four rails. In some probability, the resulting state will be a ZZ measurement of the two qubits, which in turn will entangle the other qubits from the rest of the cluster.

In this analysis, I use the code in section 8.1. Assuming we start from some general state and focusing the two qubits that go to the fusion gate:

$$|\psi\rangle = (a_H^\dagger * f_1 + a_V^\dagger * f_2) * (b_H^\dagger * f_3 + b_V^\dagger * f_4)\,|vaccum\rangle \tag{6.1}$$

Which is actually:

$$|\psi\rangle = f_1 f_3\,|H,H\rangle + f_1 f_4\,|H,V\rangle + f_2 f_3\,|V,H\rangle + f_2 f_4\,|V,V\rangle \tag{6.2}$$

After they go through the beam splitter, the linear operation of it brought them to the state:

$$\begin{aligned}|\psi\rangle =& 0.25(c_H^{\dagger\,2} * f_2 * f_3 + \\ & c_H^\dagger * c_V^\dagger * (f_1 * f_3 - f_2 * f_4) \\ & - c_H^\dagger * d_H^\dagger * (f_1 * f_3 - f_2 * f_4) \\ & + 2 * c_H^\dagger * d_V^\dagger * f_2 * f_3 \\ & - c_V^{\dagger\,2} * f_1 * f_4 \\ & + 2 * c_V^\dagger * d_H^\dagger * f_1 * f_4 \\ & + c_V^\dagger * d_V^\dagger * f_1 * f_3 \\ & - c_V^\dagger * d_V^\dagger * f_2 * f_4 \\ & - d_H^{\dagger\,2} * f_1 * f_4 \\ & - d_H^\dagger * d_V^\dagger * (f_1 * f_3 - f_2 * f_4) \\ & + d_V^{\dagger\,2} * f_2 * f_3)\,|vaccum\rangle\end{aligned} \tag{6.3}$$

Which enlarges the cluster up to local Hadamard gates. Notice that each term implies a different measurement result at the output. So after measurement, some of the terms will vanish depending on the result. So if we assume that the clicks were 00 / 01 / 10 / 11, the state will collapse to one of the following, depending on the measurement result:

$$\begin{aligned}&00 : 0.5i(-f_1 f_4 + f_2 f_3)\,|vaccum\rangle \\ &01 : 0.5i(-f_1 f_3 - f_2 f_4)\,|vaccum\rangle \\ &10 : 0.5i(f_1 f_3 + f_2 f_4)\,|vaccum\rangle \\ &11 : 0.5i(-f_1 f_4 + f_2 f_3)\,|vaccum\rangle\end{aligned} \tag{6.4}$$

Notice that those options are equivalent to a ZZ measurement with the following:

- Result of +1 that indicated even ZZ measurement (case 01 and 10)
- Result of -1 that indicated odd ZZ measurement (case 00 and 11)

Now let's wonder what happens when the beam splitter is not accurate. When the terms of the superposition are not accurate, some terms do not cancel each other.

In the non-ideal case, the operator is not an ideal ZZ anymore, but it actually collapses the state into a superposition of ZZ=+1 eigenstate and ZZ=-1 eigenstate. For example, in $R_Y(0.1rad)$ rotation to each beam splitter:

$$\begin{aligned}
&00 : (0.001f_1f_3 + f_1f_4(0.025 - 0.499i) + f_2f_3(-0.075 + 0.494i) - 0.004f_2f_4)\,|vaccum\rangle \\
&01 : (-f_1f_3(0.025 + 0.499i) - 0.001f_1f_4 - 0.004f_2f_3 + f_2f_4(0.025 - 0.499i))\,|vaccum\rangle \\
&10 : (f_1f_3(-0.025 + 0.499i) - 0.004f_1f_4 - 0.001f_2f_3 + f_2f_4(0.025 + 0.499i))\,|vaccum\rangle \\
&11 : (-0.004f_1f_3 - f_1f_4(0.025 + 0.499i) + f_2f_3(0.075 + 0.494i) + 0.001f_2f_4)\,|vaccum\rangle
\end{aligned} \tag{6.5}$$

Notice in the negative rotation case $R_Y(-0.1rad)$ rotation to each beam splitter:

$$\begin{aligned}
&00 : (0.001f_1f_3\text{-}f_1f_4(0.025\text{+}0.499i) + f_2f_3(\text{+}0.075 + 0.494i) - 0.004f_2f_4)\,|vaccum\rangle \\
&01 : (\text{+}f_1f_3(0.025\text{-}0.499i) - 0.001f_1f_4 - 0.004f_2f_3\text{-}f_2f_4(0.025\text{+}0.499i))\,|vaccum\rangle \\
&10 : (f_1f_3(0.025 + 0.499i) - 0.004f_1f_4 - 0.001f_2f_3 + f_2f_4(-0.025 + 0.499i))\,|vaccum\rangle \\
&11 : (-0.004f_1f_3 + f_1f_4(0.025 - 0.499i) + f_2f_3(-0.075 + 0.494i) + 0.001f_2f_4)\,|vaccum\rangle
\end{aligned} \tag{6.6}$$

6.2 Effect of Plus-Minus on Fusion of Clusters

In this section, we will analyze the effect of coherent errors on the fusion of clusters, and see how using the plus-minus technique, helps us to reduce errors.

Fusion of Small Clusters using Plus-Minus Technique

Let us focus on the fusion of linear clusters, which also creates a linear cluster. Let us also assume that the fusions were type-2 fusion gates with successful cases (we can always look at the subgroup of the successful fusions).

Graphically we can describe it by:

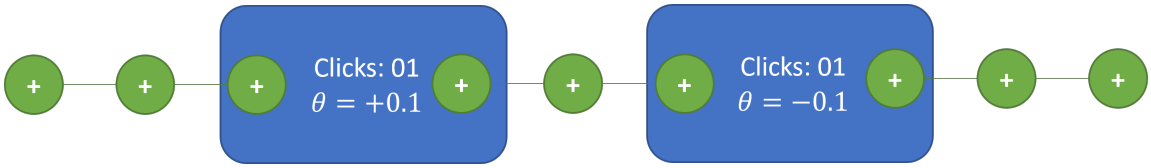


Figure 6.2: The operation is taking a small linear cluster. Each of the Mini-Clusters is described as qubits in $|+\rangle$ state (using Hadamard gate) with CZ between the connected qubits. The fusion gate is making a destructive measurement which depends on the result of the clicks in the detectors and on the error on the beam splitters of the fusion gate.

