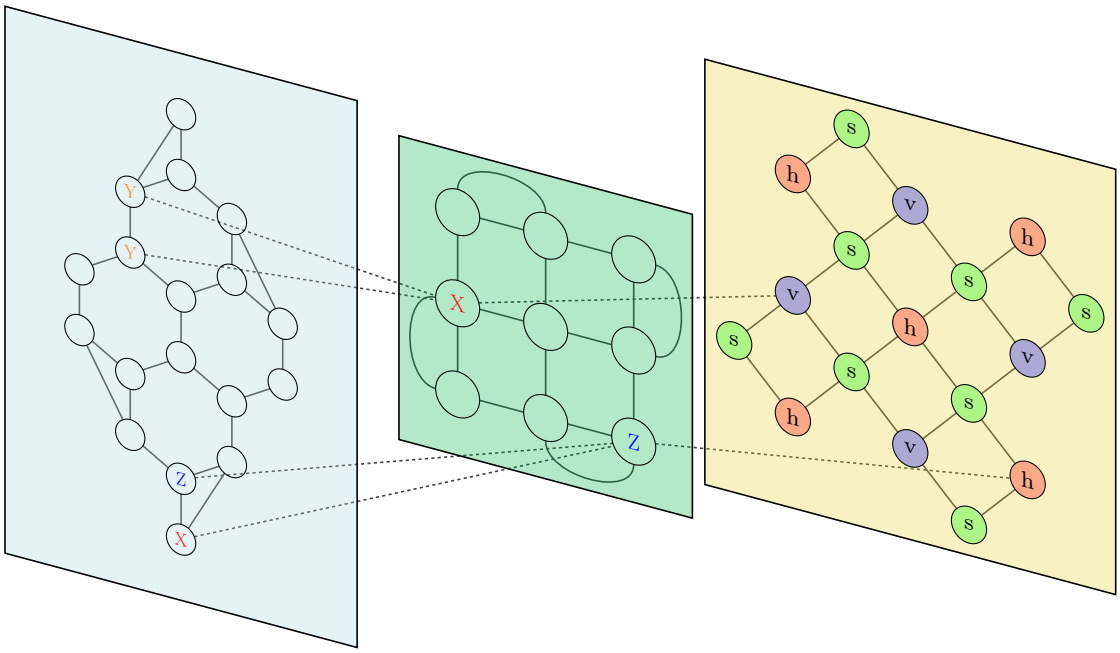
CHALMERS
UNIVERSITY OF TECHNOLOGY



Tensor network based decoders for topological stabilizer codes with diverse qubit error rates

Master's thesis in EMMNano

YINZI XIAO

DEPARTMENT OF NANOTECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Master's thesis 2023

Tensor network based decoders for topological stabilizer codes with diverse qubit error rates

YINZI XIAO

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Nanotechnology
Chalmers University of Technology
Gothenburg, Sweden 2023

Tensor network based decoders for topological stabilizer codes
with diverse qubit error rates
YINZI XIAO

Supervisor: Mats Granath, University of Gothenburg
Examiner: Mats Granath, University of Gothenburg

Master's Thesis 2023
Department of Nanotechnology
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Diagram illustration of algorithm steps in BSV decoder of XYZ^2 code.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2023

Tensor network based decoders for topological stabilizer codes
with diverse qubit error rates
YINZI XIAO
Department of Nanotechnology
Chalmers University of Technology

Abstract

To get a reliable and robust computing result for quantum computers, quantum error correction (QEC) is necessary, and various protection codes have been developed. Among them topological stabilizer codes are gaining promise for QEC, because their global properties provided by the topological code structure help protect the logical qubits from noises. While only local measurements are required which makes them possible to be implemented in existing quantum hardware, a decoder needs to be applied to map measurements to error correcting operations. Improving the decoder helps in reducing the logical failure rate of the code. In this master thesis project, a tensor network based decoder which supports decoding certain types of stabilizer codes with diverse qubit error rates is developed, and is used to research the scenarios for stabilizer codes with diverse error rates and the multi-level decoding for concatenated codes such as the XYZ^2 code.

Keywords: quantum error correction, surface code, tensor network, non-IID error.

Acknowledgements

First I want to express my sincere gratitude to the examiner and supervisor of my master thesis project, Prof. Mats Granath. Thank you for all the guidance and help you offered during this project. You can always provide good suggestions when I were struggling in various kind of problems during the project, and I hope that I can learn from you to become a better researcher in my future study. I also want to thank Basudha Srivastava and Moritz Lange for their valuable suggestions and discussions. Then I want to thank all of my friends here in Gothenburg, especially Veronika Beliaeva and Eduardo Bardales. We have been classmates and good friends for almost two years during our master study, and my life in Gothenburg would be much less joyful if I don't have your company. Finally I want to thank my parents. Even though I could only see you through video calls, I know for sure that you are standing by my side and giving me support, as you've been always doing since I was a little kid.

The simulations and part of the codes written in this project are based on the python package *qecsim*. Computations were enabled by resources provided by the Chalmers Centre for Computational Science and Engineering (C3SE).

Yinzi Xiao, Gothenburg, June 2023

Contents

List of Figures

1
Introduction

This chapter introduces the relevant background knowledge for this thesis. Section 1.1 briefly discusses some necessary knowledge for quantum computing; Section 1.2 introduces the theory and some examples of quantum error correcting codes; Section 1.3 gives an introduction of tensor networks and section 1.4 discusses the simulation software and packages used in this thesis.

1.1 Quantum Computing

This section introduces the basic knowledge needed for quantum computing, which includes the axioms of quantum mechanics, the representation of quantum states and quantum operators, the concept of qubit (quantum bit) and common quantum operators used in quantum computing and quantum error correction.

1.1.1 Axioms of quantum mechanics

There are multiple sets of axioms which build the foundation of quantum mechanics. The following relates closely to the content in this thesis.

Axiom of state and superposition: A quantum system is related to a complex vector space $\mathbb{H}$ (Hilbert space). The properties of a quantum system can be completely described by a quantum state vector $|\phi\rangle$ in this Hilbert space. As a result, any quantum state can be formed by the superposition of two or more quantum states; and the superposition of any two or more quantum states will form a new quantum state.

Axiom of observable: For every physical property $\mathcal{A}$ of a quantum system, there exists an associated Hermitian operator $\hat{A}$ (which is usually called an observable) which acts in the Hilbert space $\mathbb{H}$ of this quantum system. Each measurement result of this physical property must be an eigenvalue of this Hermitian operator, and the corresponding quantum state for this eigenvalue is the state of this quantum system after measurement.

Axiom of evolution: The time evolution of a closed quantum system can be described by a unitary operator $\hat{U}(t, t_0)$, such that for any state $|\phi_0\rangle$, the state $|\phi\rangle$ can be obtained by: $|\phi\rangle = \hat{U}(t, t_0)\,|\phi_0\rangle\,; \forall t, t_0$.

Axiom of composite system: Consider two quantum system A and B. If the Hilbert space of system A and B is denoted as $\mathbb{H}_A$ and $\mathbb{H}_B$, the Hilbert space of the

composite system AB is the tensor product of $\mathbb{H}_A$ and $\mathbb{H}_B$: $\mathbb{H}_A \otimes \mathbb{H}_B$. If the state of system A and B is $|\phi_A\rangle$ and $|\psi_B\rangle$, the state of composite system AB is $|\phi_A\rangle \otimes |\psi_B\rangle$.

1.1.2 Representation

In this thesis, quantum states and operators are usually represented in the form of Dirac notation or vectors and matrices. This section introduces the denotation and calculation of these two representations.

1.1.2.1 Dirac notation

State:
In Dirac notation, a quantum state is denoted as a *ket*: $|\phi\rangle$, which represents a vector in the Hilbert space $\mathbb{H}$. According to the axiom of superposition, the superposition of several quantum states will form another state inside the Hilbert space:

$$|c\rangle = a\,|a\rangle + b\,|b\rangle\,, \tag{1.1}$$

where a and b are complex numbers and $|a\rangle$ and $|b\rangle$ are state vectors.

For each Hilbert space, there's a corresponding dual vector space $\mathbb{H}^*$, and the dual vectors in this space is denoted as a *bra*: $\langle\phi|$. This dual vector space is used to describe properties of the inner product between states:

$$\langle u|v\rangle = \langle v|u\rangle^* \in \mathbb{C}. \tag{1.2}$$

If a ket $|u\rangle$ satisfies $\langle u|u\rangle = 1$, then this ket is called *normalized*. If two kets $|u\rangle$ and $|v\rangle$ satisfy $\langle u|v\rangle = 0$, then these two kets are called *orthogonal* with each other.

Operator:
In Dirac notation, an operator is denoted as a capital character with a hat: $\hat{O}$. Sometimes it is also common to skip the hat for convenience: O. An operator acts on a quantum state, usually transforming it into another different state. If the result of the action is the same state with some complex value, then the state and value are called eigenstate and eigenvalue of this operator.

An operator can also act on a bra state in the dual vector space. Consider an operator $\hat{A}$ which acts on a ket state $|\phi\rangle$ in $\mathbb{H}$ and generates a new ket state $|\phi'\rangle = \hat{A}\,|\phi\rangle$. The corresponding bra state $\langle\phi'|$ in $\mathbb{H}^*$ is defined as:

$$\langle\phi'| = \langle\phi|\,\hat{A}^\dagger, \tag{1.3}$$

where $\hat{A}^\dagger$ is called the Hermitian conjugate of operator $\hat{A}$. If an operator $\hat{H}$ satisfies: $\hat{H} = \hat{H}^\dagger$, this operator is called a *Hermitian operator*. Every physical property must correlate with a Hermitian operator (also called an observable), and all the eigenvalues of a Hermitian operator are real numbers.

Operators can also be generated by the outer product of ket and bra states:

$$|u\rangle\langle v| = (|v\rangle\langle u|)^\dagger. \tag{1.4}$$

It's easy to see that $|u\rangle\langle v|$ is indeed an operator by acting it on a state $|\psi\rangle$:

$$(|u\rangle\langle v|)|\psi\rangle = |u\rangle(\langle v|\psi\rangle) = (\langle v|\psi\rangle)|u\rangle, \tag{1.5}$$

since it transforms $|\psi\rangle$ into $|u\rangle$.

1.1.2.2 Linear Algebra

Orthonormal complete basis:
Consider a discrete Hilbert space $\mathbb{H}$. It is possible to find a collection of states $\{|\phi_i\}\rangle = \{|\phi_1\rangle, |\phi_2\rangle, \cdots, |\phi_N\rangle\}$, such that every state in $\mathbb{H}$ can be represented by this collection:

$$|\Psi\rangle = \sum_{i=1}^{N} c_i |\phi_i\rangle, \forall |\Psi\rangle \in \mathbb{H}. \tag{1.6}$$

If all states in this collection are *orthonormal*:

$$\langle\phi_i|\phi_j\rangle = \delta_{ij}, \tag{1.7}$$

then this collection of basis is called the *orthonormal complete basis*. This basis is very useful: since it's complete, it can describe all states in $\mathbb{H}$; since it's orthonormal, the calculation rules between states and operators are very easy and are identical with those in linear algebra.

Vector:
Because the norm of a state doesn't influence any of its physical properties, one can require every state in a Hilbert space to be normalized. Then by Equation 1.6, every state in this Hilbert space can be uniquely represented by the coefficients $\{c_i\}$. This yields the column vector representation for a ket state:

$$|\Psi\rangle = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{pmatrix}. \tag{1.8}$$

Similarly, a row vector representation can be used for the bra states in the dual vector space:

$$\langle\Psi| = (c_1^* \;\; c_2^* \;\; \cdots \;\; c_N^*). \tag{1.9}$$

It's natural to define the inner product of a bra and a ket state under this representation:

$$\langle\Psi|\Psi\rangle = (c_1^* \;\; c_2^* \;\; \cdots \;\; c_N^*) \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{pmatrix} = |c_1|^2 + |c_2|^2 + \cdots + |c_N|^2, \tag{1.10}$$

since the basis is orthonormal.

Matrix:
Operators in the Hilbert space can also be represented by the basis:

$$\hat{O} = \sum_{i=1}^{N} \sum_{j=1}^{N} |\phi_i\rangle \langle\phi_i| \hat{O} |\phi_j\rangle \langle\phi_j| , \tag{1.11}$$

by using this property of orthonormal complete basis:

$$\sum_{i=1}^{N} |\phi_i\rangle \langle\phi_i| = \hat{I}, \tag{1.12}$$

where $\hat{I}$ is the identity operator.

Therefore, by denoting the coefficients $\langle\phi_i| \hat{O} |\phi_j\rangle$ as c_{ij}, one can represent the operator coefficients in a matrix form. Because these coefficients identically define the operator, this matrix actually represents the operator in the given basis:

$$\hat{O} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1N} \\ c_{21} & c_{22} & \cdots & c_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ c_{N1} & c_{N2} & \cdots & c_{NN} \end{pmatrix}. \tag{1.13}$$

It's easy to verify that the calculation for the operator acting on bra and ket states under this representation follows the rules in linear algebra as well. Note that this representation is not unique, since one can choose different basis for the given Hilbert space. Then, the representation for the same state or operator may change due to the different choice for basis.

1.1.3 Qubit

The *qubit* (quantum bit) is the elementary unit in quantum computing, which is essentially a two-level quantum system. Based on the type of quantum system, there are several types of qubits, including superconducting qubits, spin qubits, photonic qubits and so on. Those will not be discussed in this thesis, but only the mathematical or abstract properties of a qubit.

1.1.3.1 Classical bit

Before discussing the qubit, it's good to first discuss the classical bit. The classical bit is the basic unit in classical information theory. It has two states, which are often denoted 0 and 1. A classical bit can either be in state 0 or state 1, and can only be in one state at a time.

1.1.3.2 Qubit

Unlike a classical bit, a qubit is defined in a two-dimensional Hilbert space $\mathbb{H}^2$. It has two computational basis states, which are often denoted $|0\rangle$ and $|1\rangle$, and infinitely many states, which can be represented as the superposition of $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \tag{1.14}$$

where α and β are complex numbers, and are often required to obey $|\alpha|^2 + |\beta|^2 = 1$ in order to normalize $|\phi\rangle$.

By the definition given in Section 1.1.2.2, the basis states can be written in vector notation as:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad ; \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \tag{1.15}$$

and $|\psi\rangle$ can also be written in vector notation as:

$$|\psi\rangle = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}. \tag{1.16}$$

The bra state $\langle\psi|$ in dual vector space can be written as:

$$\langle\psi| = \alpha^*(1\ \ 0) + \beta^*(0\ \ 1) = (\alpha^*\ \ \beta^*). \tag{1.17}$$

The inner product of $|\psi\rangle$ and $\langle\psi|$ is:

$$\langle\psi|\psi\rangle = (\alpha^*\ \ \beta^*) \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = |\alpha|^2 + |\beta|^2, \tag{1.18}$$

which is often set to be 1 to normalize the state.

1.1.3.3 Bloch sphere

The Bloch sphere is a good tool to visualize a qubit. In order to map a qubit to the Bloch sphere, one can first transform α and β by two real parameters θ and ϕ:

$$|\psi\rangle = cos\frac{\theta}{2}\,|0\rangle + e^{i\phi}sin\frac{\theta}{2}\,|1\rangle = \begin{pmatrix} cos\frac{\theta}{2} \\ e^{i\phi}sin\frac{\theta}{2} \end{pmatrix}, \tag{1.19}$$

where $0 \leq \theta \leq \pi$ and $0 \leq \phi \leq 2\pi$.

It is easy to prove that this transformation still satisfies the normalization of the state. After this transformation, all (pure) qubit states can be mapped to the surface of a sphere, which is called the Bloch sphere (Fig. 1.1).

The north and south pole of the Bloch sphere represent the two basis states $|0\rangle$ and $|1\rangle$. Parameter θ portrays the amplitude of the coefficients and ϕ portrays the relative phase of the coefficients. Rather than the computational basis $|0\rangle$ and $|1\rangle$, one can also choose other bases to represent the Hilbert space, for example the two states on the positive and negative side of x axis:

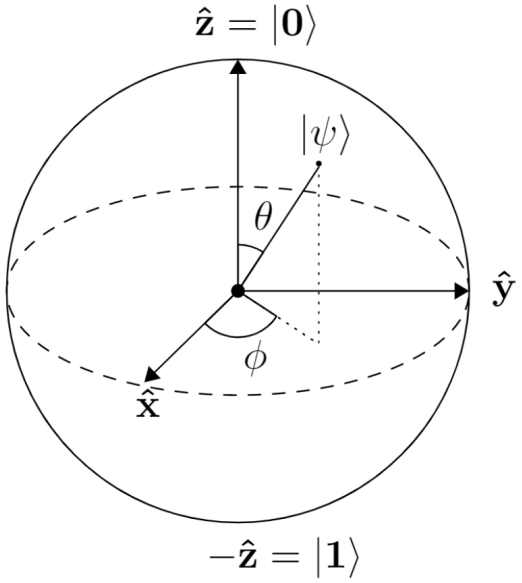


Figure 1.1: The Bloch sphere.

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad ; \quad |-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle), \tag{1.20}$$

and the two states on the positive and negative side of y axis:

$$|+i\rangle = \frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle) \quad ; \quad |-i\rangle = \frac{1}{\sqrt{2}}(|0\rangle - i\,|1\rangle). \tag{1.21}$$

Here they are represented by $|0\rangle$ and $|1\rangle$, but since they are complete bases, they can also represent $|0\rangle$ and $|1\rangle$. For example:

$$|0\rangle = \frac{1}{\sqrt{2}}(|+\rangle + |-\rangle) \quad ; \quad |1\rangle = \frac{1}{\sqrt{2}}(|+\rangle - |-\rangle). \tag{1.22}$$

1.1.4 Operators

As mentioned in Section 1.1.2 before, operators can act on states and transform them into other states. In quantum computing, it's also common to call an operator a *gate*, corresponding to the term used in classical computing. This section will introduce a few important operators which act on a single qubit.

1.1.4.1 Identity operator

The identity operator is often denoted as $\hat{I}$ or simply just I by convenience. The result of its action on a state is still the same state. In the case of a single qubit, one can represent it with basis states $|0\rangle$ and $|1\rangle$:

$$I = |0\rangle\langle 0| + |1\rangle\langle 1| = \begin{pmatrix} 1 \\ 0 \end{pmatrix}(1\ \ 0) + \begin{pmatrix} 0 \\ 1 \end{pmatrix}(0\ \ 1) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{1.23}$$

For simplicity, operators in the rest of this thesis will mainly use the matrix representation of the computational basis. It's easy to observe that the matrix representation of the identity operator is just the identity matrix, which acts on any vectors trivially.

1.1.4.2 Pauli operators

Pauli operators are very important in qubit computation. They can be written as:

$$X = |0\rangle\langle 1| + |1\rangle\langle 0| = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \tag{1.24}$$

$$Y = -i\,|0\rangle\langle 1| + i\,|1\rangle\langle 0| = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \tag{1.25}$$

$$Z = |0\rangle\langle 0| - |1\rangle\langle 1| = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{1.26}$$

X, Y, Z are also often written as $\sigma_1, \sigma_2, \sigma_3$. Identity operator I is sometimes written as σ_0.

The eigenvalues for all Pauli operators are either 1 or -1, and the eigenstates for each Pauli operator are $|+\rangle$ and $|-\rangle$ for X, $|+i\rangle$ and $|-i\rangle$ for Y and $|0\rangle$ and $|1\rangle$ for Z. Therefore, the spectrum decompositions of Pauli operators are:

$$\begin{aligned} X &= |+\rangle\langle +| - |-\rangle\langle -|\,, \\ Y &= |+i\rangle\langle +i| - |-i\rangle\langle -i|\,, \\ Z &= |0\rangle\langle 0| - |1\rangle\langle 1|\,. \end{aligned} \tag{1.27}$$

Based on the chosen basis, the form of these Pauli operators can actually transform from each other. For example if one choose $|+\rangle$ and $|-\rangle$ to be the basis:

$$|+\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad ; \quad |-\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \tag{1.28}$$

then the matrix representation of X in this basis will become:

$$X = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \tag{1.29}$$

while Z will become:

$$Z = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}. \tag{1.30}$$

Besides what's discussed above, Pauli operators also have the following properties:

Pauli operators are Hermitian:

$$X = X^\dagger, \qquad Y = Y^\dagger, \qquad Z = Z^\dagger. \tag{1.31}$$

The inverse of Pauli operators are themselves:

$$X^2 = Y^2 = Z^2 = I. \tag{1.32}$$

Pauli operators anti-commute with each other:

$$\{X, Y\} = \{Y, Z\} = \{Z, X\} = O, \tag{1.33}$$

where $\{A, B\} = AB + BA$ is defined as the anticommutor of the two operators A and B, and O is the empty operator, which acts on any state and results in nothing.