Ideally, the operator resulting from this fusions chain can be thought of as :

$$U = I^{\otimes 2} \otimes (ZZ) \otimes I^{\otimes 1} \otimes (ZZ) \otimes I^{\otimes 2} \tag{6.7}$$

In the case of the non-ideal fusion gate, the resulting total operator after fusion will be a superposition of terms with ZZ=+1 result and ZZ=-1 result:

$$U = I^{\otimes 2} \otimes (a * ZZ_{-} + b * ZZ_{-}) \otimes I^{\otimes 1} \otimes (a * ZZ_{-} + b * ZZ_{-}) \otimes I^{\otimes 2} \tag{6.8}$$

It might be that with the right choice of signs of the error angle, some equivalent terms will cancel each other. So we can use the Plus-Minus Technique that was mentioned in section 4.3 to make destructive superpositions of unwanted terms. We will assume that the beam splitters of the fusion gates have a coherent error with a positive or negative angle of rotation around some axis.

For example, we can reduce errors in the following case of the fusion of three linear clusters from size 2:

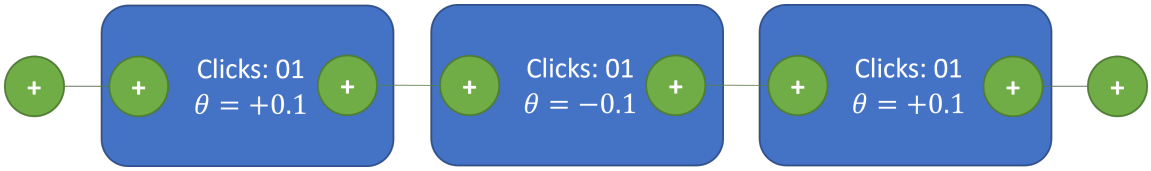


Figure 6.3: Fusion of 3 Clusters from size 2, using fusions with different signs for the error angle

For longer chains, we can look for the +- combination with the best result. In order to do that:

1. We define criteria for the quality of the solution. We would like to know how far is the fusion operator from the ideal operator with zero rotation error. This criterion is by Schmidt Decomposition:

$$D = \sqrt{Tr((U - U_{ideal})^{\dagger}(U - U_{ideal}))} \tag{6.9}$$

2. We iterate on every combination of negative and positive error angles, and extract the one with the smallest D.

The code for this process is attached at section 8.4

Results

We ran for different :

- Mini-cluster sizes of 2,3,4,5
- Fusion gates amount of up to nine fusions
- Error axis of X, Y, and Z axis
- Absolute value of the error angle was always $0.1 rad$
- Different cases of clicks (00/01/10/11) in the fusion gates. All fusion with the same clicks.

.

And examined the improvement ratio of D that was defined in Eq. 6.9 between the naive case and the best combination result case.

$R_X(\theta)$ Error Results

- In the case that the clicks of the fusion gate were 00 / 11, the error sign did not affect the result.
- In the case that the clicks of the fusion gate were 01 / 10 - The improvement for the best case was minor, less than a factor of 1.05 to the D ratio between the worst case to the naive case.
- It was interesting to see that in the 01 clicks case, the best result was all negative angles, and the worst was the positive angle. And vice versa for the 10 clicks case

.

$R_Y(\theta)$ Error Results

- In the case that the clicks of the fusion gate were 01 / 10, the error sign did not affect the result.
- In the case that the clicks of the fusion gate were 00 / 11 - we significantly improved using the best combination of Plus-Minus.

.

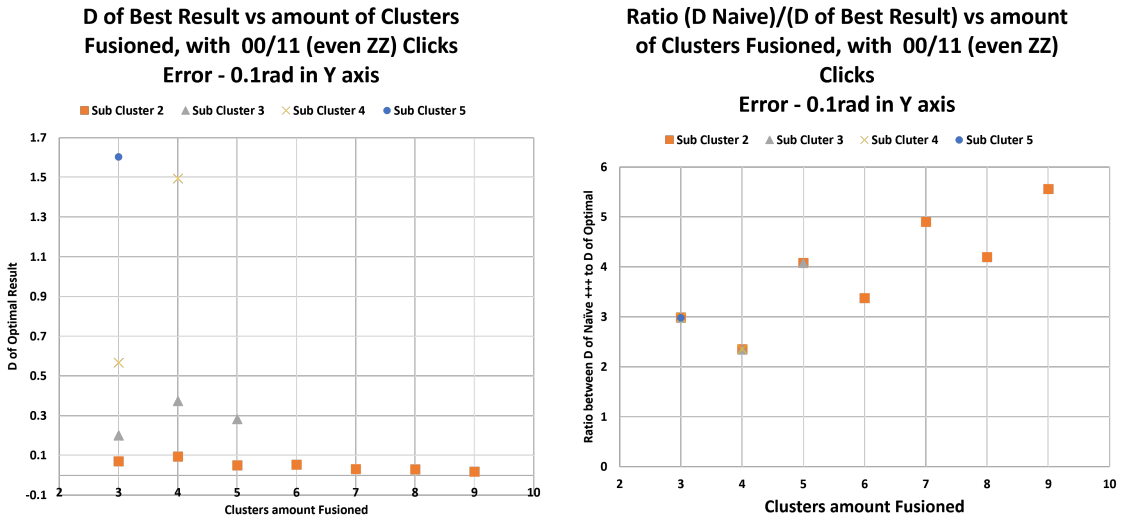


Figure 6.4: Left - D of the best result as a function of fusions amount, for different mini-cluster size cases 2,3,4,5. Right - the ration between D_{Naive} to D_{Best}

As the amount of fusions increases, we get higher and higher D ratio improvement. We can also notice that the improvement ratio does not depend on the size of the mini-cluster. About the best and worst combinations, we noticed that for 00/11 clicks, the all-positive (+++..++) combination is always the worst and that the patterns for the best combination look like those with an equal amount of plus and minus signs.

$R_Z(\theta)$ Error Results

- In all cases, the sign of error did not have any effect on the result.

.

Best Internal Sign Combination for One Fusion Gate

We also checked the fusion gate with the best D when running over all 2^5 combinations of the sign of the beam splitters inside the fusion gate when they are not all the same.

- For $R_X(\theta)$ error model, we notice that the sign of error did not affect the result at all
- For $R_Y(\theta)$ error model, we notice an improvement factor of 1.52 between the worst to the best case. The worst case was the same for all the cases where the sign of beam splitter 0 was different from the sign of beam splitter 1, and it was the best when they were equal.
- For $R_Z(\theta)$ error model, we notice an improvement factor of 1.75 between the worst to the best case. The worst case was the same for all the cases where the sign of beam splitter 3 was different from the sign of beam splitter 4, and it was the best when they were equal.