The product of two different Pauli operators will give the third Pauli operator:

$$XY = iZ, \qquad YZ = iX, \qquad ZX = iY. \tag{1.34}$$

The three Pauli operators, along with the identity operator, form a complete basis for all operators in the single-qubit Hilbert space $\mathbb{H}^2$. Therefore, any single-qubit operator A can be represented as a linear combination of Pauli operators and the identity operator:

$$A = a_0 I + a_1 X + a_2 Y + a_3 Z, \tag{1.35}$$

where a_i are complex coefficients. If all a_i are real, then the operator is a Hermitian operator.

Using Pauli operators as generators, one can get the single-qubit Pauli group $\mathcal{P}_1$:

$$\mathcal{P}_1 = \langle X, Y, Z \rangle = \{cf | f \in \{I, X, Y, Z\}, c \in \{\pm 1, \pm i\}\}. \tag{1.36}$$

1.1.4.3 Other common operators

The Hadamard operator (gate) H can be obtained by adding the X and Z gate:

$$H = \frac{1}{\sqrt{2}}(X + Z) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}. \tag{1.37}$$

The Hadamard gate can transform states between bases $|0\rangle$, $|1\rangle$ and $|+\rangle$, $|-\rangle$, and is often used to create superposition at the beginning of a quantum algorithm.

The Z gate gives a phase factor -1 to the state $|1\rangle$. With similar effects, the S gate and the T gate respectively give a phase factor i and $exp(i\pi/4)$ to the state $|1\rangle$:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \qquad T = \begin{pmatrix} 1 & 0 \\ 0 & exp(i\pi/4) \end{pmatrix}. \tag{1.38}$$

They also have the relation:

$$Z = S^2, \qquad S = T^2. \tag{1.39}$$

1.1.5 Composite system

1.1.5.1 States in a composite system

Consider two qubits A and B. The bases for A and B are $\{|0\rangle_A, |1\rangle_A\}$ and $\{|0\rangle_B, |1\rangle_B\}$, separately. Since these two bases can represent all pure states for qubit A and B:

$$\begin{aligned} |\psi\rangle_A &= a_0 |0\rangle_A + a_1 |1\rangle_A , \\ |\psi\rangle_B &= b_0 |0\rangle_B + b_1 |1\rangle_B , \end{aligned} \tag{1.40}$$

then consider the composite system AB. Based on the axiom in Section 1.1.1, any product states in the system AB can be written as:

$$\begin{aligned} |\psi\rangle_A \otimes |\psi\rangle_B =& a_0 b_0 |0\rangle_A \otimes |0\rangle_B + a_0 b_1 |0\rangle_A \otimes |1\rangle_B \\ &+ a_1 b_0 |1\rangle_A \otimes |0\rangle_B + a_1 b_1 |1\rangle_A \otimes |1\rangle_B . \end{aligned} \tag{1.41}$$

Therefore, one can find that the basis for composite system AB can simply be the tensor product of the bases in system A and B:

$$\{|i\rangle_A \otimes |j\rangle_B\} = \{|i\rangle_A |j\rangle_B\} = \{|ij\rangle\}, \qquad i, j = 0, 1, \tag{1.42}$$

where the right two forms are usually used for simplicity.

By the definition of tensor product in linear algebra, it's also easy to write the vector form of the composite system basis:

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad |01\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad |10\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad |11\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}. \tag{1.43}$$

One can generalize this to an n-qubit system, where the basis is:

$$\{|i\rangle^{\otimes n} | i = 0, 1\} = \{\underbrace{|00\cdots0\rangle, |00\cdots1\rangle, \cdots, |11\cdots1\rangle}_{2^n}\}, \tag{1.44}$$

which can be written in the vector form in a 2^n vector space.

1.1.5.2 Operators in the composite system

Similar to states, operators in the composite system can be written as a (sum of) tensor product of operators in each subsystem. Using the two qubit system AB as an example, and consider a CNOT gate which flips qubit B (target qubit) if the state of qubit A (control qubit) is $|1\rangle$ and acts trivially if the state of control qubit is $|0\rangle$. It can be written as:

$$CNOT = |0\rangle_A \langle 0|_A \otimes I_B + |1\rangle_A \langle 1|_A \otimes X_B. \tag{1.45}$$

Expressing everything above with the linear algebra representation, the matrix form of the CNOT gate is:

$$CNOT = \begin{pmatrix} 1 \\ 0 \end{pmatrix} (1 \;\; 0) \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} (0 \;\; 1) \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{1.46}$$

For n-qubit systems, Pauli operators can be generalized as n-fold tensor products of single-qubit Pauli operators, and they form the n-qubit Pauli group $\mathcal{P}_n$:

$$\mathcal{P}_n = \{cf_1 \otimes f_2 \cdots \otimes f_n | f_j \in \{I, X, Y, Z\}, c \in \{\pm 1, \pm i\}\}. \tag{1.47}$$

For elements in the n-qubit Pauli group, it's common to omit the tensor product sign $\otimes$ and even the identity operators which act on some qubits to simplify the expression. For example, the below expressions equivalently denote an n-qubit operator where Z operator acts on the second and the third qubit:

$$\underbrace{I \otimes Z \otimes Z \cdots \otimes I}_{n} = IZZ \cdots I = Z_2 Z_3. \tag{1.48}$$

1.2 Quantum Error Correction

Based on the content introduced in the last section about quantum computing, this section introduces the concept of quantum error correction (QEC), some basic examples of QEC, the stabilizer formalism and the surface code.

1.2.1 Classical error correction

Basically, the strategy used in classical error correction is straightforward: encode one bit redundantly by copying the initial bit several times, such that a few errors occurring in the copied bits won't affect the encoded information overall.

The simplest classical error correction code is the three-bit repetition code. It simply encodes bits by repeating them:

$$0_L = 000; \qquad 1_L = 111. \tag{1.49}$$

Here 0_L and 1_L are the encoded bits, which are also called *codewords*.

Only bit-flip errors could happen on classical bits. Consider a bit-error happens on the second bit, and transforms the codewords as follows:

$$0_L \to 010; \qquad 1_L \to 101. \tag{1.50}$$

Now one observes the bit string after the error happening, and finds that the second bit has a different value. When the error probability is not large, it's natural to conclude that the error happened on the second bit, since it has a different value

from the other two bits. Then, to correct the error, one manually flips the second bit again, and the bit string is corrected from the bit-flip error and returns to the correct codeword.

Consider the *physical error probability* p for each of the three bits. The probability for one bit getting flipped is $3p(1-p)^2$, and the probabilities for two bits getting flipped is $3p^2(1-p)$. Therefore, if $3p(1-p)^2 > 3p^2(1-p)$, which is $p < 1/2$, it is more reasonable to consider that the error bit is the one which has a different bit value rather than the other two bits which have the same bit values.

Hence, as long as only one error occurs on these three bits, one can always correct it. Then, the *logical failure rate* P_f for the encoded bit would be the sum of probabilities where two errors or three errors happen on the bit string: $p_L = 3p^2(1-p) + p^3$. If $p_L < p$, which leads to $p < 1/2$, the three-bit repetition code will reduce the error probability.

1.2.2 Challenges of quantum error correction

It's not possible to correct errors on qubits with the same strategy used in classical error correction. The major challenges of QEC are:

1. Qubits can not be copied. By the no-cloning theorem [1], there doesn't exist a unitary operator which can transform a state $|\phi\rangle$ into an arbitrary unknown state $|\psi\rangle$: $U|\phi\rangle|\psi\rangle \overset{\times}{=} c|\psi\rangle|\psi\rangle$. For classical error correction, the essential part of the encoding strategy is copy.
2. Qubits lose superposition after measurements. After measuring a quantum state, it will collapse into an eigenstate of the measurement observable. For example, after measuring Z on $\alpha|0\rangle + \beta|1\rangle$, the state collapses into either $|0\rangle$ or $|1\rangle$ and thus loses its superposition.
3. Qubits have infinite possible errors. Unlike classical bits, which only have bit-flip errors, any single-qubit operator can be an error acting on a qubit. Therefore, there is a lot more to consider for quantum errors.

The study of QEC is to tackle these challenges and find a way to correct quantum errors while protecting the encoded information. Following a different strategy from classical encoding and using the quantum properties of qubits, different kinds of QEC codes are created for correcting quantum errors. The next section will start from the simplest QEC code, and then introduce some definitions in QEC.

1.2.3 Three-qubit repitition code

The three-qubit repitition code is the simplest QEC code. Here we introduce two types of three-qubit codes, that one can only correct for a single bit-flip (X) error and the other can only correct for a single phase-flip (Z) error. However, by combining these two types of codes together, it can correct any possible single-qubit errors. We start with the three-qubit bit-flip code.

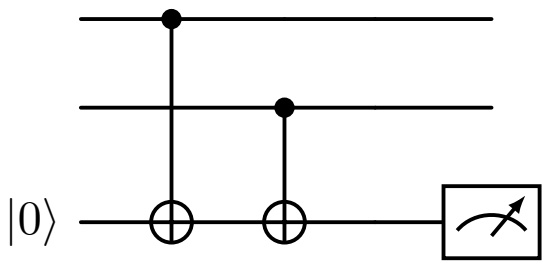


Figure 1.2: Circuit diagram for the parity measurement.

1.2.3.1 Three-qubit bit-flip code

Consider an arbitrary single-qubit state $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$. In order to protect it from bit-flip (X) errors, one encodes it with three qubits as:

$$|\phi\rangle_3 = \alpha|000\rangle + \beta|111\rangle . \tag{1.51}$$

Note that the encoded (logical) state is not a repeated clone of state $|\phi\rangle$, which is $|\phi\rangle^{\otimes 3} = (\alpha|0\rangle + \beta|1\rangle)^{\otimes 3}$.

Here $|0\rangle_L = |000\rangle$ and $|1\rangle_L = |111\rangle$ are called the *codeword basis states*. The state space spanned by the codeword basis states is called the *codespace*. Any error-free logical states fall into the codespace.

Error action
Consider an X error happens on the second qubit and transforms the logical state into:

$$X_2|\phi\rangle_3 = (I \otimes X \otimes I)(\alpha|000\rangle + \beta|111\rangle) = \alpha|010\rangle + \beta|101\rangle . \tag{1.52}$$

Error detection
It's not possible to measure a single qubit in the logical state, since the superposition will be destroyed after the measurement. For example, if one measures Z for the second qubit and gets the value 1, the logical state will collapse from $\alpha|010\rangle + \beta|101\rangle$ to $|010\rangle$, and it's impossible to get the coefficients back from this state anymore.

However, one can do the so called *parity measurement*, by adding an ancillary qubit and using CNOT gates as shown in Fig. 1.2. In the circuit diagram, each line represents a single-qubit state; the dots and cross circles respectively represent the control parts and target parts of CNOT gates, and the meter represents the measurement of the ancillary qubit in the computational basis (measurement of Z).

This parity measurement is equivalent to measuring the observable $Z \otimes Z$ on the first two qubits:

$$\begin{aligned} (Z \otimes Z)|00\rangle &= |00\rangle , \qquad (Z \otimes Z)|11\rangle = |11\rangle , \\ (Z \otimes Z)|01\rangle &= -|01\rangle , \qquad (Z \otimes Z)|10\rangle = -|10\rangle . \end{aligned} \tag{1.53}$$

It gives the same result for the parity measurement, where the even parity is associated with eigenvalue +1, and the odd parity is associated with eigenvalue -1.

Therefore, in the three-qubit bit-flip code, one performs parity measurement Z_1Z_2 and Z_2Z_3 on the logical qubit. Given that the logical qubit after X_2 error is $|\phi\rangle_{3,err} = \alpha|010\rangle + \beta|101\rangle$, the result of the parity measurement is -1 for both Z_1Z_2 and Z_2Z_3:

$$\begin{aligned} Z_1Z_2|\phi\rangle_{3,err} &= -|\phi\rangle_{3,err}\,, \\ Z_2Z_3|\phi\rangle_{3,err} &= -|\phi\rangle_{3,err}\,. \end{aligned} \tag{1.54}$$

The important thing is that these measurements won't destroy the superposition of the logical qubit, since $|\phi\rangle_{3,err}$ is an eigenstate for both Z_1Z_2 and Z_2Z_3.

Error decoding
The result of the parity measurements is called a *syndrome*. If the logical state stays in the codespace, all syndromes should be +1; if a syndrome is -1, this syndrome is said to be lighted or lit. Here two syndromes are both lighted, which means the second qubit has a different value from the other two. Two types of errors X_2 and X_1X_3 correspond to these syndrome values. If the error probability for each qubit is the same and not too large, it's more likely that the error is X_2 than X_1X_3.

Error correction
After getting the conclusion that most likely an X error happens on the second qubit, one caan just apply an X gate to the second qubit to correct the error state back to the initial state:

$$X_2|\phi\rangle_{3,err} = (I\otimes X\otimes I)(\alpha|010\rangle + \beta|101\rangle) = \alpha|000\rangle + \beta|111\rangle\,. \tag{1.55}$$

1.2.3.2 Three-qubit phase-flip code

A phase-flip error Z gives an additional π phase to $|1\rangle$, but it also flips $|+\rangle$ and $|-\rangle$ with each other:

$$\begin{aligned} Z|0\rangle &= |0\rangle \qquad & Z|1\rangle &= -|1\rangle\,; \\ Z|+\rangle &= |-\rangle \qquad & Z|-\rangle &= |+\rangle\,. \end{aligned} \tag{1.56}$$

Therefore, using a similar strategy, the encoded qubit of the three-qubit phase-flip code is given by:

$$|\phi\rangle_3 = \alpha|+++\rangle + \beta|---\rangle\,. \tag{1.57}$$

By analysing the measurement results of X_1X_2 and X_2X_3, one can correct a single Z errors by applying another Z gate to the same qubit where error occurs.

1.2.3.3 Nine-qubit Shor code

The nine-qubit Shor code [2] combines the three-qubit bit-flip and phase-flip codes into one code:

$$|\phi\rangle_9 = \alpha\,|+\rangle_3\,|+\rangle_3\,|+\rangle_3 + \beta\,|-\rangle_3\,|-\rangle_3\,|-\rangle_3\,, \tag{1.58}$$

where

$$\begin{aligned} |+\rangle_3 &= \frac{|000\rangle + |111\rangle}{\sqrt{2}}; \\ |-\rangle_3 &= \frac{|000\rangle - |111\rangle}{\sqrt{2}}. \end{aligned} \tag{1.59}$$

By this combination, the Shor code can correct for single-qubit X,Z and XZ errors. Since Y can be expressed as the product of X and Z, noticing that any single-qubit error can be seen as a rotation of a qubit on the Bloch sphere, and measurements will project it to the discrete X, Y or Z errors, the Shor code can correct for any single-qubit errors.

1.2.4 Stablizer formalism

The three-qubit code, the Shor code, along with the surface code which will be discussed in the next subsection, are all examples of a particular class of QEC codes called the *stabilizer codes*. The stabilizer formalism is a great tool to study these codes. It has clear rules, and can be used to avoid the complex expression for many-qubit states.

Define $\mathcal{H} = (\mathbb{C}^2)^{\otimes n}$ as the Hilbert space of all *data qubits*, which refers to the qubits used to encode the logical qubits in the QEC codes. The codespace $\mathcal{H}_0$ is a subspace of $\mathcal{H}$. It is defined as

$$\mathcal{H}_0 = \{\psi \in \mathcal{H} : g\psi = \psi | \forall g \in \mathcal{G}\}, \tag{1.60}$$

where $\mathcal{G}$ is the *stabilizer group*. It is an Abelian subgroup of the n-qubit Pauli group $\mathcal{P}$, such that for any $g_i, g_j \in \mathcal{G}$, $g_i g_j = g_j g_i$, i.e. $[g_i, g_j] = 0$.

A stabilizer code is defined by a minimal complete set of m generators g_i of $\mathcal{G}$:

$$\mathcal{G} = \langle g_1, g_2, \cdots, g_m \rangle \tag{1.61}$$

and a pair of logical operators $(X_{L,j}, Z_{L,j})$ for each encoded logical qubit j which satisfies

$$X_{L,j}, Z_{L,j} \in \mathcal{C}(\mathcal{G}) \backslash \mathcal{G} \quad \text{and} \quad X_{L,j} Z_{L,j} = -Z_{L,j} X_{L,j}, \tag{1.62}$$

where $\in \mathcal{C}(\mathcal{G})\backslash\mathcal{G}$ means in $\mathcal{C}(\mathcal{G})$ but not in $\mathcal{G}$. By the above definitions, the stabilizer code encodes $k = n - m$ logical qubits with n data qubits. The code distance d is defined as

$$d = \min_{P \in \mathcal{C}(\mathcal{G}) \backslash \mathcal{G}} wt(P), \tag{1.63}$$

where $wt(P)$, called the *weight* of P, is the number of qubits that P acts on non-trivially, where P is some logical operator of this stabilizer code. The stabilizer code

can detect up to $d-1$ errors and correct up to $(d-1)/2$ errors. The code is labeled by $[[n, k, d]]$.

To decode the stabilizer codes, parity measurements are done on $g_1, g_2, \cdots, g_m$. The measurement results are eigenvalues ± 1, and can be described by a syndrome $s = (s_1, s_2, \cdots, s_m)$, where $(-1)^{s_i} = \pm 1$. A recovery operator needs to be given to bring the corrupted state back to the codespace based on the syndrome s.

1.2.5 Surface code

Topological stabilizer codes are a class of stabilizer code where logical qubits are encoded with topological properties. Among all the topological stabilizer codes, the surface code (which contains the planar code with a fixed boundary and the toric code with a periodic boundary) [3, 4, 5] may be the most famous one. A $[[n, 1, d]]$ surface code [5] is defined by a $d \times d$ lattice with $n = d^2 + (d-1)^2$ data qubits. Fig. 1.3 shows an example of $d = 5$ surface code. As one can see from the left figure, stabilizers of the surface code can be classified into two types. The first type is called the plaquette stabilizer, where 4 data qubits and their edges form a square, and the stabilizer will act with Pauli Z operator on these 4 qubits; the second type is called the vertex stabilizer, where 4 data qubits and their edges form a cross, and the stabilizer will act with Pauli X operator on these 4 qubits. On the lattice boundary the stabilizers only have weight 3 instead of 4.

Logical X and Z operators of the surface code can be defined as a vertical line of X operators and a horizontal line of Z operators, as shown in the center and right figures of Fig. 1.3. It is not hard to find out that these logical operators commute with all stabilizers but anti-commute with each other, which satisfies the stabilizer formalism. In Chapter 2 we will introduce some other topological stabilizer codes, but they are all based on the surface code.

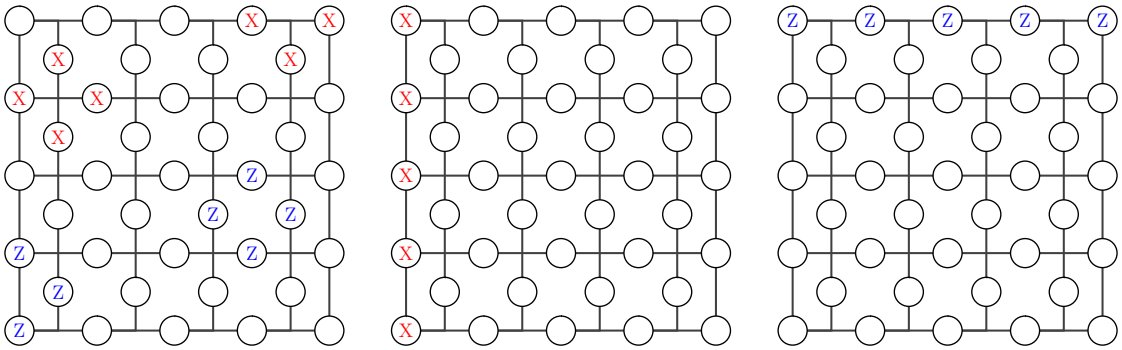


Figure 1.3: Surface code with distance $d = 5$. The left, center and right figures respectively show the stabilizer, logical X operator and logical Z operator of the code.

1.3 Tensor Network

One can think of tensors as high-dimensional arrays. Vectors and matrices can be considered as 1-dimensional and 2-dimensional arrays, while tensors can have more dimensions. Tensors are often denoted by a capital character with several indices, where the number of indices represents their dimension. For example a 3-dimensional tensor can be denoted as T_{ijk}.

However in the tensor algebra, this kind of notation is usually not very convenient, since tensor contractions often need a lot of symbols. Here is an example: $C_{il} = \sum_j \sum_k A_{ijk} B_{jkl}$. By using the *tensor diagram*, things will be a lot easier. Here are two rules for tensor diagrams according to [6]:

1. Tensors are denoted by shapes, and tensor indices are denoted by lines emanating from these shapes.
2. Connecting two index lines implies a contraction, or summation over the connected indices.

Based on these two rules, Fig. 1.4 shows the diagram representation for the vector v_i, the matrix M_{ij} and the tensor T_{ijk}, and Fig. 1.5 shows the diagram representation for tensor contractions $\sum_j M_{ij} v_j$ and $\sum_j T_{ijkl} V_{jm}$.

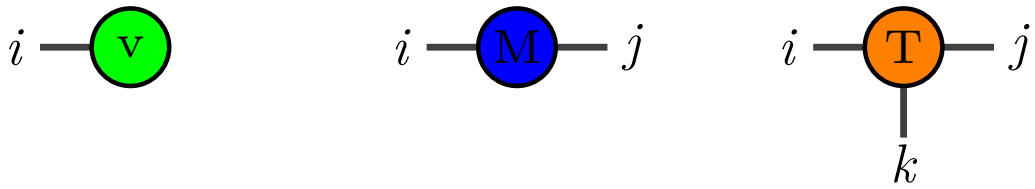


Figure 1.4: Example diagrams for low-order tensors: vector v_i (left), matrix M_{ij} (center) and three-dimensional tensor T_{ijk} (right).

Here, we introduce two special tensor networks which will be later used in Section 2.5. The *matrix product state* (MPS) [7] represents a tensor with N indices by factorizing it into a chain-like product of three-index tensors. A MPS can be expressed in the tensor diagram notation in Fig. 1.6. Alternatively, this MPS can be expressed in traditional expression as

$$T^{ijklm} = \sum_{b_1, b_2, \cdots, b_5} A^i_{b_1} A^j_{b_1 b_2} A^k_{b_2 b_3} A^l_{b_3 b_4} A^m_{b_4}, \tag{1.64}$$

where b_i, called the bond dimensions or virtual dimensions, are contracted, and each tensor A can be different from each other. The inner product of two MPSs in the

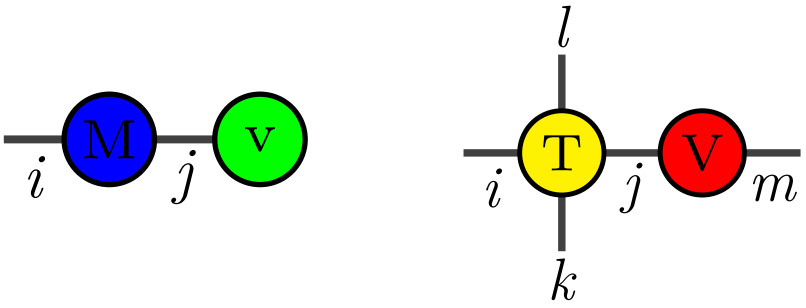


Figure 1.5: Example diagrams for tensor contractions.

same space will result in a number.

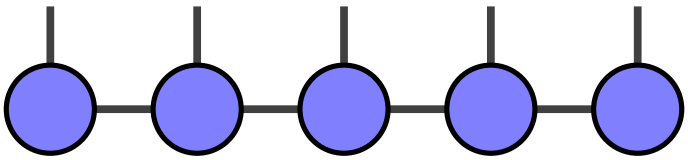


Figure 1.6: Example diagram for MPS.

A *matrix product operator* (MPO) is a tensor network where each tensor has two external indices as well as two internal indices contracted with neighboring tensors in a sequence manner. A MPO can be expressed in the tensor diagram notation in Fig. 1.7 and be expressed in traditional expression as

$$O^{ijklm}_{i'j'k'l'm'} = \sum_{b_1,b_2,\cdots,b_5} A^{i}_{i'b_1} A^{j}_{j'b_1b_2} A^{k}_{k'b_2b_3} A^{l}_{l'b_3b_4} A^{m}_{m'b_4}. \tag{1.65}$$

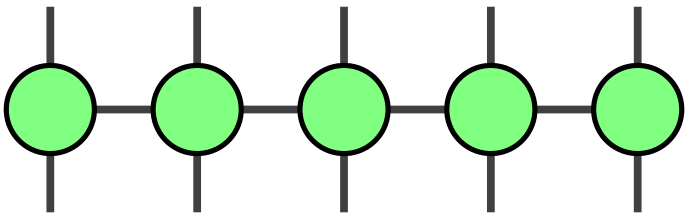


Figure 1.7: Example diagram for MPO.

Intuitively, if one thinks of an MPS as a 'vector' in some high dimensional space, an MPO is the generalization to the case of a 'matrix' acting in the same space. Therefore, when an MPO acts on an MPS in the same space, the result will be a new MPS.

1.4 Simulation

The simulation results and the code construction are based on the python package *qecsim* [8], [9] and its extension [11]. *Qecsim* is a Python3 library for simulating quantum error correction using stabilizer codes, allowing additional definitions of codes, error models and decoders. Based on this package, we develop codes and decoders for the YZZY code and the XYZ^2 code, which are further discussed in Chapter 3. The simulations are run on clusters provided by Chalmers Centre for Computational Science and Engineering (C3SE).

2

Theory

This chapter mainly discusses the theoretical background of this thesis. Section 2.2 introduces the XZZX code, which is a variant of the standard surface code; Section 2.3 then introduces the YZZY code, which is very similar to the XZZX code yet has a connection to the XYZ^2 code which is discussed in Section 2.5. Section 2.4 introduces maximum-likelihood decoders and Section 2.5 introduces the MPS decoder, an approximate maximum-likelihood decoder based on tensor networks.

2.1 XZZX code

Before discussing the XZZX code, it's good to first introduce the *rotated surface code* [10]. The rotated surface code is generated by rotating the code lattice 45° with respect to the grid of the standard surface code introduced in Subsection 1.2.5. After the operation, the resulting code can be presented equivalently in Fig. 2.1, which is an example of the rotated surface code with distance $d = 5$. If we consider a square code, i.e. the number of rows and columns are the same in the code lattice, then $[[n, k, d]] = [[d^2, 1, d]]$ for the rotated surface code.

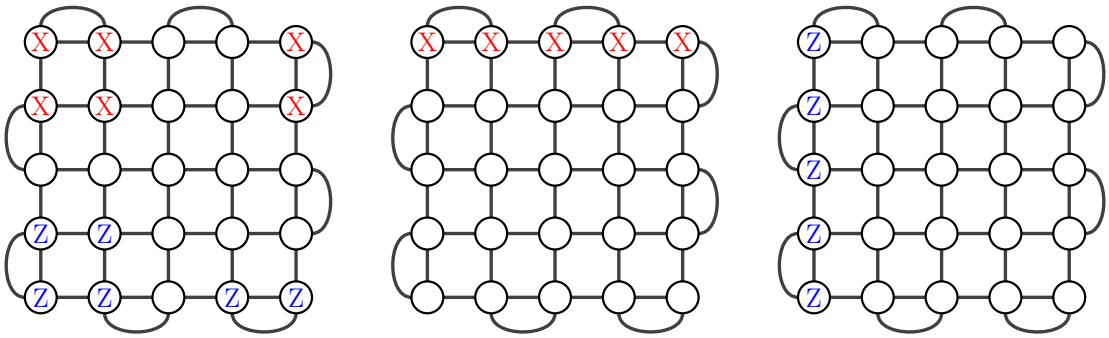


Figure 2.1: Rotated surface code with distance $d = 5$. The left, center and right figures respectively show the stabilizer, logical X operator and logical Z operator of the code.

In the rotated surface code, there exist two kinds of stabilizers: plaquette stabilizers and boundary stabilizers. Plaquette stabilizers are formed with four-qubit operations, which are XXXX and ZZZZ alternating on neighbouring plaquettes. For example, in the left side of Fig. 2.1, the left-top plaquette has a stabilizer XXXX,

and the left-bottom plaquette has a stabilizer ZZZZ. For a distance d rotated surface code, there are $(d-1)^2$ plaquette stabilizers. Boundary stabilizers are formed with two-qubit operations XX and ZZ. They are drawn as arc connections between some neighbouring qubits, and XX(ZZ) boundary stabilizers are next to ZZZZ(XXXX) plaquettes. For a distance d rotated surface code, there are $2(d-1)$ boundary stabilizers. Examples of the logical operator X and Z for the rotated surface code are also shown in Fig. 2.1. It's easy to verify that these operators commute with all stabilizers but anti-commute with each other.

The XZZX code [11, 12] is an alternative code based on the rotated surface code. What it does is simply acting with a Hardmard gate on every alternating qubit. The code parameters remain the same under this transformation, so for the XZZX code it still has $[[n, k, d]] = [[d^2, 1, d]]$, which is the same for the rotated surface code. An example of the XZZX code with code distance $d = 5$ is shown in Fig. 2.2.

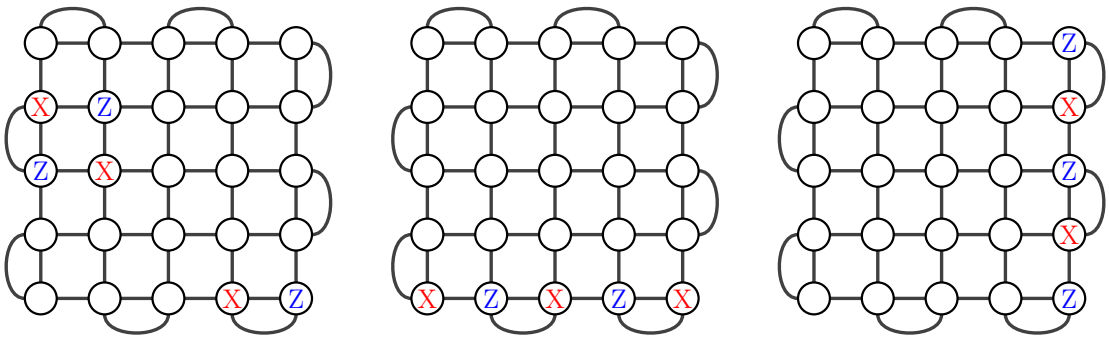


Figure 2.2: XZZX code with distance $d = 5$. The left, center and right figures respectively show the stabilizer, logical X operator and logical Z operator of the code.

The plaquette stabilizers for the XZZX code are identical in every plaquette, as XZZX. The boundary stabilizers can be seen as a 'part of' the plaquette stabilizers, which can be XZ or ZX depending on its position. Logical operators are also shown in Fig. 2.2. Comparing with the rotated surface code, there are just additional Hardamard gates acting on alternating qubits in the code lattice.

2.2 YZZY code

The YZZY code has exactly the same structure as the XZZX code. The only difference is that X and Y operators are exchanged in these two codes. Therefore, all the X operators in stabilizers and logical operators of the XZZX code becomes Y operators in the YZZY code. For the same reason, the X logical operator in the XZZX code becomes the Y operator in the YZZY code after transformation, but the X logical operator in the YZZY code can be easily constructed by multiplying logical Y and Z operators. One example of the YZZY code with $d = 5$ is shown in

Fig. 2.3.

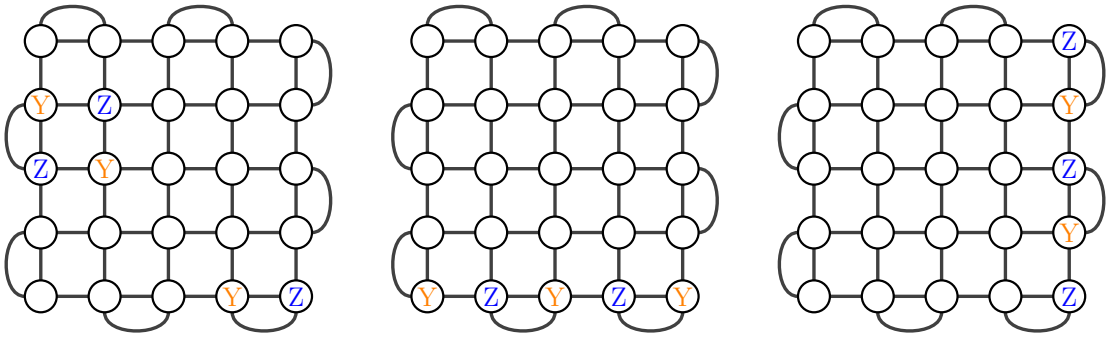


Figure 2.3: YZZY code with distance $d = 5$. The left, center and right figures respectively show the stabilizer, logical Y operator and logical Z operator of the code.

Starting from the YZZY code, one can obtain the XYZ^2 code [13] by doubling the number of qubits with each pair of qubits stabilized by an XX operator.

First consider mapping the qubit states from $|0\rangle$ to $|{+}{+}\rangle$ and $|1\rangle$ to $|{-}{-}\rangle$. Pauli operators acting on the single qubit will be mapped to:

$$\begin{aligned} I &\rightarrow II \ or \ XX, \\ X &\rightarrow ZZ \ or \ YY, \\ Y &\rightarrow YZ \ or \ ZY, \\ Z &\rightarrow XI \ or \ IX. \end{aligned} \tag{2.1}$$

Then, consider a YZZY plaquette. After this mapping it will become a weight-six stabilizer XYZXYZ. The correlating picture is shown in Fig. 2.4. Applying this mapping to the entire YZZY code, one then obtains the XYZ^2 code. Compared with the YZZY code which has $[[n, k, d]] = [[d^2, 1, d]]$, XYZ^2 code has $[[n, k, d]] = [[2d^2, 1, d]]$, since the number of qubits has been doubled, while the logical Z operator is a pure X chain with length d across the code.

2.3 XYZ^2 code

The XYZ^2 code is a simplest version of the 'matching codes' presented by Wootton [14, 15]. The matching codes are formed on hexagonal lattices which include link stabilizers between pairs of vertices, where each vertex uniquely matches another vertex. As discussed in the last Section, the XYZ^2 code can be seen as a transformation from the YZZY code. This is also the decoding strategy that we used, and its details will be discussed in Section 3.2.6. One example of the XYZ^2 code with $d = 3$ is shown in Fig. 2.5.

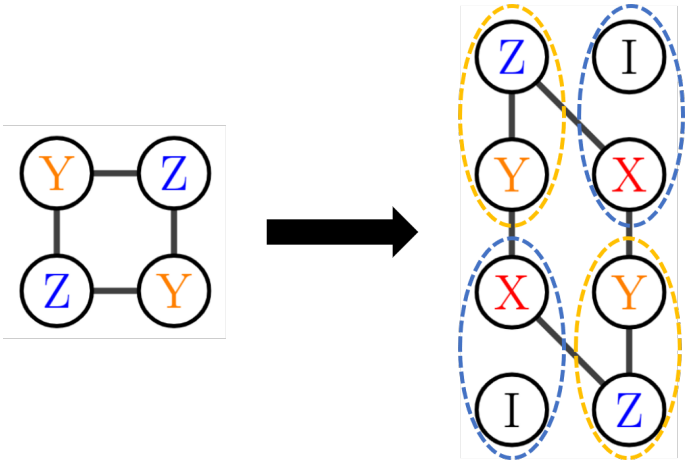


Figure 2.4: Transformation from the YZZY code to the XYZ^2 code.

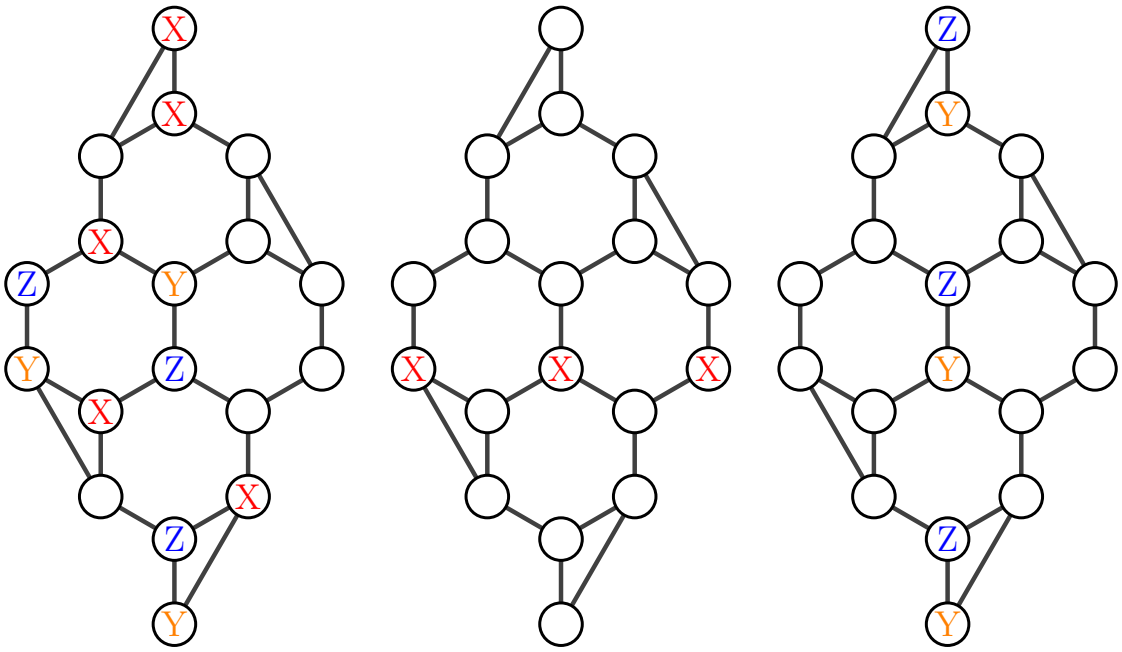


Figure 2.5: XYZ^2 code with distance $d = 3$. The left, center and right figures respectively show the stabilizer, logical X operator and logical Z operator of the code.

The stabilizers in the XYZ^2 code can be classified into three types. For every hexagonal plaquette there is a weight-six XYZXYZ stabilizer; for each vertical link there is a weight-two XX stabilizer; at the code boundary, there exists boundary XYZ stabilizers. Therefore for $2d^2$ size code, there are $(d-1)^2$ plaquette stabilizers, d^2 link stabilizers and $2(d-1)$ boundary stabilizers. In total there are $2d^2-1$ stabilizers for constraint, so the XYZ^2 code encodes one logical qubit.

The logical X operator is defined as an X chain horizontally crossing the bottom qubits on the hexagonal plaquette, therefore for $2d^2$ size code, the distance for logical X operator is d. The logical Z operator is defined as a ZY chain vertically crossing the link qubits, and its distance is $2d$ for $2d^2$ size code.

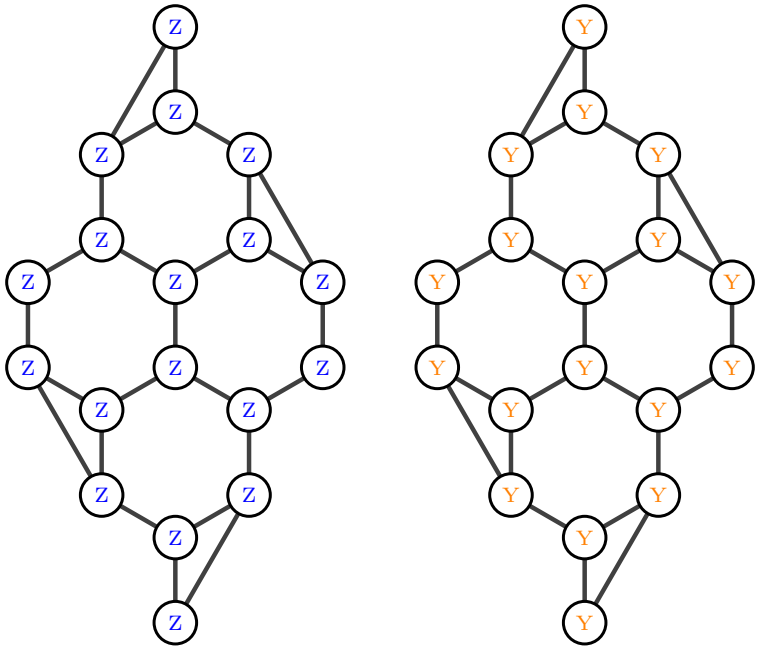


Figure 2.6: XYZ^2 code with distance $d = 3$. These two figures show that the logical Y operator in the XYZ^2 code can be expressed by pure Z or Y operators on every data qubit.

The logical Y operator can be expressed as a product of logical X and Z operators. However, in the XYZ^2 code one can use stabilizers to form the logical Y operator as pure Z or Y operations on all data qubits, as shown in Fig. 2.6. Comparing Fig. 2.5 and Fig. 2.6, it's easy to see that the pure Z operator acting on all data qubits anti-commutes with the logical X and Z operators. At the same time, one can prove that it commutes with all stabilizers in the code, therefore it's indeed the logical Y operation. Applying all the link stabilizers XX to it, one can get the other form of the logical Y operator composed by only Y operators. Note that only when Z or Y operators are acting on all data qubits can they become logical operators. This leads to the conclusion that under the pure Z or Y noise, the distance of the XYZ^2 code is $2d^2$.