Therefore, a best practice error model will ensure a full correlation in the coherent error between all five couplers.

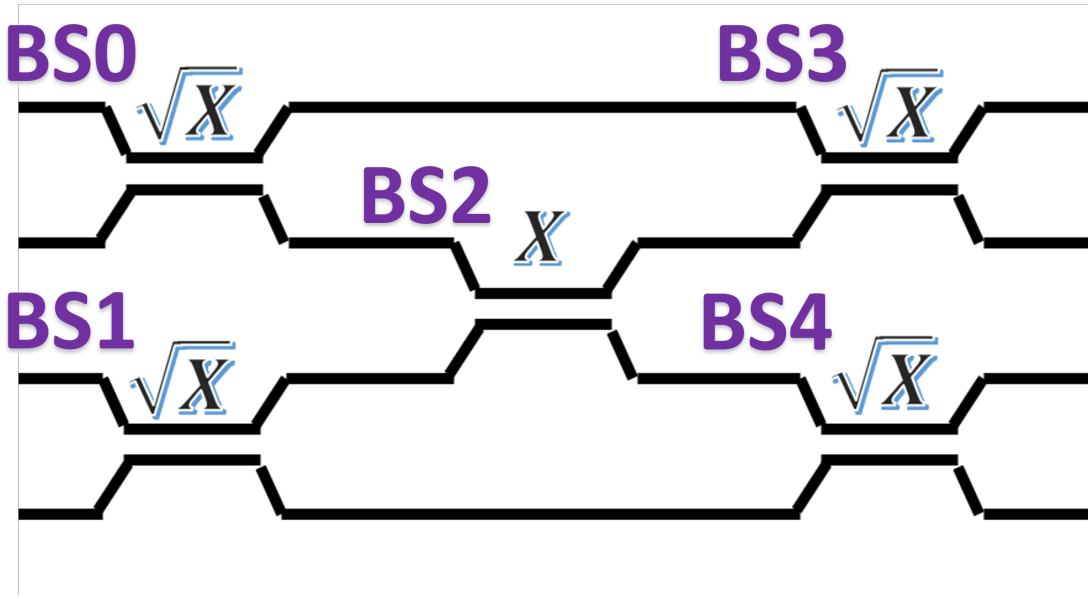


Figure 6.5: Indices for the five beam splitters of the fusion gate

7 Conclusions

The road to a useful tolerant quantum computer is very long and full of challenges. A big effort worldwide is made to suppress logical error using quantum error correction and noise reduction as explained in section 2.3.

This research examined the logical error for the first time under coherent errors, which are not identical for all the qubits. Whether they are distributed over different values as detailed in section 4.2 or whether their signs are different as explained in section 4.3. For that examination, we saw that using the method of tensor networks (as described in section 4.1), we can efficiently simulate large enough surfaces, which in this case helped us to understand the effect of coherent error suppression on the big picture- on the number of physical qubits per one logical qubit.

We found out that the composite segmented pulses approach as described in section 4.2 helps us to shrink the number of qubits by a factor of up to 6.4, which is almost an order of magnitude! Similar improvement ratios were also when we assumed x2 or x0.5 fabrication error. This improvement can give us more immunity to photon loss (higher photon loss threshold), and more work must be done to simulate a case that calculates both loss and coherent errors together and check their both effect on the logical error.

Since this simulation assumed circuit-based QC, which is equivalent to but not the same as measurement-based QC (as described in subsection6.1), more work has to be done to simulate the actual implementation of LOQC that will use the segmented composite pulse technique.

By the deep knowledge of coherent error in QEC that was developed during this research, we also found that using the plus-minus technique we invented, we can reduce the number of physical qubits by a factor of 4 as shown I Fig. 5.6.

This understanding and MBQC also lead us to examine the effect of the plus-minus technique on the fusion of cluster states. We saw that in some error models, it actually also helps us to suppress errors. This model should be expanded to examine clusters with

higher dimensions (2D and 3D). This approach might help reduce the cluster size needed to encode one logical qubit in LOQC.

More work should be performed in order to fully explore how to implement the plus-minus coherent error in the beam splitters. It can be done by playing with the beam splitters' geometry and by manipulating and choosing couplers with the matching error sign.

This plus-minus approach can also be helpful in another quantum computer implementation as cold atoms. Can we use the unwanted interaction between two atoms (as detailed in Ref. [51]) to our favor? and reduce the coherent error by coding them together in +/-. Or trapped ions that mainly suffer from a coherent dephasing Z error as detailed in Ref. [52]. Can we use it in superconductors that suffer from control errors and cross-talk (discussed in Ref. [53]). Engineering and manipulating the error in a known way will suppress logical error in those systems.

References

[1] Austin G Fowler, Matteo Mariantoni, John M Martinis, and Andrew N Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Review A*, 86(3):032324, 2012.

[2] Andrew S Darmawan and David Poulin. Tensor-network simulations of the surface code under realistic noise. *Physical review letters*, 119(4):040502, 2017.

[3] Moshe Katzman, Yonatan Piasetzky, Evyatar Rubin, Ben Barenboim, Maayan Priel, Muhammad Erew, Avi Zadok, and Haim Suchowski. Robust directional couplers for state manipulation in silicon photonic-integrated circuits. *Journal of Lightwave Technology*, 2022.

[4] Jonathan P Dowling and Gerard J Milburn. Quantum technology: the second quantum revolution. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 361(1809):1655–1674, 2003.

[5] Smriti Sharma. Quantum algorithms for simulation of quantum chemistry problems by quantum computers: an appraisal. *Foundations of Chemistry*, pages 1–14, 2022.

[6] Daniel A Lidar and Todd A Brun. *Quantum error correction*. Cambridge university press, 2013.

[7] Joschka Roffe. Quantum error correction: an introductory guide. *Contemporary Physics*, 60(3):226–245, 2019.

[8] Divyashree Y. Venkatesh, Komala Mallikarjunaiah, and Mallikarjunaswamy Srikantaswamy. A comprehensive review of low density parity check encoder techniques. *Ingénierie des Systèmes d'Information*, 27(1), 2022.

[9] Isaac H Kim, Ye-Hua Liu, Sam Pallister, William Pol, Sam Roberts, and Eunseok Lee. Fault-tolerant resource estimate for quantum chemical simulations: Case study on li-ion battery electrolyte molecules. *Physical Review Research*, 4(2):023019, 2022.