Apart from defining the vertical link stabilizer to be XX, one can also choose YY and ZZ to be the link stabilizer, but the plaquette and boundary stabilizers need to be changed as well. Fig. 2.7 shows these two variations of the XYZ^2 code. Actually these two variations can be considered as mappings from the YZZY code by replacing $|0\rangle$ to $|+i+i\rangle$ and $|1\rangle$ to $|-i-i\rangle$ for the XYZ^2 code with YY link stabilizers and replacing $|0\rangle$ to $|00\rangle$ and $|1\rangle$ to $|11\rangle$ for the XYZ^2 code with ZZ link stabilizers. Respectively, the Pauli operations acting on the single qubit will be mapped to:

$$\begin{aligned} I &\rightarrow II \ or \ YY, \\ X &\rightarrow ZZ \ or \ XX, \\ Y &\rightarrow ZX \ or \ XZ, \\ Z &\rightarrow YI \ or \ IY \end{aligned} \tag{2.2}$$

for the case of YY link stabilizers and:

$$\begin{aligned} I &\rightarrow II \ or \ ZZ, \\ X &\rightarrow XX \ or \ YY, \\ Y &\rightarrow XY \ or \ YX, \\ Z &\rightarrow ZI \ or \ IZ \end{aligned} \tag{2.3}$$

for the case of ZZ link stabilizers.

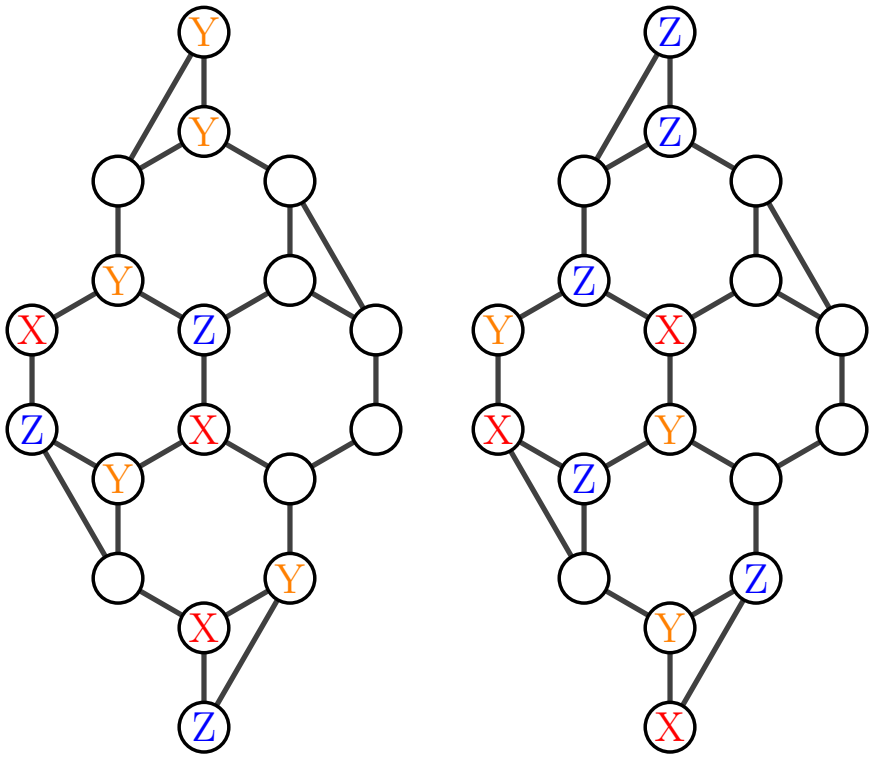


Figure 2.7: Variations of the XYZ^2 code with distance $d = 3$. Left and right figure respectively has the link stabilizers YY and ZZ.

As shown in [13], the XYZ^2 code has a unique syndrome direction for isolated X, Y and Z errors because of the structure of its hexagonal plaquette. Compared with

the XZZX code, the XYZ^2 code has a higher threshold for biased-error model and a lower sub-threshold logical failure rate under Y-bias noise with the same code size, where the threhold and logical failure rate are used for performance evaluation, and they are defined in the next section. With the same number of data qubits, the XYZ^2 code needs fewer auxiliary qubits for stabilizer measurements than the XZZX code, yet the weight-six plaquette stabilizers also need a better connectivity between data qubits.

2.4 Maximum-likelihood decoder

This section basically follows the statement in [16]. Consider a QEC code with size n. Define $\mathcal{H} = (\mathbb{C}^2)^{\otimes n}$ as the Hilbert space for all data qubits in the code and $\mathcal{P}$ the n-qubit Pauli group:

$$\mathcal{P}_n = \{cf_1 \otimes f_2 \cdots \otimes f_n | f_j \in \{I, X, Y, Z\}, c \in \{\pm 1, \pm i\}\}. \tag{2.4}$$

The stabilizers of the code belong to the stabilizer group $\mathcal{G}$, where $\mathcal{G} \subset \mathcal{P}$ and $-I \notin \mathcal{G}$. Since any state in the codespace should remain invariant under the action of any stabilizer, the codespace $\mathcal{H}_0$ is defined as:

$$\mathcal{H}_0 = \{\psi \in \mathcal{H} : g\psi = \psi | \forall g \in \mathcal{G}\}. \tag{2.5}$$

Then consider a Pauli noise which can be described by a linear map $\rho \to \mathcal{N}(\rho)$:

$$\mathcal{N}(\rho) = \sum_{f \in \mathcal{P}} P(f) f \rho f^\dagger, \tag{2.6}$$

where ρ and $\mathcal{N}(\rho)$ are the density matrix of code states before and after the action of the Pauli noise, and P is the probability distribution of the Pauli group. Because the initial code state is in the codespace $\mathcal{H}_0$, the density matrix should satisfy $g\rho g^\dagger = \rho$ for any $g \in \mathcal{G}$. Therefore, when $f\mathcal{G} = h\mathcal{G}$, one has $f\rho f^\dagger = h\rho h^\dagger$, where $f\mathcal{G} \equiv \{fg | g \in \mathcal{G}\}$ is called a *left coset* of $\mathcal{G}$ in the language of group theory. This is because when $f\mathcal{G} = h\mathcal{G}$, one can find a $g_0 \in \mathcal{G}$, so that $f = hg_0$. It's easy to show:

$$f\rho f^\dagger = (hg_0)\rho(hg_0)^\dagger = h(g_0 \rho g_0^\dagger)h^\dagger = h\rho h^\dagger. \tag{2.7}$$

Therefore, when $f = hg_0$, f and h are both elements in this coset. One can prove that $\mathcal{P}$ is a disjoint union of cosets $\mathcal{C}_i = f_i\mathcal{G}$, where f_i is a chosen element in $\mathcal{C}_i$. By classifying errors into different cosets which has the same action on the codespace, one can reform Equation 2.6 into:

$$\mathcal{N}(\rho) = \sum_i P(f_i\mathcal{G}) f_i \rho f_i^\dagger, \tag{2.8}$$

where the sum goes over all cosets of $\mathcal{G}$ and $P(f\mathcal{G})$ is the sum of probability distribution of all Pauli operators in the coset, which we define as the *coset probability*:

$$P(f\mathcal{G}) = \sum_{g \in \mathcal{G}} P(fg). \tag{2.9}$$

What the maximum-likelihood decoder (MLD) does is to calculate every coset probability based on the syndrome after measuring all stabilizers in the code, and pick one representative operator f from the coset which has the largest coset probability. As long as the actual error e is contained in the picked coset, the decoding is successful, since f can be expressed as a product of the actual error and a stabilizer: $f = eg$. Because the decoder always picks the coset with the largest coset probability as the answer, it is called the maximum-likelihood decoder.

More precisely, consider $\{g_1, \cdots, g_m\}$ as generators of the stabilizer group $\mathcal{G}$. These generators commute with each other, and measurements of these generators will give results of eigenvalues $g_i = \pm 1$, which can be described by a syndrome $s_i \in \{0, 1\}$, such that $g_i = (-1)^{s_i}$, as discussed in Section 1.2.4. Therefore, by measuring all the generators one can get a syndrome vector $s = \{s_1, \cdots, s_m\} \in \{0, 1\}^m$. Considering all generators to be independent, there will be 2^m different configurations for the syndrome vector. The full Hilbert space $\mathcal{H}$ can be decomposed into a direct sum of subspaces with different syndrome configurations:

$$\mathcal{H} = \bigoplus_{s \in \{0,1\}^m} \mathcal{H}_s, \tag{2.10}$$

where $\mathcal{H}_s = \{\psi \in \mathcal{H} : g_i\psi = (-1)^{s_i}\psi | i = 1, \cdots, m\}$, and the codespace $\mathcal{H}_0$ is just the subspace where all syndromes are zero: $s_i = 0 | i = 1, \cdots, m$. Define that a Pauli operator $f \in \mathcal{P}$ has syndrome s iff $fg_i = (-1)^{s_i} g_i f$ for all $i = 1, \cdots, m$. That is to say, when the Pauli operator commutes with a stabilizer, the stabilizer will give a 0 syndrome; when it anti-commutes with a stabilizer, the stabilizer will give a 1 syndrome. One can check that for each Pauli operator which has syndrome s, it has to be in the coset $f(s)\mathcal{C}(\mathcal{G})$, where $f(s)$ is a representative operator with syndrome s, and $\mathcal{C}(\mathcal{G})$ is called the centralizer of $\mathcal{G}$:

$$\mathcal{C}(\mathcal{G}) = \{f \in \mathcal{P} : fg = gf | \forall g \in \mathcal{G}\}. \tag{2.11}$$

It's easy to show that $\mathcal{G} \subset \mathcal{C}(\mathcal{G})$, since all stabilizers commute with each other. Not rigorously speaking, one can consider $\mathcal{C}(\mathcal{G})$ as the direct sum of the stabilizer group $\mathcal{G}$ and the logical operators, since logical operators also commute with all stabilizers. Consider a stabilizer code with one logical qubit, where $X_L, Y_L, Z_L \in \mathcal{C}(\mathcal{G}) \backslash \mathcal{G}$ are the logical operators. Each coset of $\mathcal{C}(\mathcal{G})$ can be decomposed into four disjoint cosets of $\mathcal{G}$:

$$f(s)\mathcal{C}(\mathcal{G}) = \mathcal{C}_I(s) \cup \mathcal{C}_X(s) \cup \mathcal{C}_Y(s) \cup \mathcal{C}_Z(s), \tag{2.12}$$

where

$$\begin{aligned} \mathcal{C}_I(s) = f(s)\mathcal{G}, \quad \mathcal{C}_X(s) = f(s)X_L\mathcal{G}, \\ \mathcal{C}_Y(s) = f(s)Y_L\mathcal{G}, \quad \mathcal{C}_Z(s) = f(s)Z_L\mathcal{G}. \end{aligned} \tag{2.13}$$

After the measurements of all independent stabilizer generators defined in the code, there will be a syndrome representing some information of the error, and the error state $\mathcal{N}(\rho)$ will be projected to the corresponding syndrome subspace $\mathcal{H}_s$. The syndrome can only give the information that the error e is contained in the coset

$f(s)\mathcal{C}(\mathcal{G})$. To successfully correct the error, one also needs to determine in which $\mathcal{C}_P(s)$, where $P = I, X, Y, Z$ the error is contained. Using Equation 2.8, 2.12 and 2.13, one can write the corrupted code state after syndrome measurements as:

$$\begin{aligned}\rho(s) =& P(\mathcal{C}_I(s))f(s)\rho f(s) + P(\mathcal{C}_X(s))f(s)X_L\rho X_L f(s) \\ &+ P(\mathcal{C}_Y(s))f(s)Y_L\rho Y_L f(s) + P(\mathcal{C}_Z(s))f(s)Z_L\rho Z_L f(s),\end{aligned} \tag{2.14}$$

where we assume all errors and logical operators are Hermitian for simplicity, yet the equation will have a similar form if they are not. From this equation, the effective noise model on the observed syndrome can be considered as applying an error from $f(s), f(s)X_L, f(s)Y_L, f(s)Z_L$ with probability $P(\mathcal{C}_I(s)), P(\mathcal{C}_I(s)), P(\mathcal{C}_Y(s)), P(\mathcal{C}_Z(s))$. Therefore, the best strategy for decoding is to assume the occurring error as the most likely among these four errors, i.e. the error which has the largest probability to happen. Since Pauli operators are unitary, it's equivalent to choosing one operator in the most likely coset, denoted as $\mathcal{C}_{ML}$ within four cosets $\mathcal{C}_I(s), \mathcal{C}_X(s), \mathcal{C}_Y(s), \mathcal{C}_Z(s)$, as the recovery operator. The decoding strategy can be summarised in pseudocode as follows:

Algorithm 1 Maximum-Likelihood Decoder

Input: syndrome $s \in \{0,1\}^m$
Output: recovery operator $r \in \mathcal{P}$
$f(s) \leftarrow$ one Pauli operator which has syndrome s
$P(\mathcal{C}_I(s) \leftarrow \sum_{g\in\mathcal{G}} P(f(s)g)$
$P(\mathcal{C}_X(s) \leftarrow \sum_{g\in\mathcal{G}} P(f(s)X_L g)$
$P(\mathcal{C}_Y(s) \leftarrow \sum_{g\in\mathcal{G}} P(f(s)Y_L g)$
$P(\mathcal{C}_Z(s) \leftarrow \sum_{g\in\mathcal{G}} P(f(s)Z_L g)$
$\mathcal{C}_{ML} \leftarrow \mathcal{C} \in \{\mathcal{C}_I(s), \mathcal{C}_X(s), \mathcal{C}_Y(s), \mathcal{C}_Z(s)\}$ with maximum $P(\mathcal{C})$
$r \leftarrow$ one Pauli operator in $\mathcal{C}_{ML}$

After applying the recovery operator to the corrupted code state, the final state after decoding will be $r\rho(s)r$. The MLD can correctly find the coset of $\mathcal{G}$ where the actual error is in and correct the error with probability

$$P_{success} = \sum_{s\in\{0,1\}^m} P(\mathcal{C}_{ML}(s)). \tag{2.15}$$

Performance of the stabilizer codes and decoders can be evaluated by the *logical failure rate* P_f of the codes, which is the probability that the decoder fails in correcting errors. It's easy to see that $P_f = 1 - P_{success}$. Naturally codes and decoders with lower logical failure rate have better performance.

When plotting the logical failure rate P_f versus the *error probability* p (the probability that errors can act on the code, detailed definition in Subsection 3.1.1) of the same code with several different code distances, there will be a cross point of the curves. Let the error rate of the cross point be p_c. When $p < p_c$, P_f will become smaller as code distance becomes larger; when $p > p_c$, P_f will become larger as code distance becomes larger. This is because when increasing the code distance, the code can be more robust against errors; but it also makes the decoder harder

choosing a suitable recovery operator to correct the code. Actually the error rate of the cross point is called the *threshold*, and it can also be used for performance evaluation. Codes and decoders with larger threshold have better performance.

2.5 Matrix product states-based decoder

Exact MLD is nearly unusable as code size goes larger. Therefore, approximate MLDs are used in practice. The matrix product states-based decoder presented by Bravyi, Suchara and Vargo [16] is an approximate MLD. In the following content we will call it the MPS decoder for convenience. The key idea is that the form of coset probabilities can be constructed as a tensor network defined by the code structure. During the contraction of the tensor network, one can do a truncation to approximate the calculation results. There is also another type of approximate MLD which is based on Metropolis-based Monte Carlo sampling [17, 18, 19]

This section only briefly discusses how the MPS decoder creates the tensor network for calculating coset probabilities and the approximate algorithm it used for tensor contractions. For detailed description of how the MPS decoder works, please refer to the original paper of the MPS decoder [16].

Denote $f\mathcal{G}$ one of the cosets $\mathcal{C}_I(s), \mathcal{C}_X(s), \mathcal{C}_Y(s), \mathcal{C}_Z(s)$ defined in Equation 2.8. The MLD needs to calculate the coset probability $P(f\mathcal{G})$. Denote P_1 the probability distribution of the error model. For example,

$$P_1(X) = P_1(Y) = P_1(Z) = p/3, \quad P_1(I) = 1 - p \tag{2.16}$$

for depolarizing error model with error probability p. Consider the error model as independent and identically distributed (IID), the coset probability can be expressed as

$$P(f\mathcal{G}) = \sum_{g \in \mathcal{G}} \prod_{e} P_1(f_e g_e), \tag{2.17}$$

where the sum goes over all stabilizers in the stabilizer group and the product goes over all edges of the code lattice. For example, consider the code as the standard surface code, any stabilizer g can be parameterized by binary variables $\alpha_v, \beta_p \in \{0, 1\}$ representing whether g contains vertex stabilizer A_v and plaquette stabilizer B_p which are defined in Section 1.2.5. Such that

$$g(\alpha; \beta) = \prod_{v} (A_v)^{\alpha_v} \cdot \prod_{u} (B_u)^{\alpha_u}, \tag{2.18}$$

where we define $(A_v)^0 = (B_u)^0 \equiv I$ by convention, and this equation holds since all vertex and plaquette stabilizers form a complete generator set for the stabilizer group of the surface code. Denote e some edge on the lattice which has neighbouring vertices $v(e), w(e)$ and plaquettes $p(e), q(e)$. For horizontal and vertical edges there will be two different geometric configurations, as shown in Fig. 2.8. Denote g_e the

restriction of g acting on the qubit at edge e, then g_e only depends on four binary variables $\alpha_{v(e)}, \alpha_{w(e)}; \beta_{p(e)}, \beta_{q(e)}$:

$$g_e(\alpha;\beta) = g_e(\alpha_{v(e)}, \alpha_{w(e)}; \beta_{p(e)}, \beta_{q(e)}), \tag{2.19}$$

where $g_e(i,j;k,l)$ is a function of variables $i,j,k,l \in \{0,1\}$. For example, $g_e(i,j;k,l) = X^i X^j Z^k Z^l$ for the surface code.

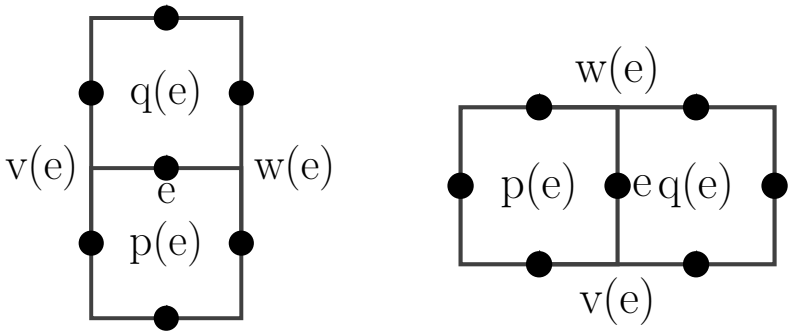


Figure 2.8: Neighbour relation for edge e with vertices $v(e), w(e)$ and plaquettes $p(e), q(e)$.

We can rewrite the expression of coset probability with parameters α, β:

$$P(f\mathcal{G}) = \sum_{\alpha}\sum_{\beta} T(\alpha;\beta), \tag{2.20}$$

where the sum goes over $\alpha, \beta \in \{0,1\}^{d(d-1)}$ corresponding to all possible configurations of α_v, β_p, and

$$T(\alpha;\beta) = \prod_e P_1(f_e g_e(\alpha_{v(e)}, \alpha_{w(e)}; \beta_{p(e)}, \beta_{q(e)})). \tag{2.21}$$

The right side of Equation 2.20 can be written as the contraction of a properly defined tensor network based on the extended lattice as shown in Fig. 2.9. There are three types of nodes s, h and v in the extended lattice. s nodes represent locations of stabilizers or ancillary qubits and h, v nodes represent data qubits at horizontal and vertical edges of the original code lattice as shown in Fig. 2.8. The edges in the extended lattice are called *links* to avoid confusion with edges in the original code lattice.

Consider a specific pair of variables α and β. Duplicate the corresponding values α_v and β_p onto every link connected to the nodes v and p of type s, then a labeling of these links can be established using the binary variable $\gamma(\alpha;\beta)$. Because of the stabilizer property, it is required that all links connected to any s node carry identical labels, and this link labeling is called *valid*. From Equation 2.21, $T(\alpha;\beta)$ is a product of objects

$$T_e(\alpha;\beta) \equiv P_1(f_e g_e(\alpha_{v(e)}, \alpha_{w(e)}; \beta_{p(e)}, \beta_{q(e)})), \tag{2.22}$$

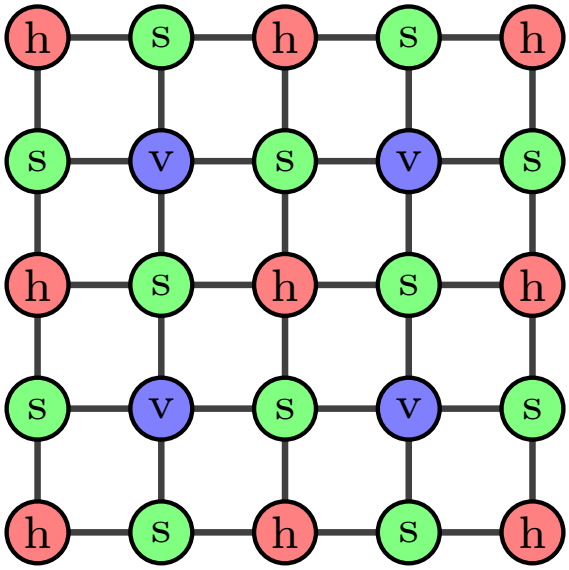


Figure 2.9: Extended surface code lattice with $d = 3$. It can be also seen as the tensor network used for calculating the coset probabilities.

where e represents horizontal or vertical edges in the original code lattice, but one can also think of them as h or v nodes in the extended lattice with a valid link labeling. By writing $T_e(\alpha;\beta)$ as a function of γ, which is $T_e(\alpha;\beta) = T_e(\gamma)$, one can rewrite Equation 2.20 as

$$P(f\mathcal{G}) = \sum_{valid\ \gamma} \prod_{e\in h,v} T_e(\gamma), \tag{2.23}$$

where the product encompasses all h and v nodes, while the summation covers only valid link labelings. To encompass all possible link labelings, one can broaden the sum by introducing additional terms $T_e(\gamma) \in 0, 1$ linked to nodes e of type s. These terms are structured so that $T_e(\gamma) = 1$ iff all links linked to node e bear identical labels, and conversely, $T_e(\gamma) = 0$ when this condition isn't met. Therefore

$$P(f\mathcal{G}) = \sum_{\gamma} \prod_{e\in h,v,s} T_e(\gamma), \tag{2.24}$$

where the product encompasses all nodes within the extended lattice, while the summation spans across all possible link labelings. Notably, each term $T_e(\gamma)$ is exclusively dependent on the labels of links connected to node e. Therefore, the right side of Equation 2.24 can be view as the contraction of a tensor network defined by the collections of tensor $T_e(\gamma)$. The tensor network will have the same diagram as Fig. 2.9, yet in the tensor network picture, each node will represent a tensor and the link between tensors represents tensor contraction. Diagrams of different tensors $T_e(\gamma)$ are shown in Fig. 2.10 and the tensor elements are shown in Equation 2.25:

$$
\begin{aligned}
s(i,j,k,l) &= \begin{cases} 1 & \text{if} \quad i=j=k=l, \\ 0 & \text{otherwise,} \end{cases} \\
h(i,j,k,l) &= P_1(f_e g_e(j,l;i,k)), \\
v(i,j,k,l) &= P_1(f_e g_e(i,k;j,l)),
\end{aligned}
\tag{2.25}
$$

where all tensor indices i, j, k, l take value 0,1. Certain indices of tensors located at the network's boundary may be absent, and it can be checked by observing the links incident to them in the network. Observe that in Equation 2.25, the arrangement of arguments in g_e is swapped between h and v. This is simply because for h nodes, vertex stabilizers are located on the left and right and plaquette stabilizers are located on the top and bottom, while for v nodes, vertex stabilizers are located on the top and bottom and plaquette stabilizers are located on the left and right.

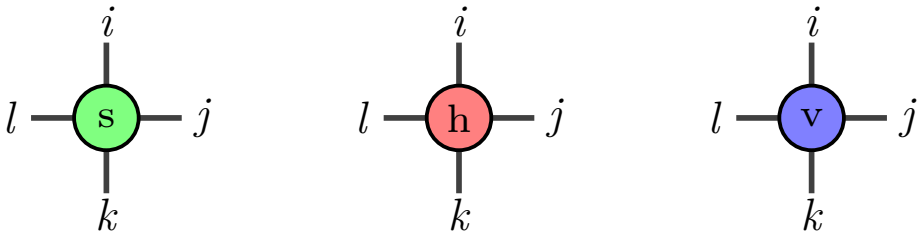


Figure 2.10: s, h and v tensors in the tensor network.

After constructing the tensor network, we then consider its contraction. Contractions can be done by rows or by columns. Here we consider the contraction by columns, but contraction by rows would be equivalent after a rotating transformation of the tensor network. Examine the partition of the tensor network, as illustrated in Figure 2.11. Define MPS(χ) and MPO(χ) as the sets of matrix product states (MPS) and matrix product operators (MPO) characterized by a sequence of $2d-1$ tensors with bond dimension χ. Then columns V_1, H_2, V_2 shown in Fig. 2.11 define MPO in MPO(2). Columns H_1, H_3 define MPS in MPS(2). In general all internal columns will define MPO and the boundary (first and last) columns will define MPS. The contraction of an adjacent pair of columns is equivalent to multiplying the corresponding MPOs. As a consequence the coset probability can be written as

$$
P(f\mathcal{G}) = \left\langle \hat{H}_d \right| \hat{V}_{d-1} \cdots \hat{H}_2 \hat{V}_1 \left| \hat{H}_1 \right\rangle, \tag{2.26}
$$

where $\hat{H}_i, \hat{V}_i$ represent the corresponding MPO(MPS) of column H_i, V_i. The right side of Equation 2.26 can be approximated. In the MPS decoder the algorithm proposed in [20] is used. The precision of the algorithm's approximation is governed by a parameter $\chi \geq 2$, and the algorithm will become exact as χ grows exponentially with the code distance d. The algorithm calculates the column product in a left-to-right manner, with each step producing a state $\psi \in \text{MPS}(\chi)$. This state can

be characterized by an array of $2d-1$ tensors, each having dimensions $2 \times \chi \times \chi$, amounting to a total of $O(d\chi^2)$ parameters. For each step, state ψ is updated by $\psi \to \hat{H}_i\psi/\hat{V}_i\psi$ according to which MPO acts on the state, and the calculation time is $O(d\chi^2)$. The action of MPO(χ) will map state ψ from MPS(χ) to MPS(2χ), therefore a truncation algorithm described in [21] is applied to confine the bond dimension to no larger than χ. The truncation occurs through the computation of the Schmidt decomposition of the state, where only the χ largest Schmidt coefficients are preserved. For more details of the truncation algorithm used in the MPS decoder, please refer to [16]. The truncation algorithm takes time $O(d\chi^3)$, and this truncation needs to be done for each step in the approximation algorithm, i.e. for each column in the tensor network. Therefore the total time running for approximation algorithm is $O(d^2\chi^3) = O(n\chi^3)$, where n is code size. The above algorithm can be summarised in Algorithm 2, where truncate() is the truncation algorithm which takes as input state $\phi \in MPS(2\chi)$ and returns state $\psi \in MPS(\chi)$ approximating ϕ.

Algorithm 2 Approximate contraction algorithm [16]

Input: Pauli operator f
Output: Approximation of $P(f\mathcal{G})$
  $\psi \leftarrow \hat{H}_1$
  for $i = 1$ to $i = d-2$ **do**
    $\psi \leftarrow \text{truncate}(\hat{V}_i\psi)$
    $\psi \leftarrow \text{truncate}(\hat{H}_{i+1}\psi)$
  end for
  $\psi \leftarrow \text{truncate}(\hat{V}_{d-1}\psi)$
  return $\langle \hat{H}_d | \psi \rangle$

The coset probability $P(f\mathcal{G})$ for given Pauli operators f, fX_L, fY_L, fZ_L will be calculated by Algorithm 2 and then used in Algorithm 1 to choose the most likely coset $\mathcal{C}_{ML}$, which has the largest coset probability $P(\mathcal{C})$.

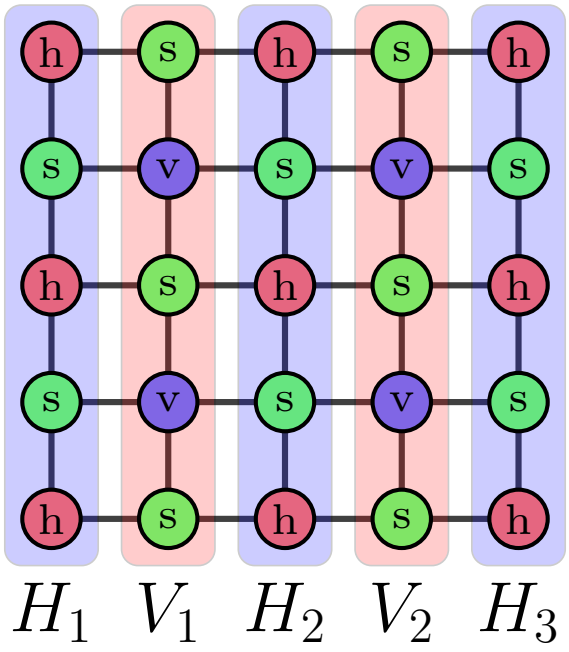


Figure 2.11: Partition of the tensor network by column. Here is an example for the $d = 3$ surface code

3
Methods

This chapter describes the methods used in this thesis. Section 3.1 describes the definition and properties of the error models; Section 3.2 describes the MPS decoder for the XYZ^2 code which is developed in the thesis project, but it also contains the description of how to build codes for the YZZY code and the decoder which supports INID error model. Section 3.3 describes the numerical method used in this thesis to get an estimated threshold.

3.1 Error Model

This section describes the error models used in this thesis. Section 3.1.1 describes standard error models that are independent and identically distributed (IID), which means different types of errors are independent with each other, and the error probabilities are the same for all data qubits in the QEC code. Section 3.1.2 introduces non-IID error model, but in here we only focus on independent and non-identically distributed error model, which means the errors are still independent for each error type, but the error probabilities can be different for each data qubit in the QEC code. Considering real devices or concatenated QEC codes, this type of error model is relevant.

3.1.1 Independent and identically distributed error model

Consider a generally IID error model, which has Pauli noise

$$\mathcal{N} = \bigotimes_{i=1}^{n} \mathcal{N}_i, \tag{3.1}$$

where

$$\mathcal{N}_i(\rho) = (1-p)\rho + p_X X\rho X + p_Y Y\rho Y + p_Z Z\rho Z \tag{3.2}$$

for all $i \in 1, \cdots, n$. $p = p_X + p_Y + p_Z$ is called the error probability or error rate, and p_X, p_Y, p_Z are the probabilities for X, Y and Z errors. Often the probability that no error happens $1-p$ is denoted as p_I. Together $\vec{p} = (p_I, p_X, p_Y, p_Z)$ is called the *probability distribution* of the error model. If there is only one type of noise which has non-zero error probability, then the error model is called the pure error model. There are three types of pure error model, which are:

Pure X/Bit-flip error model:

$$p_I = 1 - p, \quad p_X = p, \quad p_Y = 0, \quad p_Z = 0. \tag{3.3}$$

Pure Y/Bit&Phase-flip error model:

$$p_I = 1 - p, \quad p_X = 0, \quad p_Y = p, \quad p_Z = 0. \tag{3.4}$$

Pure Z/Phase-flip error model:

$$p_I = 1 - p, \quad p_X = 0, \quad p_Y = 0, \quad p_Z = p. \tag{3.5}$$

If three types of noise have the same error probability, then the error model is called the depolarizing error model:

Depolarizing error model:

$$p_I = 1 - p, \quad p_X = p/3, \quad p_Y = p/3, \quad p_Z = p/3. \tag{3.6}$$

It's straightforward to express the probability distribution for pure noise and depolarizing noise on the Pauli group. There is

$$P(f) = \begin{cases} (1-p)^{n-|f|}p^{|f|} & \text{if} f \in \mathcal{P}^P, \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

for pure noise, where $P \in \{X, Y, Z\}$, $|f| = wt(f)$ is the weight of f and $\mathcal{P}^P \subset \mathcal{P}$ represents the subgroup generated only by operator $P \in \{X, Y, Z\}$, and

$$P(f) = (1-p)^{n-|f|}(p/3)^{|f|} \tag{3.8}$$

for depolarizing noise.

In this thesis we also consider one type of error model called the biased error model [22]. It is similar with the depolarizing error model. However, it has an axis or biased error which has a larger error probability than the other two errors which have the same error probability. For example, in Z-biased error model, we define η as the ratio of Z error probability with non-Z error probabilities, such that $\eta = p_Z/(p_X + p_Y)$. Therefore we can express the error probabilities of

Z-biased error model:

$$p_I = 1 - p, \quad p_X = \frac{p}{2(\eta+1)}, \quad p_Y = \frac{p}{2(\eta+1)}, \quad p_Z = \frac{\eta p}{(\eta+1)}. \tag{3.9}$$

When $\eta = 1/2$, the error model will become the depolarizing error model; when taking the limit $\eta \to \infty$, the error model will become the pure Z error model. For the X-biased error model and Y-biased error model, define η as $p_X/(p_Y + p_Z)$ and $p_Y/(p_Z + p_X)$, the probability distribution can be similarly expressed as above. Biased error model is worth considered since it's less regular compared to the pure error model or depolarizing error model, but it can fall back to these two cases with certain η.

3.1.2 Independently and non-identically distributed error model

Here we also consider non-IID error model. To be more precise, in this thesis we only consider independently and non-identically distributed error mode, and we refer to it as INID error model showed through this thesis. A generally INID error model has Pauli noise

$$\mathcal{N} = \bigotimes_{i=1}^{n} \mathcal{N}_i, \tag{3.10}$$

where

$$\mathcal{N}_i(\rho) = (1 - p_i)\rho + p_{X,i} X\rho X + p_{Y,i} Y\rho Y + p_{Z,i} Z\rho Z, \tag{3.11}$$

where $p_i = p_{X,i} + p_{Y,i} + p_{Z,i}$ is the error probability of each data qubit in the QEC code, and they can have different values.

3.1.3 Hashing bound

The *quantum capacity* denotes the maximum rate at which quantum information can traverse a noisy quantum channel [23]. A theorem states the existence of a stabilizer code capable of achieving the *hashing bound* $R = 1 - H(\vec{p})$ for a Pauli channel [24]:

$$\rho \to p_I \rho + p_X X\rho X + p_Y Y\rho Y + p_Z Z\rho Z, \tag{3.12}$$

where $\vec{p} = (p_I, p_X, p_Y, p_Z)$ is called the *probability vector* and $H(\vec{p})$ is the *entropy* of the probability vector, which has the form:

$$\begin{aligned} H(\vec{p}) &= -\sum_{j \in \{I,X,Y,Z\}} p_j log_2 p_j \\ &= -p_I log_2 p_I - p_X log_2 p_X - p_Y log_2 p_Y - p_Z log_2 p_Z. \end{aligned} \tag{3.13}$$

For any IID error model, there is an error probability p to make hashing bound go to zero by random coding. This probability is referred to the *zero-rate hashing bound* and is served as a benchmark for code capacity thresholds.

3.2 MPS decoder for the XYZ^2 code

This section talks about the methods used for the code realization of the MPS decoder for the XYZ^2 code. In short the strategy is to first map the error chain in XYZ^2 code to the YZZY code by reversing the mapping relationship introduced in Section 2.3. This will lead to decoding INID errors on the YZZY code. Therefore, one only needs to consider building a MPS decoder for the YZZY code which also supports decoding INID error models. Fig. 3.1 illustrates the strategy pipeline.

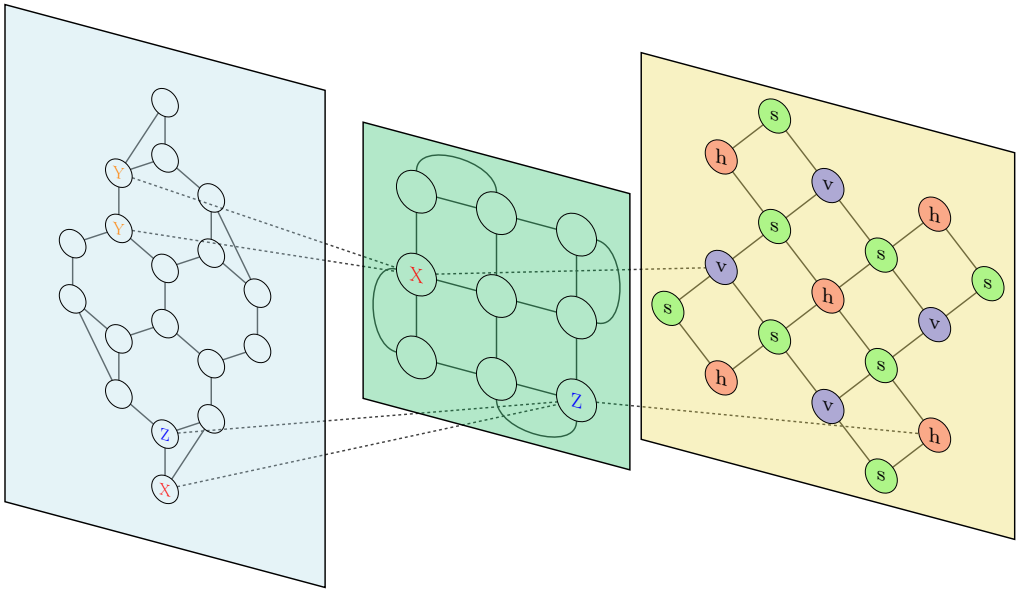


Figure 3.1: Illustration of algorithms steps in the MPS decoder for the XYZ^2 code. An error on the XYZ^2 code is mapped to a corresponding error on the YZZY code, and then the coset probability is calculated using the modified MPS decoder.

3.2.1 Code realization of the YZZY code

The Class **RotatedPlanarYZCode** for the YZZY code is based on the code of the Class **RotatedPlanarXZCode** for the XZZX code which is available in [11, 25]. Since the YZZY code and the XZZX code are highly similar, only the definitions of stabilizers and logical operators needs editing. Editing stabilizers and logical operators is followed by the content mentioned in Section 2.2 and Section 2.3: Stabilizers in the XZZX code are XZZX in every plaquette, while stabilizers in the YZZY code are YZZY in every plaquette; Logical X and Z operators in the XZZX code are defined as a horizontal and vertical chain with alternating X and Z, while in the YZZY code logical Y and Z operators are defined as a horizontal and vertical chain with alternating Y and Z. Note that in the YZZY code, logical X operator is replaced by logical Y operator to maintain its symmetry with the XZZX code. The logical X operator in the YZZY code can be defined as the product of logical Y and Z operators. After these editing, we realize the definition of the Class **RotatedPlanarYZCode**.

3.2.2 Code realization of the MPS decoder for the YZZY code

The code realization of the MPS decoder for the XZZX code in [25], defined by the Class **RotatedPlanarXZRMPSDecoder**, can be concluded as follows: Given the information of code and error syndrome, a sample recovery operator is created by applying a path of X operators between each plaquette identified by the syndrome to the code boundary. This can assure that the sample recovery operator f and the actual error e has the same syndrome, i.e. $f \in e\mathcal{C}(\mathcal{G})$. See Fig. 3.2 for an example: The actual error e on the left side will lead to the syndrome where $s_i = 1$

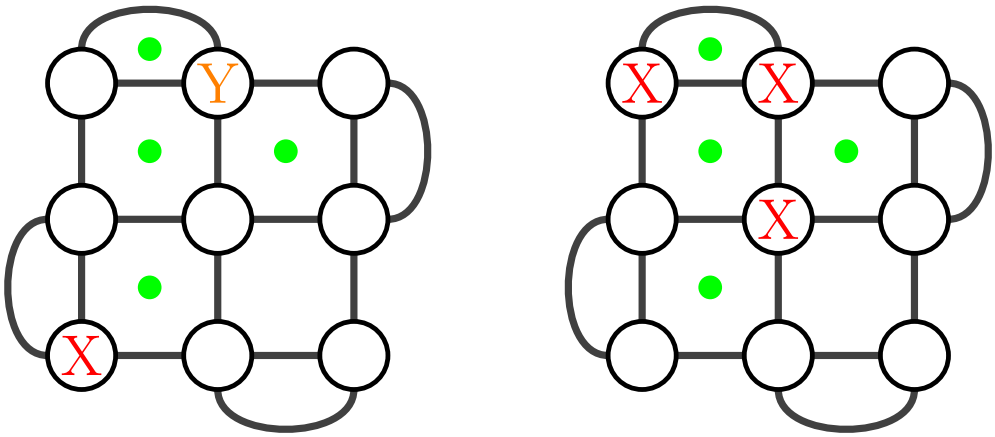


Figure 3.2: Example with the $d = 3$ XZZX code of sample recovery (right) for the error syndrome (green) caused by the actual error chain (left).

is marked in green; the sample recovery operator with all X operators constructed by the simple strategy will have the same syndrome as the actual error. Once the sample recovery operator f is found, the problem becomes to choosing an operator from sample operators f, fX_L, fY_L, fZ_L to be the final recovery operator based on Algorithm 1. Based on Algorithm 2, to calculate the approximate coset probabilities of these sample operators, one needs to first build tensor networks based on the code structure and the sample operators, and then the networks are contracted by the contraction algorithm.