[10] Daniel Greenbaum and Zachary Dutton. Modeling coherent errors in quantum error correction. *Quantum Science and Technology*, 3(1):015007, 2017.

[11] Sergey Bravyi, Matthias Englbrecht, Robert König, and Nolan Peard. Correcting coherent errors with surface codes. *npj Quantum Information*, 4(1):1–6, 2018.

[12] Eric Huang, Andrew C Doherty, and Steven Flammia. Performance of quantum error correction with coherent errors. *Physical Review A*, 99(2):022313, 2019.

[13] F Venn and B Béri. Error-correction and noise-decoherence thresholds for coherent errors in planar-graph surface codes. *Physical Review Research*, 2(4):043412, 2020.

[14] Shigeo Hakkaku, Kosuke Mitarai, and Keisuke Fujii. Sampling-based quasiprobability simulation for fault-tolerant quantum error correction on the surface codes under coherent noise. *Physical Review Research*, 3(4):043130, 2021.

[15] Joseph K Iverson and John Preskill. Coherence in logical quantum channels. *New Journal of Physics*, 22(7):073066, 2020.

[16] Elica Kyoseva, Hadar Greener, and Haim Suchowski. Detuning-modulated composite pulses for high-fidelity robust quantum control. *Physical Review A*, 100(3):032333, 2019.

[17] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[18] N Mermin David. *Quantum computer science: an introduction*. Cambridge University Press, 2007.

[19] Sajid Anis, Abby-Mitchell, Héctor Abraham, AduOffei, Rochisha Agarwal, and Gabriele Agliardi et al. Qiskit: An open-source framework for quantum computing, 2021.

[20] Cr Myers and R Laflamme. Linear optics quantum computation: an overview. *arXiv preprint quant-ph/0512104*, 2005.

[21] Kristine Boone. Concepts and methods for benchmarking quantum computers. 2021.

[22] Christopher J Wood, Jacob D Biamonte, and David G Cory. Tensor networks and graphical calculus for open quantum systems. *arXiv preprint arXiv:1111.6950*, 2011.

[23] Martin Bossert. *Channel coding for telecommunications*. John Wiley & Sons, Inc., 1999.

[24] Stefanie J Beale, Joel J Wallman, Mauricio Gutiérrez, Kenneth R Brown, and Raymond Laflamme. Quantum error correction decoheres noise. *Physical review letters*, 121(19):190501, 2018.

[25] Jeff P Barnes, Colin J Trout, Dennis Lucarelli, and BD Clader. Quantum error-correction failure distributions: Comparison of coherent and stochastic error models. *Physical Review A*, 95(6):062338, 2017.

[26] Chi-Kwong Li, Cordelia Li, Diane Christine Pelejo, and Sage Stanish. Quantum error correction scheme for fully correlated noise. *arXiv preprint arXiv:2202.12408*, 2022.

[27] John Preskill. Fault-tolerant quantum computation. In *Introduction to quantum computation and information*, pages 213–269. World Scientific, 1998.

[28] Dott Davide Vodola and Andrea Gaspari. Quantum error correction and the toric code.

[29] Simon Anders and Hans J Briegel. Fast simulation of stabilizer circuits using a graph-state representation. *Physical Review A*, 73(2):022334, 2006.

[30] Craig Gidney. Stim: a fast stabilizer circuit simulator. *Quantum*, 5:497, jul 2021.

[31] Alexander Jahn, Marek Gluza, Fernando Pastawski, and Jens Eisert. Holography and criticality in matchgate tensor networks. *Science advances*, 5(8):eaaw0092, 2019.

[32] Zhih-Ahn Jia, Yuan-Hang Zhang, Yu-Chun Wu, Liang Kong, Guang-Can Guo, and Guo-Ping Guo. Efficient machine-learning representations of a surface code with boundaries, defects, domain walls, and twists. *Physical Review A*, 99(1):012307, 2019.

[33] ChunJun Cao and Brad Lackey. Quantum lego: building quantum error correction codes from tensor networks. *PRX Quantum*, 3(2):020332, 2022.

[34] Pål Sundsøy and Egil Fjeldberg. Quantum computing: Linear optics implementations. *arXiv preprint arXiv:1607.03935*, 2016.

[35] Pieter Kok. Five lectures on optical quantum computing. In *Theoretical Foundations of Quantum Information Processing and Communication*, pages 187–219. Springer, 2010.

[36] Dripto M Debroy, Muyuan Li, Michael Newman, and Kenneth R Brown. Stabilizer slicing: coherent error cancellations in low-density parity-check stabilizer codes. *Physical review letters*, 121(25):250502, 2018.

[37] Jingzhen Hu, Qingzhong Liang, Narayanan Rengaswamy, and Robert Calderbank. Mitigating coherent noise by balancing weight-2 z-stabilizers. *IEEE Transactions on Information Theory*, 68(3):1795–1808, 2021.

[38] Sergey Bravyi and Alexander Vargo. Simulation of rare events in quantum error correction. *Physical Review A*, 88(6):062308, 2013.

[39] Austin G Fowler, Ashley M Stephens, and Peter Groszkowski. High-threshold universal quantum computation on the surface code. *Physical Review A*, 80(5):052312, 2009.

[40] Austin G Fowler. Proof of finite surface code threshold for matching. *Physical review letters*, 109(18):180502, 2012.

[41] Jing Hao Chai and Hui Khoon Ng. On the fault-tolerance threshold for surface codes with general noise. *Advanced Quantum Technologies*, 5(10):2200008, 2022.

[42] Michael Reck, Anton Zeilinger, Herbert J Bernstein, and Philip Bertani. Experimental realization of any discrete unitary operator. *Physical review letters*, 73(1):58, 1994.

[43] Jianwei Wang, Fabio Sciarrino, Anthony Laing, and Mark G Thompson. Integrated photonic quantum technologies. *Nature Photonics*, 14(5):273–284, 2020.

[44] Terry Rudolph. Why i am optimistic about the silicon-photonic route to quantum computing. *APL photonics*, 2(3):030901, 2017.

[45] Richard Jozsa. An introduction to measurement based quantum computation. *NATO Science Series, III: Computer and Systems Sciences. Quantum Information Processing-From Theory to Experiment*, 199:137–158, 2006.

[46] Daniel E Browne and Terry Rudolph. Resource-efficient linear optical quantum computation. *Physical Review Letters*, 95(1):010501, 2005.

[47] Stasja Stanisic. Universal quantum computation by linear optics, 2015.