One note for tensor networks of the rotated surface codes is that because the links between s nodes and h, v nodes are rotated (See top-left figure of Fig. 3.3), the contraction algorithm in Section 2.5 can not be implemented on the original tensor network. This can be solved by splitting each s node in the original network into 4 nodes (See top-right figure of Fig. 3.3) and then absorbing these nodes into neighbouring h or v nodes (See bottom-left figure of Fig. 3.3). After this transformation, the modified tensor network (See bottom-right figure of Fig. 3.3) fits the form on which the contraction algorithm can be applied. The details and code implementation of this tensor network transform is available in the source code of Class **RotatedPlanarRMPSDecoder** [9]. In short, the definition for Class **RotatedPlanarXZRMPSDecoder** can be described in pseudocode as shown in Algorithm 3.

Due to the symmetry of the YZZY code and the XZZX code, the overall algorithm is the same for Class **RotatedPlanarYZRMPSDecoder** and Class **RotatedPlanarXZRMPSDecoder**, as shown in Alogrithm 3. However, two parts in the code need to be modified: the first part is that for the YZZY code, the sample recovery operator is found by constructing a path of Y operators, compared with the XZZX code which uses X operators; the second part is that when creating h, v nodes for the XZZX code, a plaquette stabilizer XZZX is used for representing the restriction of g onto h, v nodes $g_{h(v)}$:

Algorithm 3 RotatedPlanarXZRMPSDecoder

Input: code C, error model EM, syndrome s
Output: recovery operator r
  $f \leftarrow$ simple recovery operator defined by s
  sample paulis $SP \leftarrow \{f, fX_L, fY_L, fZ_L\}$
  for sample pauli sp in SP **do**
    create h, v, s nodes based on C, sp and EM
    transform h, v, s nodes into H, V nodes
    calculate coset probability $P(sp\mathcal{G})$ by Algorithm 2
  end for
  pick $\mathcal{C}_{ML}$ by Algorithm 1
  $r \leftarrow sp_{ML} \in \mathcal{C}_{ML}$
  return r

$$g_{h(v)} = X^i \otimes Z^j \otimes Z^k \otimes X^l \quad \text{for the XZZX code} \tag{3.14}$$

where $i, j, k, l \in \{0, 1\}$ are the values of s node indices neighbouring the h or v node. The representation has the same form for h and v nodes because the stabilizers for the XZZX code are identical for all plaquettes. While for the YZZY code, a stabilizer YZZY is used for representing $g_{h(v)}$:

$$g_{h(v)} = Y^i \otimes Z^j \otimes Z^k \otimes Y^l \quad \text{for the YZZY code.} \tag{3.15}$$

In addition there are some other minor modifications in the source code not discussed in the thesis. To compare the source codes, see [25] and [26].

3.2.3 Correctness check of the MPS decoder for the YZZY code

Several test cases are used to check the correctness of the MPS decoder for the YZZY code. Due to the symmetry of the XZZX and the YZZY code, the performance of the XZZX and the YZZY code with the MPS decoder should have the same performance if the error models used in these two codes satisfy:

$$\begin{aligned} p_{I,XZZX} &= p_{I,YZZY}, \\ p_{X,XZZX} &= p_{Y,YZZY}, \\ p_{Y,XZZX} &= p_{X,YZZY}, \\ p_{Z,XZZX} &= p_{Z,YZZY}, \end{aligned} \tag{3.16}$$

where $(p_{I,XZZX}, p_{X,XZZX}, p_{Y,XZZX}, p_{Z,XZZX}) = \vec{p}_{XZZX}$ is the probability distribution of the error model used in the XZZX code and $(p_{I,YZZY}, p_{X,YZZY}, p_{Y,YZZY}, p_{Z,YZZY}) = \vec{p}_{YZZY}$ is the probability distribution of the error model used in the YZZY code.

Here three (pairs of) error models are used for the consistency check: pure X(Y), pure Z and depolarizing error model. Fig. 3.4, Fig. 3.5 and Fig. 3.6 show the logical failure rate P_f of pure X(Y), pure Z and depolarizing error model as a function of

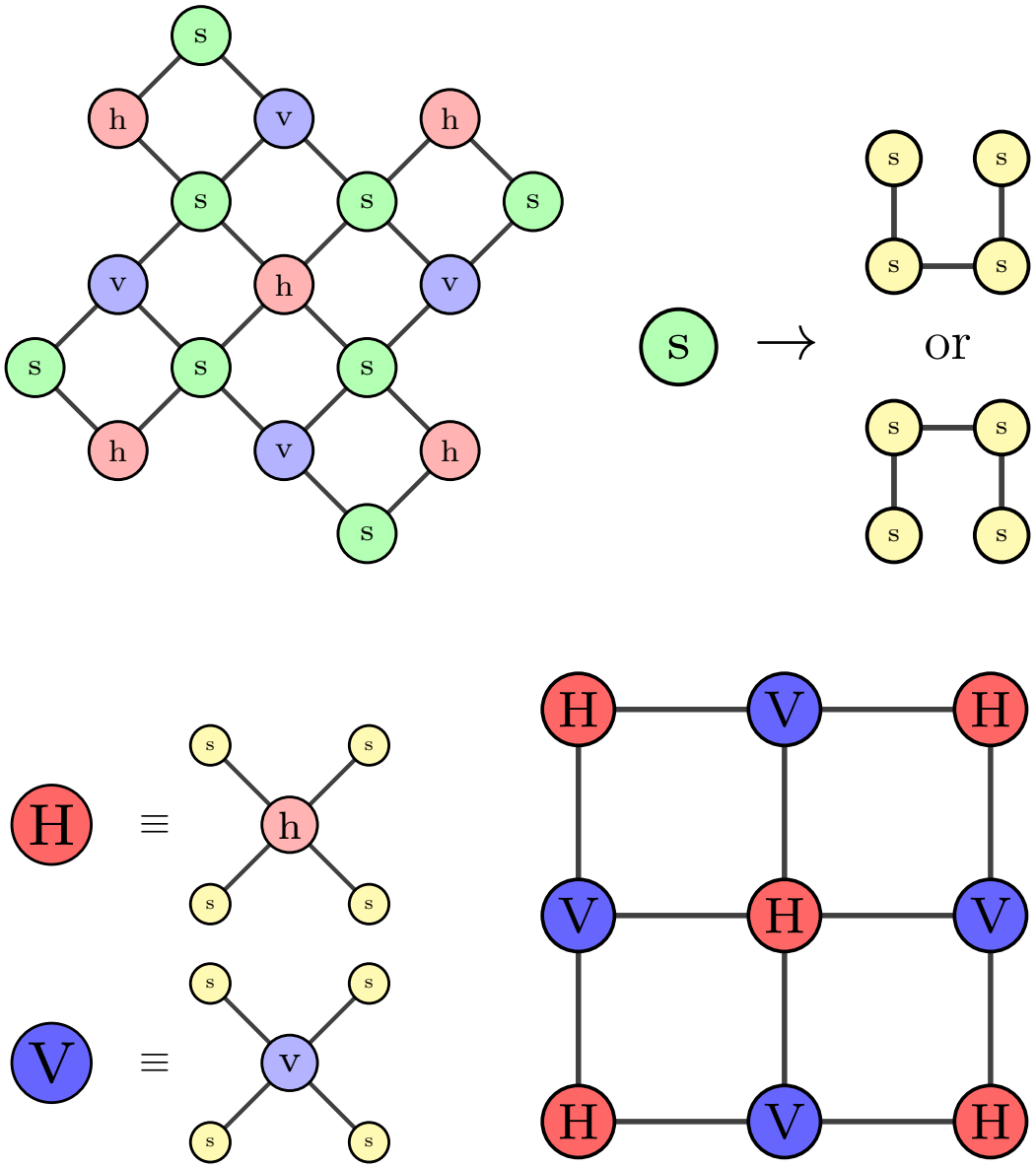


Figure 3.3: Transformation of tensor network based on rotated surface codes. The original tensor network extending from rotated surface codes (top-left) is inconvenient for tensor contractions. A transformation of splitting every s node into 4 nodes (top-right) and absorbing these nodes into neighbouring h or v nodes (bottom-left) is then applied to the original tensor network. The transformed tensor network (bottom-right) is available for implementing the contraction algorithm of the MPS decoder.

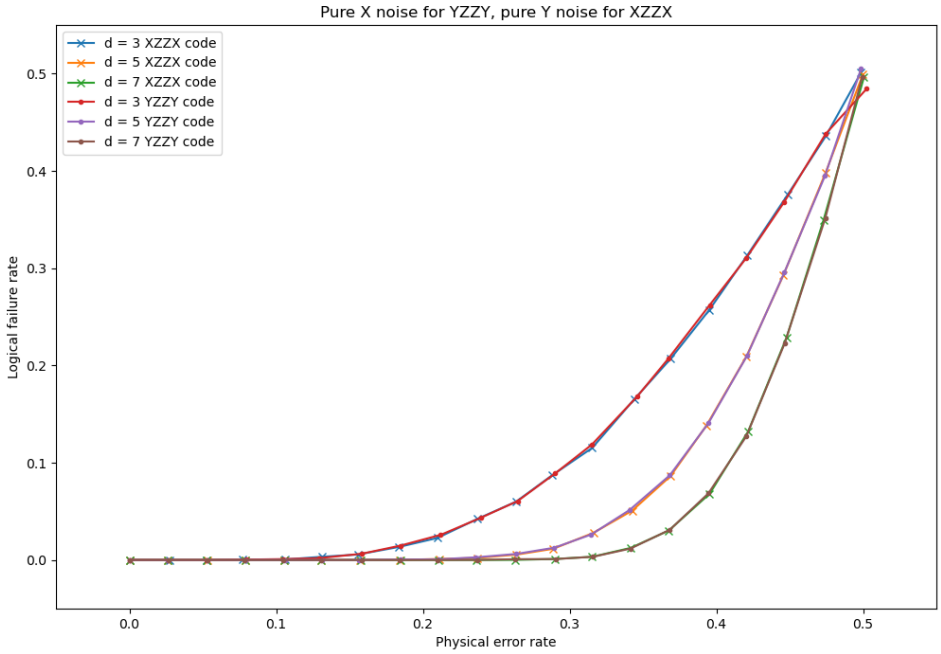


Figure 3.4: Logical failure rate vs physical error rate, pure X(Y) noise comparing the XZZX and YZZY codes and decoders.

physical error rate p for the XZZX and YZZY code with 3 different code distances d.

All curves representing the same code distance d are the same for the XZZX code and the YZZY code with their corresponding error models in all three figures. This is a strong evidence that the codes written for the YZZY code and the MPS decoder for the YZZY code work properly.

3.2.4 Code realization of the MPS decoder with INID error model

Modifying the MPS decoder to INID error models is very easy to achieve in principle: notice that the way for creating tensor networks in Class **RotatedPlanarYZRMPSDecoder** can be describe in pseudocode as shown in Algorithm 4.

Here the probability distribution $\vec{p} = (p_I, p_X, p_Y, p_Z)$ is the probability distribution of Pauli operators (I,X,Y,Z) of the error model with certain error rate p. When creating the tensor network, the same $\vec{p}$ is used for all nodes, i.e. all data qubits, since the nodes in modified tensor network corresponds to data qubits in the code lattice. For INID error model, the probability distribution can be represented by a list of $\vec{p_i}$, where the list size equals the number of data qubits in the code.

Therefore when creating the tensor network, use the list of probability distribution

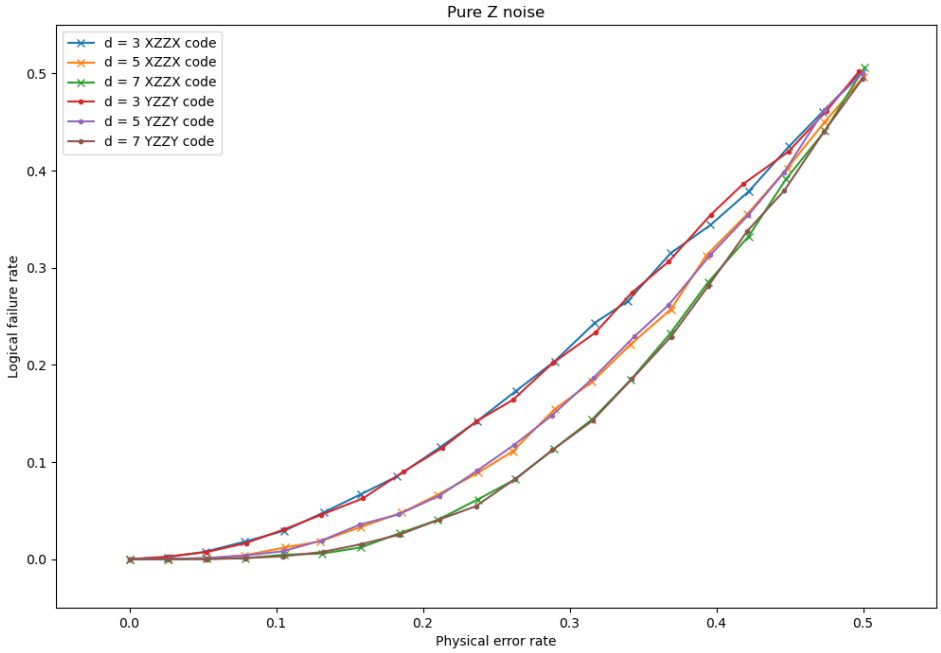


Figure 3.5: Logical failure rate vs physical error rate, pure Z noise comparing the XZZX and YZZY codes and decoders.

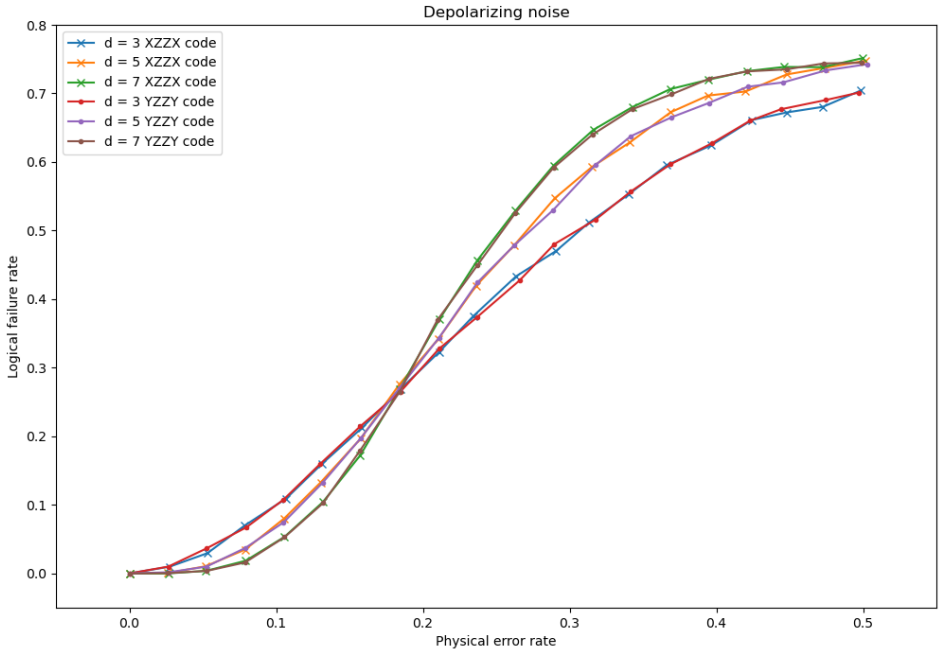


Figure 3.6: Logical failure rate vs physical error rate, depolarizing noise comparing the XZZX and YZZY codes and decoders.

Algorithm 4 Creating tensor network

Input: code C, probability distribution $\vec{p}$, sample pauli f
Output: tensor network tn

```
tn ← empty numpy array with size defined by C
for node i in tn do
  if node i = H then
    create H node based on p⃗ and f
    tn[i] ← H node
  else if node i = V then
    create V node based on p⃗ and f
    tn[i] ← V node
  end if
end for
return tn
```

$List(\vec{p})$ instead of just $\vec{p}$ as the input variable, and use the i^{th} probability distribution $\vec{p}_i$ when creating the i^{th} node of the tensor network. This is the key point for supporting decoding INID error model, and some other parts in the code of Class **RotatedPlanarYZRMPSDecoder** also needs to adapt to this change. The modified code will form another Class **RotatedPlanarYZRINIDMPSDecoder**, not only because it can decode for INID error model, but also when using it, one has to use the list of probability distributions $List(\vec{p})$ rather than just the error model EM as one of the input variables. So the ways of using these two decoders are different. However, when all elements in $List(\vec{p})$ are identical, the INID error model will coincide with the IID error model.

3.2.5 Correctness check of the MPS decoder with the INID error model

To check the correctness of the MPS decoder with the INID error model, we consider pure Y noise on the YZZY code. For the YZZY code, there's only one logical operator containing only Pauli Y, which is the diagonal $Y^{\otimes d}$ operator. An example of this operator is shown in the left side of Fig. 3.7. As a consequence, for any syndrome there exists only two error chains in different classes, and they differ only on the diagonal (Fig. 3.7). Therefore, the logical failure rate for the YZZY code under pure Y noise should be irrelevant with non-diagonal qubits. Then assume one of the diagonal qubit is ideal and never has error. This will allow the decoder never fail for decoding, since there will be only one error chain excluding the ideal qubit that corresponds to the syndrome.

Now consider the following four error models. The first error model is the pure Y error model as discussed in Subsection 3.1.1:

$$\vec{p} = (1-p, 0, p, 0) \qquad \text{for all data qubits,} \tag{3.17}$$

where $\vec{p} = (p_I, p_X, p_Y, p_Z)$ is the probability distribution, and p, the error rate, is

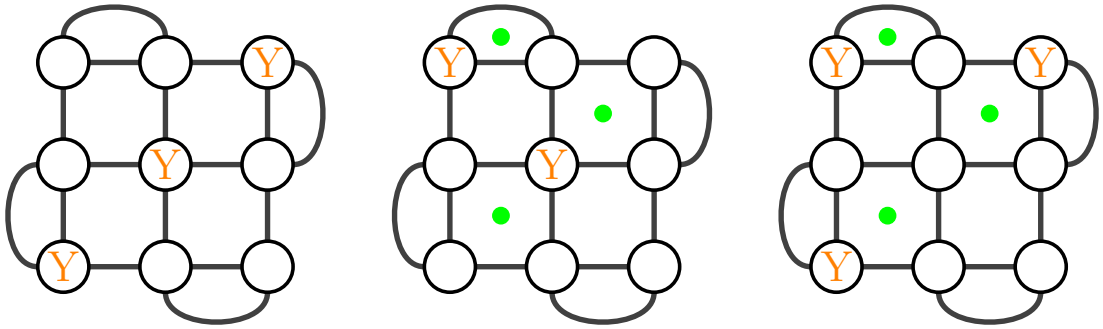


Figure 3.7: The diagonal $Y^{\otimes d}$ operator (left) is the only logical operator containing only Pauli Y in the YZZY code. As a consequence, for pure Y noise any syndrome has only two error chains in different classes (center, right) which differ only on the diagonal.

a variable. The second and third error model are INID error models, where the non-diagonal data qubits have a fixed error rate and the error rate for diagonal data qubits is a variable:

$$\begin{aligned} \vec{p} &= (1-p, 0, p, 0) \qquad \text{for diagonal data qubits;} \\ \vec{p} &= (0.9, 0, 0.1, 0) \qquad \text{otherwise} \end{aligned} \tag{3.18}$$

for the second error model and

$$\begin{aligned} \vec{p} &= (1-p, 0, p, 0) \qquad \text{for diagonal data qubits;} \\ \vec{p} &= (0.5, 0, 0.5, 0) \qquad \text{otherwise} \end{aligned} \tag{3.19}$$

for the third error model.

The fourth error model is also INID. It is almost the same as the pure Y error model, except that the first diagonal data qubit is ideal and its error rate is 0:

$$\begin{aligned} \vec{p} &= (1, 0, 0, 0) \qquad \text{for the first diagonal data qubit;} \\ \vec{p} &= (1-p, 0, p, 0) \qquad \text{otherwise.} \end{aligned} \tag{3.20}$$

According to the discussion above, if we plot the curves of the logical failure rate versus error rate for these four error models, the first, second and third error model should have the same curve, because only the error rate for non-diagonal data qubits are different in these error models, and they don't contribute to the decoding failures; the fourth error model should always have a zero failure rate, because one of the diagonal data qubits is ideal, the syndrome and error chain are in one-to-one correspondence and the decoding will never fail.

If the MPS decoder with the INID error model works correctly, it should give the same results as discussed above. In Fig. 3.8 and Fig. 3.9 for two different code distances, we can see that the curves for the first, second and the third error models are identical, and the failure rate for the fourth error model is always zero. These

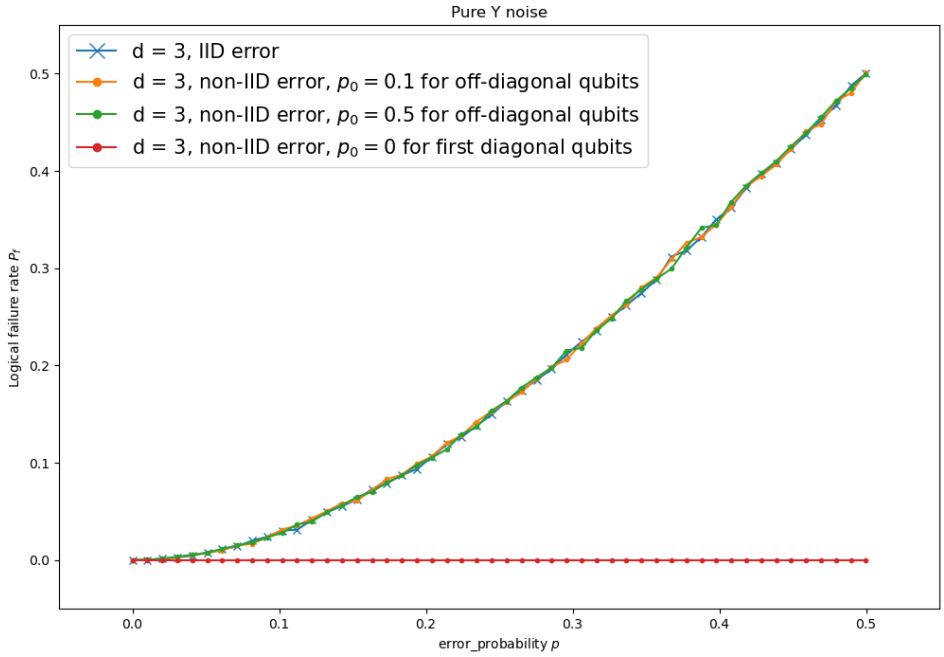


Figure 3.8: Logical failure rate vs error probability, (modified) pure Y noise for $d = 3$ the YZZY code.

results strongly support the correctness of the MPS decoder with the INID error model.

3.2.6 Code realization of the MPS decoder for the XYZ^2 code

The strategy of realizing the MPS decoder for the XYZ^2 code can be illustrated with Fig. 3.1: First consider XYZ^2 code with some IID error model with probability distribution $\vec{p} = (p_I, p_X, p_Y, p_Z)$; then consider an error chain, that follows this error model, which is acting on the XYZ^2 code. Using the mapping relation mentioned in Section 2.3, the XYZ^2 code, along with the error chain and probability distribution, can be mapped to the YZZY code.

Consider one vertical link in the XYZ^2 code. There can be 16 different cases for the action of the error chain on it:

$$\begin{aligned} &II, XX, ZZ, YY, YZ, ZY, XI, IX, \\ &ZI, YX, IZ, XY, XZ, IY, YI, ZX, \end{aligned} \tag{3.21}$$

where these 16 cases are divided into 2 groups: the first group corresponds to the *without link syndrome* case, i.e. the error chain commutes with the link stabilizer XX, and the second group correspond to the *with link syndrome* case, i.e. it anti-commutes with XX.

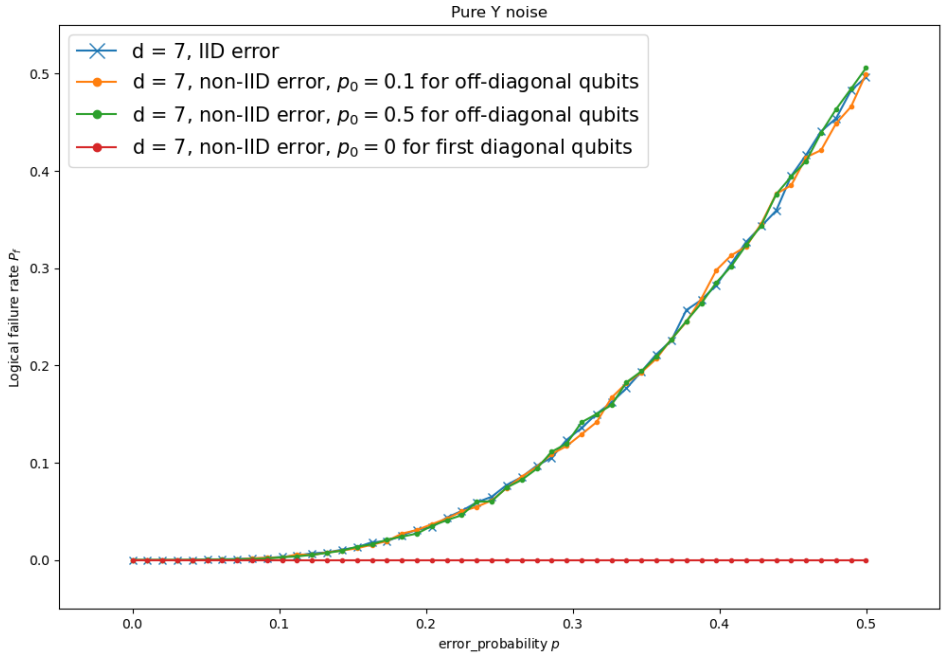


Figure 3.9: Logical failure rate vs error probability, (modified) pure Y noise for $d = 7$ the YZZY code.

For the *without link syndrome* case, one can just reverse the mapping relation in Section 2.3 to map a two-qubit Pauli operator on the vertical link to a single-qubit Pauli operator:

$$\begin{aligned} I &\leftarrow II \text{ or } XX, \\ X &\leftarrow ZZ \text{ or } YY, \\ Y &\leftarrow YZ \text{ or } ZY, \\ Z &\leftarrow XI \text{ or } IX, \end{aligned} \tag{3.22}$$

with a mapped probability distribution for this single qubit:

$$\begin{aligned} p'_I &= p_I p_I + p_X p_X, \\ p'_X &= p_Z p_Z + p_Y p_Y, \\ p'_Y &= p_Y p_Z + p_Z p_Y, \\ p'_Z &= p_X p_I + p_I p_X. \end{aligned} \tag{3.23}$$

For the *with link syndrome* case, it is not possible to directly map the two-qubit Pauli operator to a single-qubit Pauli operator, since they are not in the mapping relation in Section 2.3. However, notice that if one puts an extra Z operator on the first qubit of the vertical link, the *with link syndrome* case can be transformed into the *without link syndrome* case. For example, $ZI \rightarrow II, YX \rightarrow XX$, etc. Therefore,

by purposely adding an extra Z operator on the first qubit of the vertical link, one can map a two-qubit Pauli operator to a single-qubit Pauli operator:

$$\begin{aligned} I &\leftarrow ZI \text{ or } YX, \\ X &\leftarrow IZ \text{ or } XY, \\ Y &\leftarrow XZ \text{ or } IY, \\ Z &\leftarrow YI \text{ or } ZX, \end{aligned} \tag{3.24}$$

with a mapped probability distribution for this single qubit:

$$\begin{aligned} p'_I &= p_Z p_I + p_Y p_X, \\ p'_X &= p_I p_Z + p_X p_Y, \\ p'_Y &= p_X p_Z + p_I p_Y, \\ p'_Z &= p_Y p_I + p_Z p_X. \end{aligned} \tag{3.25}$$

This leads to two different probability distributions for the two cases. By doing this mapping to every vertical link in the XYZ^2 code, the error chain and probability distribution in the XYZ^2 code are mapped to the YZZY code. However, because the probability distribution of each qubit in the YZZY code can be in either of the two cases, it corresponds to a INID error model. After mapping to the YZZY code with the INID error model, one can just use the MPS decoder for the YZZY code with the INID error model as discussed in Subsection 3.2.2 and 3.2.4 to give a suggested recovery operator for the corresponding error. The overall decoding strategy is described in Algorithm 5, where an error e is generated by the given error model; the probability distribution $\vec{p}$ is defined by the given error model; the function Map() maps the error and probability distribution of the IID error model in the XYZ^2 code to errors and probability distributions of the INID error model in the YZZY code.

Note that even though here we only discussed decoding the XYZ^2 code with the IID error model, decoding the XYZ^2 code with the INID error model can also be realized with ease: the only thing needs modifying is to change the input probability distribution into a list of probability distributions which can describe the INID error model.

3.3 Error threshold

The numerical method introduced in [27] is used in this thesis to give an estimation of the error threshold p_c. Define a correlation length $\xi = (p - p_c)^{-\nu}$, where ν is a critical exponent characterizing the divergence of ξ at $p = p_c$. For code distance d large enough, the performance of the code is expected to be controlled by the ratio d/ξ. The logical failure rate is a function of the scaling variable

$$x = (p - p_{c_0}) d^{\frac{1}{\nu_0}}. \tag{3.26}$$

Consider doing Taylor expansion around the actual threshold p_c, i.e. $x = 0$, the function can be truncated to the few first power series. Here we are using a quadratic model, where $f = Ax^2 + Bx + C$. By fitting all the data from different code distances

Algorithm 5 MPS decoder for XYZ^2 code

Input: code C, error e, probability distribution $\vec{p}$
Output: recovery operator r

```
e_map, List(p⃗) ← Map(e, p⃗)
map C from XYZ² code to the YZZY code
get syndrome s from e_map
f ← simple recovery operator defined by s
sample paulis SP ← {f, fX_L, fY_L, fZ_L}
for sample pauli sp in SP do
    tn ← empty numpy array with size defined by C
    for node i in tn do
        if node i = H then
            create H node based on List(p⃗)[i] and sp
            tn[i] ← H node
        else if node i = V then
            create V node based on List(p⃗)[i] and sp
            tn[i] ← V node
        end if
    end for
    calculate coset probability P(spG) by Algorithm 2
end for
pick C_ML by Algorithm 1
r ← sp_ML ∈ C_ML
return r
```

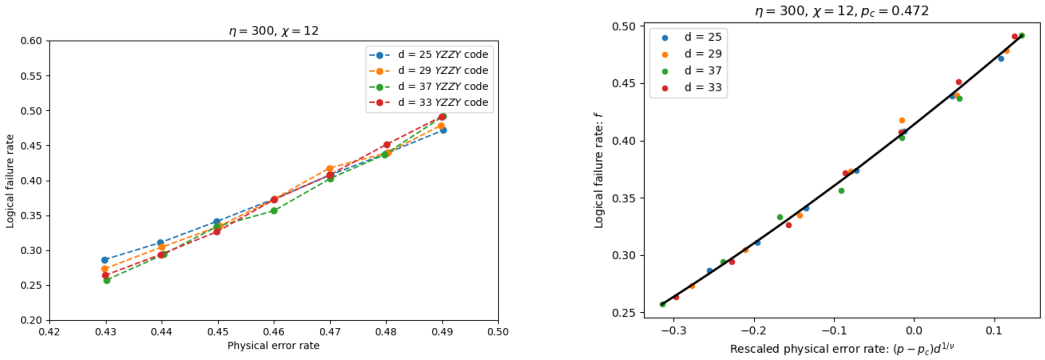


Figure 3.10: Example of estimation method for error threshold.

into this model, we can find p_c and v which make the best fit of this model. This method can be described in pseudocode as follows:

Algorithm 6 Estimation of error threshold

Input: code distance d, error rate p, logical failure rate p_f
Output: estimation of error threshold $p_{c_{est}}$
  for all data with different code distance d **do**
    $x \leftarrow (p - p_{c_0})d^{1/\nu_0}$
    $f \leftarrow Ax^2 + Bx + C$
  end for
  fit p_f with function f
  $p_{c_{est}} \leftarrow$ optimized fitting parameter $p_{c_{opt}}$ for p_{c_0}
  return $p_{c_{est}}$

Since the truncation for Taylor expansion is only valid when the error rate p in all data is close to the actual threshold p_c, in practise we first plot the logical failure rate p_f versus the error rate p for several code distances d with a wide range of p to estimate the threshold p_c by observation; then use some error rates p which lie in a narrow range of p symmetric to p_c, along with their corresponding d and p_f, as the fitting data. Fig. 3.10 gives an example of this fitting method, where the left figure shows p_f versus p, and the right figure shows p_f versus the scaling variable x and the fitting function $f(x)$ versus x with the optimized parameters. From the right figure, all the re-scaled data fit well with the quadratic model, and gives an estimation value for threshold $p_c = 0.472$, which is reasonable referred to the left figure.

4

Result

This chapter presents the main results of this thesis project. Section 4.1 presents the results we get for the XYZ^2 code by using the MPS decoder with pure error models. These results are compared with the analytical results discussed in [13] as the evidence of correctness for the MPS decoder for the XYZ^2 code; Section 4.2 presents the results we get for the XYZ^2 code by using the MPS decoder with depolarizing and biased error models. These results are compared with the results from the EWD decoder [19, 28]. These results are used to study the performance of the MPS decoder for the XYZ^2 code; Section 4.3 shows the threshold of the XYZ^2 code and the YZZY code as a function of η for X, Y and Z-biased error models.

4.1 Results of the MPS decoder for the XYZ^2 code compared with analytical results

As discussed in [13], the logical failure rate for the pure X noise can be analytically given by

$$P_{f,X}(p) = \sum_{n=\lceil d/2 \rceil}^{d} C_d^n p_o^n p_e^{d-n}, \tag{4.1}$$

where $\lceil x \rceil$ is the ceiling function, which represents the least integer larger than or equal to x; $C_d^n = d!/((d-n)!n!)$ is the combinatorial number of (d, n); p_o and p_e stand for the probability of errors with odd parity and even parity on a link, which is given by the following equation:

$$\begin{aligned} p_o &= 2p(1-p), \\ p_e &= (1-p)^2 + p^2. \end{aligned} \tag{4.2}$$

The logical failure rate for the pure Y/Z noise can be analytically given by

$$P_{f,Y/Z}(p) = \sum_{n=N/2}^{N} C_N^n p^n (1-p)^{N-n} - \frac{1}{2} C_N^{N/2} p^{N/2} (1-p)^{N/2}, \tag{4.3}$$

where $N = 2d^2$ is the number of data qubits in the XYZ^2 code. Fig. 4.1, Fig. 4.2 and Fig. 4.3 show the failure rate P_f versus the error rate p for the pure X, pure Y and pure Z noise for the XYZ^2 code with several different code distances. The solid lines with point marker represent the simulation results obtained by the MPS decoder for the XYZ^2 code, where $\chi = 12$; 50 different error rates and 10000 runs

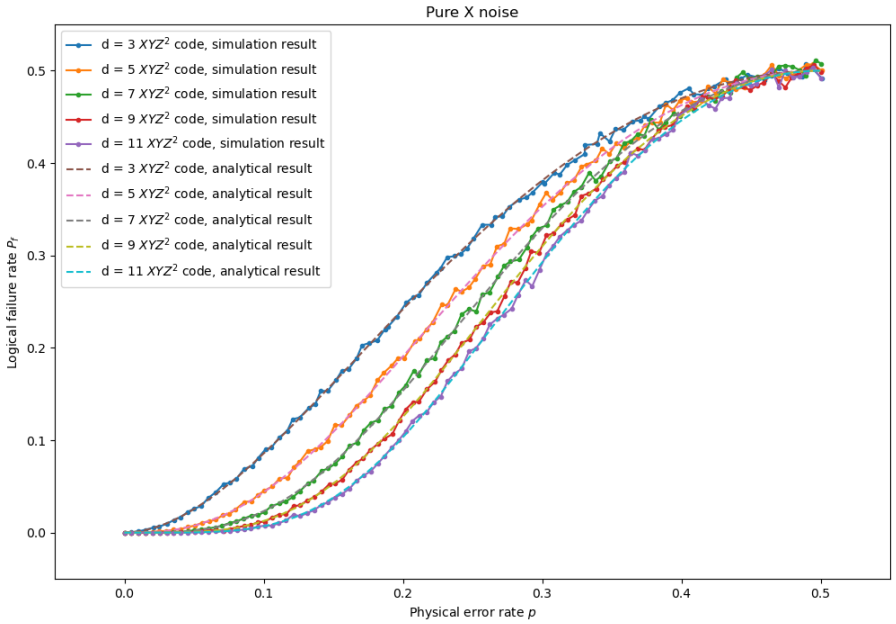


Figure 4.1: Logical failure rate vs physical error rate, pure X noise for the XYZ^2 code comparing simulation results (solid lines with point marker) with analytical results (dashed lines).

are used for the simulation of each error rate, and the dashed lines represent the analytical results. As we can see, the simulation results show great consistency with the analytical results, except for some small fluctuations due to the limited number of running cases for the simulation.

4.2 Results of the MPS decoder for the XYZ^2 code compared with results obtained by the EWD decoder

We also use the MPS decoder for the XYZ^2 code with depolarizing, X-bias, Y-bias and Z-bias error models with $\eta = 10$. Since analytical expressions for these error models are not known, we use another decoder, called effective weight and degeneracy (EWD) decoder [19] for performance comparison.

The plots of the logical failure rate as a function of physical error rate under different error models for the XYZ^2 code obtained by the MPS and EWD decoders are shown in Fig. 4.4, Fig. 4.5, Fig. 4.6 and Fig. 4.7. The solid lines with point marker represent the simulation results obtained by the MPS decoder, where $\chi = 12$; 30 different error rates and 10000 runs are used for the simulation of each error rate; the dashed lines represent the results obtained by the EWD decoder, where 25 different error rates and 10000 runs are used for the simulation of each error rate. From these

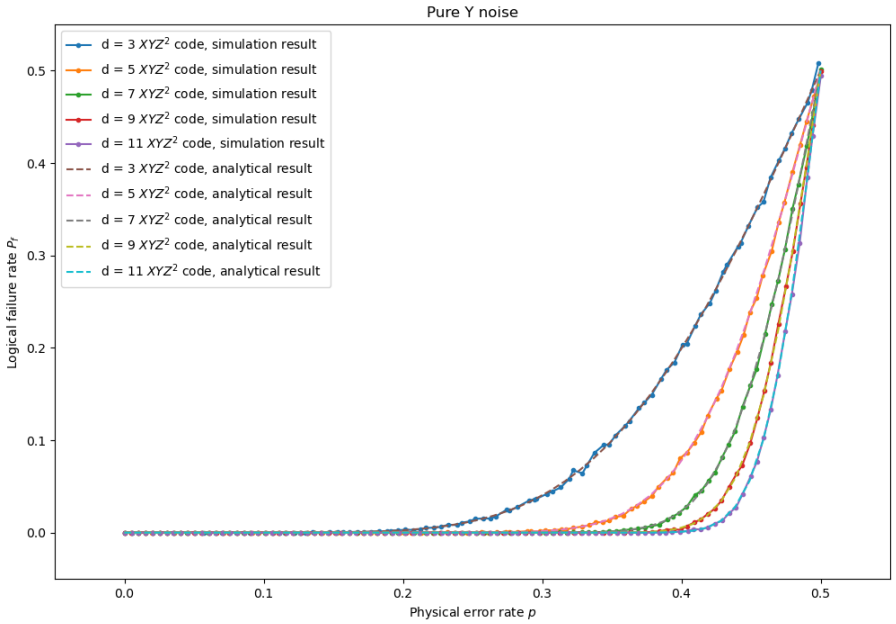


Figure 4.2: Logical failure rate vs physical error rate, pure Y noise for the XYZ^2 code comparing simulation results (solid lines with point marker) with analytical results (dashed lines).

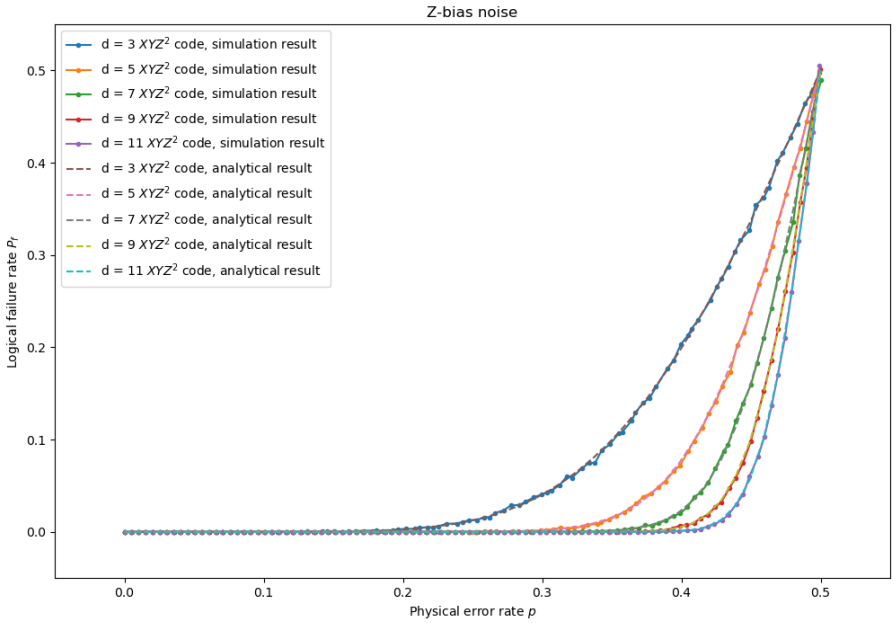


Figure 4.3: Logical failure rate vs physical error rate, pure Z noise for the XYZ^2 code comparing simulation results (solid lines with point marker) with analytical results (dashed lines).