[48] Sara Bartolucci, Patrick Birchall, Hector Bombin, Hugo Cable, Chris Dawson, Mercedes Gimeno-Segovia, Eric Johnston, Konrad Kieling, Naomi Nickerson, Mihir Pant, et al. Fusion-based quantum computation. *arXiv preprint arXiv:2101.09310*, 2021.

[49] Hector Bombin, Isaac H Kim, Daniel Litinski, Naomi Nickerson, Mihir Pant, Fernando Pastawski, Sam Roberts, and Terry Rudolph. Interleaving: Modular architectures for fault-tolerant photonic quantum computing. *arXiv preprint arXiv:2103.08612*, 2021.

[50] James M Auger, Hussain Anwar, Mercedes Gimeno-Segovia, Thomas M Stace, and Dan E Browne. Fault-tolerant quantum computation with nondeterministic entangling gates. *Physical Review A*, 97(3):030301, 2018.

[51] Dolev Bluvstein, Harry Levine, Giulia Semeghini, Tout T Wang, Sepehr Ebadi, Marcin Kalinowski, Alexander Keesling, Nishad Maskara, Hannes Pichler, Markus Greiner, et al. A quantum processor based on coherent transport of entangled atom arrays. *Nature*, 604(7906):451–456, 2022.

[52] Jie Zhang, Wei Wu, Chun-wang Wu, Jian-guo Miao, Yi Xie, Bao-quan Ou, and Ping-xing Chen. Discrimination and estimation for dephasing sources of trapped ion qubits. *Applied Physics B*, 126(1):1–5, 2020.

[53] Irfan Siddiqi. Engineering high-coherence superconducting qubits. *Nature Reviews Materials*, 6(10):875–891, 2021.

8 Appendices

8.1 Fusion Gate Analysis

```
import sympy as sp
from sympy import *
import numpy as np
from FusionGate import FusionGate
sp.init_printing(order='lex',pretty_print=False, wrap_line=False)
theta = symbols(r'\theta')

f1, f2, f3, f4 = symbols('f_1 f_2 f_3 f_4')
a0, a1, b0, b1, c0, c1, d0, d1 = symbols('a_H^\dagger a_V^\dagger '
                                         'b_H^\dagger b_V^\dagger '
                                         'c_H^\dagger c_V^\dagger '
                                         'd_H^\dagger d_V^\dagger')
vaccum = symbols('\ket{vaccum}')

# input state
input = (f1 * a0 + f2 * a1)*(f3 * b0 + f4 * b1)

# simplifyed input state
(simplify(
            collect(expand(input),
                    [c0 * d0, c0 * d1, c1 * d0, c0 * c1, d0 * d1])))

fusion_gate = FusionGate(error_angle=0.1, error_axis=[0, 1, 0])
print(fusion_gate)
clicks = "00"
```

```
ZZ = fusion_gate.U[int(clicks, 2)]
print("ZZ operator is : ", str(ZZ))

# round function for sympy
def round_expr(expr, num_digits):
    return expr.xreplace({n : round(n, num_digits) for n in expr.atoms(Number)})
num_digits = 3

# before measurement
round_expr(fusion_gate.print_latex_fusion_gate_state(), 3)

# after measurement
round_expr(fusion_gate.print_latex_fusion_gate_state("00"), 3)
round_expr(fusion_gate.print_latex_fusion_gate_state("01"), 3)
round_expr(fusion_gate.print_latex_fusion_gate_state("10"), 3)
round_expr(fusion_gate.print_latex_fusion_gate_state("11"), 3)
```

8.2 Fusion Gate Object

```
SYMBOLIC = False

import sympy as sp
from sympy import *
import numpy as np
from photonic_circuit import PhotonicCircuit

if SYMBOLIC:
    Iden = sp.eye(2)
    X = sp.Matrix([[0, 1], [1, 0]])
    Y = sp.Matrix([[0, -1j], [1j, 0]])
    Z = sp.Matrix([[1, 0], [0, -1]])
else:
    I = np.eye(2)
    X = np.array([[0, 1], [1, 0]])
    Y = np.array([[0, -1j], [1j, 0]])
    Z = np.array([[1, 0], [0, -1]])
```

```
class FusionGate:

    def __init__(self, error_angle=0, error_axis=None):
        if SYMBOLIC:
            if error_axis is None:
                error_axis = [1, 0, 0]

            self.error_axis = error_axis
            self.error_angle = error_angle

            self.R_error = sp.cos(error_angle / 2) * I + 1j * sp.sin(
                error_angle / 2) * \
                           (error_axis[0] * X + error_axis[0] * Y + error_axis[
                               0] * Z)
            X_sqrt = sp.cos(np.pi / 4) * I - sp.sin(np.pi / 4) * 1j * X

            circuit = PhotonicCircuit(4)
            circuit.apply_coupler(X_sqrt * self.R_error, 0, 1)
            circuit.apply_coupler(X_sqrt * self.R_error, 2, 3)
            circuit.apply_coupler(-1j * X_sqrt * self.R_error, 1, 2)
            circuit.apply_coupler(X_sqrt * self.R_error, 0, 1)
            circuit.apply_coupler(X_sqrt * self.R_error, 2, 3)
            self.basis = [[0, 2], [0, 3], [1, 2], [1, 3]]
            self.U = circuit.project_on_computation_basis(self.basis)

        else:
            if error_axis is None:
                error_axis = [1, 0, 0]
            self.error_axis = error_axis
            self.error_angle = error_angle

            self.R_error = np.cos(error_angle / 2) * I + 1j * np.sin(
                error_angle / 2) * \
                           (error_axis[0] * X + error_axis[1] * Y + error_axis[
```

```
                                2] * Z)
            X_sqrt = np.cos(np.pi / 4) * I - np.sin(np.pi / 4) * 1j * X

            circuit = PhotonicCircuit(4)
            circuit.apply_coupler(np.matmul(X_sqrt, self.R_error), 0, 1)
            circuit.apply_coupler(np.matmul(X_sqrt, self.R_error), 2, 3)
            circuit.apply_coupler(-1j * np.matmul(X, self.R_error), 1, 2)
            circuit.apply_coupler(np.matmul(X_sqrt, self.R_error), 0, 1)
            circuit.apply_coupler(np.matmul(X_sqrt, self.R_error), 2, 3)
            self.basis = [[0, 2], [0, 3], [1, 2], [1, 3]]
            self.U = circuit.project_on_computation_basis(self.basis)

    def getUFromClicks(self, clicks):
        if SYMBOLIC:
            return self.U.row(int(clicks, 2))
        else:
            return self.U[int(clicks, 2)]

    def print_latex_fusion_gate_state(self, clicks=""):
        # Symbols
        f1, f2, f3, f4 = symbols('f_1 f_2 f_3 f_4')
        a0, a1, b0, b1, c0, c1, d0, d1 = symbols('a_H^\dagger a_V^\dagger '
                                                 'b_H^\dagger b_V^\dagger '
                                                 'c_H^\dagger c_V^\dagger '
                                                 'd_H^\dagger d_V^\dagger')
        U = symbols('U')

        # U Matrix
        if clicks == "":
            U = Matrix(self.U)

            U
            simplify(U)

            # in out
            a_out = Matrix([[c0],
```

```
                        [c1],
                        [d0],
                        [d1], ])

            # Mul
            a_in = U * a_out
            a0, a1, b0, b1 = a_in[0], a_in[1], a_in[2], a_in[3]

            # state
            state = (f1 * a0 + f2 * a1) * (f3 * b0 + f4 * b1)

        else:
            U = Matrix((self.getUFromClicks(clicks)).transpose())
            a_out = Matrix([[f1 * f3],
                            [f1 * f4],
                            [f2 * f3],
                            [f2 * f4], ])
            state = U[0] * a_out[0] + U[1] * a_out[1] + U[2] * a_out[2] + U[3] *
                    a_out[
                        3]

        return (simplify(
            collect(expand(state),
                    [c0 ** 2, c1 ** 2, d0 ** 2, d1 ** 2, c0 * d0, c0 * d1,
                     c1 * d0,
                     c0 * c1, d0 * d1])))




def __str__(self):
    s = ""
    s += "\n_____________________Fusion:_____________________\n"
```

```
        s += "U : \n"
        if self.error_angle == 0 or SYMBOLIC:
            round_res = 7
        else:
            round_res = int(7+np.log10(1/self.error_angle))
        if not SYMBOLIC:
            s += str(np.round(self.U, round_res))
        else:
            s += str(sp.simplify(self.U))
        s += "\n\nBasis :\n"
        s += str(self.basis)

        s += "\n\nR error : \n"
        if not SYMBOLIC:
            s += str(np.round(self.R_error, round_res))
        else:
            s += str(self.R_error)
        s += "\n_____________________________________________________"

        return s

# fusion_gate = FusionGate(error_angle=0.1, error_axis=[0, 1, 0])
# print(fusion_gate)
# clicks = "00"
# ZZ = fusion_gate.U[int(clicks, 2)]
# print("ZZ operator is : ", str(ZZ))
```

8.3 Photonic Circuit Object

```
SYMBOLIC = False
import sympy as sp
import numpy as np

class PhotonicCircuit:
    def __init__(self, n):
        self.n = n
```

```
        if SYMBOLIC:
            self.unitary = sp.eye(self.n)
        else:
            self.unitary = np.eye(self.n, dtype=np.complex128)

    def apply_coupler(self, coupler, wg1, wg2):
        if SYMBOLIC:
            coup_matrix = sp.eye(self.n)
            coup_matrix[wg1, wg1] = coupler.row(0).col(0)
            coup_matrix[wg1, wg2] = coupler.row(0).col(1)
            coup_matrix[wg2, wg1] = coupler.row(1).col(0)
            coup_matrix[wg2, wg2] = coupler.row(1).col(1)

            self.unitary = coup_matrix * self.unitary
        else:
            coup_matrix = np.eye(self.n, dtype=np.complex128)
            coup_matrix[wg1, wg1] = coupler[0, 0]
            coup_matrix[wg1, wg2] = coupler[0, 1]
            coup_matrix[wg2, wg1] = coupler[1, 0]
            coup_matrix[wg2, wg2] = coupler[1, 1]
            self.unitary = np.matmul(coup_matrix, self.unitary)

    def project_on_computation_basis(self, basis):
        m = len(basis)  # 2^Number of qubits

        if SYMBOLIC:
            logical_unitary = sp.zeros(m, m)
        else:
            logical_unitary = np.zeros((m, m), dtype=np.complex128)
        for i in range(m):
            for j in range(m):
                v_ind = [basis[i][0], basis[i][1]]
                u_ind = [basis[j][0], basis[j][1]]
                logical_unitary[i, j] = self.unitary[v_ind[0], u_ind[0]] * \
                                        self.unitary[v_ind[1], u_ind[1]] \
                                        + self.unitary[v_ind[0], u_ind[1]] * \
```

```
                                        self.unitary[v_ind[1], u_ind[0]]
        return logical_unitary
```

8.4 Find Best combination

```
import os

import openpyxl
import xlsxwriter

SYMBOLIC = False

from Cluster import Cluster
from FusionGate import FusionGate


def best_combo_with_vs_without_rotation(fi, fp, fm, cluster_size, clicks,
                                        fusions_amount, rot_state_angle,
                                        rot_state_axis=[1, 0, 0]):
    runOverPMCombinations(fi, fp, fm, cluster_size, clicks, fusions_amount,
                          rot_state_angle=0, rot_state_axis=rot_state_axis)
    runOverPMCombinations(fi, fp, fm, cluster_size, clicks, fusions_amount,
                          rot_state_angle=rot_state_angle,
                          rot_state_axis=rot_state_axis)


def runOverPMCombinations(fi, fp, fm, cluster_size, clicks, fusions_amount,
                          rot_state_angle=0, rot_state_axis=[1, 0, 0]):
    print("---------------------------------------")
    print("Finding combination for ", str(fusions_amount + 1), "cluster of",
          str(cluster_size), "  Clicks are ", clicks)
    print("State change from cluster is theta=", str(rot_state_angle))
    print("Statrting...")
    c12naive = Cluster.fuseLinearClusters(size_of_cluster=cluster_size,
                                          clicks=clicks,
                                          fusions=[fp] * fusions_amount,
```

```
                                        rot_state_angle=rot_state_angle,
                                        rot_state_axis=rot_state_axis)
c12i = Cluster.fuseLinearClusters(size_of_cluster=cluster_size,
                                  clicks=clicks,
                                  fusions=[fi] * fusions_amount,
                                  rot_state_angle=rot_state_angle,
                                  rot_state_axis=rot_state_axis)
worst_case = float("-Inf")
best_case = float("Inf")
for i in range(2 ** fusions_amount):
    b = format(i, '0' + str(fusions_amount) + 'b')
    fusions_list = []
    for bit in b:
        if bit == '0':
            fusions_list += [fp]
        else:
            fusions_list += [fm]
    c12pm = Cluster.fuseLinearClusters(size_of_cluster=cluster_size,
                                       clicks=clicks, fusions=fusions_list,
                                       rot_state_angle=rot_state_angle,
                                       rot_state_axis=rot_state_axis)
    # print("for: "+str(b)+" PM dist from ideal : " + str(c12pm - c12i))
    if c12pm - c12i > worst_case:
        worst_case = c12pm - c12i
        worst_case_raw = b
    if c12pm - c12i < best_case:
        best_case = c12pm - c12i
        best_case_raw = b

naive_case = c12naive - c12i
print("Naive (00..0) dist from ideal : " + str(naive_case))

print("Worst case is : " + worst_case_raw + " value: " + str(worst_case))
print("Best case is : " + best_case_raw + " value: " + str(best_case))
print("---------------------------------------")
```

```
# Add row to excel
result_folder = r"C:\PycharmProjects\LOQCGateSimulation\outputs"
result_file_name = r"C:\PycharmProjects\LOQCGateSimulation\outputs\results.xlsx"
if not (os.path.isfile(result_file_name) and os.access(result_file_name,
                                                       os.R_OK)):
    if not os.path.exists(result_folder):
        os.makedirs(result_folder)
    workbook = xlsxwriter.Workbook(result_file_name)
    worksheet = workbook.add_worksheet()
    workbook.close()

book = openpyxl.load_workbook(result_file_name)

sheet_name = "axis-" + str(fp.error_axis[0]) + "," + str(
    fp.error_axis[1]) + "," + str(fp.error_axis[2]) + " theta-" + str(
    fp.error_angle)
if not sheet_name in book.sheetnames:
    ws = book.create_sheet(sheet_name)
else:
    ws = book.get_sheet_by_name(sheet_name)

data = {2: "clicks", 3: "SubCluster Size", 4: "Cluster Amount",
        5: "Worst Combo", 6: "Best Combo", 7: "Naive Result",
        8: "Worst Result", 9: "Best Result",
        10: "Naive/Best", 11: "Worst/Best", 12: "Theta", 13: "Angle Axis"}
row_index = 1
for col, value in data.items():
    ws.cell(row=row_index, column=col, value=value)
book.save(filename=result_file_name)

data = {2: clicks, 3: cluster_size, 4: fusions_amount + 1,
        5: worst_case_raw, 6: best_case_raw,
        7: naive_case, 8: worst_case, 9: best_case,
        10: naive_case / best_case, 11: worst_case / best_case,
        12: str(fp.error_angle), 13: str(fp.error_axis)}
```

```
    row_index = ws.max_row + 1
    ws.insert_rows(row_index)
    for col, value in data.items():
        ws.cell(row=row_index, column=col, value=value)

    book.save(filename=result_file_name)


if __name__ == '__main__':
    print("Init Fusion Gates...")

    error_angle = 0.1
    error_axis = [1, 0, 0]
    fm = FusionGate(error_angle=-error_angle, error_axis=error_axis)
    fp = FusionGate(error_angle=error_angle, error_axis=error_axis)
    fi = FusionGate(error_angle=0, error_axis=error_axis)

    # print("0-0 ~ 0-0")
    # c12p = Cluster.fuseLinearClusters(size_of_cluster=2, clicks='00', fusions=
    # print("0-0 ~ 0-0")
    # c12m = Cluster.fuseLinearClusters(size_of_cluster=2, clicks='00', fusions=
    # c12i = Cluster.fuseLinearClusters(size_of_cluster=2, clicks='00', fusions=
    # print("0-0 ~ 0-0")
    # c12naive = Cluster.fuseLinearClusters(size_of_cluster=3, clicks='00', fusi
    # c12pm = Cluster.fuseLinearClusters(size_of_cluster=3, clicks='00', fusions
    # c12i = Cluster.fuseLinearClusters(size_of_cluster=3, clicks='00', fusions=
    # print("PM dist from ideal : " + str(c12pm - c12i))
    # print("Naive dist from ideal : " + str(c12naive - c12i))

    for clicks in ["00", "01", "10", "11"]:
        for mini_c_size in [2]:
            for fusions_amount in range(2, 9):
                runOverPMCombinations(fi, fp, fm, cluster_size=mini_c_size,
                                      clicks=clicks,
                                      fusions_amount=fusions_amount,
                                      rot_state_angle=0,
```

```
                                    rot_state_axis=[1, 0, 0])

    for clicks in ["00", "01", "10", "11"]:
        for mini_c_size in [3]:
            for fusions_amount in range(2, 5):
                runOverPMCombinations(fi, fp, fm, cluster_size=mini_c_size,
                                    clicks=clicks,
                                    fusions_amount=fusions_amount,
                                    rot_state_angle=0,
                                    rot_state_axis=[1, 0, 0])

    for clicks in ["00", "01", "10", "11"]:
        for mini_c_size in [4]:
            for fusions_amount in range(2, 4):
                runOverPMCombinations(fi, fp, fm, cluster_size=mini_c_size,
                                    clicks=clicks,
                                    fusions_amount=fusions_amount,
                                    rot_state_angle=0,
                                    rot_state_axis=[1, 0, 0])

    for clicks in ["00", "01", "10", "11"]:
        for mini_c_size in [5]:
            for fusions_amount in range(2, 3):
                runOverPMCombinations(fi, fp, fm, cluster_size=mini_c_size,
                                    clicks=clicks,
                                    fusions_amount=fusions_amount,
                                    rot_state_angle=0,
                                    rot_state_axis=[1, 0, 0])
```

8.5 Cluster Class

```
SYMBOLIC = False
import sympy as sp
from sympy.physics.quantum import TensorProduct
import numpy as np
from dec2bin import *
```

```
from sympy.core.rules import Transform

from FusionGate import FusionGate

I = np.eye(2)
X = np.array([[0, 1], [1, 0]])
Y = np.array([[0, -1j], [1j, 0]])
Z = np.array([[1, 0], [0, -1]])


class Cluster:

    def __init__(self, n, U=None):
        self.res = 3
        if isinstance(n, int):
            if SYMBOLIC:
                x = (1 / 2 ** 0.5) * sp.Matrix([1, 1])
                for _ in range(1, n):
                    x = TensorProduct(x, sp.Matrix([1, 1]))

                for i in range(0, n - 1):
                    x = Cluster.CZ(x, i, i + 1)

                # normalize = np.sqrt(sum(x * np.conj(x)))
                # x = x / normalize

                self.q_state = x
            else:
                x = (1 / 2 ** 0.5) * np.array([1, 1])
                for _ in range(1, n):
                    x = np.kron(x, ([1, 1]))

                for i in range(0, n - 1):
                    x = Cluster.CZ(x, i, i + 1)

                # normalize = np.sqrt(sum(x * np.conj(x)))
```

```
            # x = x / normalize

            self.q_state = x
        else:
            self.q_state = n
        if U is not None:
            self.U = U
        else:
            self.U = np.eye(len(self.q_state))

    @staticmethod
    def CZ(x, n1, n2):
        for i in range(len(x)):
            b = format(i, '0' + str(int(np.log2(len(x)))) + 'b')
            if b[n1] == '1' and b[n2] == '1':
                x[i] = -x[i]
        return x

    def rotate(self, index, theta, axis):
        matrix1 = np.eye(2 ** index)
        matrix2 = np.cos(theta / 2) * I + 1j * np.sin(theta / 2) * \
                  (axis[0] * X + axis[1] * Y + axis[2] * Z)
        matrix3 = np.eye(2 ** (int(np.log2(len(self.q_state))) - index - 1))
        matrix_total = np.kron(matrix1, np.kron(matrix2, matrix3))
        self.q_state = np.matmul(matrix_total, self.q_state)

    def fusion(self, other, fusion_gate, clicks):
        # clicks - '00'/'01'/'10'/'11'
        # fusion_gate - object

        if SYMBOLIC:
            n1 = int(np.log2(len(self.q_state))) - 1
            U1 = sp.eye(2 ** n1)
            n2 = int(np.log2(len(other.q_state))) - 1
            U2 = sp.eye(2 ** n2)
```

```
            new_c_q_state = TensorProduct(self.q_state, other.q_state)

            # ZZ = fusion_gate.U[int(clicks, 2)]
            ZZ = fusion_gate.U.row(int(clicks, 2))

            U = TensorProduct(U1, sp.Matrix([ZZ]))
            U = TensorProduct(U, U2)
            normalize = np.sqrt(
                sum(np.multiply(new_c_q_state, np.conj(new_c_q_state))))
            U = U * (1 / normalize)
            new_c_q_state = U * new_c_q_state

            # normalize = sp.sqrt(sum(sp.matrices.dense.matrix_multiply_elementw
            # new_c_q_state = new_c_q_state / normalize

            # phase1 = sp.log(new_c_q_state[0]).as_real_imag()[1]
            # new_c_q_state = new_c_q_state * sp.exp(-1j * phase1)

            # TODO U for symbolic
            raise NotImplementedError("SYMBOLIC not implemented")
        else:

            n1 = int(np.log2(len(self.q_state))) - 1
            U1 = np.eye(2 ** n1)
            n2 = int(np.log2(len(other.q_state))) - 1
            U2 = np.eye(2 ** n2)

            new_c_q_state = np.kron(self.q_state, other.q_state)

            ZZ = fusion_gate.U[int(clicks, 2)]

            U = np.kron(U1, ZZ)
            U = np.kron(U, U2)
            normalize = np.sqrt(
                sum(np.multiply(new_c_q_state, np.conj(new_c_q_state))))
            U = U * (1 / normalize)
```

```
            new_c_q_state = np.matmul(U, new_c_q_state)

            phase1 = np.angle(new_c_q_state[0])
            new_c_q_state = new_c_q_state * np.exp(-1j * phase1)

            original_u = np.kron(self.U, other.U)
            mid_u = np.kron(np.kron(np.eye(int(len(self.q_state) * 0.5)), ZZ),
                            np.eye(int(len(other.q_state) * 0.5)))
            new_U = np.matmul(mid_u, original_u)

        return Cluster(new_c_q_state, new_U)

    @staticmethod
    def fuseLinearClusters(size_of_cluster, clicks, fusions, rot_state_angle=0,
                           rot_state_axis=[1, 0, 0]):
        if isinstance(size_of_cluster, list):
            # assert(len(size_of_cluster) == len(fusions)+1)
            c_out = Cluster(size_of_cluster[0])
            c_out.rotate(1, rot_state_angle, rot_state_axis)
            for i in range(1, len(fusions) + 1):
                other_cluster = Cluster(size_of_cluster[i])
                other_cluster.rotate(1, rot_state_angle, rot_state_axis)
                c_out = c_out.fusion(other=other_cluster,
                                     fusion_gate=fusions[i - 1], clicks=clicks)
            return c_out
        else:
            c_out = Cluster(size_of_cluster)
            c_out.rotate(1, rot_state_angle, rot_state_axis)
            for i in range(1, len(fusions) + 1):
                other_cluster = Cluster(size_of_cluster)
                other_cluster.rotate(1, rot_state_angle, rot_state_axis)
                c_out = c_out.fusion(other=Cluster(size_of_cluster),
                                     fusion_gate=fusions[i - 1], clicks=clicks)
            return c_out

    def __str__(self):
```

```
        s = ""
        s += "Size: " + str(int(np.log2(len(self.q_state))))
        s += "\nQ State:\n"
        for i in range(0, len(self.q_state)):
            if SYMBOLIC:
                x = sp.simplify(self.q_state[i].xreplace(
                    Transform(lambda x: x.round(self.res),
                              lambda x: isinstance(x, float))))
                s += str(x) + "|" + format(i, '0' + str(
                    int(np.log2(len(self.q_state)))) + 'b') + "â©\n"
            else:
                s += str(round(self.q_state[i], self.res)) \
                     + "|" + format(i,
                                    '0' + str(
                                        int(np.log2(
                                            len(self.q_state)))) + 'b') + "â©\n"
        return s

    def __sub__(self, other):

        diff_mat = self.U - other.U
        return np.abs(np.sqrt(
            np.trace(np.matmul(diff_mat, np.conj(np.transpose(diff_mat))))))

        # yaron = (1/len(self.U))*np.trace(np.matmul(np.conj(np.transpose(self.U
        # return yaron

        # if SYMBOLIC:
        #     p1 = sp.outer(self.q_state, sp.conjugate(self.q_state))
        #     p2 = sp.outer(other.q_state, sp.conjugate(other.q_state))
        #
        #     return sp.simplify(0.5 * sp.trace(abs(p1 - p2)))
        # else:
        #
        #     # p1 = sum(np.multiply(self.q_state, np.conj(self.q_state)))
        #     p1 = np.outer(self.q_state, np.conj(self.q_state))
```

```
#     # p2 = sum(np.multiply(other.q_state, np.conj(other.q_state)))
#     p2 = np.outer(other.q_state, np.conj(other.q_state))
#
#     return 0.5 * np.trace(abs(p1 - p2))
```