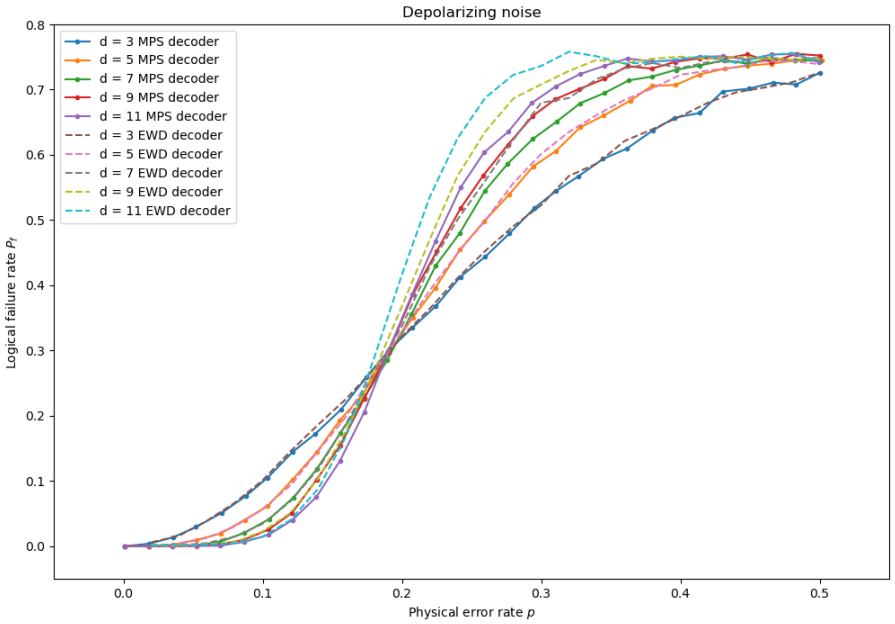


Figure 4.4: Logical failure rate vs physical error rate, depolarizing noise for the XYZ^2 code comparing two approximate maximum-likelihood decoders.

plots, we can see that the results obtained by the MPS decoder share the same trend with those obtained by the EWD decoder. However, we find that for larger code distances, the MPS decoder gives a lower sub-threshold logical failure rate and larger threshold than the EWD decoder. As there is an (unknown) lower bound to the logical failure rate given by the exact MLD, this means that the MPS decoder is more accurate.

4.3 Threshold of the XYZ^2 code compared with the YZZY code

To estimate the threshold as a function of η for the XYZ^2 code and the YZZY code, we get the results in Fig. 4.8, Fig. 4.9 and Fig. 4.10 for the X-biased, Y-biased and Z-biased noise with the zero-hashing bound for comparison. The expression for the hashing bound is discussed in Subsection 3.1.3 and the numerical method used for the threshold estimation is discussed in Section 3.3, where four different code distances $d = 25, 29, 33, 37$ with 7 different error rates around the estimated threshold for each code distance and 10000 runs for the simulation of each error rate are used. These results match with the results in the XZZX code paper [11], if one considers the symmetry of the XZZX code and the YZZY code, and extend them to the XYZ^2 code.

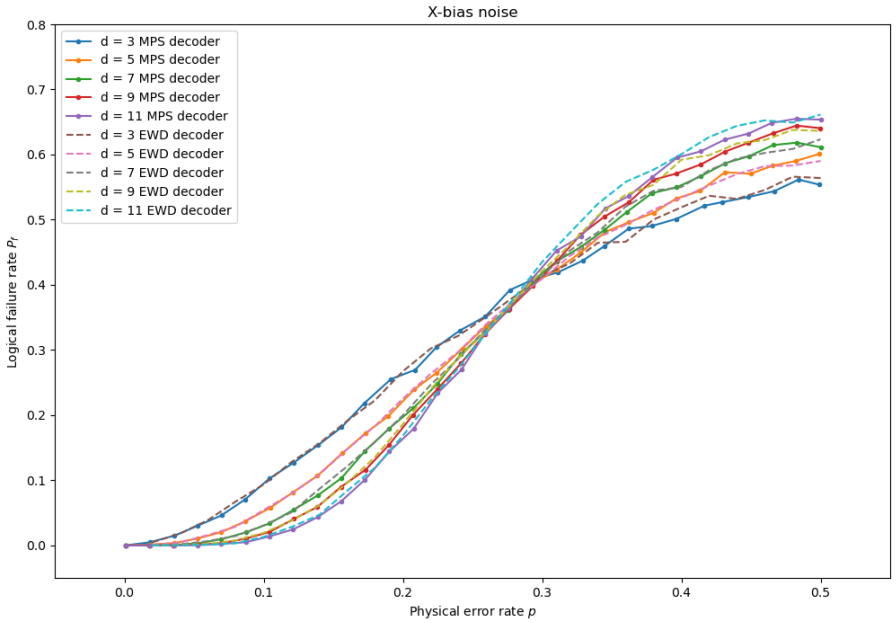


Figure 4.5: Logical failure rate vs physical error rate, X-bias $\eta = 10$ noise for the XYZ^2 code comparing two approximate maximum-likelihood decoders.

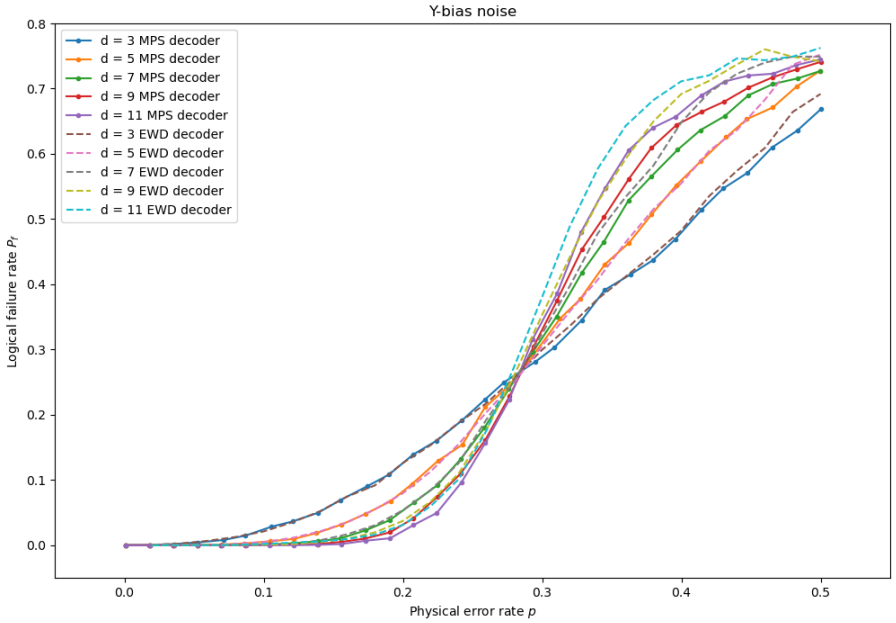


Figure 4.6: Logical failure rate vs physical error rate, Y-bias noise $\eta = 10$ for the XYZ^2 code comparing two approximate maximum-likelihood decoders.

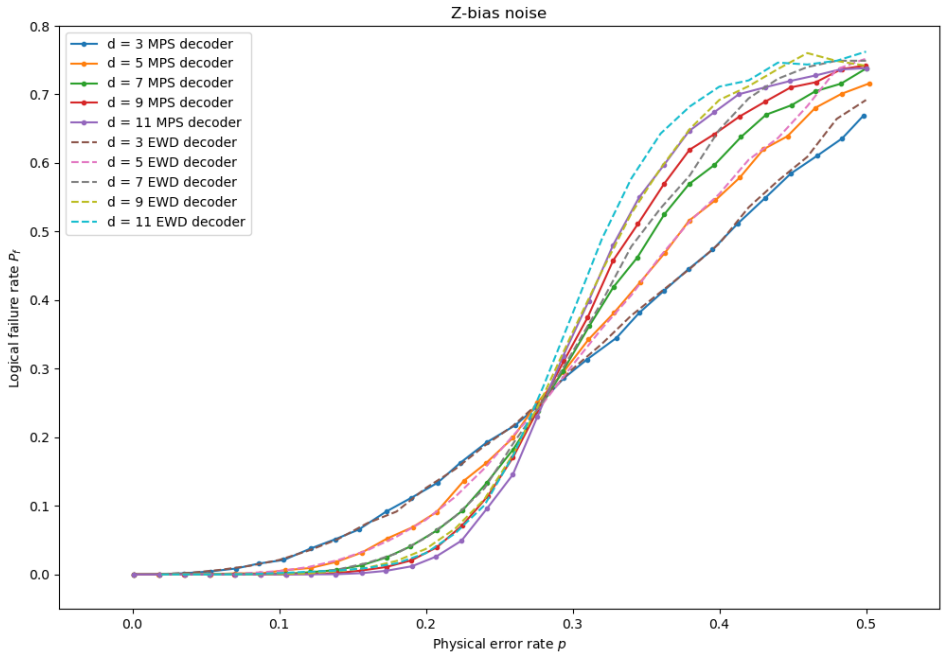


Figure 4.7: Logical failure rate vs physical error rate, Z-bias noise $\eta = 10$ for the XYZ^2 code comparing two approximate maximum-likelihood decoders.

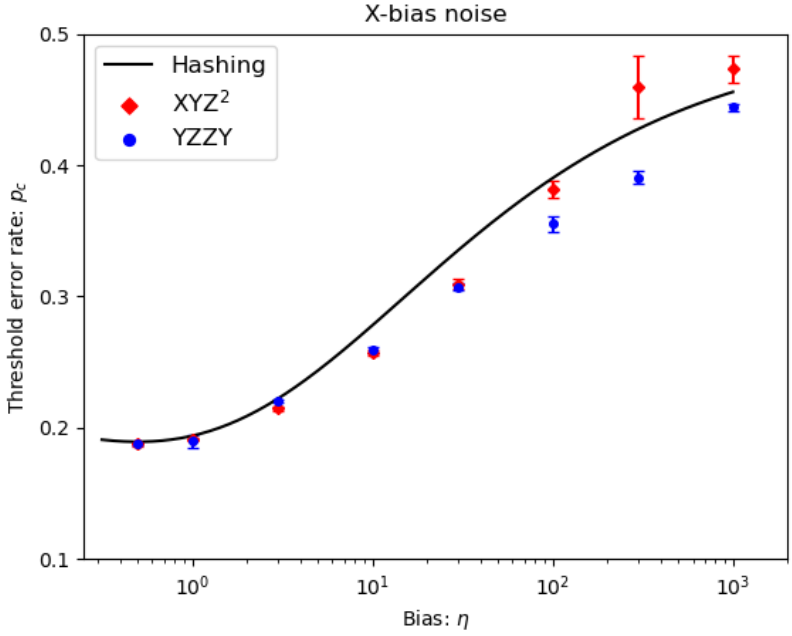


Figure 4.8: Estimated threshold p_c for the XYZ^2 and the YZZY code as a function of bias η of X-biased noise.

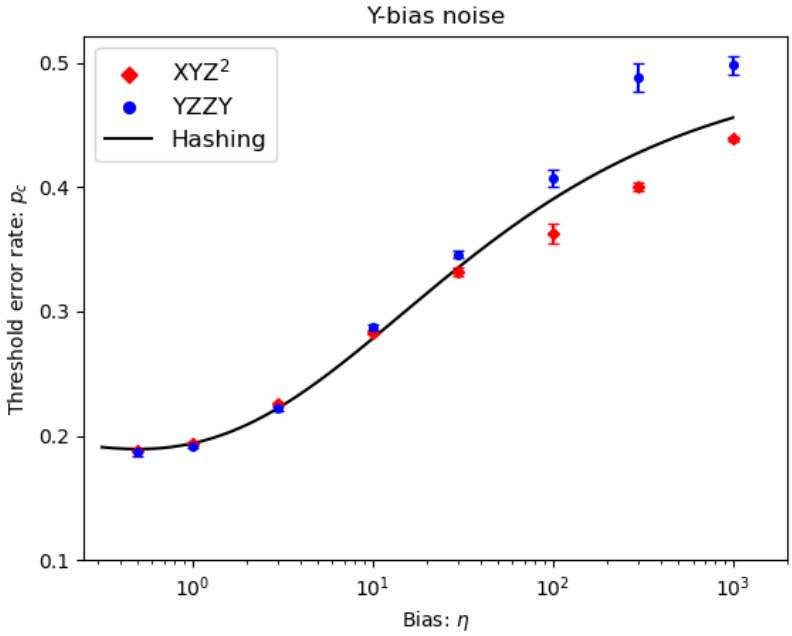


Figure 4.9: Estimated threshold p_c for the XYZ^2 and the YZZY code as a function of bias η of Y-biased noise.

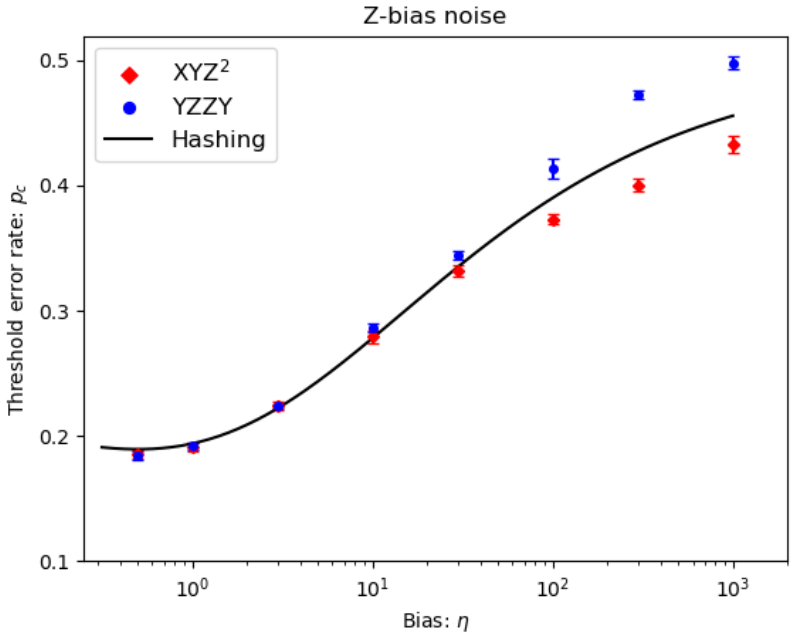


Figure 4.10: Estimated threshold p_c for the XYZ^2 and the YZZY code as a function of bias η of Z-biased noise.

5
Conclusion

This thesis project studied the principle of the tensor network based MPS (matrix product states-based) decoder and the code structure of *qecsim* python package, and developed codes for the MPS decoder for the YZZY code; the MPS decoder with the INID (independent but non-identical distributed) error model; the MPS decoder for the XYZ^2 code, and some other practical features based on *qecsim* and its extension code *qsdxzzx*. The correctness of the MPS decoder for the YZZY code is checked by comparing simulation results with the XZZX code; the correctness of the MPS decoder with the INID error model is checked by observing results for several special INID error models.

Combining the MPS decoder for the YZZY code with the INID error model and the mapping algorithm for the XYZ^2 and YZZY code, we develop a two-step decoder to decode the XYZ^2 code. The performance of the XYZ^2 code for several error models is studied with this two-step decoder. For pure error models, the results fit well with analytical results; for depolarizing and biased noise, the results obtained by the MPS decoder have lower sub-threshold logical failure rate and larger threshold than the results obtained by the EWD (effective weight and degeneracy) decoder. The threshold as a function of different error models is also studied. With the construction of these codes, other studies focusing on the YZZY and the XYZ^2 code on INID error models with the MPS decoder can be done.

One note is that the results shown in this thesis may not be fully correct, because we have found that for biased noise with large η, the estimated threshold is strongly dependent on the code distances d used in the numerical method: when d becomes larger, estimated threshold will become lower. Also, to obtain a statistically more reliable result, the number of simulation runs needed increases for large η, because when η and d are both large, the difference of logical failure rates between different d is very small, and the data fluctuation will significantly affect the accuracy of the estimated threshold. The reason for the threshold dropping with d seems very interesting, and is being further investigated [29].

Bibliography

[1] Wootters, W. K., & Zurek, W. H. (1982). A single quantum cannot be cloned. Nature, 299(5886), 802-803.

[2] Shor, P. W. (1995). Scheme for reducing decoherence in quantum computer memory. Physical review A, 52(4), R2493.

[3] Kitaev, A. Y. (2003). Fault-tolerant quantum computation by anyons. Annals of physics, 303(1), 2-30.

[4] Bravyi, S. B., & Kitaev, A. Y. (1998). Quantum codes on a lattice with boundary. arXiv preprint quant-ph/9811052.

[5] Fowler, A. G., Mariantoni, M., Martinis, J. M., & Cleland, A. N. (2012). Surface codes: Towards practical large-scale quantum computation. Physical Review A, 86(3), 032324.

[6] https://tensornetwork.org/diagrams/

[7] Oseledets, I. V. (2011). Tensor-train decomposition. SIAM Journal on Scientific Computing, 33(5), 2295-2317.

[8] Tuckett, D. K. (2020). Tailoring surface codes: Improvements in quantum error correction with biased noise (Doctoral dissertation).

[9] David K. Tuckett, qecsim: Quantum error correction simulator, 2021, https://qecsim.github.io/.

[10] Horsman, C., Fowler, A. G., Devitt, S., & Van Meter, R. (2012). Surface code quantum computing by lattice surgery. New Journal of Physics, 14(12), 123011.

[11] Bonilla Ataides, J. P., Tuckett, D. K., Bartlett, S. D., Flammia, S. T., & Brown, B. J. (2021). The XZZX surface code. Nature communications, 12(1), 2172.

[12] Wen, X. G. (2003). Quantum orders in an exact soluble model. Physical review letters, 90(1), 016803.

[13] Srivastava, B., Kockum, A. F., & Granath, M. (2022). The XYZ^2 hexagonal stabilizer code. Quantum, 6, 698.

[14] Wootton, J. R. (2015). A family of stabilizer codes for anyons and Majorana modes. Journal of Physics A: Mathematical and Theoretical, 48(21), 215302.

[15] Wootton, J. R. (2017). Demonstrating non-Abelian braiding of surface code defects in a five qubit experiment. Quantum Science and Technology, 2(1), 015006.

[16] Bravyi, S., Suchara, M., & Vargo, A. (2014). Efficient algorithms for maximum likelihood decoding in the surface code. Physical Review A, 90(3), 032326.

[17] Wootton, J. R., & Loss, D. (2012). High threshold error correction for the surface code. Physical review letters, 109(16), 160503.

[18] Hutter, A., Wootton, J. R., & Loss, D. (2014). Efficient Markov chain Monte Carlo algorithm for the surface code. Physical Review A, 89(2), 022326.

[19] Hammar, K., Orekhov, A., Hybelius, P. W., Wisakanto, A. K., Srivastava, B., Kockum, A. F., & Granath, M. (2022). Error-rate-agnostic decoding of topological stabilizer codes. Physical Review A, 105(4), 042616.
[20] Murg, V., Verstraete, F., & Cirac, J. I. (2007). Variational study of hard-core bosons in a two-dimensional optical lattice using projected entangled pair states. Physical Review A, 75(3), 033605.
[21] Schollwöck, U. (2011). The density-matrix renormalization group in the age of matrix product states. Annals of physics, 326(1), 96-192.
[22] Aliferis, P., & Preskill, J. (2008). Fault-tolerant quantum computation against biased noise. Physical Review A, 78(5), 052331.
[23] Lloyd, S. (1997). Capacity of the noisy quantum channel. Physical Review A, 55(3), 1613.
[24] Wilde, M. M. (2013). Quantum information theory. Cambridge university press.
[25] https://bitbucket.org/qecsim/qsdxzzx/
[26] https://github.com/yinzi-xiao/yzzy-xyz-code
[27] Wang, C., Harrington, J., & Preskill, J. (2003). Confinement-Higgs transition in a disordered gauge theory and the accuracy threshold for quantum memory. Annals of Physics, 303(1), 31-58.
[28] https://github.com/QEC-project-2020/EWD-QEC
[29] Srivastava, B., Xiao, Y. Z., & Granath, M. (2023). Code-capacity thresholds for surface codes under biased noise. In preparation.

DEPARTMENT OF SOME SUBJECT OR TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY