An Accurate Analytical Framework for Computing Fault-tolerance Thresholds Using the [[7,1,3]] Quantum Code

by

Andrew J. Morten

Submitted to the Department of Physics
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Physics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2005

Author ....
Department of Physics
August 26, 2005

Certified by ....
Isaac Chuang
Associate Professor, Department of Physics
Thesis Supervisor

Accepted by ....
David E. Pritchard
Senior Thesis Coordinator, Department of Physics

An Accurate Analytical Framework for Computing Fault-tolerance Thresholds Using the [[7,1,3]] Quantum Code

by

Andrew J. Morten

Submitted to the Department of Physics
on August 26, 2005, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Physics

Abstract

In studies of the threshold for fault-tolerant quantum error-correction, it is generally assumed that the noise channel at all levels of error-correction is the depolarizing channel. The effects of this assumption on the threshold result are unknown. We address this problem by calculating the effective noise channel at all levels of error-correction specifically for the Steane [[7,1,3]] code, and we recalculate the threshold using the new noise channels. We present a detailed analytical framework for these calculations and run numerical simulations for comparison. We find that only X and Z failures occur with significant probability in the effective noise channel at higher levels of error-correction. We calculate that when changes in the noise channel are accounted for, the value of the threshold for the Steane [[7,1,3]] code increases by about 30 percent, from .00030 to .00039, when memory failures occur with one tenth the probability of all other failures. Furthermore, our analytical model provides a framework for calculating thresholds for systems where the initial noise channel is very different from the depolarizing channel, such as is the case for ion trap quantum computation.

Thesis Supervisor: Isaac Chuang
Title: Associate Professor, Department of Physics

Acknowledgments

The length of my acknowledgements list is in indirect proportion to my gratitude toward those acknowledged. I would not have completed this work without the support of Prof. Isaac Chuang and Andrew Cross. I would like to thank Prof. Isaac Chuang for introducing me to the problem and for providing me an opportunity to work with his group at the Media Lab. I learned a great deal from interactions with his research group, especially with Andrew Cross. I cannot thank Andrew Cross enough – for providing my with a chunk of code that eventually became my QEC simulator, for many useful and eye-opening conversations about quantum error-correction whenever and wherever I needed them, for letting me use what would otherwise have been his computing cycles, and for his support during the final writing of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Quantum fault-tolerance is the key to a successful physical realization of a large-scale quantum computation. Using concatenated quantum error-correcting codes [17, 18, 10], it has been shown that as long as the noise in a system is below a certain threshold, arbitrarily long fault-tolerant quantum computations can be performed[2, 9, 11, 13, 8, 3]. The Steane [[7,1,3]] is the most promising among the small quantum error-correction codes. Many studies of the threshold for the Steane [[7,1,3]] code have been carried out[22, 14, 2, 19, 16, 21].

Previous estimates of the threshold for the Steane [[7,1,3]] code have assumed that the noise channel is the depolarizing noise channel at *all* levels of concatenation. In the depolarizing channel, the three types of errors X, Y, and Z occur with equal probability. In a concatenated code error-correction procedure, every level of concatenation has its own effective noise channel, which can be very different from the depolarizing channel. No detailed study of the effects of changes in the noise channel on the threshold has been done.

In this thesis we answer the following two questions: What is the effective noise channel at different levels of concatenation of the Steane [[7,1,3]] code? More importantly, how does the estimate of the threshold change when the different noise channels are taken into account?

We answer these two questions using an analytical model. Additionally, we conduct simulations to verify the accuracy of our model. Because our analytical model must distinguish between X, Y, and Z errors, it is necessarily more detailed than the models used for previous estimates of the threshold for the Steane [[7,1,3]] code. We contribute to the ongoing study of the Steane [[7,1,3]] code by providing this new, richer analytical model.

1.1 Outline

In the next Chapter of this thesis, we present some background in quantum computation and quantum error correction that will be used in later sections. We present only what is needed for an understanding of the later sections, and we introduce concepts in a way that assumes only some familiarity with quantum mechanics and classical computation.

In Chapter 3 we describe the model we use to calculate the threshold for the Steane [[7,1,3]] code. Modeling choices include the quantum circuit used for error-correction, the replacement rule that prescribes how to construct circuits for concatenated codes, and the noise model.

The main achievement of this thesis is the analytical model which we present in Chapter 4, along with the tables in Appendices A and B. This very detailed model is used to determine the noise channel at all levels of concatenation and the resulting threshold.

We then make predictions using our analytical model and compare a subset of the predictions to numerical simulation results in Chapter 5. We wrote code that generates quantum computer assembly code instructions for the Steane [[7,1,3]] code that were input to a program called ARQ, a quantum computer simulator. The ARQ code generator and some sample output ARQ code are given in Appendices D and E.

In the last Chapter we review our results and discuss limitations of and possible improvements to our analytical model.

Chapter 2

Background

In Section 2.1 we introduce the network model of quantum computation and the stabilizer formalism. The network model is the quantum mechanical generalization of the theory of classical circuits. In the network model, the classical bits 0 and 1 get replaced by the quantum states $|0\rangle$ and $|1\rangle$, and classical logic gates get replaced by unitary transformations. We use the network model to represent our quantum error-correction routine. The stabilizer formalism of quantum mechanics has to do with representing the state of a system with a complete set of commuting observables. Stabilizer circuits and the propagation of errors through them have an efficient mathematical description. We will use the stabilizer formalism in describing how to construct our error-correction circuits, why they work, and how we can simulate them efficiently on a classical computer.

In Section 2.2 we introduce quantum error correction. Because of the properties of quantum measurement, quantum errors can by "digitized," so they appear as bit or phase flips on a subset of qubits. Cleverly used classical error-correcting codes can then be applied to correct these errors. First we introduce quantum noise and the noise model used throughout the analysis. Next we explain the theory behind the Steane [[7,1,3]] error-correcting code by discussing classical error correction and a group of quantum error-correction codes, called CSS codes. After that we explain how to use the stabilizer formalism to construct and understand the quantum circuits for the [[7,1,3]] code. Finally, we explain the threshold result for quantum computation and summarize previous work on the Steane [[7,1,3]] code.

2.1 Quantum Computation

We give a general overview of the network model of quantum computation and the stabilizer formalism. See [12] for much of the material we present here.

2.1.1 Network Model

In this thesis we restrict ourselves to the network model of quantum computation. Other models for quantum computation exist, such as cluster states [15] and adiabatic evolution [7], but the network model is suitable for our present study of the concatenation of the Steane [[7,1,3]] code. These other models have been shown to be equivalent to the network model.

The theory of quantum computation in the network model [6] is the quantum mechanical generalization of the theory of classical circuits. In the classical circuit model, a circuit of logical gates acts on input bits to produce output bits. If the set of logical gates is *universal*, then any possible classical computation can be achieved in the classical circuit model (more precisely, any function $f : \mathbb{Z}_2^m \rightarrow \mathbb{Z}_2^n$ can be evaluated). In the quantum network model, a "circuit" of unitary transformations (gates) acts on input quantum states to produce output quantum states. If the set of unitary gates is *universal*, then any possible quantum computation can be achieved in the quantum network model (more precisely, any specified state can be created with arbitrary precision).

The inputs and outputs in the quantum network model are quantum states. The Hilbert space of these states is a tensor product of two-level systems, and the eigenstates of each two-level system are written as $|0\rangle$ and $|1\rangle$. The states in these two-level systems are called quantum bits, or *qubits*, in reference to their classical analogue. A physical example of a qubit would be a spin 1/2 particle with $|0\rangle \equiv |\downarrow\rangle$ and $|1\rangle \equiv |\uparrow\rangle$.

The gates in a quantum circuit are all unitary transformations, as required by the postulates of quantum mechanics. In our quantum error correction circuit, we assume that a few elementary quantum gates are available to the quantum computer: the identity I; the Pauli gates X, Y, and Z; the Hadamard gate H; cnot (control-X), cz (control-Z), and the Toffoli gate. We list the definitions of the identity gate, Pauli gates, and the Hadamard gate here in matrix representation in the $\{|0\rangle, |1\rangle\}$ basis:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{2.1}$$

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \tag{2.2}$$

The cnot and cz gates act on two qubits: a *control* qubit and a *target* qubit. They apply the X and Z gates, respectively, to the target qubit when the control qubit is $|1\rangle$, and do nothing when the control qubit is $|0\rangle$. This defines their behavior on the basis states $\{|00\rangle, |10\rangle, |01\rangle, |11\rangle\}$, so their behavior has been fully specified on all input two qubit states.

The Toffoli gate is a three qubit gate that acts as a cnot with two control qubits that must both be $|1\rangle$ for the X to be applied to the target.

A set of universal quantum gates is for quantum computation in the network model is {X,Y,Z,H,cnot,cz,Toffoli}. This is not a minimal set; these are the six fundamental gates that we assume can be carried out by the quantum computer in our model.

2.1.2 Stabilizer Formalism

We use the stabilizer formalism because it offers a compact representation of a certain subspace of quantum states. It allows us to simulate quantum error correction networks efficiently on a classical computer.

A *stabilizer circuit* is a circuit that consists only of gates that are in the normalizer of the Pauli group, and single qubit measurements. Included in this list of gates are the X, Y, Z, cnot, and cz gates. The only gate in our universal family of gates not included in this list is the Toffoli gate. The Gottesmann-Knill Theorem [8] states that any stabilizer circuit can be simulated efficiently on a classical computer, as long as the initial state is a stabilizer state.. The error-correction circuit we use for the Steane [[7,1,3]] code is a stabilizer circuit.

If the quantum state $|\psi\rangle$ satisfies $U|\psi\rangle = |\psi\rangle$ for some unitary gate U, we say that U *stabilizes* $|\psi\rangle$. For example, the state $|0\rangle$ is stabilized by the Pauli operator Z, and the state $H|0\rangle = (|0\rangle + |1\rangle)/\sqrt{2}$ is stabilized by the Pauli operator X. In fact, the states in these two examples are the unique states (up to a global phase) that are stabilized by their respective gates.

The Pauli group G_1 on one qubit is defined to consist of the identity (I), the three Pauli operators (X,Y,Z), and all operators created by multiplying the above operators by ± 1 or $\pm i$. The Pauli group G_n on n qubits is defined to consist of all n-fold tensor products of elements of G_1.

A vector space V of quantum states is *stabilized* by a subgroup S of the Pauli group G_n on n qubits if every state in V is stabilized by every operator in S. Any subset of S that generates S is called a set of *stabilizer generators* for V. If V contains a single quantum state with m qubits, then a set of m independent stabilizer generators uniquely defines the state (up to phase).

In our numerical simulations, we keep track of the stabilizer generators of the quantum system, rather than the state itself. We always keep track of the minimum number of stabilizer generators such that the state is uniquely specified (up to a phase). The stabilizer of the quantum system evolves as follows. If the current state is $|\psi\rangle$ with stabilizer g, then after a unitary gate, the state becomes $U|\psi\rangle = Ug|\psi\rangle = UgU^\dagger U|\psi\rangle$, so the new stabilizer is $UgU^\dagger$. Because all of the gates we use in our simulations (X, Y, Z, H, cnot, cz) are in the normalizer of the Pauli group G_n, we always have $UgU^\dagger \in G_n$. As long as the input state is stabilized by a subset of the Pauli group, the evolving state is always stabilized by a subset of the Pauli group.

Measurements also affect the stabilizer, but as long as the measurements are in the computational basis (that is, measurements of the operators X or Z), then the stabilizer remains a subset of the Pauli group after measurement. We only use single qubit measurements in the computational basis in our circuits. So, we conclude that we can efficiently simulate our error correction networks on a classical computer.

2.2 Quantum Error Correction

Quantum fault-tolerance is an essential ingredient for the physical realization of a quantum computer. Quantum fault-tolerance has three requirements: (1) we must be able to prepare encoded states, (2) we must correct errors on those states, and (3) we must control the spread of errors through our circuits.

In this section we present some background in quantum error correction. The purpose of this section is to provide the background in error correction needed by the rest of this thesis, so we limit the discussion to topics that will later be used.

In Section 2.2.1 we describe the quantum noise model, and how it can be interpreted using a set of discrete errors. Then in Sections 2.2.2, and 2.2.3. we explain how these errors can be corrected using circuits that are themselves noisy. In Section 2.2.4 we explain how to construct the circuits for the Steane [[7,1,3]] error-correction code. We end with Section 2.2.5 explaining the threshold result and summarizing previous work on the Steane [[7,1,3]] code.

2.2.1 Quantum Noise Model

Noise in a quantum network is not as simple as in the classical network, where the only possible error is a bit flip. In a noisy quantum network, there is a continuous spectrum of errors that can occur on a quantum state, because the quantum states are specified by two complex numbers (subject to normalization). Despite the continuous spectrum of errors, quantum error correction can be achieved by correcting only a small set of discrete errors [17, 18].

To represent quantum noise, we use the density operator formulation of quantum mechanics. In the density operator formulation, the state $|\psi\rangle$ is represented by the outer product $|\psi\rangle\langle\psi|$. If the state is unknown, but known to be $|\psi_1\rangle$, $|\psi_2\rangle$, ... or $|\psi_n\rangle$

with probabilities p_1, p_2, ... p_n, respectively, then the associated density operator is

$$\rho = \sum_{i=1}^{n} p_i \left|\psi_i\right\rangle\!\left\langle\psi_i\right| . \tag{2.3}$$

Such an operator is called a *mixed state.*

The application of the gate U to a mixed state ρ transforms the density operator into $U\rho U^\dagger$

The quantum noise model we use is called the *depolarizing channel.* In a depolarizing channel, a single qubit is replaced by the completely mixed state $I/2$ with probability p, and left unchanged with probability $1-p$. If the density operator of the single qubit state before the depolarizing channel is ρ, then the density operator after the depolarizing channel is

$$\begin{aligned} D(\rho) &= (1-p)\rho + p\frac{I}{2} \\ &= (1-p')\rho + \frac{p'}{3}(X\rho X + Y\rho Y + Z\rho Z), \end{aligned} \tag{2.4}$$

where we used the fact that for arbitrary ρ, $I = (\rho + X\rho X + Y\rho Y + Z\rho Z)/2$, and we defined $p' \equiv 3p/4$.

Equation 2.4 can be interpreted (density operators can have multiple valid interpretations) as is the identity gate being applied with probability p' and each Pauli gate being applied with probability $p'/3$. In this interpretation we call the application of the Pauli gate X an *X error*, the application of the Pauli gate Y a *Y error*, and the application of the Pauli gate Z a *Z error.*

From now on, whenever we talk about quantum noise, we simply refer to the probability of X, Y, and Z errors.

Before we continue on to discuss error-correction, we explain how noise errors propagate through a circuit. This is very important to understanding how error correction works (and also why we need it).

When there is an X error on a single qubit state state $|\psi\rangle$, then after the application of a Hadamard gate the new state is $H(X|\psi\rangle) = Z(H|\psi\rangle)$. This is interpreted as a Z error on the expected (without noise) state $H|\psi\rangle$. So, Hadamard gates *propagate* X errors to Z errors. They also propagate Z errors to X errors and Y errors to Y errors.

We can similarly determine that cz gates propagate X errors on the control qubit to Z errors on the target qubit and propagate X errors on the target qubit to Z errors on the control qubit. Cnot gates propagate X errors on the control qubit to X errors on the target qubit but propagate Z errors on the target qubit to Z errors on the control qubit. These facts are used in the construction of the syndrome extraction networks designed in Section 2.2.4.

2.2.2 Classical Error Correction

Quantum noise must be corrected in order for quantum computations to be fault-tolerant. Quantum error correcting codes have been designed for this purpose. The Steane [[7,1,3]] quantum error-correcting code belongs to the collection of Calderbank-Shor-Steane (CSS) codes [4], which are based on classical linear codes. In this section we discuss classical linear codes, using the codes that lead to the [[7,1,3]] quantum code as ongoing examples. Much of the theory in this section and the next is borrowed from [12].

The noise in classical error correction consists of bit flip errors: 0 becomes 1 with some probability, and 1 becomes 0 with some probability. A simple code for protecting against single bit flip errors is to represent the bit 0 by three bits 000 and the bit 1 by three bits 111. Then, if a single bit flip occurs, majority voting corrects the error.

In general, classical linear codes use n bits to store k bits of information. A linear code is specified by an n by k generator matrix G with entries in $\mathbb{Z}_2$ (zeros and ones with addition modulo 2, i.e. binary numbers). The n bit codewords are created from the k bit words by the operation Gx, where x is the k bit word represented as a column vector. For example, the generator matrix for the [[7,4,3]] Hamming code C_1,

$$G(C_1) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}, \tag{2.5}$$

creates 7 bit codewords out of 4 bit words.

The [[7,4,3]] code is an [[n,k,d]] linear code, where d is the *Hamming distance* of the code. The distance of a code is the minimum distance between codewords, where the distance between two codewords is defined to be the number of bits at which the codewords differ. For example, the codewords generated by G are 0000000, 1010101, 0110011, 1100110, 0001111, 1011010, 0111100, 1101001, 1111111, 0101010, 1001100, 0011001, 1110000, 0100101, 1000011, and 0010110, where every pair of codewords differs at at least three locations. Because the Hamming distance is three, if only one bit of a codeword of [[7,4,3]] is flipped, we can determine which was the original codeword. Codes can correct $t \equiv (d-1)/2$ errors and detect $d-1$ errors.

To determine which was the original codeword and how to correct for it, we use the parity check matrix H associated with G. The parity check matrix is an $n-k$ by n matrix with linearly independent rows such that $Hx = 0$ for every codeword x, and it can be found directly from G. If a single error occurs on the jth bit of any codeword x, then parity check matrix reveals the error that occurred on the new codeword $x' = x + e_j$, using $Hx' = H(x + e_j) = 0 + He_j = He_j$, where e_j is column vector of zeros with a one on the jth bit. The vectors He_j are called *syndromes*. The syndromes reveal the location of the bit flip errors and are unique because H has linearly independent rows.

The parity check matrix for the [[7,4,3]] code is

$$H(C_1) = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}. \tag{2.6}$$

An interesting property of this code is that if an error occurs on the jth qubit, then He_j is the binary representation of j.

Lastly, we describe an important classical linear code that can be constructed from any given classical linear code C: the dual of C, denoted $C^\perp$, is defined to consist of all codewords orthogonal to C. Equivalently, $C^\perp$ is defined by having the generator matrix H^T. Dual codes are used in the creation of CSS codes, of which the Steane [[7,1,3]] is a specific example.

2.2.3 CSS Codes and the [[7,1,3]] Code

A useful class of quantum error-correcting codes is the Calderbank-Shor-Steane [4] codes. CSS codes are based on classical linear codes and their duals. Given two classical linear codes C_1 and C_2 of the type $[n,k_1]$ and $[n,k_2]$ such that $C_2 \subset C_1$ and such that C_1 and $C_2^\perp$ correct t errors, we can construct an $[[n,k_1-k_2]]$ quantum code that corrects t errors. As part of our ongoing example of the Steane [[7,1,3]] code, we choose C_1 as defined in the previous section, and $C_2 = C_1^\perp$. These codes are [7,4,3] and [7,3,4] codes, respectively, so they combine to form a [[7,1,3]] quantum error-correction code, the Steane code. We describe what this quantum error-correction code is and how it works.

For every codeword $x \in C_1$ we define a quantum state

$$|x + C_2\rangle \equiv \sum_{y \in C_2} |x + y\rangle\,, \tag{2.7}$$

up to a normalization constant. Explicitly, the state (2.7) is either

$$|0\rangle_L \equiv \frac{1}{\sqrt{8}}(|0000000\rangle + |1010101\rangle + |0110011\rangle + |1100110\rangle + |0001111\rangle + |1011010\rangle + |0111100\rangle + |1101001\rangle), \tag{2.8}$$

or

$$|1\rangle_L \equiv \frac{1}{\sqrt{8}}(|1111111\rangle + |0101010\rangle + |1001100\rangle + |0011001\rangle + |1110000\rangle + |0100101\rangle + |1000011\rangle + |0010110\rangle), \tag{2.9}$$

depending on the value of $x \in C_1$.

The eight states in the expression for $|0\rangle_L$ are the codewords of the classical linear code $C_2 \subset C_1$. The eight states in the expression for $|1\rangle_L$ are the codewords in C_1 that are not in C_2.

If X errors are represented by the vector e_X with bits set to one where X errors occur, and Z errors are represented by the vector e_Z with bits set to one where Z errors occur, then the quantum state in Equation 2.7 becomes

$$\sum_{y \in C_2} (-1)^{(x+y)\cdot e_Z} |x + y + e_X\rangle . \tag{2.10}$$

Because $Y = iXZ$, Y errors are automatically corrected when X and Z errors are corrected.

The X error syndrome can be determined by using reversible quantum computation and ancilla to create the state

$$\sum_{y \in C_2} (-1)^{(x+y)\cdot e_Z} |x + y + e_X\rangle \underbrace{|H(C_1)e_X\rangle}_{\text{ancilla}}, \tag{2.11}$$

followed by measurement of the ancilla. The quantum circuit that achieves this is designed in Section 3.2. The syndrome is used to correct the bit flip errors by applying X gates on the appropriate qubits.

The Z error syndrome can be determined by first applying Hadamards to all of the data qubits, producing the state

$$\sum_{z \in C_2^{\perp}} (-1)^{x \cdot z} |z + e_Z\rangle , \tag{2.12}$$

after some mathematical manipulations and using the definition of dual space $C_2^{\perp}$. The syndrome (using $H(C_2^{\perp})$ instead of $H(C_1)$) is transferred to the ancilla and measured as in the X error correction. Note that by applying the Hadamard gates, we turned Z errors into X errors. Since $C_1 = C_2^{\perp}$ in the case of the Steane [[7,1,3]] code, we can use the same circuit for Z error correction as we did with X error correction, except Hadamards are applied to the data before and after Z the syndrome extraction.

2.2.4 Circuit Construction

In this section we use the theory of CSS codes to construct three circuits used in the Steane [[7,1,3]] error-correction code: G, the *preparation* network, which prepares the state $|0\rangle_L$; V, the *verification* network, which verifies that there are no X errors on the qubits that make up $|0\rangle_L$; and S, the *syndrome extraction* network, which uses ancilla qubits to extract the classical error syndrome from the data qubits. The gates we use are the same as in [21].

First, we construct the preparation network G, given in Figure 2-1. The preparation network constructs the state $|0\rangle_L$, which is a superposition of all codewords defined by the generator matrix for C_2,

$$G(C_2) = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{matrix} , \tag{2.13}$$

where we have labeled the columns 0 through 6 to correspond to qubits $|qa_0\rangle$ through $|qa_6\rangle$ in Figure 2-1.

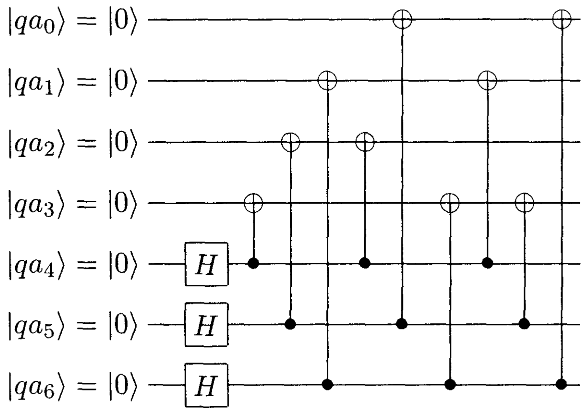


Figure 2-1: This is the circuit for the preparation network, G. It prepares the logical zero state, $|0\rangle_L$. It is used in the error-correction circuit (see Section 3.2) to prepare ancilla qubits in the state $|0\rangle_L$.

First we apply Hadamard gates to qubits 4, 5, and 6. This puts these three states into a superposition of all possible three bit words. Because rows 4, 5, and 6 in $G(C_2)$ form a 3×3 identity matrix, the last three qubits in each seven qubit codeword correspond to the three bits from which the codeword was derived using $G(C_2)$. This makes determining what state to put the other qubits in quite easy. Reading off from the three columns of $G(C_2)$: if qubit 4 is $|1\rangle$ then qubits 1, 2, and 3 need to be flipped; if qubit 5 is $|1\rangle$ then qubits 0, 2, and 3 need to be flipped; and if qubit 6 is $|1\rangle$ then qubits 0, 1, and 3 need to be flipped. We apply nine cnot gates according to the above three rules. Because $G(C_2)$ is a linear code and {001, 010, 100} forms a basis for the input bits to $G(C_2)$, this circuit correctly constructs a superposition of all codewords generated by $G(C_2)$.

Next, using $H(C_2)$ we construct the verification network shown in Figure 2-3. The verification network verifies that that there are no X errors on the logical qubit $|0\rangle_L$. This is accomplished by measuring all stabilizer generators of $|0\rangle_L$ that anti-commute with X errors. There are four such (independent) stabilizers, and a measurement result of 0 (meaning that the measured operator stabilizes the state) for all of them means that there is no X error. As can determined by reading off the rows of the parity matrix $H(C_2)$,

$$H(C_2) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}, \tag{2.14}$$

the stabilizer generators that anti-commute with X errors are

$$\text{ZIIIIZZ, IZIIZIZ, IIZIZZI, and IIIZZZZ} \tag{2.15}$$

(the other three stabilizer generators are XIIIIXX, IXIIXIX, and IIXIXXI, which commute with X errors).

In general, to measure a single qubit (unitary, hermitian) operator M, you apply a Hadamard on the ancilla prepared in the state $|0\rangle$, followed by a control-M gate with control on the ancilla, followed by a Hadamard and measurement on the ancilla. This also projects the the measured qubits into the eigenspace of the measured eigenvalue. For example, the measurement of the operator Z is depicted in Figure 2-2.

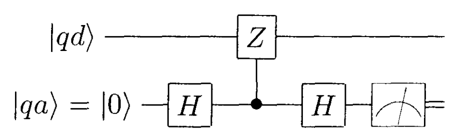


Figure 2-2: This circuit measures the operator Z on the qubit $|q_d\rangle$ and projects $|q_d\rangle$ into an eigenstate of Z with the measured eigenvalue.

The circuit V in Figure 2-3 measures the four stabilizer generators that anti-commute with X errors. Each of the four verification qubits is used to measure on of the generators.

The matrix $H(C_1)$ is not the only parity check matrix for C_1. Indeed, any matrix formed by adding together rows of $H(C_1)$ would be equally valid. However, as explained in [20], putting $H(C_1)$ in the form (I,A) ensures that the derived verification network does not leave correlated errors on the qubits of $|0\rangle_L$.

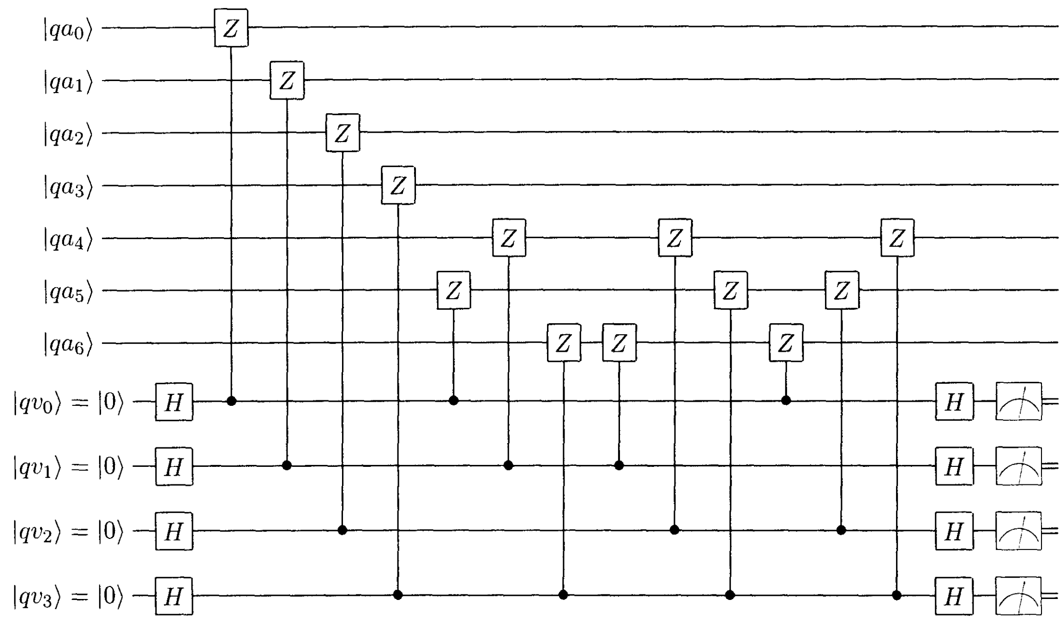


Figure 2-3: The verification network V checks for X errors on the state $|0\rangle_L$ and gives four zero measurement results if no X errors are detected.

The third and last circuit we construct is the syndrome extraction network. There are two syndrome extraction networks: one that detects X errors, and one that detects Z errors. We explain how to construct the Z error syndrome extraction network, shown in Figure 2-4. The construction of the X error syndrome extraction is very similar.

First, a logical cnot gate is performed with the ancilla in the state $|0\rangle_L$ as control and the logical data qubit as target. A logical cnot gate is just seven cnot gates acting transversally on the data and ancilla. Because the ancilla is in the state $|0\rangle_L$, the logical cnot gate does affect the logical data qubit. However, X errors on the ancilla propagate to X error on the qubits , and Z errors on the data qubits propagate to Z errors on the ancilla qubits.

Next, seven Hadamard gates are applied to the ancilla qubits, transforming Z errors into X errors. This is followed by Pauli Z measurements of all the ancilla. If there is no error, the result of the measurement will be in the code C_1. The reason for this is because the Hadamard gates are actually a logical Hadamard gate that transforms the state $|0\rangle_L$ into the state $(|0\rangle_L + |1\rangle_L)/\sqrt{2}$, which is a superposition of all of the codewords in C_1. A classical syndrome extraction is done on the measurement

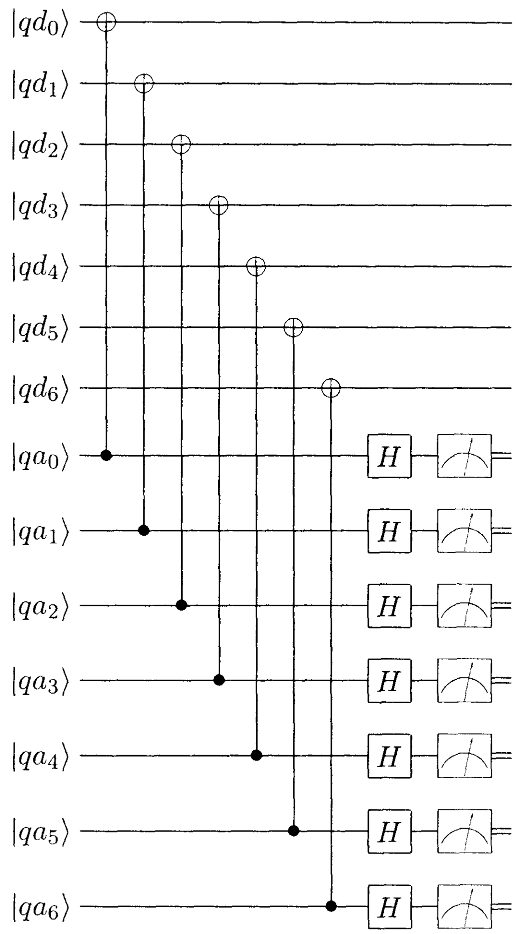


Figure 2-4: The syndrome extraction network S consists of three time steps. The above network is the syndrome extraction for Z error correction. The syndrome extraction network for X error correction is the same, except with each cnot replaced by cz.

results to determine if any of the ancilla were flipped. If there is exactly one Z error on the data qubits coming into the the syndrome extraction network, and no other failures occur, it will be detected by the classical syndrome extraction on the measurement results.

The syndrome extraction for X error-correction is the same, except that the seven transversal cnot gates get replaced by seven transversal cz gates which propagate X errors on the data to Z errors on the ancilla.

Note that in Section 2.2.3 we explained how to perform Z error-correction by first applying Hadamards to the data, then correcting X errors, and then reversing the

Hadamards on the data. On the surface this would appear to be different from our our present construction of the Z syndrome extraction network, but it is not: the sequence of gates (Hadamard on data)(cnot)(Hadamard on data) is equivalent to the gate cz.

This concludes our construction of the gates G, V, and S. We explain how these gates are used together in a full error-correcting circuit when we describe our model in Chapter 3.

2.2.5 Fault Tolerant Thresholds

A particularly effective method for quantum error correction is to take a quantum error correction code and concatenate it. That is, the code is applied to the code itself, ad infinitum, or (more physically) until a desired success probability is achieved. The process of concatenation is explained in further detail in Section 3.1.

One of the most important achievements of the theory of quantum fault-tolerance is the proof of various *threshold theorems*, originally proved by Aharonov and Ben-Or [2], Kitaev [9], and Knill, Laflamme, and Zurek [11], and improved by Preskill [13], Gottesman [8], and Aliferis, Gottesman, and Preskill [3]. The basic idea of each threshold theorem is that as long as the noise level of a quantum computation is below a certain constant threshold that is independent of the computation size, then arbitrarily long quantum computations can be performed using concatenated codes.

The value of the threshold for the [[7,1,3]] code has been estimated by several authors, with estimates varying between 10^{-6} and 3×10^{-2}. Zalka [22] estimated the threshold to be about 10^{-3} and argued that it might still be larger. Preskill [14] estimated a threshold of about 3×10^{-4}. Aharonov and Ben-Or [2] estimated the threshold to be 10^{-6} using a quantum circuit that did not require classical computation.

The above estimates were calculated before Steane found improved ancilla preparation circuits [19, 20] that eliminate the need for repeated measurements during ancilla preparation. With the new circuits, Steane estimated the threshold to be on the order of 10^{-3}.

Reichardt [16] used a modified version of Steane's ancilla preparation network (using error detection as well as error correction) to increase the threshold estimate to about 10^{-2}, at the cost of creation of more ancilla.

Svore, Terhal, and DiVincenzo [21] used the same circuits as Steane, but performed a more detailed analysis of the threshold by separating the types of noise according to types of gates and analytically approximating the new level of each type of noise upon code concatenation. They estimated the threshold to be about 3×10^{-4} when all error rates are the same and the memory error rate is a factor of 10 smaller. Some of our work, especially Section 4.3, was based on their analysis.

In the above estimates, it was assumed (implicitly or explicitly) that the noise could be modeled as depolarizing noise at all levels of the concatenated code. Little work has been published regarding the change in the distribution of errors and the possible effects on the threshold. The threshold that we present in this thesis (see Section 5.5) is the first to consider the effects of changing noise channels on the threshold.

Chapter 3

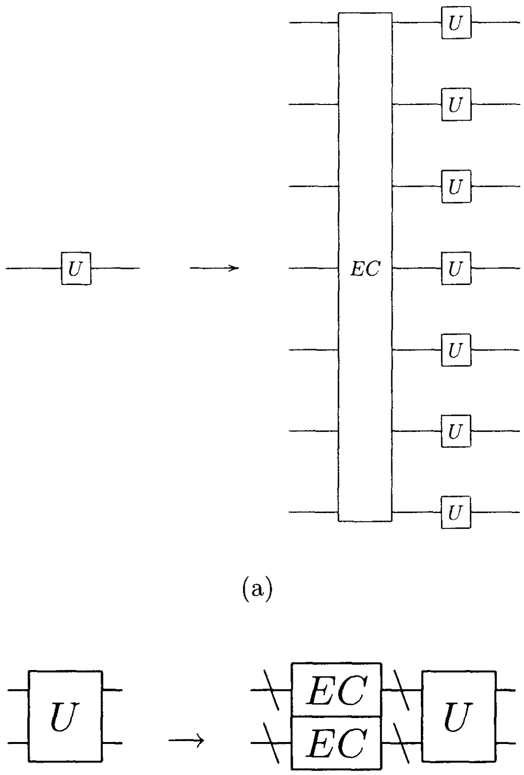
The Model

A detailed analysis of the effective noise channel of fault-tolerant quantum computation is difficult to carry out in general due to the many parameters of that noise channel and the numerous classes of codes and circuit constructions. For this reason, we have chosen to focus on the smallest CSS code correcting one quantum error (the [[7,1,3]] code), the generalized depolarizing channel, and the most efficient known fault-tolerance constructions for CSS codes. Both the code and its constructions were introduced in Chapter 2. As we will see in Chapter 4, this choice leads to a tractable analysis that is prototypical of all CSS fault-tolerance analyses.

In this chapter, we lay out the model we have chosen for recursively simulating fault-tolerant gates, correcting errors on logical qubits, and modeling faults in circuits. Section 3.1 describes so-called *replacement rules*, recursive rules for inserting fault-tolerant gates in place of basic gates. Section 3.2 details the fault-tolerant error-correction subroutine that appears in each fault-tolerant gate. Finally, Section 3.3 enumerates the modeling decisions that abstract the quantum computer hardware and its environment-induced noise.

3.1 Replacement Rule

To obtain an encoded circuit, we replace every gate U by a circuit that encodes U via a *replacement rule*. Figure 3-1 shows the replacement rule for single qubit and two



(a)

(b)

Figure 3-1: (a) The replacement rule for a single qubit gate. (b) The replacement rule for a two qubit gate.

qubit gates. In the replacement rule for a qubit gate U_i, each qubit gets replaced by seven qubits followed by an error-correction subroutine, and the gate U_i gets replaced by a new gate $\mathbf{U_i}$ that acts transversally on all the qubits. EC represents the error-correction circuit, which we describe in the next Section 3.2. The replacement rule is applied L times to construct a level L concatenated code.

The replacement rule is applied to every *location*. A location for our purposes is either a one qubit gate, a two qubit gate, a preparation (creation of the zero state), a measurement of the Pauli Z operator, or a wait gate. We list the replacement rule for each type of location:

1. one qubit gate: see Figure 3-1(a)

2. two qubit gate: see Figure 3-1(b)

3. preparation: a preparation of the state $|0\rangle$ gets replaced by a circuit that prepares the logical zero state $|0\rangle_L$. We do not concern ourselves with the construction of this circuit, because we will later just assume that a preparation fails with about the same probability as a single qubit gate.

4. measurement: a measurement of the Z operator on a single qubit gets replaced by a measurement of the ZZZZZZZ operator on a logical qubit and classical processing involving the parity check matrix. The seven qubit measurement is accomplished by using seven transversal Z measurements.

5. wait gate: A wait gate is a single qubit gate, so Figure 3-1(a) gives its replacement rule.

3.2 Error Correction Circuit

The general layout of the X or Z error-correction circuit is shown in Figure 3-2. The gates S^j in the figure mean either S_x^j for X error-correction or S_z^j for Z error-correction.

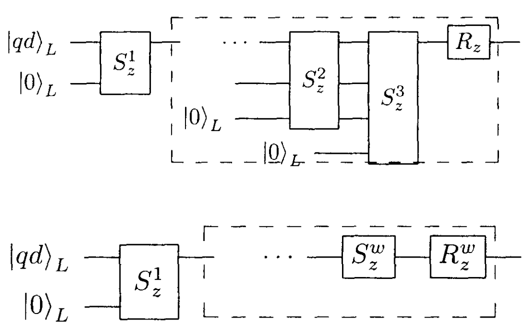


Figure 3-2: The error correction routine finds and corrects errors on the seven data qubits in the logical state $|qd\rangle_L$ with the aid of multiple copies of ancilla qubits in the logical zero state $|0\rangle_L$. The second half of the circuit is on of two possibilities, depending on whether the first syndrome extraction S_z^1 was zero or non-zero. If the syndrome is non-zero, then two more syndromes are collected (middle circuit), but if the syndrome is zero, no more syndromes are collected and the data qubits wait (righmost circuit) during the syndrome extraction circuit acting on other qubits.

We explain the error-correction routine step-by-step. In the following explanation, S^j is to be replaced by either S_x^j or S_z^j depending on whether the error-correction routine is X or Z, respectively.

1. The ancilla qubits are prepared via the preparation network G, and verified by the verification network V. The number of ancilla prepared is usually referred to as n_{rep}. We assume that n_{rep} is large enough so that enough ancilla consistently pass the verification network for the successful completion of the rest of the error correction.

2. The (X or Z)-error syndrome is extracted by S^1. Classical processing is done on the measurement results to determine the syndrome.

3. If the syndrome is non-zero, then two more syndromes are extracted via second and third applications of the S network: S^2 and S^3. The ancilla qubits that come into S^2 wait during the network S^1, and the ancilla qubits that come into S^3 wait during S^1 and S^2.

4. If a majority of the syndrome extractions agree, then an X or Z gate is applied to the agreed upon qubit while the other six data qubits wait. This is the recovery gate R. If there is no majority agreement, no further steps are taken (as in [21] but not as in [19]).

5. If the syndrome is zero, then the data waits for an amount of time equal to the total length of S^2 and S^3. The gate for these six time steps of waiting is called S^w.

6. Also if the syndrome is zero, all data qubits wait during the possible recovery of data qubits in other blocks. The gate for waiting during recovery is called R^w.

The circuits for G, V, and S were designed in Section 2.2.4 and are listed in Appendix C.

The full error-correction circuit, EC, consists of two copies of the circuit in Figure 3-2, one for X error-correction (S_x^j) and one for Z error-correction (S_z^j).

Some error-correction circuits will have Z error-correction followed by X error-correction. Other error-correction circuits will have X error-correction followed by Z error-correction. The rule that determines the appropriate order is that the first error correction corrects the error that is more likely to be on the qubits. Thus, the order of error-corrections remains the same after every gate except the Hadamard gate, after which the error-corrections are swapped, because Hadamard gates swap X and Z errors.

A few error-correction circuits will actually need to have three error-correction steps: S_x, followed by S_z, followed by S_x; or S_z, followed by S_x, followed by S_z. The rule that determines when this happens is that before every cz gate, the last error-correction must be S_x on both qubits, and before every cnot gate, the last error-correction on the control qubit must be S_x and the last error-correction on the target qubit must be S_z. The reason for prescribing the last error-correction before a two qubit gate is to minimize the probability of an error propagating from one logical qubit to the other. Qubits being error-corrected elsewhere need to wait during the third error-correction.

The order of error-corrections for each gate can be chosen to minimize the number of places where three consecutive error-correction steps are required. When the error-correction routine is itself error-corrected, three consecutive error-correction steps are required only when a cnot follows a cz or a cz follows a cnot and only on the data qubits. This happens infrequently enough that we approximate the failure rate of the error-correction gate by assuming that it consists of only two error-correction steps.

3.3 Modeling Choices

Somewhat following [21], a noise error can occur at any of five types of *locations* in the circuit: a single qubit gate with failure rate γ_1; a two qubit gate, γ_2; a single qubit wait (or memory) gate, γ_w; a preparation gate, γ_p; and a single qubit measurement of the Pauli Z operator, γ_m.

We model noisy locations as follows. At a location i, the corresponding gate (or procedure in the case of preparation or measurement) is performed perfectly with probability $(1 - \gamma_i)$, and a failure occurs with probability γ_i

As in [19] we distinguish between *failures* and *errors*. A *failure* is an imperfection caused by a single gate, while an *error* is an imperfection on a single qubit as a result of a failure. A single failure may cause multiple errors when the failure is on a two qubit gate.

The noise model we adopt assumes that failures are uncorrelated and stochastic.

The single qubit failures are X, Y, and Z, which occur with equal probability in the depolarizing channel. They are labeled by the failures they cause and are defined to occur *before* the erroneous gate. For example, an X failure on a Hadamard gate causes an X error to occur *before* the gate, which becomes a Z error after propagating through the Hadamard gate.

The two qubit failures are IX,IY,IZ,XI,XX,XY,XZ,YI,YX,YY,YZ,ZI,ZX,ZY,ZZ, which occur with equal probability in the depolarizing channel. They are labeled by the pair of errors they cause, and are defined to occur *before* the erroneous two qubit gate. The two qubit gates that appear in the error-correction circuit always have as inputs one data qubit and one ancilla qubit, with the ancilla as control. We define the order of the single qubit errors in each pair to be control-target (or ancilla-data).

We need to define failures as coming *before* their corresponding gates. The reason we make this seemingly arbitrary decision will be made apparent in the Analysis Chapter 4.

In addition to our choice of noise model, we make the following modeling choices:

- We assume that the time it takes to do a measurement followed by any necessary classical processing on the result takes one time step.

- We do not concern ourselves with the method of preparation of the single state $|0\rangle$. We call the preparation of the state $|0\rangle$ a preparation gate, which fails with probability γ_p (the error occurring after the preparation). We discover that magnitude of γ_p has very little affect on the threshold, so we just set $\gamma_p = \gamma_1$ at all levels error-correction as an approximation.

- Each type of location can have a different noise channel, though the noise channel for every type of location is depolarizing at level zero of the error-correction.

- At each level of error-correction, we assume that the noise channel for a given type of location is the same for every instance of that type of location. This is not true in general (for example, when the initial noise channel is heavily weighted toward X or Z failures, then the effective noise channel of a given instance of a location depends on whether that location immediately follows a Hadamard gate, which swaps X and Z errors). However, the assumption is fairly accurate when the initial noise channel is depolarizing, as will be shown in Section 5.4.

Chapter 4

Analytical Approximation

In this Chapter we provide an analytical model for studying higher level noise channels and the threshold for the Steane [[7,1,3]] code. A novel feature of our model is that its input noise channel is not necessarily depolarizing, and it predicts the noise channel at the next level of error-correction. Also, our model meticulously accounts for incoming errors, calculating separately the probabilities of X, Y, and Z errors coming into the X error-correction subroutine and into the Z error-correction subroutine. Furthermore, our model *exactly* counts all pairs of errors that could lead to a logical error when estimating the threshold.

We begin the chapter with a section explaining the overall structure of our analysis. Then, after we set up some notation in Section 4.2, we proceed to calculate the probability that the verification network passes with and without errors (Section 4.3), the probabilities of incoming errors on the data (Section 4.4), the effective nose channel at all of levels of error-correction (Section 4.5), and finally the threshold (Section 4.6). The results of our analytical model with some comparisons to simulations are given in the following Chapter 5.

4.1 Analysis Overview

We calculate the threshold for the Steane [[7,1,3]] code step-by-step, using the results of each section in each of the following sections, calculating the higher level noise channels along the way and eventually deriving a method for calculating the threshold in the last section. We believe it is instructive to give an overview of the analytical model in reverse order, explaining first how to calculate the threshold, and then explaining how to calculate the quantities used to calculate the threshold.

To calculate the threshold in Section 4.6 we only need to know the noise channel for each type of gate for every level of error-correction

We calculate the noise channel for each type of gate at every level of concatenation in Section 4.5. The noise channel is determined by calculating the probabilities of logical X, Y, and Z failures (in the case of a single qubit gate). The probability of a logical failure in an error-correction circuit EC depends on whether or not there is an incoming error on the data into EC. With knowledge of the probabilities of incoming errors on the data, the probabilities of logical failures can be determined by counting the number of ways one or two more failures in addition to the incoming errors can cause a logical failure.

We calculate the probabilities of incoming errors on the data into EC in Section 4.4. We do so by solving six linear equations in six unknowns. Each probability of an incoming error on the data is calculated in terms of the probabilities of the other incoming errors on the data along with the probabilities of incoming errors on the *ancilla*. We calculate the probabilities of incoming errors on the ancilla into EC in Section 4.3, the first section in our determination of the threshold.

4.2 Notation

This section sets up the mathematical notation used in the following analysis.

4.2.1 The Error Correction Network

We take the order of error correction to be Z error correction followed by X error correction for notational purposes. There is no loss of generality here as long as Z failures and X errors always occur with equal probability.

The Z error correction gates are

$$S_z^1,\ S_z^2,\ S_z^3,\ S_z^w,\ R_z, \text{ and } R_z^w, \tag{4.1}$$

where S_z^1, S_z^2, and S_z^3 are the three syndrome extraction circuits; S_z^w is the collection of wait gates if there are no second and third syndrome extractions; R_z is the recovery if there is a detected error; and R_z^w is the collection of wait gates that take the place of recovery if there is no detected error.

In addition, the gates V_z^1, V_z^2, and V_z^3 are the verification networks that precede S_z^1, S_z^2, and S_z^3. We define V^2 and V^3 to be the concatenation of the V^1 network with the additional wait gates on the ancilla that occur during the first syndrome, and the first two syndromes, respectively.

Any gate can be divided into its individual time steps by adding an extra superscript specifying the time step. For example, the gate during the first time step of S_z^1 is $S_z^{1,1}$, and the gate during last two time steps is $S_z^{1,t>1}$.

Similarly, the X error correction gates are

$$S_x^1,\ S_x^2,\ S_x^3,\ S_x^w,\ R_x, \text{ and } R_x^w, \tag{4.2}$$

with V_x^1, V_x^2, and V_x^3 defined analogously.

4.2.2 Failure Rates

The failure rates for single qubit gates, two qubit gates, wait gates, preparation gates, and measurements, at level ℓ of concatenation are $\gamma_1(\ell)$, $\gamma_2(\ell)$, $\gamma_w(\ell)$, $\gamma_p(\ell)$, and $\gamma_m(\ell)$, respectively. Note that the case $\ell = 0$ corresponds to the failure rates for the depolarizing channel defined in Section 3.3.

We denote the probability of a specific failure by adding that failure as a superscript. Then each failure rate is the sum of the probabilities over all specific failures:

$$
\begin{aligned}
\gamma_1 &= \gamma_1^X + \gamma_1^Y + \gamma_1^Z, \\
\gamma_w &= \gamma_w^X + \gamma_w^Y + \gamma_w^Z, \\
\gamma_p &= \gamma_p^X + \gamma_p^Y + \gamma_p^Z, \\
\gamma_m &= \gamma_m^X + \gamma_m^Y + \gamma_m^Z, \\
\gamma_2 &= \gamma^{IX} + \gamma^{XI} + \gamma^{IZ} + \gamma^{ZI} \\
&\quad + \gamma^{IY} + \gamma^{YI} \\
&\quad + \gamma^{XX} + \gamma^{XY} + \gamma^{XZ} + \gamma^{YX} + \gamma^{YY} + \gamma^{YZ} + \gamma^{ZX} + \gamma^{ZY} + \gamma^{ZZ},
\end{aligned} \tag{4.3}
$$

where we left out the dependence on ℓ, since the above equations hold at all levels of concatenation.

The list of possible two qubit errors is long, so we divide the list into three kinds of failures:

$$
\begin{aligned}
\gamma_2^{IW} &\equiv \gamma^{IX} = \gamma^{XI} = \gamma^{IZ} = \gamma^{ZI}, \\
\gamma_2^{IY} &\equiv \gamma^{IY} = \gamma^{YI}, \\
\gamma_2^{AB} &\equiv \gamma^{XX} = \gamma^{XY} = \gamma^{XZ} = \gamma^{YX} = \gamma^{YY} = \gamma^{YZ} = \gamma^{ZX} = \gamma^{ZY} = \gamma^{ZZ}.
\end{aligned} \tag{4.4}
$$

We chose these three categories based on the expectation that each will occur with a very different probability at higher levels of error correction. For level one error correction, we expect γ_2^{IW} to be approximately one order of magnitude greater than γ_2^{IY} and approximately two orders of magnitude greater than γ_2^{AB}, which we will treat as zero when we calculate the threshold.

4.2.3 Probabilities

In the coming analysis, we write out many probabilities. To write each one out in rigorous mathematical notation would take up a lot of space and would make the longer equations difficult to interpret. For this reason, we have developed a well-defined shorthand for nearly all of the probabilities that occur in our analysis.

We write nearly all probabilities of the form

$$\mathbb{P}([\text{no}]\ [\text{inc}]\ \textit{errors}\ [\text{caused}]\ \text{on}\ A_1, A_2, ...\ \textit{qubits}), \tag{4.5}$$

The [no] and the [inc] are optional; *errors* is a list of errors; $A_1, A_2, ...$ is a list of gates; and *qubits* is either "data" or "anc." To save space, *qubits* is "data" when not specified.

The above shorthand is intended to have a rather intuitive meaning, so do not get bogged down by the following definitions. We provide the following four definitions for the purpose of mathematical rigor and to avoid ambiguity:

$$\mathbb{P}(\text{no}\ \textit{errors}\ [\text{caused}]\ \text{on}\ A_{,}A_2, ...1\ \textit{qubits}) \equiv \tag{4.6}$$

"the probability that no errors in *errors* occur on the *qubits* during the gates $A_1, A_2, ...$"

$$\mathbb{P}(\text{no inc}\ \textit{errors}\ \text{on}\ A_1, A_2, ...\ \textit{qubits}) \equiv \tag{4.7}$$

"the probability that there are no errors in the list *errors* on the *qubits* incoming into each of the gates $A_1, A_2, ...$"

$$\mathbb{P}(\textit{errors}\ [\text{caused}]\ \text{on}\ A_1, A_2, ...\ \textit{qubits}) \equiv \tag{4.8}$$

"the probability that at least one error in *errors* occurs on the *qubits* during at least one of the gates $A_1, A_2, ...$"

$$\mathbb{P}(\text{inc}\ \textit{errors}\ \text{on}\ A_1, A_2, ...\ \textit{qubits}) \equiv \tag{4.9}$$

"the probability that there is at least one error in *errors* on the *qubits* incoming into at least one of the gates $A_1, A_2, ...$"

Sometimes we will find it useful to refer to the probability that a certain error is left on a qubit *after* a gate due to the gate. When we want to refer to errors left on qubits, we insert the word *caused* into the statement of the probability. For example, a Z error caused on a Hadamard gate is the same as an X failure on a Hadamard gate and occurs with probability γ_1^X, because Hadamard gates propagate X errors to Z errors. Similarly, an XZ error caused by a cz gate is the same as an XI failure on a cz gate and occurs with probability γ_2^{IX} rather than γ_2^{AB}.

For example, $\mathbb{P}$(no X,Y on S_x^1, S_x^2, S_x^3) means that no X or Y failure occurs during the gates S_x^1, S_x^2, and S_x^3.

The comma separated list of gates exists to save space, and can be eliminated using the following rule:

$$\mathbb{P}(\text{no [inc] } \textit{errors} \text{ on } A_1, A_2, \ldots\ \textit{qubits}) \approx \prod_i \mathbb{P}(\text{no [inc] } \textit{errors} \text{ on } A_i\ \textit{qubits}). \quad (4.10)$$

Equation 4.10 is an equality when the failures on each gate are independent. The equality holds for level zero of the error-correction, since we assumed that intial failures were uncorrelated and stochastic. However, at higher levels of error-correction, failure rates need not be independent. A logical failure during one error-correction routine can inrease the probabilitiy of a logical failure on the next error-correction routine via an incoming error on the data. We assume that this scenario has little affect on the threshold and take Equation 4.10 to be an equality. We do so only to simplify our analysis.

When there is just one gate in the list of gates, the probability can be looked up in Appendix A. The rule 4.10 is useful because it can reduce most probabilities in the following sections to a product or sum of probabilities that can be looked up in Appendix A.

4.3 Alpha

We calculate alpha, the probability that the verification network passes. Along with alpha, we calculate the probabilities that the verification network passes with various errors (X or Y; Z or Y; X,Y, or Z). Errors on passed ancilla can propagate to the data and can cause incorrect syndrome extractions, affecting the crash probability.

Our approximation follows [21], except that more attention is paid to the details of the verification network. Similar to [21], but treating X, Y, and Z as distinct errors, alpha can be expressed exactly as

$$\begin{aligned}
\alpha &= \mathbb{P}(\text{pass and no inc X,Y on } S^1 \text{ anc}) + \mathbb{P}(\text{pass and inc X,Y on } S^1 \text{ anc}) \\
&= \mathbb{P}(\text{pass and no inc X,Y on } S^1 \text{ anc}) + \mathbb{P}(\text{pass and no inc Z,Y on } S^1 \text{ anc}) \\
&\quad -\mathbb{P}(\text{pass and no inc X,Y,Z on } S^1 \text{ anc}) + \mathbb{P}(\text{pass and inc Y on } S^1 \text{ anc}) \\
&\quad +\mathbb{P}(\text{pass and inc X and Z on } S^1 \text{ anc}) \\
&\quad -\mathbb{P}(\text{pass and inc X and Y and Z on } S^1 \text{ anc}),
\end{aligned} \tag{4.11}$$

where S^1 is either S^1_z or S^1_x.

The last two terms of Equation 4.11 are set to zero in our approximation. They require at least two errors to occur in V^1, whereas the other four terms require only one or zero. We keep the first four terms.

To approximate the first three terms analytically, we determine which single gate failures in G and V^1 lead to "good" outcomes for the corresponding probability. For a simple example, a Z failure on the ancilla during the last time step of V^1 is a single gate failure that leads to a good outcome for $\mathbb{P}$(pass and no inc X,Y on S^1 anc) but a bad outcome for $\mathbb{P}$(pass and no inc Z,Y on S^1 anc). Tables 4.1, 4.2, and 4.3 list the failures in G and V^1 that cause "good" outcomes.

Usually, there is a "bad" outcome exactly when one of the following happens: an X,Y error is left on the ancilla by G; an X,Y error is caused on the ancilla in V^1 and propagates to the verification qubits; a failure on a verification qubit leads to an X,Y error left on the verification qubits just before measurement in V^1; or a failure in G or V^1 leaves an undesired error on the ancilla at the end of V^1.

Some of the entries in the tables are not obvious. For example, an XI error on the first cnot in G does not lead to any errors coming out of G, even though one would expect the X error to propagate to several ancilla qubits. In general we need to consider the stabilizer of the state that a error occurs on, because in this case the stabilizer of the control qubit is X, so the X "error" has no effect.

Table 4.1 lists the failures in G that lead to good outcomes for $\mathbb{P}$(pass and no inc X,Y on S^1anc) and $\mathbb{P}$(pass and no inc Z,Y on S^1 anc).

	top prep	bot. prep	had	early cnot	mid. cnot	late cnot	early wait	late wait
Pass no X,Y	Z	XYZ	XYZ	IZ,ZI,ZZ	IZ,ZI,ZZ	IZ,ZI,ZZ	Z	Z
Pass no Z,Y	Z	Z	Z	XI,YZ,IZ	ZZ	-	Z	-
# of gates	4	3	3	3	1	5	5	2

Table 4.1: This table lists the failures in the G network that lead to good outcomes for the probabilities $\mathbb{P}$(pass and no inc X,Y on S^1 anc) and $\mathbb{P}$(pass and no inc Z,Y on S^1 anc). The bot. prep. gates are the preparation gates followed by Hadamards, and the top prep. gates are the ones that are not followed by Hadamards. The early cnot gates are the three in the second time step, the mid. cnot gate is the cnot gate in the third time step that still acts on a $|0\rangle$, while the late cnot gates include all others.

	prep/ early had	late had	early cZ	mid cZ	late cZ	ms	early anc. wait	late anc. wait	early ver. wait	late ver. wait
Pass no X	Z	X	XZ,IZ XI	XZ,IZ, XI	XZ,IZ, XI	Z	Z	Z	X	X
Pass no ZY	Z	X	ZX	-	XZ,ZX, YY	Z	-	X	-	X
# of gates	4/4	4	4	6	3	4	3	26	2	1

Table 4.2: This table lists the failures in the V^1 network that lead to good outcomes for the probability $\mathbb{P}$(pass and no inc X,Y on S^1 anc) and $\mathbb{P}$(pass and no inc Z,Y on S^1 anc). ms is short for measurement gate.

Table 4.2 does the same for V^1. Using these two tables we calculate that

$\mathbb{P}$(pass and no inc X,Y on S^1 anc) =

$$(1-\gamma_p^X-\gamma_p^Y)^4(1-2\gamma_2^{IW}-2\gamma_2^{IY}-8\gamma_2^{AB})^9(1-\gamma_w^X-\gamma_w^Y)^7(1-\gamma_p^X-\gamma_p^Y)^4$$
$$\times(1-\gamma_1^X-\gamma_1^Y)^4(1-\gamma_1^Z-\gamma_1^Y)^4(1-2\gamma_2^{IW}-2\gamma_2^{IY}-8\gamma_2^{AB})^7(1-\gamma_2)^6$$
$$\times(1-\gamma_m^X-\gamma_m^Y)^4(1-\gamma_w^X-\gamma_w^Y)^{29}(1-\gamma_w^Z-\gamma_w^Y)^3 \quad (4.12)$$

and

$\mathbb{P}$(pass and no inc Z,Y on S^1 anc) =

$$(1-\gamma_p^X-\gamma_p^Y)^7(1-\gamma_1^X-\gamma_1^Y)^3(1-2\gamma_2^{IW}-2\gamma_2^{IY}-8\gamma_2^{AB})^3$$
$$\times(1-4\gamma_2^{IW}-2\gamma_2^{IY}-8\gamma_2^{AB})^1(1-\gamma_2)^5(1-\gamma_w^X-\gamma_w^Y)^5(1-\gamma_w)^2$$
$$\times(1-\gamma_p^X-\gamma_p^Y)^4(1-\gamma_1^X-\gamma_1^Y)^4(1-\gamma_1^Z-\gamma_1^Y)^4(1-4\gamma_2^{IW}-2\gamma_2^{IY}-8\gamma_2^{AB})^4$$
$$\times(1-\gamma_2)^6(1-4\gamma_2^{IW}-2\gamma_2^{IY}-6\gamma_2^{AB})^3(1-\gamma_m^X-\gamma_m^Y)^4(1-\gamma_w)^5$$
$$\times(1-\gamma_w^Z-\gamma_w^Y)^{27} \quad (4.13)$$

	prep	early had	last had	last cZ	other cZ, cnots	meas	early waits	last ver. wait	other waits
Pass no X,Y,Z	Z	Z	X	XZ	-	Z	Z	X	-
# of gates	11	7	4	4	18	4	5	1	33

Table 4.3: This table lists the failures in the G and V^1 networks that lead to good outcomes for the probabilities $\mathbb{P}$(pass and no inc X,Y,Z on S^1 anc). The last cZ gates are the cZ gates that are the last to act on each verification qubit.

Table 4.3 lists the failures in G and V^1 that lead to good outcomes for $\mathbb{P}$(pass and no inc X,Y,Z on S^1 anc). Using Table 4.3 we calculate that

$$
\begin{aligned}
&\mathbb{P}(\text{pass and no inc X,Y,Z on } S^1 \text{ anc}) = \\
&\qquad \left(1-\gamma_p^X-\gamma_p^Y\right)^{11}\left(1-\gamma_1^X-\gamma_1^Y\right)^7\left(1-\gamma_1^Z-\gamma_1^Y\right)^4 \\
&\qquad \times\left(1-4\gamma_2^{IW}-2\gamma_2^{IY}-8\gamma_2^{AB}\right)^4\left(1-\gamma_2\right)^{18}\left(1-\gamma_m^X-\gamma_m^Y\right)^4 \\
&\qquad \times\left(1-\gamma_w^X-\gamma_w^Y\right)^5\left(1-\gamma_w^Z-\gamma_w^Y\right)^1\left(1-\gamma_w\right)^{33}.
\end{aligned}
\tag{4.14}
$$

Finally, we approximate $\mathbb{P}($ pass and inc Y on S^1 anc$)$. The ancilla pass with a Y error only when an ZY failure (causes IY error) occurs on the first four control-Z gates, when a ZY or YX failure (causes IY or XY error) occurs on the last three control-Z gates, or when a Y failure occurs on the waiting ancilla after the control-Z gates. Thus,

$$
\mathbb{P}(\text{pass and inc Y on } S^1 \text{ anc}) \approx 10\gamma_2^{AB}+26\gamma_w^Y. \tag{4.15}
$$

4.4 Incoming Errors on Data

In this section we derive a set of linear equations for the probabilities of incoming errors into S_z^1 and S_x^1 in the steady state. We calculate separately the probabilities of X, Y, and Z errors on the data coming into S_z^1 and S_x^1. The probabilities to be derived are

$$
\begin{aligned}
P_z^X &\equiv \mathbb{P}(\text{inc X on } S_z^1), \\
P_z^Y &\equiv \mathbb{P}(\text{inc Y on } S_z^1), \\
P_z^Z &\equiv \mathbb{P}(\text{inc Z on } S_z^1), \\
P_x^X &\equiv \mathbb{P}(\text{inc X on } S_x^1), \\
P_x^Y &\equiv \mathbb{P}(\text{inc Y on } S_x^1), \\
\text{and } P_x^Z &\equiv \mathbb{P}(\text{inc Z on } S_x^1).
\end{aligned}
\tag{4.16}
$$

We could have chosen to make the approximations $P_z^X = P_x^Z$, $P_z^Y = P_x^Y$, and $P_z^Z = P_x^X$, but we did not because (1) it is simply not true because of the gate U_i, (2) it is interesting to discover by how much they differ, and (3) the approximation is not needed to simplify the theory, since either way we need only to write two equations to represent all of them.

First, we find P_z^X, the probability that there is an X error on the data coming into S_z^1. The same equation is used to find P_z^Y, by replacing the letter X by Y where indicated by the symbol [Y].

$$
\begin{aligned}
&\mathbb{P}(\text{no inc X[Y] on } S_z^1) = 1 - P_z^{X[Y]} = \\
&\quad [\mathbb{P}(\text{no inc X,Y on } S_x^1)\times \\
&\quad\quad [\mathbb{P}(\text{no inc Z,Y on } S_x^1 \text{ anc})\times \\
&\quad\quad\quad [\mathbb{P}(\text{no Z,Y caused on } S_x^{1,1} \text{ anc})\times \\
&\quad\quad\quad\quad [\mathbb{P}(\text{no X,Y caused on } S_x^{1,t>1} \text{ anc}) \\
&\quad\quad\quad\quad\quad\quad \times \mathbb{P}(\text{no X[Y] caused on } S_x^1, S_x^w, R_x^w, U_i | \text{no Z,Y caused on } S_x^{1,1} \text{ anc}) \\
&\quad\quad\quad\quad + (1 - \mathbb{P}(\text{no X,Y caused on } S_x^{1,t>1} \text{ anc})) \\
&\quad\quad\quad\quad\quad\quad \times \mathbb{P}(\text{no X[Y] caused on } S_x^{2,t>1}, S_x^3, R_x, U_i)] \\
&\quad\quad\quad + (1 - \mathbb{P}(\text{no Z,Y caused on } S_x^{1,1} \text{ anc}))\mathbb{P}(\text{no X[Y] caused on } S_x^{2,t>1}, S_x^3, R_x, U_i)] \\
&\quad\quad + (1 - \mathbb{P}(\text{no inc Z,Y on } S_x^1 \text{ anc}))\mathbb{P}(\text{no X[Y] caused on } S_x^{2,t>1}, S_x^3, R_x, U_i)] \\
&\quad + (1 - \mathbb{P}(\text{no inc X,Y on } S_x^1))\mathbb{P}(\text{no X[Y] caused on } S_x^1, S_x^2, S_x^3, R_x, U_i)]
\end{aligned}
\tag{4.17}
$$

For there to be no incoming X[Y] error on S_z^1, the following must occur: (1) If there is an incoming X or Y error on the preceding S_x^1 (we assume this causes a non-zero syndrome), then there must be no X[Y] error caused on the data before S_z^1. (2) If there is *not* an incoming X or Y error on the preceding S_x^1, then either (a) there is no error on the ancilla (so the syndrome is zero) and there is no X[Y] error caused before S_z^1 or (b) an error on the ancilla causes a non-zero syndrome and there is no *uncorrectable* X[Y] error caused before S_z^1. Equation 4.17 expresses the above reasoning in the precise mathematical notation set up in Section 4.2.

Note that X[Y] errors cannot be caused on the data in S_x^1 via the propagation of errors from the ancilla, so X[Y] errors on the data must be caused by failures on the data only. This makes the equation for $P_z^{X[Y]}$ somewhat simpler than the equation we will later write for P_z^Z.

To obtain the equation for $P_x^{Z[Y]}$ from Equation 4.17, swap the labels x and z everywhere they occur, swap the errors X and Z whenever they refer to errors on the data (but not when they refer to errors on the ancilla), and remove every instance of the gate U_i.

Now we have four equations after writing only one, but we have introduced the unknown quantity, $\mathbb{P}(\text{no inc X,Y on } S_x^1)$ (along with $\mathbb{P}(\text{no inc Z,Y on } S_z^1)$ in the corresponding equation for $P_x^{Z[Y]}$). Before proceeding to write the equation for P_z^Z, we find this quantity in terms of the original six.

$$\begin{aligned}
\mathbb{P}(\text{no inc X,Y on } S_x^1) &= 1 - \mathbb{P}(\text{inc X on } S_x^1 \text{ or inc Y on } S_x^1) \\
&= 1 - P_x^X - P_x^Y + \mathbb{P}(\text{inc X on } S_x^1 | \text{inc Y on } S_x^1)\mathbb{P}(\text{inc Y on } S_x^1) \\
&\approx 1 - P_x^X - P_x^Y + P_x^X P_x^Y \\
&\approx 1 - P_x^X - P_x^Y.
\end{aligned} \tag{4.18}$$

The last approximation in Equation 4.23 allows the system of equations to be linear and only causes an error of about .5 percent on $P_z^{X[Y]}$, which is itself only about .5 percent near threshold. The second to last approximation assumes that the events of an incoming X error and an incoming Y error are independent, which they are not, but we expect the intersection of the two events to be relatively small (because an incoming X and Y error requires two independent failures instead of just one).

Similarly, we approximate

$$\mathbb{P}(\text{no inc Z,Y on } S_z^1) \approx 1 - P_z^Z - P_z^Y. \tag{4.19}$$

Second, we find P_z^Z, the probability that there is a Z error on the data coming into S_z^1.

$$\begin{aligned}
&\mathbb{P}(\text{no inc Z on } S_z^1) = 1 - P_z^Z = \\
&\quad \mathbb{P}(\text{no inc X,Y on } S_x^1 \text{ anc})\mathbb{P}(\text{no inc Z,Y on } S_x^1)\times \\
&\quad [\mathbb{P}(\text{no inc X,Y on } S_x^1 | \text{no inc Z,Y on } S_x^1)\times \\
&\qquad [\mathbb{P}(\text{no inc Z on } S_x^1 \text{ anc})\times \\
&\qquad\quad [\mathbb{P}(\text{no Z,Y caused on } S_x^{1,1} \text{ anc})\times \\
&\qquad\qquad [\mathbb{P}(\text{no X,Y caused on } S_x^{1,t>1} \text{ anc}) \\
&\qquad\qquad\quad \times \mathbb{P}(\text{no Z caused on } S_x^1, S_x^w, R_x^w, U_i | \text{no Z,Y caused on } S_x^{1,1} \text{ anc}) \\
&\qquad\qquad + (1 - \mathbb{P}(\text{no X,Y caused on } S_x^{1,t>1} \text{ anc})) \\
&\qquad\qquad\quad \times \mathbb{P}(\text{no Z,Y caused on } S_x^1 | \text{no Z,Y caused on } S_x^1 \text{ anc}) \\
&\qquad\qquad\quad \times \mathbb{P}(\text{no Z caused on } S_x^2, S_x^3, R_x^w, U_i)\mathbb{P}(\text{no inc X,Y on } S_x^2, S_x^3 \text{ anc})] \\
&\qquad\quad + (1 - \mathbb{P}(\text{no Z,Y caused on } S_x^{1,1} \text{ anc})) \\
&\qquad\qquad \times \mathbb{P}(\text{no Z,Y caused on } S_x^1 | \text{Z,Y caused on } S_x^{1,1} \text{ anc}) \\
&\qquad\qquad \times \mathbb{P}(\text{no Z caused on } S_x^2, S_x^3, R_x^w, U_i)\mathbb{P}(\text{no inc X,Y on } S_x^2, S_x^3 \text{ anc})] \\
&\qquad + (1 - \mathbb{P}(\text{no inc Z on } S_x^1 \text{ anc}))\mathbb{P}(\text{no Z,Y caused on } S_x^1) \\
&\qquad\quad \times \mathbb{P}(\text{no Z caused on } S_x^2, S_x^3, R_x^w, U_i)\mathbb{P}(\text{no inc X,Y on } S_x^2, S_x^3 \text{ anc})] \\
&\quad + (1 - \mathbb{P}(\text{no inc X,Y on } S_x^1 | \text{no inc Z,Y on } S_x^1))\mathbb{P}(\text{no Z,Y caused on } S_x^1) \\
&\qquad \times \mathbb{P}(\text{no Z caused on } S_x^2, S_x^3, R_x, U_i)\mathbb{P}(\text{no inc X,Y on } S_x^2, S_x^3 \text{ anc})]
\end{aligned} \tag{4.20}$$

For there to be no incoming Z error on S_z^1, the following must occur: There must be no incoming Z or Y error on the data preceding S_x^1 and no incoming X or

Y errors on the ancilla coming into S_x^1. Also, (1) If there is an incoming X or Y error on the preceding S_x^1 (we assume this causes a non-zero syndrome), then there must be no Z error caused on the data before S_z^1. (2) If there is *not* an incoming X or Y error on the preceding S_x^1, then either (a) there is no error on the ancilla (so the syndrome is zero) and there is no Z error caused before S_z^1 or (b) an error on the ancilla causes a non-zero syndrome and there is no *uncorrectable* Z error caused before S_z^1. Equation 4.20 expresses the above reasoning in the precise mathematical notation set up in section 4.2.

Note that Z errors can be caused on the data in S_x^1 via the propagation of errors from the ancilla, so this is included in the calculation of P_z^Z. Also, note that the events of various errors caused on the ancilla are not independent of the events various errors caused on on the data, due to the two qubit gates, so conditional probabilities must sometimes be used.

To obtain the equation for $P_x^{Z[Y]}$ from Equation 4.20, swap the labels x and z everywhere they occur, swap the errors X and Z whenever they refer to errors on the data (but not when they refer to errors on the ancilla), and remove every instance of the gate U_i.

We now have six equations, but again we have introduced some new probabilities, which we now approximate:

$$\mathbb{P}(\text{no inc Z,Y on } S_x^1) \approx 1 - P_x^Z - P_x^Y, \tag{4.21}$$

and

$$\begin{aligned} &\mathbb{P}(\text{no inc Z,Y on } S_x^1) \times [\mathbb{P}(\text{no inc X,Y on } S_x^1 | \text{no inc Z,Y on } S_x^1) \\ &= \mathbb{P}(\text{no inc X,Y,Z on } S_x^1) \\ &\approx 1 - P_x^X - P_x^Y - P_x^Z. \end{aligned} \tag{4.22}$$

Similarly, we approximate

$$\mathbb{P}(\text{no inc X,Y on } S_z^1) \approx 1 - P_z^X - P_z^Y, \tag{4.23}$$

and

$$\begin{aligned} &\mathbb{P}(\text{no inc X,Y on } S_z^1) \times [\mathbb{P}(\text{no inc Z,Y on } S_z^1 | \text{no inc X,Y on } S_z^1) \\ &= \mathbb{P}(\text{no inc X,Y,Z on } S_z^1) \\ &\approx 1 - P_z^X - P_z^Y - P_z^Z, \end{aligned} \tag{4.24}$$

By substituting in Equations 4.18, 4.19, 4.21, 4.22, 4.23,and 4.24 into the six equations represented by 4.17 and 4.20, we obtain six linear equations in P_z^X, P_z^Y, P_z^Z, P_x^X, P_x^Y, and P_x^Z, as desired. All of the other terms in Equations 4.17 and 4.20 can be simplified using the rule 4.10 and/or looked up in Table A.1 in Appendix A.

Though we do not pursue it in this paper, our theory can give the values of P_z^X, P_z^Y, P_z^Z, P_x^X, P_x^Y, and P_x^Z in the non-steady state. In the non-steady state, the quantities P_z^X, etc. would be labeled in temporal order: ${}_1P_z^X$, ${}_2P_z^X$, ${}_3P_z^X$, etc. The quantities ${}_nP_z^X$, ${}_nP_z^Y$, and ${}_nP_z^Z$ would be linear in ${}_{(n-1)}P_x^X$, ${}_{(n-1)}P_x^Y$, and ${}_{(n-1)}P_x^Z$, which would be linear in ${}_{(n-1)}P_z^X$, ${}_{(n-1)}P_z^Y$, and ${}_{(n-1)}P_z^Z$, and so on until we reach ${}_1P_z^X = {}_1P_z^Y = {}_1P_z^Z = 0$ (if there were initially no errors on the data). This would give an easily solvable system of $6n$ linear equations.

Our analysis of incoming errors assumed that the order of error-corrections was always Z error-correction followed by X error-correction, but when the gate being error corrected is a Hadamard gate, the order of error-corrections gets reversed. This would suggest that our analysis breaks down whenever a Hadamard gate is error-corrected, but our analysis does still hold – as long as X and Z failures occur with equal probability. When X and Z failures occur with equal probability, we are free to relabel the errors so that the first error-correction needed is Z error-correction, so our analysis holds.

The assumption that X and Z errors occur with equal probability puts a limit on the type of initial noise channel we can model (one of the reasons we restrict ourselves to depolarizing noise). However, if we lifted this assumption we would have other problems such as (1) the effective noise channel for a logical gate would depend on whether the preceding logical gate swaps errors, so the noise channel of a gate would depend on the circuit it belongs to, and (2) there might be better error-correction procedures that take into account the different noise channel, such as correcting the more likely error more often. These problems would make the analysis less tractable, so we keep the assumption that the initial noise channel is depolarizing.

4.5 Noise Channels

In this section we calculate the probabilities of logical X, Y, and Z failures on single qubit gates and measurements; and logical IX, IY, and IZ failures on two qubit gates. That is, we find the level $(\ell+1)$ noise channel in terms of the level ℓ noise channel.

4.5.1 Single Qubit Gate

In a [[7,1,3]] code error-correction routine, two X errors or one X and one Y error causes a logical X failure when S_x detects the two errors and misinterprets them as a single error, correcting the wrong qubit. Similarly, two Z errors or one Z and one Y error gets misinterpreted by S_z and cause a logical Z failure. Two Y errors cause a logical Y failure.

A *logical failure* is defined to occur whenever a failure occurs that puts the logical qubit into an uncorrectable state (a state that would be misinterpreted and incorrectly "corrected" by a noiseless error-correction circuit). At least two failures are needed to cause a logical failure.

For each logical error we approximate its probability of occurring by counting the ways in which exactly two failures (calling incoming errors on the data or ancilla "failures" for our present purpose). For example, one way to cause a logical X error is to have an incoming X or Y error before S_x^1 and an X error anytime before $S_x^{2,2}$. Another way to have a logical X failure is to have two X errors occur after $S_x^{1,1}$. In the last example, the logical X failure by our definition occurs at the time of the second error, *not* when the logical X state gets created by a misinterpreted syndrome in the following error-correction.

We then sum over all of the possible pairs of locations the probability of both errors occurring:

$$\gamma_i(\ell+1) = \sum_{a \geq b,\ \text{pair causes logical error}} \gamma_a(\ell)\gamma_b(\ell), \tag{4.25}$$

where a and b sequentially label all of the gates in the error correction network that act on the data, and also label the probabilities for incoming errors on the data or ancilla.

This sum implicitly assumes that as long as the two failures under consideration occur, there will be a logical failure regardless of what happens elsewhere in the network. This is a small over-approximation.

When counting the pairs of errors that lead to a logical failure, a pair of errors that act on the same qubit are not counted. Such pairs of errors cancel and do not cause a logical failure. If we counted these pairs of errors, we would over-count by about $1/7 \approx 14\%$ and expect our calculated failure rates to be inaccurate by the same percentage. So, the first order effect of the cancellation of errors is taken into account in our calculation of the failure rates.

As one last detail, we approximate the probability that there are two incoming X (or Y or Z) errors on the ancilla as $(6/7)(\gamma)^2$, where γ is the probability of one error coming in. This assumes the events are independent, which is almost but not the case.

Here we do the counting for and calculate the probability of a logical X or Y error on a single logical qubit. We count all pairs of failures that cause a logical failure. The counting for all logical errors is done in Appendix B. The counting is *exact*.

We calculate $\gamma_i^X(\ell+1) + \gamma_i^Y(\ell+1)$, the probability of a level $(\ell+1)$ X or Y error as follows:

$$
\begin{aligned}
\gamma_i^X(\ell+1)+\gamma_i^Y(\ell+1) \approx\ & \mathbb{P}(\text{logical X,Y failure}|\text{no inc X,Y,Z on } S_z^1)(\ell) \\
& + \mathbb{P}(\text{logical X,Y failure}|\text{inc X on } S_z^1)(\ell) \\
& + \mathbb{P}(\text{logical X,Y failure}|\text{inc Y on } S_z^1)(\ell) \\
& + \mathbb{P}(\text{logical X,Y failure}|\text{inc Z on } S_z^1)(\ell)
\end{aligned} \tag{4.26}
$$

We assume that no logical failure occurred in the previous error correction. For this reason, we do not need to consider the case of two incoming errors.

First, we consider the case that there are no incoming X,Y or Z errors on S_z^1. This occurs with probability $1-P_z^X-P_z^Y-P_z^Z$. In this case, there must be a pair of errors that cause a logical X or Y error.

Table 4.4 counts the pairs of errors that lead to a logical X or Y error. The errors indicated in the table are the errors *caused* on the qubits. The columns indicate the location of the first error and the rows indicate the location of the second error. Usually filled in each cell is the error that must by caused by both gates (or list of errors from which one must be caused by each gate). The errors in some cells are followed by 1 or 2, meaning that the specified error(s) must be the first or second error, respectively. In such a case, the other error is assumed to be in the list XY. The additional designations "s" and "n" indicate that the error causes further syndromes to be extracted (s) or must *not* cause further syndromes to be extracted (n). Such a designation is needed when, for example, the first error is on $S_x^{1,1}$ and the second error could be either on $S_x^{2,1}$ or S_x^w. The designation of "sn" indicates that both errors must cause a non-zero syndrome or both errors must not cause a non-zero syndrome.

The probabilities of the errors in each cell can be looked up in Appendix A.

Each cell corresponds to a pair of errors that may occur on many distinct pairs of qubits. The errors in the diagonal cells are errors that occur in the same gate, so they should be counted

$$
\binom{7}{2}T^2 = 21T^2 \tag{4.27}
$$

times for a gate with T time steps.

	inc	$S_z^{1,1}$	$S_z^{1,t>1}$	S_z^w	R_z^w	$S_x^{1,1}$	$S_x^{1,t>1}$	S_x^w	R_x^w	U_i
inc	XY									
$S_z^{1,1}$	XY	XY								
$S_z^{1,t>1}$	XY	XY	XY							
$S_z^{2,1}$	Y1	XYs1	-							
$S_z^{2t>1}$	Y1	XYs1	-							
S_z^3	Y1	XYs1	-							
S_z^w	XY	XYn1	XY	XY						
R_z^w	XY	XY	XY	XY	XY					
$S_x^{1,1}$	XY	XY	XY	XY	XY	XYsn				
$S_x^{1,t>1}$	XY	XY	XY	XY	XY	XY	XY			
$S_x^{2,1}$	XYs2	XYs2	XYs2	XYs2	XYs2	XYs	-			
$S_x^{2,t>1}$	-	-	-	-	-	-	-			
S_x^3	-	-	-	-	-	-	-			
S_x^w	-	-	-	-	-	XYn1	XY	XY		
R_x^w	-	-	-	-	-	XYn1	XY	XY	XY	
U_i	-	-	-	-	-	XYn1	XY	XY	XY	XY

Table 4.4: This table lists all pairs of errors that cause a logical X or Y error. The columns indicate the location of the first error and the rows indicate the location of the second error. The column and row labeled "inc" correspond to incoming errors on S_z^1 ancilla.

Order of errors is already prescribed in the off-diagonal cells, so the errors in those cells should be counted

$$(7)(6)T_1T_2 = 42T_1T_2 \tag{4.28}$$

times, where T_1 and T_2 are the number of time steps in the corresponding gates.

If one of the gates is "inc", then the error in the cell should be counted $6T$ times. If both gates are "inc", then the error should be counted $6/7$ times.

Second, we consider the case that there is an incoming X error on S_z^1. This occurs with probability P_z^X. For a logical X or Y error to occur, there must be an X or Y error before $S_x^{2,1}$. This fact is represented in Table 4.5.

Third, we consider the case that there is an incoming Y error on S_z^1. This occurs with probability P_z^Y. For a logical X or Y error to occur, there must be an X or Y error before $S_x^{2,1}$, as represented in Table 4.5.

Finally, we consider the case that there is an incoming Z error on S_z^1. This case is negligible, since two X or Y errors would still be required to cause a logical X or Y

	inc	$S_z^{1,1}$	$S_z^{1,t>1}$	S_z^2	S_z^3	S_z^w	R_z^w	$S_x^{1,1}$	$S_x^{1,t>1}$	$S_x^{2,1}$
inc X	XY	XY	XY	-	-	XY	XY	XY	XY	XYs
inc Y	XY	XY	XY	XY	XY	-	XY	XY	XY	XYs

Table 4.5: This table lists all errors that cause a logical X or Y failure, given either an incoming X error (first row) or an incoming Y error (second row). See Appendix B for an explanation of the designation "s".

error. This concludes our determination of $\gamma_i^X(\ell+1) + \gamma_i^Y(\ell+1)$.

In the same manner as above, we calculate $\gamma_i^Z(\ell+1) + \gamma_i^Y(\ell+1)$ and $\gamma_i^Y(\ell+1)$ using the tables in Appendix B. From these three values we easily find $\gamma_i^X(\ell+1)$, $\gamma_i^Y(\ell+1)$, and $\gamma_i^Z(\ell+1)$, giving the effective noise channel for a single qubit gate (Hadamard or wait) at the next level of error-correction.

4.5.2 Two Qubit Gate

For two qubit gates, the event that there is a logical failure on one of the qubits is *almost* independent of the event that there is a logical failure on the other qubit, since the only gate that acts on both logical qubits is the logical U_i gate. This makes the failures XI, IX, ZI, IZ, YI, and IY much more likely than any of the other possible failures.

Only when one or two of the failures occur on the U_i gate can a different failure occur due to two failures. The probability of two failures on U_i is negligible compared with the probability of two failures before U_i that cause a logical failure. The probability of one failure on U_i is less negligible, but it must occur on the same qubit as a single error that propagates to both logical qubits. The probability of an error propagating from one logical qubit to another is minimized by reordering the error-correction. This makes the probability negligible compared with the probabilities for XI, IX, ZI, IZ, YI, and IY failures.

So the only non-negligible failure rates for a level one (or higher) two qubit gate are γ_2^{IX}, γ_2^{XI}, γ_2^{IY}, γ_2^{YI}, γ_2^{IZ}, and γ_2^{ZI}, and their values are

$$\begin{aligned} \gamma_2^{IX} &= \gamma_2^{XI} = \gamma_1^X \text{ (with } U_1 \text{ replaced by } U_2\text{)}, \\ \gamma_2^{IY} &= \gamma_2^{YI} = \gamma_1^Y \text{ (with } U_1 \text{ replaced by } U_2\text{)}, \\ \gamma_2^{IZ} &= \gamma_2^{ZI} = \gamma_1^Z \text{ (with } U_1 \text{ replaced by } U_2\text{)}, \end{aligned} \tag{4.29}$$

where γ_1^X, γ_1^Y, and γ_1^Z are calculated the same way as in the above section, but with U_1 replaced by U_2 everywhere in the calculation.

Because the replacement of U_1 by U_2 can only have a small affect on the overall failure rate, we expect that the failure rate of a level one (or higher) two qubit gate to be very close to twice the failure rate of a single qubit gate:

$$\gamma_2(\ell > 0) \approx 2\gamma_2^{IX}(\ell) + 2\gamma_2^{IY}(\ell) + 2\gamma_2^{IZ}(\ell) \approx 2\gamma_1(\ell). \tag{4.30}$$

Equation 4.29 gives the effective noise channel for a two qubit gate at the next level of error-correction.

4.5.3 Measurement

The probability of a logical measurement error is very simple, since the measurement is done immediately:

$$\begin{aligned} \mathbb{P}(\text{logical X,Y failure}) &= \mathbb{P}(\text{no inc X,Y,Z on } S_z^1)(6/7)\mathbb{P}(\text{X,Y on meas.})^2 \\ &\quad + \mathbb{P}(\text{inc X,Y on } S_z^1)(6/7)\mathbb{P}(\text{X,Y on meas.}), \\ \mathbb{P}(\text{logical Z,Y failure}) &= \mathbb{P}(\text{no inc X,Y,Z on } S_z^1)(6/7)\mathbb{P}(\text{Z,Y on meas.})^2 \\ &\quad + \mathbb{P}(\text{inc Z,Y on } S_z^1)(6/7)\mathbb{P}(\text{Z,Y on meas.}), \\ \mathbb{P}(\text{logical Y failure}) &= \mathbb{P}(\text{no inc X,Y,Z on } S_z^1)(6/7)\mathbb{P}(\text{Y on meas.})^2 \\ &\quad + \mathbb{P}(\text{inc Y on } S_z^1)(6/7)\mathbb{P}(\text{Y on meas.}). \end{aligned} \tag{4.31}$$

Written out explicitly, Equation 4.31 becomes

$$\begin{aligned}
\gamma_m^X(\ell+1)+\gamma_m^Y(\ell+1) &= (1-P_z^X-P_z^Y-P_z^Z)(6/7)(\gamma_m^X+\gamma_m^Y)^2 \\
&\quad + (P_z^X+P_z^Y)(6/7)(\gamma_m^X+\gamma_m^Y), \\
\gamma_m^Z(\ell+1)+\gamma_m^Y(\ell+1) &= (1-P_z^X-P_z^Y-P_z^Z)(6/7)(\gamma_m^Z+\gamma_m^Y)^2 \\
&\quad + (P_z^Z+P_z^Y)(6/7)(\gamma_m^Z+\gamma_m^Y), \\
\gamma_m^{Y,n}(\ell+1) &= (1-P_z^X-P_z^Y-P_z^Z)(6/7)(\gamma_m^Y)^2 \\
&\quad + (P_z^Y)(6/7)(\gamma_m^Y),
\end{aligned} \tag{4.32}$$

where every term on the right had side of Equation 4.32 is calculated at level ℓ.

Equation 4.32 gives the effective noise channel for a measurement gate at the next level of error-correction.

4.5.4 Preparation

As one of our modeling choices, we assume that

$$\begin{aligned}
\gamma_p(\ell) &= \gamma_1(\ell), \\
\gamma_p^X(\ell) &= \gamma_1^X(\ell), \\
\gamma_p^Y(\ell) &= \gamma_1^Y(\ell), \\
\gamma_p^Z(\ell) &= \gamma_1^Z(\ell).
\end{aligned} \tag{4.33}$$

We can make such an approximation, because even a factor of ten in the value of γ_p has negligible effect on the other failure rates (the effect is typically less than $\pm 5\times 10^{-5}$ on the other failure rates).

4.6 Threshold

Using the failure rates calculated in Section 4.5 we can easily calculate the threshold. Given the level zero failure rates (γ_1^X, γ_1^Y, γ_1^Z, γ_2^{IW}, γ_2^{IY}, γ_2^{AB}, γ_w^X, γ_w^Y, γ_w^Z, γ_m^X, γ_m^Y,

γ_m^Z, γ_p^X, γ_m^Y, and γ_m^Z), we calculate the level one failure rates (same list), from which we calculate the level two failure rates, and so on.

We determine whether the initial set of failure rates was above or below threshold by repeating the above procedure until each failure rate is above its initial value or each failure rate is below its initial value. This gives an eleven dimensional threshold surface. We calculate a two-dimensional cross-section of this surface in Section 5.5.

We should note that the set of gets that we have analyzed to determine the threshold is not universal. For universality, we would have to include a gate such as the Toffoli gate. We assume that the existence of the Toffoli gate in an error-corrected circuit has little affect the threshold. We think this amounts to assuming that the Toffoli appears infrequently enough that the correlated errors it can cause are about as likely as those for a cnot gate. As with the cnot gate, the X and Z error corrections would be reordered to minimize the probability of correlated errors.

Chapter 5

Results

What is the effective noise channel at different levels of concatenation of the Steane [[7,1,3]] code? How does the estimate of the threshold change when the different noise channels are taken into account? Within the assumptions of our model, we answer these two questions in this Chapter.

Whenever possible, we carry out numerical simulations to provide support for the accuracy of our analytical model. In Section 5.1, we explain our method for numerical simulation. In the following sections, we present our results in the same order as they were predicted in the analysis: Section 5.2 compares our predictions for alpha and the probabilities of incoming errors on the ancilla to numerical simulations; Section 5.3 does the same for the probabilities of incoming errors on the data; Section 5.4 predicts the effective noise channels at all levels of code concatenation, answering our first question; and Section 5.5 predicts the value of the threshold with and without changes in the noise channel, answering our second question.

5.1 Numerical Simulations

We conduct numerical simulations to test the accuracy of our analytical model. We do not simulate more than one level of error correction (that would require too many computing cycles). However, in addition to simulating level one error-correction with depolarizing noise, we simulate level one error-correction with the noise channel that our analytical model predicts for higher level error-correction. In this way we effectively simulate higher level error-correction, assuming that our analytical model is sufficiently accurate.

We modified a quantum computer simulator called ARQ, created by A. Cross [5], which uses stabilizer simulations given in [1]. The program ARQ takes as input a sequence of commands that specifies qubits, the gates that act on them, and simple classical processing. ARQ language specifications are given at the beginning of Appendix E. ARQ uses stabilizers to track the state of a quantum system, and can efficiently simulate any stabilizer circuit.

We wrote Python code that generates ARQ code for simulating the quantum error-correction circuits that we have chosen for our model. We have included the ARQ code generator in Appendix D and included some sample output (ARQ code) in Appendix E.

5.2 Alpha

In order to calculate the threshold and noise channels, we derived equations for the probabilities of incoming errors on the ancilla in Section 4.3. In our notation, these were the probabilities $\mathbb{P}(\text{inc X on } S^1 \text{ anc})$, $\mathbb{P}(\text{inc Y on } S^1 \text{ anc})$, and $\mathbb{P}(\text{inc Z on } S^1 \text{ anc})$. In this Section we compare our analytical estimates of these probabilities to the results of our numerical simulations. We find that they are in precise agreement.

We derived the probabilities of incoming errors on the ancilla by first deriving the probability that the verification network passes with and without errors. The probability that the verification network passes, which we called α, is plotted in Figure 5-1 along with our numerical results.

In Figure 5-1 we plot alpha versus the gate failure rate γ, where we define $\gamma \equiv \gamma_1 = \gamma_2 = \gamma_m = \gamma_p = 10\gamma_w$. We plotted α twice: once using the depolarizing channel, and once using a channel with equally weighted X and Z errors (but no Y failures) on single qubit gates and equally weighted IX, IZ, XI, and ZI failures (but no on other failures) on two qubit gates. This second channel is approximately the effective noise

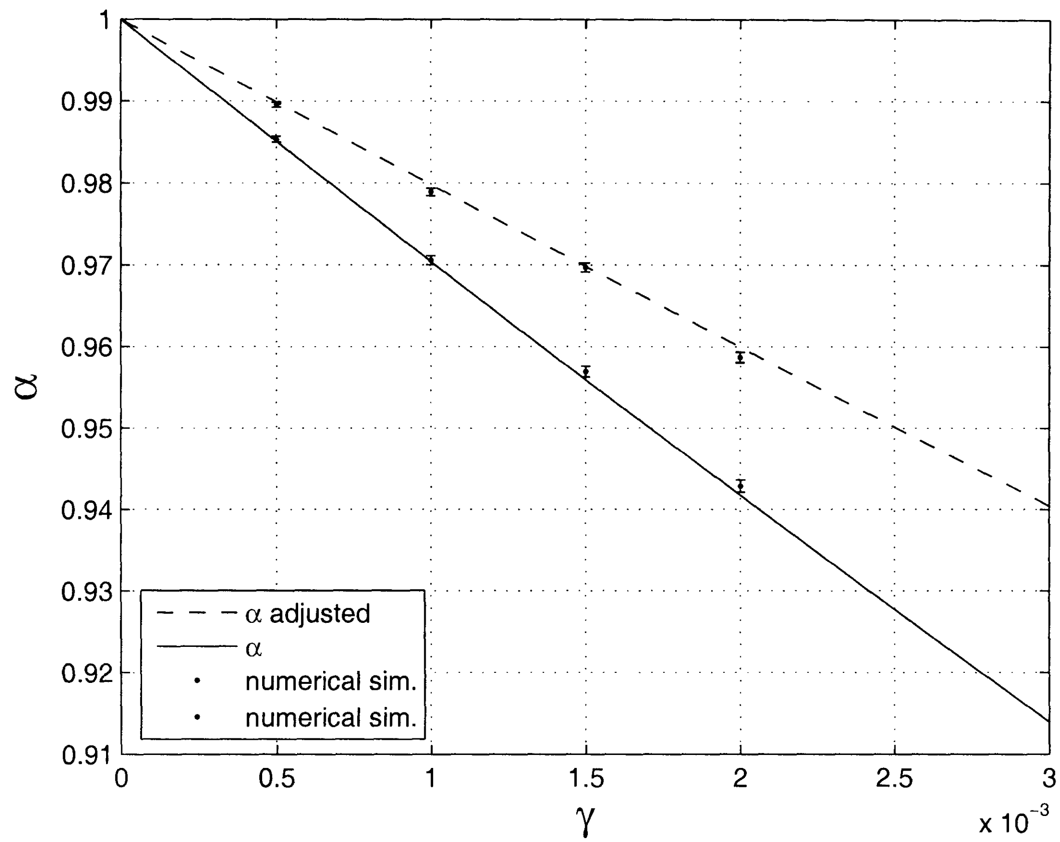


Figure 5-1: This is a graph for α, the probability that the verification network passes, versus the failure rate $\gamma \equiv \gamma_1 = \gamma_2 = \gamma_m = \gamma_p = 10\gamma_w$. It is plotted twice: once assuming the depolarizing channel (solid line), and once assuming the effective channel that we calculate in Section 5.4 to be the higher level noise channel ("adjusted" dashed line).

channel at higher level error-correction that we calculate in Section 5.4. In Figure 5-1, we call the value of α for the higher level noise channel "α adjusted." We find that the value of α is higher for the higher level noise channel. This is mainly because the probability of an X or Y error changes from $2\gamma/3$ to $\gamma/2$ for single qubit gates and from $8/15$ to $1/4$ for two qubit gates.

The essential probabilities for determining the threshold and higher level noise channels were the probabilities of incoming errors on the the ancilla, which are plotted in Figure 5-2. Again, we plot two results, one set of results for depolarizing noise, and one set of results for the higher level noise that we calculate in Section 5.4.

In Figure 5-2 we plot two probabilities: $\mathbb{P}$(inc X,Y on S^1 anc) and $\mathbb{P}$(inc Z,Y on S^1 anc). The first, $\mathbb{P}$(inc X,Y on S^1 anc), is important because X and Y errors on the ancilla propagate to errors on the data. The second, $\mathbb{P}$(inc Z,Y on S^1 anc), is important because Z and Y errors on the ancilla cause non-zero syndrome measurements.

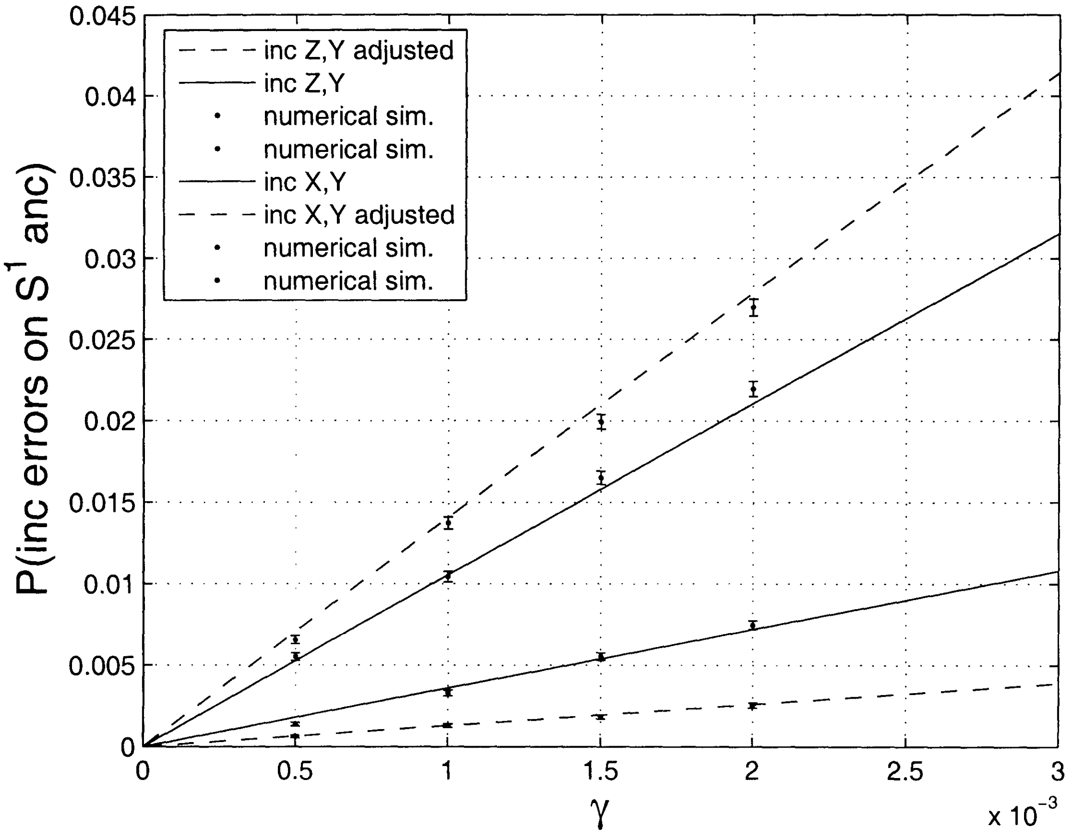


Figure 5-2: This is a graph of the probabilities of incoming errors on the ancilla coming into S^1 versus the failure rate $\gamma \equiv \gamma_1 = \gamma_2 = \gamma_m = \gamma_p = 10\gamma_w$. The probabilities are plotted twice: once assuming the depolarizing channel (solid lines), and once assuming the effective channel that we calculate in Section 5.4 to be the higher level noise channel ("adjusted" dashed lines). The legend indicates the the order of the plotted probabilities as they appear in the graph from top to bottom.

We find that the probability of an incoming X or Y error on the ancilla is lower for the higher level noise channel. This means that there is a lower probability of an error propagating to the data than would be predicted using the depolarizing channel. However, we find that the probability of an incoming Z or Y on the ancilla is actually higher for the higher level noise channel. This means that there is a higher probability of obtaining a wrong syndrome measurement during S.

We explain how we obtained the numerical results. For each probability, we ran 10^5 simulation trials of the preparation network (G and V^1) for each data point. For alpha, we merely counted the number of times verification succeeded. For the other probabilities, we used stabilizer generators to detect the errors that occurred. For $\mathbb{P}$(inc X,Y on S^1 anc) we compared the set of stabilizer generators of the data qubits at the end of V^1 to the set of stabilizer generators for the state $|0\rangle_L$ with one X error (seven possibilities) and the set of stabilizer generators for the state $|0\rangle_L$ with one Y error (also seven possibilities).

Similarly, for $\mathbb{P}$(inc Z,Y on S^1 anc) we compared the set of stabilizer generators of the data qubits at the end of V^1 to the set of stabilizer generators for the state $|0\rangle_L$ with one Z error and the set of stabilizer generators for the state $|0\rangle_L$ with one Y error.

5.3 Incoming Errors on Data

In Section 4.4 we solved a set of six linear equations to solve for the steady state incoming error probabilities (P_z^X, P_z^Y, P_z^Z, P_x^X, P_x^Y, and P_x^Z). We plot our analytical estimates and simulation results for the first three of these probabilities for the case $U_i = I$ (single qubit identity gate) in Figure 5-3.

As in the preceding Section, we calculated each probability twice, once using the depolarizing channel and once using a channel with equally weighted X and Z errors but no Y errors. This second channel was approximately the effective channel for higher level error-correction that we calculate in Section 5.4.

For each probability, we ran 10^5 simulation trials of a network consisting of six error-corrected identity gates (for each data point). Six consecutive error-corrections were used to ensure that the steady state distribution of incoming errors was achieved. We found that five was large enough to ensure steady state.

As in the preceding Section 5.2, we determined the probabilities by comparing the set of stabilizer generators for the data qubits to the set of stabilizer generators for $|0\rangle_L$ with a single qubit error. We compared the stabilizer generators immediately before either S_z^1 or S_x^1 during the sixth error-correction.

5.4 Noise Channels

In Section 4.5 we calculated the probabilities of logical X, Y, and Z errors. This gave us the effective noise channel at any level of error-correction. We plot our analytical estimates of these probabilities given depolarizing noise in Figure 5-4. The noise channel is given by the relative probabilities of logical X, Y, and Z errors.

We find that after one level of concatenation, the probability of a Y error is an order of magnitude less than the probability of an X or Z error. After two levels of concatenation, the probability of a logical Y error is negligible: the effective noise channel at all higher levels of concatenation is approximately one half X error and one half Z error. Also n Figure 5-4 we plot the probabilities of logical X, Y, and Z errors assuming that the noise channel is already this higher level noise channel.

We cannot easily compare the probabilities of the three logical errors to numerical simulations, because no matter what single logical qubit state we create, the state is always stabilized by one of the logical errors. That means that the logical Y error is always indistinguishable from either the logical X or logical Z error, when using stabilizers to distinguish them. However, we can conduct numerical simulations to determine $\mathbb{P}$(logical X,Y failure) and $\mathbb{P}$(logical Z,Y failure), the first of which we compare to our analytical model in Figure 5-4.

5.5 Threshold

As explained in Section 4.6, the threshold that our model predicts is an eleven dimensional surface in noise parameter space. It would be intractable to represent that surface here, so we present a two dimensional cross-section in Figure 5-5.

We chose a cross-section where $\gamma_{else} \equiv \gamma_1 = \gamma_2 = \gamma_p = \gamma_m$ and γ_w are the independent initial failure rates. We could have set $\gamma_w = \gamma_1/10$, obtaining a one-dimensional cross-section of the threshold as other authors do, but wait gates appear far more often than the other gates in our error-correction circuit, so the effect on the threshold of changes in γ_w is greater. For this reason, we thought it would be useful to show how the threshold depends on γ_w.

Because our intent is to show how the value of the threshold changes when we take into account the changes in the noise channel, we plot three thresholds under three different assumptions: the noise channel is depolarizing at all levels of concatenation (this gives the lowest threshold result); the noise channel is always one half X, one half Z, and no Y at all levels of concatenation (this gives the highest threshold result); and the noise channel is initially depolarizing but changes at each level according to our analytical model (this gives the middle threshold result).

Our result is that the value of the threshold for the Steane [[7,1,3]] code changes by $\approx 30\%$ from 3.0×10^{-4} to 3.9×10^{-4} when $10\gamma_w = \gamma_1 = \gamma_2 = \gamma_p = \gamma_m$. Our result for the case where we assume depolarizing noise every level is in excellent agreement with [21].

We analyze our threshold result a little more. Figure 5-6 graphs the failure rates of level $\ell + 1$ gates in terms of the level ℓ failure rate $\gamma = \gamma 1 = \gamma 2 = \gamma_m = \gamma_p = 10\gamma_w$. The solid lines give the results when the level ℓ noise is depolarizing, while the dashed lines give the results when the level ℓ noise is equally weighted X and Z errors only. We find that level $\ell + 1$ are much lower in the latter case (with the exception of $\gamma_m(\ell + 1)$, which is so small we do not really care about it).

Tables 5.1 and 5.2 show in detail the behavior of the noise channels just below threshold (the adjusted threshold, 3.9×10^{-4}). Rows 1-4 of each table show the error rates assuming changing noise channels at each level of error correction. Rows 5-8 show the error rates assuming depolarizing noise at each level of error correction.

First we look at rows 1-4. The characteristic behavior is that the failure rates γ_1^X and γ_2^{IX} initially jump down, while the failure rate γ_w^X jumps up to the same level as γ_1^X and γ_2^{IX}, since all three of these gates get replaced by approximately the same

γ_1^X	γ_1^Y	γ_2^{IX}	γ_2^{IY}	γ_2^{AB}	γ_w^X	γ_w^Y	γ_m^X	γ_m^Y
1.300	1.300	.2600	.2600	.2600	.1300	.1300	1.300	1.300
.5305	.0688	.5143	.0657	0	.4645	.0571	.0126	.0025
.5118	.0028	.5273	.0028	0	.5089	.0027	.0001	.0001
.4764	.0000	.4914	.0000	0	.4763	.0000	.0000	.0000
1.300	1.300	.2600	.2600	.2600	.1300	.1300	1.300	1.300
.3766	.3766	.1459	.1459	.1459	.3287	.3287	.0093	.0093
.5580	.5580	.2284	.2284	.5551	.5551	.5551	.0001	.0001
1.517	1.517	.6216	.6216	.6216	1.517	1.517	.0000	.0000

Table 5.1: This table shows the behavior of the noise channels just below threshold. Rows 1-4 give the noise channels for successive levels of error-correction in our model. Rows 5-8 give the noise channels for successive levels of error-correction assuming that the noise channel is depolarizing at each level.

γ_1	γ_2	γ_w	γ_m
3.900	3.900	.3900	3.900
1.130	2.189	.9861	.0278
1.026	2.115	1.021	.0002
.9529	1.966	.9526	.0000
3.900	3.900	.3900	3.900
1.130	2.189	.9861	.0278
1.674	3.427	1.665	.0003
4.551	9.325	4.550	.0000

Table 5.2: This table is the same as Table 5.1, except only the full failure rate for each type of gate is presented.

error-correction circuit. If γ_1^X and γ_2^{IX} jumped down far enough, they will be below threshold, as is barely the case in our example. The failure rates γ_1^Y, γ_2^{IY}, γ_w^Y, and γ_m^Y all jump down to about 1/9 to 1/8 of their corresponding X failure rates. The jump of γ_1^Y and γ_2^{IY} down between rows 2 and 3 is what causes γ_2^{IX} and γ_w^X to start decreasing again after a slightly increasing between rows 2 and 3. The measurement failure rates become negligible rather quickly.

Now we look at rows 5-8. Rows 5 and 6 are the same as rows 1 and 2 because both started with depolarizing noise. For each row in 5-8, each location failure rate is spread out evenly among the possible failures. Logical X failures occur when there are two X errors *or* one X error and one Y error. Similarly, logical Z failures occur when there are two Z errors *or* one Z error and one Y error. Y errors contribute to both logical failure rates, so when the failure rates are spread out among X, Y, and Z errors, this increases the probability of logical failrates. Row 8 shows that $\gamma = 3.9 \times 10^{-4}$ is above threshold when assuming depolarizing noise at all levels.

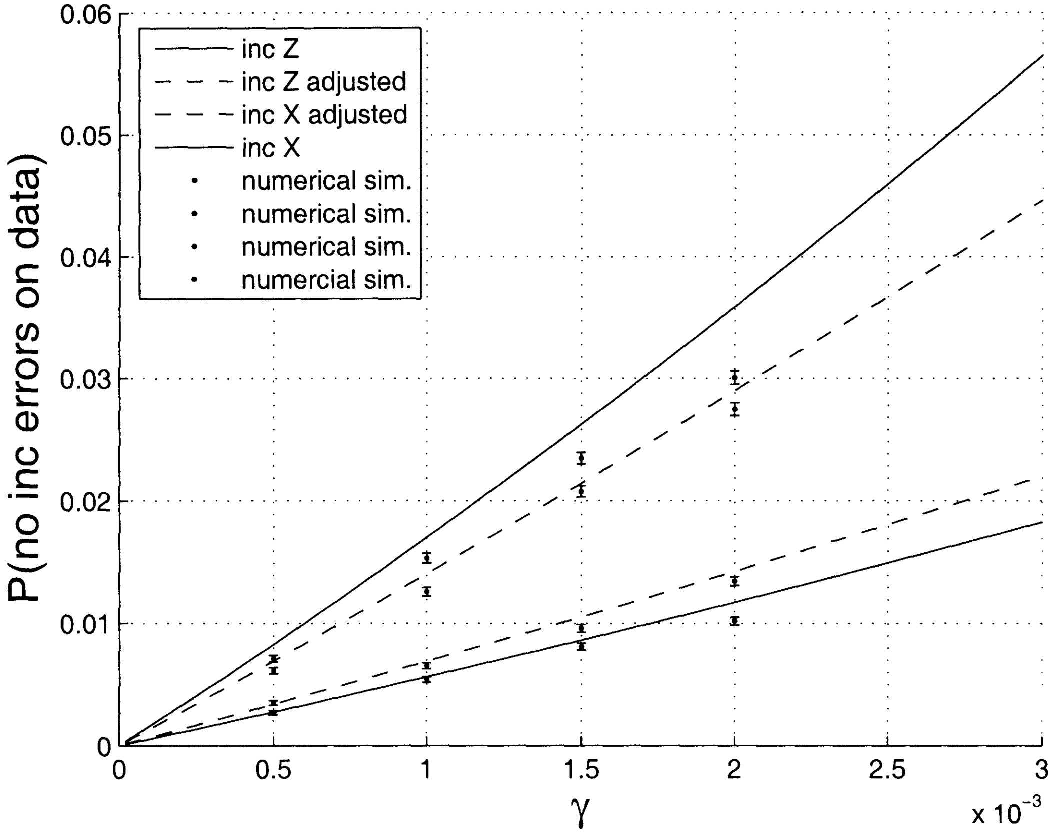


Figure 5-3: This is a graph of the probabilities of incoming X, Y, and Z errors into Z error-correction: P_z^X, P_z^Y, and P_z^Z. They are plotted against the failure rate $\gamma \equiv \gamma_1 = \gamma_2 = \gamma_m = \gamma_p = 10\gamma_w$ The probabilities are plotted twice: once assuming the depolarizing channel (solid lines), and once assuming the effective channel that we calculate in Section 5.4 to be the higher level noise channel (dashed lines). The legend indicates the the order of the plotted probabilities as they appear in the graph from top to bottom.

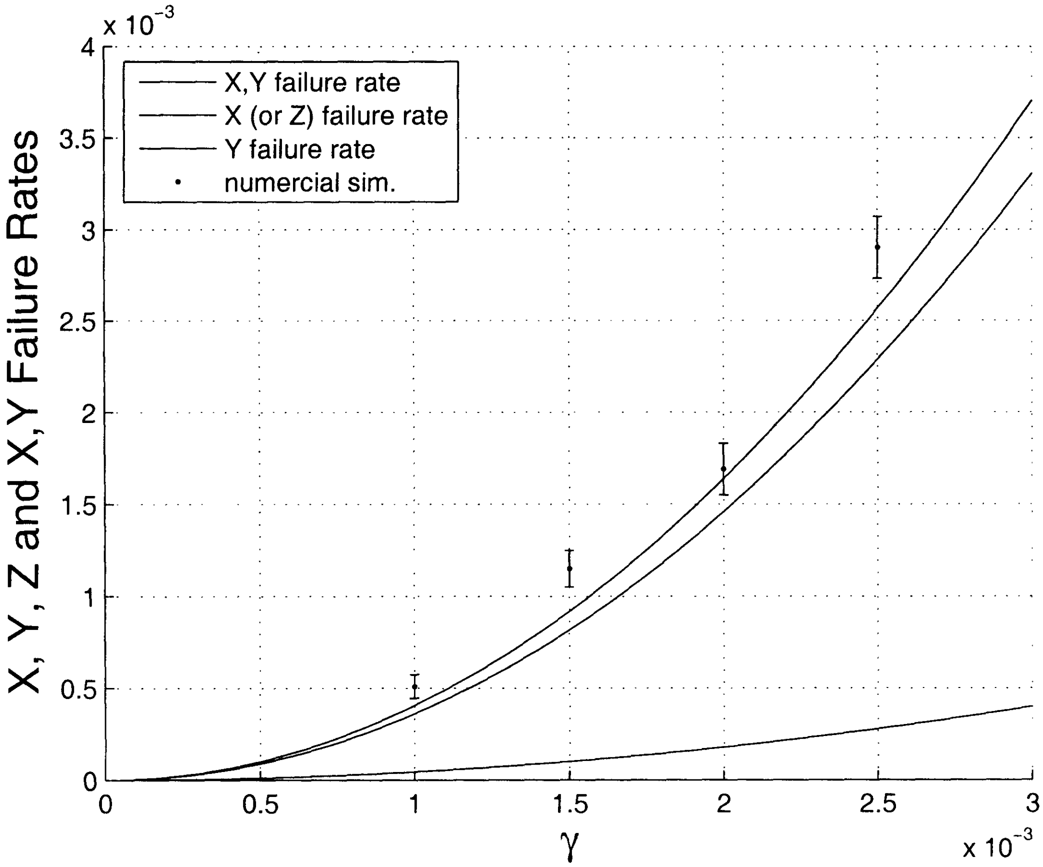


Figure 5-4: The is a graph the effective noise channel (the separate probabilities of X, Y, and Z errors) at level $\ell+1$ versus the failure rate $\gamma \equiv \gamma 1 = \gamma 2 = \gamma_m = \gamma_p = 10\gamma_w$ at level ℓ. The probabilities are plotted twice: once assuming the depolarizing channel, and once assuming the effective channel that we calculate in Section 5.4 to be the higher level noise channel. We also plot the probability of an X *or* Y error (denoted X,Y error) and compare to numerical simulation. The legend indicates the the order of the plotted probabilities as they appear in the graph from top to bottom.

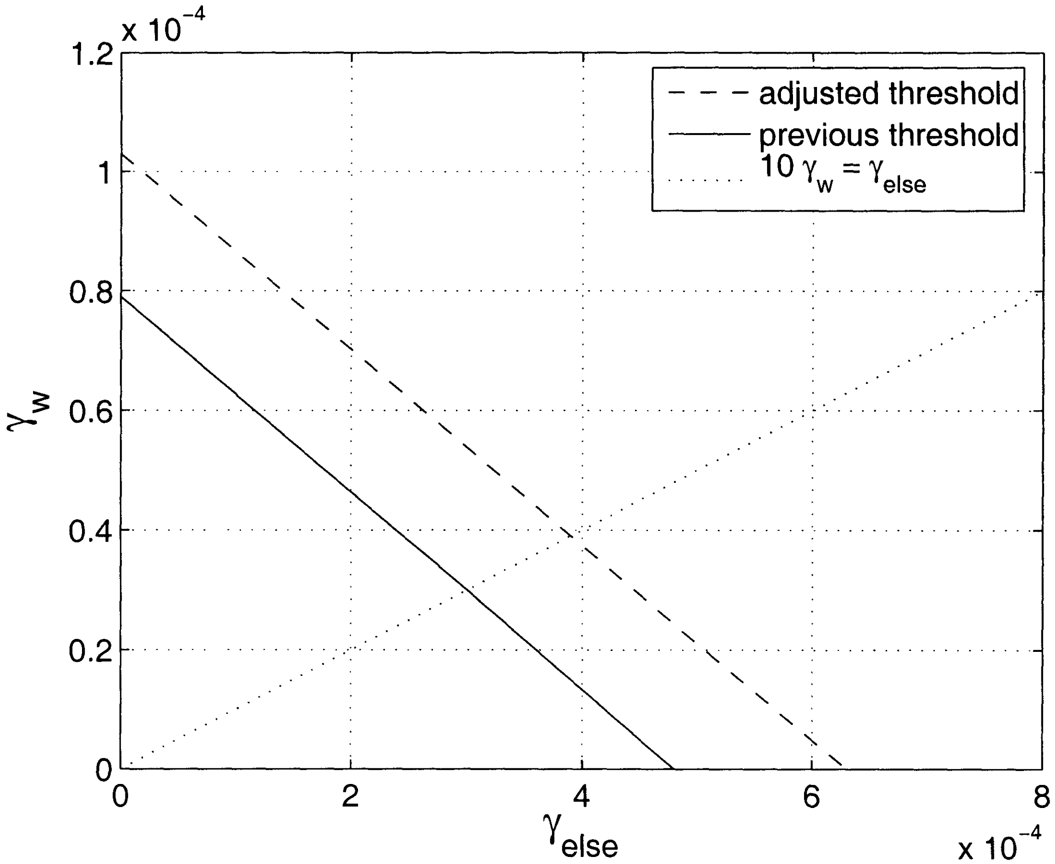


Figure 5-5: This is a graph of the threshold for the Steane [[7,1,3]] code. The horizontal axis is $\gamma_{else} \equiv \gamma_1 = \gamma_2 = \gamma_p = \gamma_m$ and the vertical axis is γ_w. The solid line is the threshold result assuming depolarizing noise at all levels of error-correction. The dashed line is the threshold result when the noise channel changes according to our analytical model. Along the line $\gamma_{else} = 10\gamma_w$, the threshold increases from 3.0×10^{-4} to 3.9×10^{-4} when we take into consideration the changing noise channel.

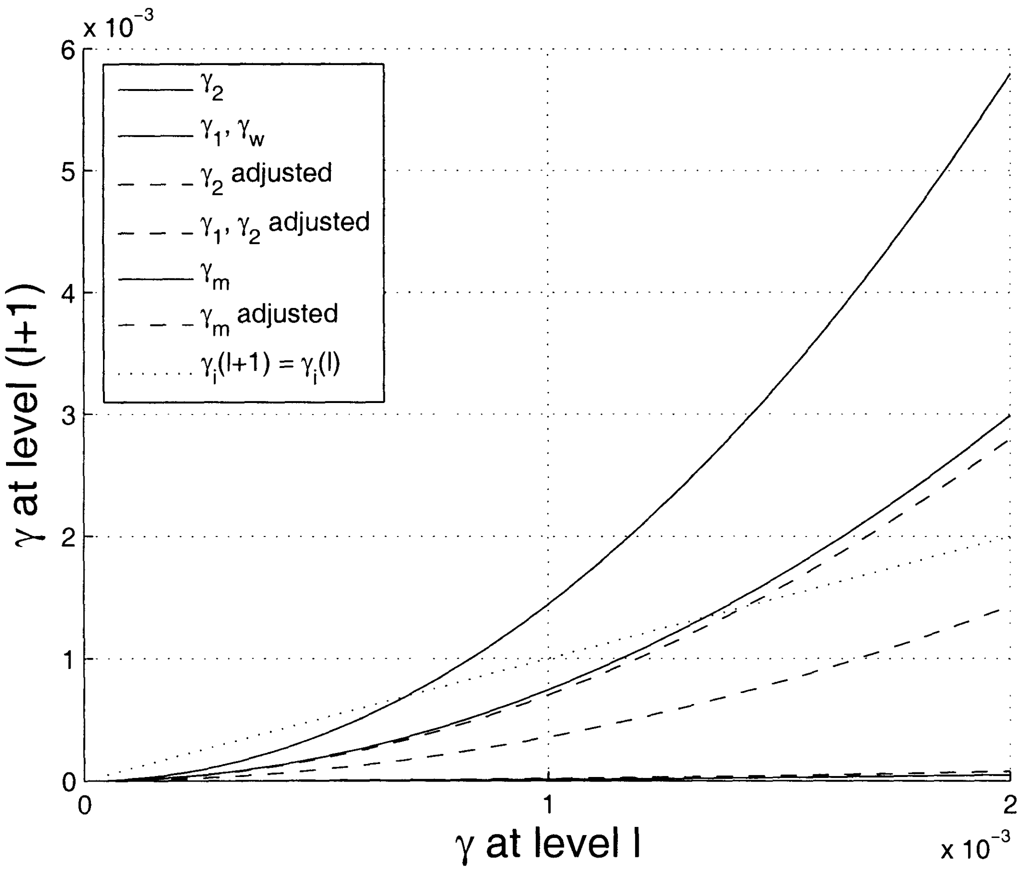


Figure 5-6: This is a graph of the failure rates at level $\ell + 1$ in terms of the failure rate $\gamma \equiv \gamma 1 = \gamma 2 = \gamma_m = \gamma_p = 10\gamma_w$ at level ℓ. The probabilities are plotted twice: once assuming the depolarizing channel, and once assuming the effective channel that we calculate in Section 5.4 to be the higher level noise channel. The legend indicates the the order of the plotted probabilities as they appear in the graph from top to bottom.

Chapter 6

Conclusions and Further Directions

We developed an analytical model that determines the effective noise channel for each type of gate at each level of concatenation of the Steane [[7,1,3]] code. We used the model to determine the effects of the changing noise channel on the threshold. We found that Y errors quickly drop out of the effective noise channel for all types of gates at levels of error-correction beyond level one. The effect this had on the threshold was to increase it by 30%. We also found the threshold to be 3.9×10^{-4}, which is an order of magnitude lower than the rough estimate in [19], but in good agreement with the estimate in [21].

Our analytical model has the novel feature that it calculates separately the probabilities of incoming X, Y, and Z errors into the X and Z error-correction routines. It calculates each set of probabilities in terms of the same set of probabilities in the previous error-correction, setting up an easily solvable system of linear equations. The power if this method was not fully utilized in the current thesis, and we note some possible extensions here.

The first extension is of practical interest. In a physical realization of a quantum computer, the noise channel can be very different from the depolarizing channel (X, Y, and Z failures can be weighted unequally). When this is the case, it may be possible to design new or modify existing error-correction routines to increase the threshold. Any analysis of the benefits of particular error-correction routine will necessarily involve a detailed analysis of the change in the noise channel at higher levels of error-correction, for which we set up a framework in our analysis.

In our thesis we limited the possible set of initial noise channels by assuming that X and Z failures always occur with equal probability. If we did not make this assumption, then the failure rate of any gate would depend on whether the preceding gate was a Hadamard (which swaps X and Z errors). Then we would not be able to assign a given threshold to an entire class of gates, since the failure rate of any particular gate would depend on the circuit it belongs to. This circuit dependence of the threshold can be calculated using the system of linear equations we set up in Section 4.4.

The second extension would be to generalize our analysis to arbitrary CSS codes. Construction of efficient error-correction networks using the generator matrices and parity check matrices was already explained in [19]. The main difficulty would lie in determining the incoming errors into each error-correction routine. The system of linear equations derived would be much larger and more complicated, but tractable if calculated by computer. It would be very interesting to find out how larger CSS codes affect the noise channel and how the new noise channel affects the threshold.

Appendix A

Probabilities

Table A.1 in conjunction with rule A.1 lists all of the probabilities used in the analysis. For a description of our notation, see Section 4.2.

$$\mathbb{P}(\text{no [inc] } \textit{errors} \text{ on } A_1, A_2, \ldots\ \textit{qubits}) = \prod_i \mathbb{P}(\text{no [inc] } \textit{errors} \text{ on } A_i\ \textit{qubits}). \quad \text{(A.1)}$$

expression	causes of error	$\mathbb{P}(expression)$
no W,Y on $S_x^{j,t>1}$	W,Y	$\left(1-\gamma_w^W-\gamma_w^Y\right)^{14}$
no W[Y] on $S_x^{j,t>1}$	W[Y]	$\left(1-\gamma_w^{W[Y]}\right)^{14}$
no W[Y] on S_x^w	W[Y]	$\left(1-\gamma_w^{W[Y]}\right)^{42}$
no W[Y] on R_x	W[Y]	$\left(1-\gamma_1^{W[Y]}\right)\left(1-\gamma_w^{W[Y]}\right)^6$
no W[Y] on R_x^w	W[Y]	$\left(1-\gamma_w^{W[Y]}\right)^7$
no W[Y] on U_1	W[Y]	$\left(1-\gamma_1^{W[Y]}\right)^7$
no Z,Y caused on $S_x^{j,1}$ anc	XX,XY,YI,YZ, ZI,ZZ,IX,IY	$\left(1-2\gamma_2^{IW}-2\gamma_2^{IY}-4\gamma_2^{AB}\right)^7$
no Z,Y caused on $S_x^{j,t>1}$ anc	Z,Y; X,Y	$\left(1-\gamma_1^Z-\gamma_1^Y\right)^7\left(1-\gamma_m^X-\gamma_m^Y\right)^7$
no X[Y] caused on $S_x^{j,1}$	IX[IY],ZX[ZY], XY[XX],YY[YX]	$\left(1-\gamma_2^{IW[IY]}-3\gamma_2^{AB}\right)^7$
no X[Y] caused on $S_x^{j,1}\vert$ no Z,Y caused on $S_x^{j,1}$ anc	(ZX[ZY],ZX[ZY]) XX,XY,YI,YZ, ZI,ZZ,IX,IY	$\left(1-2\gamma_2^{IW}-2\gamma_2^{IY}-6\gamma_2^{AB}\right)^7/$ $\left(1-2\gamma_2^{IW}-2\gamma_2^{IY}-4\gamma_2^{AB}\right)^7$ $\approx\left(1-2\gamma_2^{AB}\right)^7$
no X[Y] caused on U_2	IX[IY],ZX[ZY] XY[XX],YY[YX]	$\left(1-\gamma_2^{IW[IY]}-3\gamma_2^{AB}\right)^7$
no Z caused on $S_x^{j,1}$	XI,YI,ZZ,IZ	$\left(1-2\gamma_2^{IW}-\gamma_2^{IY}-\gamma_2^{AB}\right)^7$
no Z,Y caused on $S_x^{j,1}$	XI,XX,YI,YX ZY,ZZ,IY,IZ	$\left(1-2\gamma_2^{IW}-2\gamma_2^{IY}-4\gamma_2^{AB}\right)^7$
no Z caused on $S_x^{j,1}\vert$ no Z,Y caused on $S_x^{j,1}$ anc	(XI,IZ) XX,XY,YI,YZ, ZI,ZZ,IX,IY	$\left(1-4\gamma_2^{IW}-2\gamma_2^{IY}-4\gamma_2^{AB}\right)^7/$ $\left(1-2\gamma_2^{IW}-2\gamma_2^{IY}-4\gamma_2^{AB}\right)^7$ $\approx\left(1-2\gamma_2^{IW}\right)^7$
no Z,Y caused on $S_x^{j,1}\vert$ no Z,Y caused on $S_x^{j,1}$ anc	(XI,IZ,YX,ZY) XX,XY,YI,YZ, ZI,ZZ,IX,IY	$\left(1-4\gamma_2^{IW}-2\gamma_2^{IY}-6\gamma_2^{AB}\right)^7/$ $\left(1-2\gamma_2^{IW}-2\gamma_2^{IY}-4\gamma_2^{AB}\right)^7$ $\approx\left(1-2\gamma_2^{IW}-2\gamma_2^{AB}\right)^7$
no Z,Y caused on $S_x^{1,1}\vert$ Z,Y caused on $S_x^{1,1}$ anc	YI,ZZ,IY,XX XX,XY,YI,YZ, ZI,ZZ,IX,IY	$(2\gamma_2^{IY}+2\gamma_2^{AB})/$ $2\gamma_2^{IW}+2\gamma_2^{IY}+4\gamma_2^{AB}$
no Z caused on U_2	XI,IZ,YI,ZZ	$\left(1-2\gamma_2^{IW}-\gamma_2^{IY}-\gamma_2^{AB}\right)^7$

Table A.1: All probabilities needed for the calculation of P_z^X, P_z^Y, P_z^Z, P_x^X, P_x^Y, and P_x^Z given in Section 4.4 and the calculations of the failure rates in Section 4.5 can bee looked up in this table. In the top section, the label W can be replaced by either X or Z. The gate U_2 is taken to be a cz gate.

Appendix B

Counting Tables for Failure Rate Estimates

Tables B.1, B.3, and B.5 list all pairs of errors that cause a logical X or Y error; a logical Z or Y error; and a logical Y error, respectively. The columns indicate the location of the first error and the rows indicate the location of the second error. The column and row labeled "inc" correspond to incoming errors on S_z^1 ancilla. The counting is exact.

Usually filled in each cell is the error that must occur on both gates (or list of errors from which one must occur on each gate). The errors in some cells are followed by 1 or 2, meaning that the specified error(s) must be the first or second error, respectively. In such a case, the other error is assumed to be in the list XY (for Table B.1), ZY (for Table B.3), or Y (for Table B.5). The additional designations "s" and "n" indicate that the error causes further syndromes to be extracted (s) or must *not* cause further syndromes to be extracted (n). Such a designation is needed when, for example, the first error is on $S_x^{1,1}$ and the second error could be either on $S_x^{2,1}$ or S_x^w. The designation of "sn" indicates that both errors must cause a non-zero syndrome or both errors must not cause a non-zero syndrome.

Tables B.2, B.4, and B.6 list all single errors that cause a logical X or Y error; a logical Z or Y error; and a logical Y error, respectively, given various single incoming errors.

	inc	$S_z^{1,1}$	$S_z^{1,t>1}$	S_z^w	R_z^w	$S_x^{1,1}$	$S_x^{1,t>1}$	S_x^w	R_x^w	U_i
inc	XY									
$S_z^{1,1}$	XY	XY								
$S_z^{1,t>1}$	XY	XY	XY							
$S_z^{2,1}$	Y1	XYs1	-							
$S_z^{2t>1}$	Y1	XYs1	-							
S_z^3	Y1	XYs1	-							
S_z^w	XY	XYn1	XY	XY						
R_z^w	XY	XY	XY	XY	XY					
$S_x^{1,1}$	XY	XY	XY	XY	XY	XYsn				
$S_x^{1,t>1}$	XY	XY	XY	XY	XY	XY	XY			
$S_x^{2,1}$	XYs2	XYs2	XYs2	XYs2	XYs2	XYs	-			
$S_x^{2,t>1}$	-	-	-	-	-	-	-			
S_x^3	-	-	-	-	-	-	-			
S_x^w	-	-	-	-	-	XYn1	XY	XY		
R_x^w	-	-	-	-	-	XYn1	XY	XY	XY	
U_i	-	-	-	-	-	XYn1	XY	XY	XY	XY

Table B.1: This table lists all pairs of errors that cause a logical X or Y error. The columns indicate the location of the first error and the rows indicate the location of the second error. The column and row labeled "inc" correspond to incoming errors on the S_z^1 ancilla.

	inc	$S_z^{1,1}$	$S_z^{1,t>1}$	S_z^2	S_z^3	S_z^w	R_z^w	$S_x^{1,1}$	$S_x^{1,t>1}$	$S_x^{2,1}$
inc X	XY	XY	XY	-	-	XY	XY	XY	XY	XYs
inc Y	XY	XY	XY	XY	XY	-	XY	XY	XY	XYs

Table B.2: This table lists all errors that cause a logical X or Y failure, given either an incoming X error (first row) or an incoming Y error (second row). The column labeled "inc" corresponds to an incoming error on the S_z^1 ancilla.

	$S_z^{1,1}$	$S_z^{1,t>1}$	S_z^w	R_z^w	inc	$S_x^{1,1}$	$S_x^{1,t>1}$	S_x^w	R_x^w	U_i
$S_z^{1,1}$	ZYsn									
$S_z^{1,t>1}$	ZY	ZY								
$S_z^{2,1}$	ZYs	-								
$S_z^{2t>1}$	-	-								
S_z^3	-	-								
S_z^w	ZYn1	ZY	ZY							
R_z^w	ZYn1	ZY	ZY	ZY						
inc	ZYn1 XY2	XY2	XY2	XY2	XY					
$S_x^{1,1}$	ZYn1	ZY	ZY	ZY	XY1	ZY				
$S_x^{1,t>1}$	ZYn1	ZY	ZY	ZY	XY1	ZY	ZY			
$S_x^{2,1}$	Yn1	Y1	Y1	Y1	Y1	Ys1	-			
$S_x^{2,t>1}$	Yn1	Y1	Y1	Y1	Y1	Ys1	-			
S_x^3	Yn1	Y1	Y1	Y1	Y1	Ys1	-			
S_x^w	Zn1	Z1	Z1	Z1	X1	ZYn1	ZY	ZY		
R_x^w	ZYn1	ZY	ZY	ZY	XY1	ZY	ZY	ZY	ZY	
U_i	ZYn1	ZY	ZY	ZY	XY1	ZY	ZY	ZY	ZY	ZY

Table B.3: This table lists all pairs of errors that cause a logical Z or Y error. The columns indicate the location of the first error and the rows indicate the location of the second error. The column and row labeled "inc" correspond to incoming errors on the S_x^1 ancilla.

	$S_z^{1,1}$	$S_z^{1,t>1}$	$S_z^{2,1}$
inc Z	ZY	ZY	ZYs
inc Y	ZY	ZY	ZYs

Table B.4: This table lists all errors that cause a logical Z or Y failure, given either an incoming Z error (first row) or an incoming Y error (second row).

	$S_z^{1,1}$	$S_z^{1,t>1}$	S_z^w	R_z^w	$S_x^{1,1}$	$S_x^{1,t>1}$	S_x^w	R_x^w	U_i
$S_z^{1,1}$	Ysn								
$S_z^{1,t>1}$	Y	Y							
$S_z^{2,1}$	Ys	-							
$S_z^{2t>1}$	-	-							
S_z^3	-	-							
S_z^w	Ys1	Y	Y						
R_z^w	Ys1	Y	Y	Y					
$S_x^{1,1}$	Ys1	Y	Y	Y	Ysn				
$S_x^{1,t>1}$	Ys1	Y	Y	Y	Y	Y			
$S_x^{2,1}$	Ys1 Ys2	Ys2	Ys2	Ys2	Ys	-			
$S_x^{2,t>1}$	-	-	-	-	-	-			
S_x^3	-	-	-	-	-	-			
S_x^w	-	-	-	-	Yn1	Y	Y		
R_x^w	-	-	-	-	Y	Y	Y	Y	
U_i	-	-	-	-	Yn1	Y	Y	Y	Y

Table B.5: This table lists all pairs of errors that cause a logical Y error. The columns indicate the location of the first error and the rows indicate the location of the second error.

	$S_z^{1,1}$	$S_z^{1,t>1}$	$S_z^{2,1}$
inc Y	Y	Y	Ys

Table B.6: This table lists all errors that cause a logical Y failure given an incoming Y error.

Appendix C

Error Correction Circuits

These are the circuits used to construct the error-correction routine for the Steane [[7,1,3]] code. The use of these circuits in error-correction is explained in detail in Section 3.2

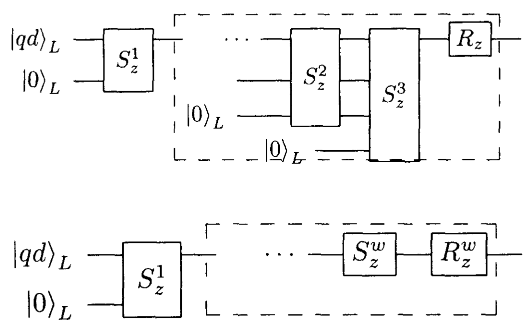


Figure C-1: The error correction routine finds and corrects errors on the seven data qubits in the logical state $|qd\rangle_L$ with the aid of multiple copies of ancilla qubits in the logical zero state $|0\rangle_L$. The second half of the circuit is on of two possibilities, depending on whether the first syndrome extraction S_z^1 was zero or non-zero. If the syndrome is non-zero, then two more syndromes are collected (middle circuit), but if the syndrome is zero, no more syndromes are collected and the data qubits wait (righmost circuit) during the syndrome extraction circuit acting on other qubits.

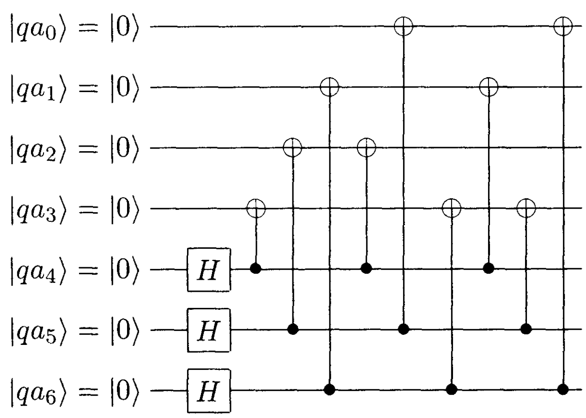


Figure C-2: This is the circuit for the preparation network, G. It prepares the logical zero state, $|0\rangle_L$. It is used in the error-correction circuit (see Section 3.2) to prepare ancilla qubits in the state $|0\rangle_L$.

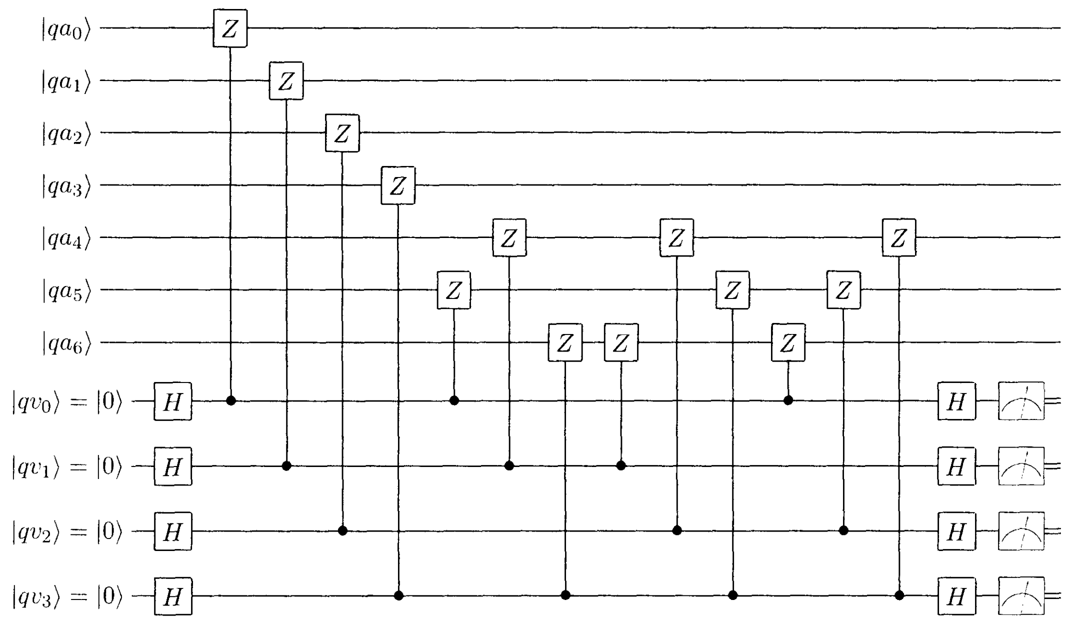


Figure C-3: The verification network V checks for X errors on the state $|0\rangle_L$ and gives four zero measurement results if no X errors are detected.

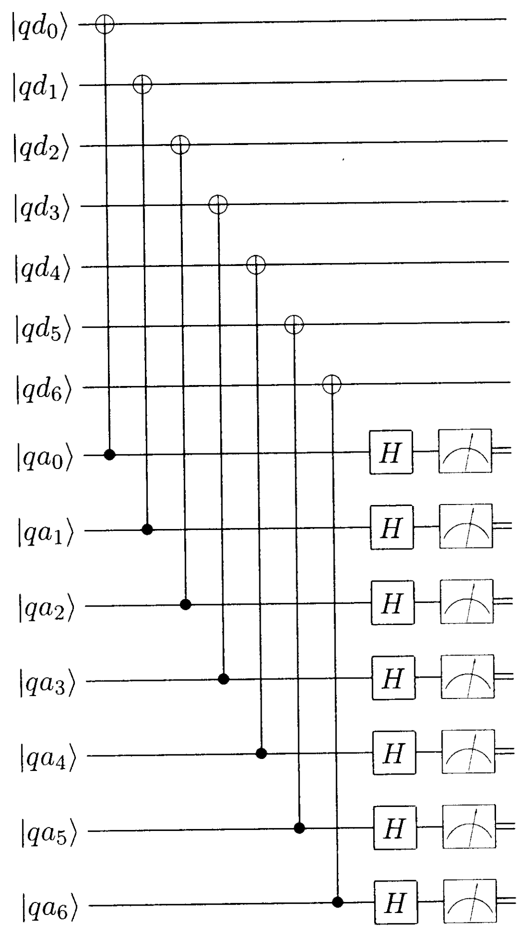


Figure C-4: The syndrome extraction network S consists of three time steps. The above network is the syndrome extraction for Z error correction. The syndrome extraction network for X error correction is the same, except with each cnot replaced by cz.

Appendix D

ARQ Code Generator for [[7,1,3]] Quantum Code

In this Appendix we present our code which generates ARQ code for an arbitrarily concatenated error-correction circuit for the Steane [[7,1,3]] code. The inputs to our code generator are g, the type of gate to be error-corrected; L, the level of code concatenation; s, the number of syndromes to collect if the first is non-zero; s', the number of syndromes that must agree for error-correction; and t, the number of repetitions of gate g. In our simulations, we only used the case $L = 1$, $s = 3$, and $s' = 2$, though our code generator allows for more generality.

The main program appears at the end of the code. It takes as inputs the above mentioned quantities and then outputs the appropriate ARQ code via print statements. The function *recover*, which writes code for the error-correction circuit EC, is called first, followed by the transversal version of the gate being error-corrected. *recover* calls the functions G, V, and S to print out the error-correction circuit.

At the end of our code (but before the main program) we provide a set of functions that measure the stabilizer generators of the data to find errors on the data.

```
#!/usr/bin/env python
# nft.py
# Andrew Morten <amorten@mit.edu>. Andrew Cross <awcross@mit.edu>
#
# Generates ARQ code for the [[7.1.3]] nonlocal fault-tolerance model
# given in quant-ph/0410047 by Svore, Terhal. DiVincenzo.

import sys
from math import *
import time

# version information
global version_info
version_info = "nft.py version 1.1 amorten@mit.edu, last modified 26 Aug 2005"

global L # level of code concatenation, i.e. an M_L simulating circuit is created
global s # number of additional syndromes to collect if the first syndrome is nonzero
global s_prime # number of syndromes that must agree for error correction

# counters for creating unique labels for jump targets
global counterPrepare
counterPrepare = 0
global counterRecover
counterRecover = 0
global counterSyndrome
counterSyndrome = 0
global counterMeasure
counterMeasure = 0

# global lists of qubit and cbit variables
global data, data_c, data_t, ancilla, verify1, verify2, verify3, verify4, vbits,\
               xsyndrome0, xsyndrome1, xsyndrome2,\
               zsyndrome0, zsyndrome1, zsyndrome2, meas, meas_H

# qubit and cbit declarations. setup instructions
#
# Returns lists of physical qubit and cbit names:
# data - 1 logical qubit @ L
# data1 - 1 logical qubit @ L
# data2 - 1 logical qubit @ L
# ancilla - list of logical qubits @ [L,L-1,...,1]
# verify1 - 1 logical qubit @ L-1
# verify2 - 1 logical qubit @ L-1
# verify3 - 1 logical qubit @ L-1
# vbits - a list of verification cbits
# xsyndrome0 - list of s syndrome cbits
# xsyndrome1 - list of s syndrome cbits
# xsyndrome2 - list of s syndrome cbits
# zsyndrome0 - list of s syndrome cbits
# zsyndrome1 - list of s syndrome cbits
# zsyndrome2 - list of s syndrome cbits

def declare_all_but_data():

        ancilla = []
        ancilla_temp = []
        verify1 = []
        verify2 = []
        verify3 = []
        verify4 = []
        verify1_temp = []
```

```
verify2_temp = []
verify3_temp = []
verify4_temp = []
vbits = []
xsyndrome0 = []
xsyndrome1 = []
xsyndrome2 = []
zsyndrome0 = []
zsyndrome1 = []
zsyndrome2 = []
meas_H = []
meas = []
meas_temp = []

print "#------------------------------------------------------------"
print "# declare(L=%d, s=%d)"%(L,s)
print "#"
print "# measurement bits(4+%dx7 cbits)"%L
for i in range(4):
        print "\tbit\tmeas_H_%d"%(i)
        meas_H.append("meas_H_%d"%(i))
for l in range(L):
        for i in range(7):
                print "\tbit\tmeas_%d_%d"%(l+1,i)
                meas_temp.append("meas_%d_%d"%(l+1,i))
        meas.append(meas_temp[:])
print "# temporary cbits used in measurement (3 cbits)"
for i in range(4): print "\tbit\ttemp%d"%i
print "# temporary cbits used in syndrome extraction (7 cbits)"
for i in range(7): print "\tbit\tsc%d"%i
print "# qubits and cbits that must be passed into G. V. S. etc"
print "# verify cbit"
print "\tbit\tv"
vbits.append("v")
print "# syndome bits. 2x3x%d of them"%s
for i in range(s):
        print "\tbit\txs0%d"%i
        print "\tbit\txs1%d"%i
        print "\tbit\txs2%d"%i
        xsyndrome0.append("xs0%d"%i)
        xsyndrome1.append("xs1%d"%i)
        xsyndrome2.append("xs2%d"%i)
        print "\tbit\tzs0%d"%i
        print "\tbit\tzs1%d"%i
        print "\tbit\tzs2%d"%i
        zsyndrome0.append("zs0%d"%i)
        zsyndrome1.append("zs1%d"%i)
        zsyndrome2.append("zs2%d"%i)
print "#ancilla qubits (1 logical qubit @ L=%d)"%L
for l in range(L):
        for i in range(7**(l+1)):
                print "\tqubit\tqa%d_%d"%(l+1,i)
                ancilla_temp.append("qa%d_%d"%(l+1,i))
        ancilla.append(ancilla_temp[:])
print "# verification qubits (3 logical qubits @ L-1=%d)"%(L-1)
for l in range(L):
        for i in range(int(7**l)):
                print "\tqubit\tv0_%d_%d"%(l+1,i)
                print "\tqubit\tv1_%d_%d"%(l+1,i)
                print "\tqubit\tv2_%d_%d"%(l+1,i)
                print "\tqubit\tv3_%d_%d"%(l+1,i)
```

```
                    verify1_temp.append("v0_%d_%d"%(l+1,i))
                    verify2_temp.append("v1_%d_%d"%(l+1,i))
                    verify3_temp.append("v2_%d_%d"%(l+1,i))
                    verify4_temp.append("v3_%d_%d"%(l+1,i))
            verify1.append(verify1_temp[:])
            verify2.append(verify2_temp[:])
            verify3.append(verify3_temp[:])
            verify4.append(verify4_temp[:])
        print "# other initialization code"
        full_add_init()
        find_best_syndrome_init()
        print "# other"
        print "\tnoise\tdepolarize"
        print "\tbit\tmagic"

        return ancilla,verify1,verify2,verify3,verify4,vbits,xsyndrome0,xsyndrome1,xsyndrome2,
            zsyndrome0,zsyndrome1,zsyndrome2,meas,meas_H


def declare_data_qubit(label):

        data = []

        print "# data qubits and ancilla qubits (2 logical qubits @ L=%d)"%L
        for i in range(7**L):
                print "\tqubit\tqd_%s_%d"%(label,i)
                data.append("qd_%s_%d"%(label,i))

        return data



##########################################################################3###########
# Transversal gates
#


def h(q):

        for x in q: print "\th\t%s"%x

def X(q):
        #if len(q)<7:
        #        print "\th\t%s"%q
        #else:
        for x in q: print "\tx\t%s"%x

def Z(q):
        #if len(q)<7:
        #        print "\th\t%s"%q
        #else:
        for x in q: print "\tz\t%s"%x

def cnot(qc,qt):

        for i in range(len(qc)):
                print "\tcnot\t%s,%s"%(qc[i],qt[i])

def cz(qc,qt):
```

```
        for i in range(len(qc)):
                print "\tcz\t%s,%s"%(qc[i],qt[i])

def wait(q):

        for x in q: print "\twl\t%s"%x

def identity(q):

        for x in q: print "\tid\t%s"%x

def measure(q,b):

        if len(q) == 1:
                print "\tmeasure\t%s,%s"%(b,q[0])
        else:
                global counterMeasure
                mynumber1 = counterMeasure
                counterMeasure = counterMeasure + 1
                level=int(log(len(q))/log(7))
                for i in range(7):
                        measure(q[i],meas[level-1][i])
                H = [[1,0,0,0,0,1,1],[0,1,0,0,1,0,1],[0,0,1,0,1,1,0],[0,0,0,1,1,1,1]]
                for i in range(4):
                        print "\tset\t%s,0"%(meas_H[i])
                        for j in range(7):
                                if H[i][j]:
                                        print "\txor\t%s,%s,%s"%(meas_H[i],meas_H[i],meas[level
                                            -1][j])

                print "\tset\t%s,0"%(b)

                #First check if Hv=0 (implies outcome=0)
                print "\tset\ttemp0,1"
                for i in range(4):
                        print "\txor\ttemp1,%s,1"%(meas_H[i])
                        print "\tand\ttemp0,temp0,temp1"
                print "\tif\ttemp0"
                print "\tjump\tmeasure_end_%d"%(mynumber1)

                #If not, then check if parity of Hv is zero (implies outcome=1)
                print "\tset\ttemp0,1"
                for i in range(4):
                        print "\txor\ttemp0,temp0,%s"%(meas_H[i])
                print "\tif\ttemp0"
                print "\tjump\tmeasure_outcome1_%d"%(mynumber1)

                #If not, check if Hv is [1110] (implies outcome=1, otherwise 0)
                print "\tset\ttemp0,1"
                for i in range(3):
                        print "\tand\ttemp0,temp0,%s"%(meas_H[i])
                print "\txor\ttemp1,%s,1"%(meas_H[3])
                print "\tand\ttemp0,temp0,temp1"
                print "\tif\ttemp0"
                print "\tjump\tmeasure_outcome1_%d"%(mynumber1)
                print "\tjump\tmeasure_end_%d"%(mynumber1)

                print "\tlabel\tmeasure_outcome1_%d"%(mynumber1)
                print "\tset\t%s,1"%(b)
                print "\tlabel\tmeasure_end_%d"%(mynumber1)
```

```
##########################################################################3#######
# Logical qubit manipulations
#

# Split a logical qubit q (a list of strings) into a list of lists of strings.
# The inner lists are (n-1)-blocks of the n-block q.
def split_qubit(q):

        out = []
        if len(q) == 1: return q          # just return single qubits
        L = int(log(len(q))/log(7))
        for i in range(7):
                out.append(q[int(i*7**(L-1)):int((i+1)*7**(L-1))])
        return out

##################################################################
# Recovery network elements
#

# Preparation (sans verification)
# q is a list of 7 logical qubits (lists)
# or a list of 1 physical qubit (string)
# noiseType is one of "none", "NFT", "full"
#       none -> no noise at all
#       NFT  -> subordinate preparation introduces errors like a single qubit gate replacement
    rule
#       full -> all gate and wait failures
def G(q,noiseType):

        print "#G ACTING ON",q
        print "#-----------------------------------------------"
        print "# G preparation network, noiseType = %s"%noiseType
        print "# acting on a %d-block of M_%d"%(int(log(len(q))/log(7)),L)
        if len(q) == 1:
                if noiseType != "full": print "\tnoise\toff"

                print "\tmeasure\ttemp0,%s"%q[0]
                print "\tif\ttemp0"
                print "\tx\t%s"%q[0]
                if noiseType == "NFT":
                        print "\tidentity\t%s"%q[0]
                if noiseType != "full": print "\tnoise\ton"
        else:
                # timestep 0
                for i in range(7):
                        if noiseType != "full":
                                G(split_qubit(q[i]),"none")
                        else:
                                G(split_qubit(q[i]),"full")
                        if noiseType == "NFT": identity(q[i])
                # timestep 1
                if noiseType == "none": print "\tnoise\toff"
                if noiseType != "none": recover(q[0],noiseType)
                h(q[6])
                if noiseType != "none": recover(q[1],noiseType)
                h(q[5])
                if noiseType != "none": recover(q[2],noiseType)
                h(q[4])
                if noiseType != "none":
                        recover(q[3],noiseType)
```

```
                        wait(q[3])
                if noiseType != "none":
                        recover(q[4],noiseType)
                        wait(q[2])
                if noiseType != "none":
                        recover(q[5],noiseType)
                        wait(q[1])
                if noiseType != "none":
                        recover(q[6],noiseType)
                        wait(q[0])

                # timestep 2-4
                interact = [[3,2,1],[2,0,3],[1,3,0]]
                for x in interact:
                        if noiseType != "none": recover(q[0],noiseType)
                        if noiseType != "none": recover(q[x[0]],noiseType)
                        cnot(q[4],q[x[0]])
                        if noiseType != "none": recover(q[1],noiseType)
                        if noiseType != "none": recover(q[x[1]],noiseType)
                        cnot(q[5],q[x[1]])
                        if noiseType != "none": recover(q[2],noiseType)
                        if noiseType != "none": recover(q[x[2]],noiseType)
                        cnot(q[6],q[x[2]])
                        z = map(lambda y:y not in x+[4,5,6],range(7))
                        for y in range(7):
                                if z[y]:
                                        if noiseType != "none":
                                                recover(q[y],noiseType)
                                                wait(q[y])
                if noiseType == "none": "\tnoise\ton"

# Verification
# q is a list of 7 logical qubits (lists)
# v0, v1, and v2 are logical qubits (lists)
# b is a cbit name
# noiseType is one of "none", "NFT", "full"
#       none -> no noise at all
#       NFT  -> subordinate preparation introduces errors like a single qubit gate replacement
    rule
#       full -> all gate and wait failures
def V(q,v0,v1,v2,v3,b,noiseType):

        print "#V ACTING ON",q,v0,v1,v2
        print "#-------------------------------------------------"
        print "# V verification network, noiseType = %s"%noiseType
        print "# acting on a %d-block of M_%d"%(int(log(len(q))/log(7)),L)
        # these two cycles are not counted
        if noiseType != "full":
                G(split_qubit(v0),"none")
                G(split_qubit(v1),"none")
                G(split_qubit(v2),"none")
                G(split_qubit(v3),"none")
        else:
                G(split_qubit(v0),"full")
                G(split_qubit(v1),"full")
                G(split_qubit(v2),"full")
                G(split_qubit(v3),"full")
        if noiseType == "NFT":
                if noiseType != "none": recover(v0,noiseType)
                identity(v0)
                if noiseType != "none": recover(v1,noiseType)
```

```
        identity(v1)
        if noiseType != "none": recover(v2,noiseType)
        identity(v2)
        if noiseType != "none": recover(v3,noiseType)
        identity(v3)
if noiseType == "none": print "\tnoise\toff"
if noiseType != "none": recover(v0,noiseType)
h(v0)
if noiseType != "none": recover(v1,noiseType)
h(v1)
if noiseType != "none": recover(v2,noiseType)
h(v2)
if noiseType != "none": recover(v3,noiseType)
h(v3)
# timestep 0-3
interact = [[0,1,2,3],[5,4,7,6],[7,6,4,5],[6,7,5,4]]
for x in interact:
        if x[0] == 7:
                if noiseType != "none":
                        recover(v0,noiseType)
                        wait(v0)
        else:
                if noiseType != "none": recover(v0,noiseType)
                if noiseType != "none": recover(q[x[0]],noiseType)
                cz(v0,q[x[0]])
        if x[1] == 7:
                if noiseType != "none":
                        recover(v1,noiseType)
                        wait(v1)
        else:
                if noiseType != "none": recover(v1,noiseType)
                if noiseType != "none": recover(q[x[1]],noiseType)
                cz(v1,q[x[1]])
        if x[2] == 7:
                if noiseType != "none":
                        recover(v2,noiseType)
                        wait(v2)
        else:
                if noiseType != "none": recover(v2,noiseType)
                if noiseType != "none": recover(q[x[2]],noiseType)
                cz(v2,q[x[2]])
        if noiseType != "none": recover(v3,noiseType)
        if noiseType != "none": recover(q[x[3]],noiseType)
        cz(v3,q[x[3]])
        z = map(lambda y:y not in x,range(7))
        for y in range(7):
                if z[y]:
                        if noiseType != "none":
                                recover(q[y],noiseType)
                                wait(q[y])
if noiseType != "none": recover(v0,noiseType)
h(v0)
if noiseType != "none": recover(v1,noiseType)
h(v1)
if noiseType != "none": recover(v2,noiseType)
h(v2)
if noiseType != "none": recover(v2,noiseType)
h(v3)
for i in range(7):
        if noiseType != "none":
                recover(q[i],noiseType)
```

```
                    wait(q[i])
        # timestep 5
        measure(split_qubit(v0),"temp0"); measure(split_qubit(v1),"temp1"); measure(split_qubit(v2
            ),"temp2"); measure(split_qubit(v3),"temp3")
        for i in range(7):
            if noiseType != "none":
                recover(q[i],noiseType)
                wait(q[i])
        # classical decode (parity)
        print "\tset\t%s,temp0"%b
        print "\tor\t%s,%s,temp1"%(b,b)
        print "\tor\t%s,%s,temp2"%(b,b)
        print "\tor\t%s,%s,temp3"%(b,b)

# Syndrome extraction
# what = "x" or "z"
# q,s are lists of 7 logical qubits (lists)
# b is a list of 3 cbits for storing the syndrome
def S(what,q,s,b,noiseType):

        print "#S ACTING ON",q,s
        print "#--------------------------------------------------------"
        print "# S syndrome extraction network"
        print "# acting on a %d-block of M_%d"%(int(log(len(q))/log(7)),L)
        # timestep 0
        if what == "x":
            for i in range(7):
                if noiseType != "none": recover(q[i],noiseType)
                if noiseType != "none": recover(s[i],noiseType)
                cz(s[i],q[i]) #order swapped from paper
        else:
            for i in range(7):
                if noiseType != "none": recover(q[i],noiseType)
                if noiseType != "none": recover(s[i],noiseType)
                cnot(s[i],q[i]) #order swapped from paper
        # timestep 1
        for i in range(7):
            if noiseType != "none": recover(s[i],noiseType)
            h(s[i])
            if noiseType != "none":
                recover(q[i],noiseType)
                wait(q[i])
        # timestep 2
        for i in range(7):
            measure(split_qubit(s[i]),"se%d"%i)
            if noiseType != "none":
                recover(q[i],noiseType)
                wait(q[i])
        # classical decode
        print "\tset\t%s,se0"%b[0]
        print "\txor\t%s,%s,se2"%(b[0],b[0])
        print "\txor\t%s,%s,se4"%(b[0],b[0])
        print "\txor\t%s,%s,se6"%(b[0],b[0])
        print "\tset\t%s,se1"%b[1]
        print "\txor\t%s,%s,se2"%(b[1],b[1])
        print "\txor\t%s,%s,se5"%(b[1],b[1])
        print "\txor\t%s,%s,se6"%(b[1],b[1])
        print "\tset\t%s,se3"%b[2]
        print "\txor\t%s,%s,se4"%(b[2],b[2])
        print "\txor\t%s,%s,se5"%(b[2],b[2])
        print "\txor\t%s,%s,se6"%(b[2],b[2])
```

```
        #print "\tif\t%s"%(b[2])
        #print "halt"

##########################################################################
# classical control support functions
#

# converts a number n to binary, with the final result
# having k binary digits.
# Returns a list l of binary digits with the most significant
# bit in the lower index location.
def toBinary(n,k=0):
        # n is number to convert
        # m is number of binary digits (>= max(digits(n)))
        if n == 0:
                l = [0]
        else:
                m = int(floor(log(n)/log(2)))
                l = (m+1)*[0]
                l[m] = 1
                n -= 2**m
                while n > 0:
                        m = int(floor(log(n)/log(2)))
                        l[m] = 1
                        n -= 2**m
        curlen = len(l)
        if k != 0: # only extend if m != 0 (i.e. default=don't)
                if k-curlen < 0: raise "badNumDigits", [k,n,curlen]
                l.extend([0]*(k-curlen)) # add the extra zeros
        l.reverse() # msb in lowest index
        return l

def counter_top(t,counter_label):

        bt = toBinary(t)
        print "#--------------------------------------------------"
        print "# counter_head_%s: do %d times"%(counter_label,t)
        for i in range(len(bt)):
                print "\tbit\tcount_%s_%d"%(counter_label,i)
                print "\tset\tcount_%s_%d,0"%(counter_label,i)
        print "\tbit\tcountercondition_%s"%counter_label
        print "\tbit\tcountertemp_%s"%counter_label
        print "\tlabel\tcountertop_%s"%counter_label
        print "\tset\tcountercondition_%s,1"%counter_label
        for x in range(len(bt)):
                if bt[x] == 1:
                        print "\tand\tcountercondition_%s,countercondition_%s,count_%s_%d"%(
                            counter_label,counter_label,counter_label,x)
                else:
                        print "\txor\tcountertemp_%s,count_%s_%d,1"%(counter_label,counter_label,x
                            )
                        print "\tand\tcountercondition_%s,countercondition_%s,countertemp_%s"%(
                            counter_label,counter_label,counter_label)
        print "\tif\tcountercondition_%s"%counter_label
        print "\tjump\tcounterbottom_%s"%counter_label

# call prior to using the full_add function
def full_add_init():

        print "#--------------------------------------------------"
```

```
        print "# full_add_init"
        print "\tbit\tfull_add_xorab"
        print "\tbit\tfull_add_andab"
        print "\tbit\tfull_add_andcxorab"
        print "\tset\tfull_add_xorab,0"
        print "\tset\tfull_add_andab,0"
        print "\tset\tfull_add_andcxorab,0"

# compute a+b+c, where c is the carry
# place the output in s and the output carry in c1
def full_add(a,b,c,s,c1):

        print "\txor\tfull_add_xorab,%s,%s"%(a,b)
        print "\tand\tfull_add_andab,%s,%s"%(a,b)
        print "\tand\tfull_add_andcxorab,full_add_xorab,%s"%c
        print "\txor\t%s,%s,full_add_xorab"%(s,c)
        print "\tor\t%s,full_add_andab,full_add_andcxorab"%c1

def counter_bottom(t,counter_label):

        bt = toBinary(t)
        print "#------------------------------------------------"
        print "# counter_bottom_%s"%counter_label
        full_add("1","count_%s_%d"%(counter_label,len(bt)-1),"0",\
                "count_%s_%d"%(counter_label,len(bt)-1),"countertemp_%s"%counter_label)
        for x in range(len(bt)-2,-1,-1):
                full_add("0","count_%s_%d"%(counter_label,x),"countertemp_%s"%counter_label,\
                        "count_%s_%d"%(counter_label,x),"countertemp_%s"%counter_label)
        print "\tjump\tcountertop_%s"%counter_label
        print "\tlabel\tcounterbottom_%s"%counter_label


def find_best_syndrome_init():

        print "# find_best_syndrome_init"
        for i in range(s):
                print "\tbit\tnot_guessed_%d"%(i)
        for i in range(3):
                print "\tbit\tguess_s%d"%(i)
        for i in range(1,s_prime+1):
                print "\tbit\tnumber_of_matches_%d"%(i)
        print "\tbit\tmatch"
        print "\tbit\tmatch_temp"

def find_best_syndrome(s0,s1,s2,label):

        print "#FIND BEST SYNDROME"
        print "#------------------------------------------------"

        for i in range(s):
                print "\tset\tnot_guessed_%d,1"%(i)
        for i in range(s-s_prime+1):
                print "#GUESSING SYNDROME %d"%(i)
                print "#------------------------------------------------"
                print "\tlabel\tguess_%d_%s"%(i,label)
                print "\tset\tguess_s0,%s"%(s0[i])
                print "\tset\tguess_s1,%s"%(s1[i])
                print "\tset\tguess_s2,%s"%(s2[i])
                print "\tset\tnot_guessed_%d,0"%(i)
                print "\tset\tnumber_of_matches_1,1"
                for i in range(2,s_prime+1):
```

```
                    print "\tset\tnumber_of_matches_%d,0"%(i)
                print "\tjump\tcompare_with_all_syndromes_%s"%(label)

    print "#COMPARE GUESS WITH ALL UNGUESSED SYNDROMES"
    print "#-----------------------------------------------------"
    print "\tlabel\tcompare_with_all_syndromes_%s"%(label)
    for i in range(1,s):
        print "\tif\tnot_guessed_%d"%(i)
        print "\tjump\tcompare_to_%d_%s"%(i,label)
        print "\tlabel\tcompared_to_%d_%s"%(i,label)
    for i in range(1,s-s_prime+1):
        print "\tif\tnot_guessed_%d"%(i)
        print "\tjump\tguess_%d_%s"%(i,label)
    if s!=1:
        print "\tjump\terror_corrected_%s"%(label)

    for i in range(1,s):
        print "#COMPARE GUESS TO SYNDROME %d"%(i)
        print "\tlabel\tcompare_to_%d_%s"%(i,label)
        print "\tset\tmatch,1"
        print "\txor\tmatch_temp,guess_s0,%s"%(s0[i])
        print "\txor\tmatch_temp,match_temp,1"
        print "\tand\tmatch,match,match_temp"
        print "\txor\tmatch_temp,guess_s1,%s"%(s1[i])
        print "\txor\tmatch_temp,match_temp,1"
        print "\tand\tmatch,match,match_temp"
        print "\txor\tmatch_temp,guess_s2,%s"%(s2[i])
        print "\txor\tmatch_temp,match_temp,1"
        print "\tand\tmatch,match,match_temp"
        print "\txor\tmatch_temp,match,1"
        print "\tif\tmatch_temp"
        print "\tjump\tcompared_to_%d_%s"%(i,label)
        print "\tset\tnot_guessed_%d,0"%(i)
        for j in range(s_prime,0,-1):
            print "\tand\tmatch_temp,number_of_matches_%d,1"%(j-1)
            print "\tset\tnumber_of_matches_%d,match_temp"%(j)
        print "\tif\tnumber_of_matches_%d"%(s_prime)
        print "\tjump\tfound_best_syndrome_%s"%(label)
        print "\tjump\tcompared_to_%d_%s"%(i,label)




# Recovery operation (error correction)
# q is a list of 7 logical qubits (lists)
# noiseType is one of "none", "NFT", "full"
#       NFT -> subordinate preparation introduces errors like a single qubit gate replacement
    rule
#       full -> all gate and wait failures
def recover(q,noiseType):


    #only recover q if q refers to more than one qubit
    if len(q) > 1:


        global counterRecover
        mynumber1 = counterRecover
        counterRecover = counterRecover + 1

        global ancilla,verify1,verify2,verify3,verify4,vbits
```

```
        global xsyndrome0, xsyndrome1, xsyndrome2
        global zsyndrome0, zsyndrome1, zsyndrome2


        # take slices of the large registers to give us ancilla
        # that are the right size
        myancilla = ancilla[int(log(len(q))/log(7)-1)][0:len(q)]
        myverify1 = verify1[int(log(len(q))/log(7)-1)][0:len(q)/7]
        myverify2 = verify2[int(log(len(q))/log(7)-1)][0:len(q)/7]
        myverify3 = verify3[int(log(len(q))/log(7)-1)][0:len(q)/7]
        myverify4 = verify4[int(log(len(q))/log(7)-1)][0:len(q)/7]

        print "#RECOVER ACTING ON", q, myancilla, myverify1, myverify2, myverify3
        print "#-----------------------------------------------------"

        q_split = split_qubit(q)

########################################
# X error correction


        global counterSyndrome
        mynumber2 = counterSyndrome
        counterSyndrome = counterSyndrome + 1

        # gather one syndrome
        prepare_until_pass(myancilla, myverify1, myverify2, myverify3, myverify4,\
                           vbits, noiseType)
        S("z", split_qubit(q), split_qubit(myancilla),\
          [zsyndrome0[0], zsyndrome1[0], zsyndrome2[0]], noiseType)

        print "\txor\ttemp0,1,%s"%zsyndrome0[0]
        print "\tand\ttemp1,temp0,1"
        print "\txor\ttemp0,1,%s"%zsyndrome1[0]
        print "\tand\ttemp1,temp1,temp0"
        print "\txor\ttemp0,1,%s"%zsyndrome2[0]
        print "\tand\ttemp1,temp1,temp0"
        print "\tif\ttemp1"
        print "\tjump\tno_ec_needed_%d"%mynumber2

        for i in range(s-1):
                prepare_until_pass(myancilla, myverify1, myverify2, myverify3, myverify4, vbits
                     , noiseType)
                if noiseType != "none":
                        for j in range(i):
                                wait(myancilla)
                                wait(myverify1)
                                wait(myverify2)
                                wait(myverify3)
                                wait(myverify4)
                S("z", split_qubit(q), split_qubit(myancilla),\
                  [zsyndrome0[i+1], zsyndrome1[i+1], zsyndrome2[i+1]], noiseType)

        find_best_syndrome(zsyndrome0, zsyndrome1, zsyndrome2, mynumber2)

        print "\tlabel\tfound_best_syndrome_%d"%(mynumber2)
        #need to error correct

        print "#ERROR CORRECTING Z"
        print "#-----------------------------------------------------"
```

```
print "\tif\tguess_s2"
print "\tjump\tcorrect_1xx_%d"%(mynumber2)
print "\tif\tguess_s1"
print "\tjump\tcorrect_01x_%d"%(mynumber2)
print "\tif\tguess_s0"
print "\tjump\tcorrect_001_%d"%(mynumber2)
#syndrome 000

print "\tjump\terror_corrected_%d"%(mynumber2)

print "\tlabel\tcorrect_1xx_%d"%(mynumber2)
print "\tif\tguess_s1"
print "\tjump\tcorrect_11x_%d"%(mynumber2)
print "\tif\tguess_s0"
print "\tjump\tcorrect_101_%d"%(mynumber2)
#syndrome 100
Z(q_split[3])
x = 3
if noiseType != "none":
        for i in range(0,x):
                wait(q_split[i])
        for i in range(x+1,7):
                wait(q_split[i])
print "\tjump\terror_corrected_%d"%(mynumber2)

print "\tlabel\tcorrect_01x_%d"%(mynumber2)
print "\tif\tguess_s0"
print "\tjump\tcorrect_011_%d"%(mynumber2)
#syndrome 010
Z(q_split[1])
x = 1
if noiseType != "none":
        for i in range(0,x):
                wait(q_split[i])
        for i in range(x+1,7):
                wait(q_split[i])
print "\tjump\terror_corrected_%d"%(mynumber2)

print "\tlabel\tcorrect_001_%d"%(mynumber2)
#syndrome 001
Z(q_split[0])
x = 0
if noiseType != "none":
        for i in range(0,x):
                wait(q_split[i])
        for i in range(x+1,7):
                wait(q_split[i])
print "\tjump\terror_corrected_%d"%(mynumber2)

print "\tlabel\tcorrect_11x_%d"%(mynumber2)
print "\tif\tguess_s0"
print "\tjump\tcorrect_111_%d"%(mynumber2)
#syndrome 110
Z(q_split[5])
x = 5
if noiseType != "none":
        for i in range(0,x):
                wait(q_split[i])
        for i in range(x+1,7):
                wait(q_split[i])
print "\tjump\terror_corrected_%d"%(mynumber2)
```

```
        print "\tlabel\tcorrect_101_%d"%(mynumber2)
        #syndrome 101
        Z(q_split[4])
        x = 4
        if noiseType != "none":
                for i in range(0,x):
                        wait(q_split[i])
                for i in range(x+1,7):
                        wait(q_split[i])
        print "\tjump\terror_corrected_%d"%(mynumber2)

        print "\tlabel\tcorrect_011_%d"%(mynumber2)
        #syndrome 011
        Z(q_split[2])
        x = 2
        if noiseType != "none":
                for i in range(0,x):
                        wait(q_split[i])
                for i in range(x+1,7):
                        wait(q_split[i])
        print "\tjump\terror_corrected_%d"%(mynumber2)

        print "\tlabel\tcorrect_111_%d"%(mynumber2)
        #syndrome 111
        Z(q_split[6])
        x = 6
        if noiseType != "none":
                for i in range(0,x):
                        wait(q_split[i])
                for i in range(x+1,7):
                        wait(q_split[i])
        print "\tjump\terror_corrected_%d"%(mynumber2)


        print "\tlabel\tno_cc_needed_%d"%mynumber2

        if noiseType != "none":
                for i in range(s-1):
                        wait(q)

        print "\tlabel\terror_corrected_%d"%(mynumber2)

############################################
# X error correction


        mynumber2 = counterSyndrome
        counterSyndrome = counterSyndrome + 1


        prepare_until_pass(myancilla, myverify1, myverify2, myverify3, myverify4,\
                            vbits, noiseType)
        S("x", split_qubit(q), split_qubit(myancilla),\
           [xsyndrome0[0], xsyndrome1[0], xsyndrome2[0]], noiseType)
        # the data waits during 6 timesteps in X, Z


        # if the syndome is nonzero, gather s total syndromes
        print "\txor\ttemp0,1,%s"%xsyndrome0[0]
        print "\tand\ttemp1,temp0,1"
```

```
print "\txor\ttemp0,1,%s"%xsyndrome1[0]
print "\tand\ttemp1,temp1,temp0"
print "\txor\ttemp0,1,%s"%xsyndrome2[0]
print "\tand\ttemp1,temp1,temp0"
print "\tif\ttemp1"
print "\tjump\tno_ec_needed_%d"%mynumber2

for i in range(s-1):
        prepare_until_pass(myancilla,myverify1,myverify2,myverify3,myverify4,vbits
             ,noiseType)
        if noiseType != "none":
                for j in range(i):
                        wait(myancilla)
                        wait(myverify1)
                        wait(myverify2)
                        wait(myverify3)
                        wait(myverify4)
        S("x",split_qubit(q),split_qubit(myancilla),\
          [xsyndrome0[i+1],xsyndrome1[i+1],xsyndrome2[i+1]],noiseType)


find_best_syndrome(xsyndrome0,xsyndrome1,xsyndrome2,mynumber2)

print "\tlabel\tfound_best_syndrome_%s"%(mynumber2)
#need to error correct
print "#ERROR CORRECTING X"
print "#----------------------------------------------"

print "\tif\tguess_s2"
print "\tjump\tcorrect_1xx_%d"%(mynumber2)
print "\tif\tguess_s1"
print "\tjump\tcorrect_01x_%d"%(mynumber2)
print "\tif\tguess_s0"
print "\tjump\tcorrect_001_%d"%(mynumber2)
#syndrome 000

print "\tjump\terror_corrected_%d"%(mynumber2)

print "\tlabel\tcorrect_1xx_%d"%(mynumber2)
print "\tif\tguess_s1"
print "\tjump\tcorrect_11x_%d"%(mynumber2)
print "\tif\tguess_s0"
print "\tjump\tcorrect_101_%d"%(mynumber2)
#syndrome 100
X(q_split[3])
x = 3
if noiseType != "none":
        for i in range(0,x):
                wait(q_split[i])
        for i in range(x+1,7):
                wait(q_split[i])
print "\tjump\terror_corrected_%d"%(mynumber2)

print "\tlabel\tcorrect_01x_%d"%(mynumber2)
print "\tif\tguess_s0"
print "\tjump\tcorrect_011_%d"%(mynumber2)
#syndrome 010
X(q_split[1])
x = 1
if noiseType != "none":
        for i in range(0,x):
```

```
                wait(q_split[i])
        for i in range(x+1,7):
                wait(q_split[i])
print "\tjump\terror_corrected_%d"%(mynumber2)

print "\tlabel\tcorrect_001_%d"%(mynumber2)
#syndrome 001
X(q_split[0])
x = 0
if noiseType != "none":
        for i in range(0,x):
                wait(q_split[i])
        for i in range(x+1,7):
                wait(q_split[i])
print "\tjump\terror_corrected_%d"%(mynumber2)

print "\tlabel\tcorrect_11x_%d"%(mynumber2)
print "\tif\tguess_s0"
print "\tjump\tcorrect_111_%d"%(mynumber2)
#syndrome 110
X(q_split[5])
x = 5
if noiseType != "none":
        for i in range(0,x):
                wait(q_split[i])
        for i in range(x+1,7):
                wait(q_split[i])
print "\tjump\terror_corrected_%d"%(mynumber2)

print "\tlabel\tcorrect_101_%d"%(mynumber2)
#syndrome 101
X(q_split[4])
x = 4
if noiseType != "none":
        for i in range(0,x):
                wait(q_split[i])
        for i in range(x+1,7):
                wait(q_split[i])
print "\tjump\terror_corrected_%d"%(mynumber2)

print "\tlabel\tcorrect_011_%d"%(mynumber2)
#syndrome 011
X(q_split[2])
x = 2
if noiseType != "none":
        for i in range(0,x):
                wait(q_split[i])
        for i in range(x+1,7):
                wait(q_split[i])
print "\tjump\terror_corrected_%d"%(mynumber2)

print "\tlabel\tcorrect_111_%d"%(mynumber2)
#syndrome 111
X(q_split[6])
x = 6
if noiseType != "none":
        for i in range(0,x):
                wait(q_split[i])
        for i in range(x+1,7):
                wait(q_split[i])
print "\tjump\terror_corrected_%d"%(mynumber2)
```

```
                print "\tlabel\tno_cc_needed_%d"%mynumber2

                if noiseType != "none":
                        for i in range(s-1):
                                wait(q)

                print "\tlabel\terror_corrected_%d"%(mynumber2)




def prepare_until_pass(q, verify1, verify2, verify3, verify4, vbits, noiseType):

        global counterPrepare
        mynumber = counterPrepare
        counterPrepare = counterPrepare + 1
        print "\tlabel\tprepare_until_%d"%mynumber

        print "#PREPARE UNTIL PASS", q, verify1, verify2, verify3, verify4
        G(split_qubit(q), noiseType)
        V(split_qubit(q), verify1, verify2, verify3, verify4,\
                        vbits[0], noiseType)
        print "\tif\t%s"%vbits[0]
        print "\tjump\tprepare_until_%d"%mynumber


############################################################################################
#CODE FOR COMPARING STABILIZERS
#

def is_logical_zero(q):
        if len(q) == 1:
                print "\tsubset\tmagic,1,q[0],Z"
        if len(q) == 7:
                print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,
                     IIIZZZZ,IZZIIZZ,ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
        print "\txor\tmagic,magic,1"
        print "\tif\tmagic"
        print "\thalt"

def zero_has_no_x_error(q):
        if len(q) == 1:
                print "\tsubset\tmagic,1,q[0],-Z"
                print "\tif\tmagic"
                print "\thalt"
        if len(q) == 7:
                print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,-ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,
                     IIIZZZZ,IZZIIZZ,-ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
                print "\tif\tmagic"
                print "\thalt"
                print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,-ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,
                     IIIZZZZ,-IZZIIZZ,ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
                print "\tif\tmagic"
                print "\thalt"
                print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,-ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,
                     IIIZZZZ,-IZZIIZZ,-ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
                print "\tif\tmagic"
                print "\thalt"
```

```
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,-ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,-IIIZZZZ,IZZIIZZ,ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,-ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,-IIIZZZZ,IZZIIZZ,-ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,-ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,-IIIZZZZ,-IZZIIZZ,ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,-ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,-IIIZZZZ,-IZZIIZZ,-ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"

def zero_has_no_y_error(q):
        if len(q) == 1:
            print "\tsubset\tmagic,1,q[0],-Z"
            print "\tif\tmagic"
            print "\thalt"
        if len(q) == 7:
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,-ZZZZZZZ,IIIXXXX,IXXIIXX,-XIXIXIX,IIIZZZZ,IZZIIZZ,-ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,-ZZZZZZZ,IIIXXXX,-IXXIIXX,XIXIXIX,IIIZZZZ,-IZZIIZZ,ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,-ZZZZZZZ,IIIXXXX,-IXXIIXX,-XIXIXIX,IIIZZZZ,-IZZIIZZ,-ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,-ZZZZZZZ,-IIIXXXX,IXXIIXX,XIXIXIX,-IIIZZZZ,IZZIIZZ,ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,-ZZZZZZZ,-IIIXXXX,IXXIIXX,-XIXIXIX,-IIIZZZZ,IZZIIZZ,-ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,-ZZZZZZZ,-IIIXXXX,-IXXIIXX,XIXIXIX,-IIIZZZZ,-IZZIIZZ,ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,-ZZZZZZZ,-IIIXXXX,-IXXIIXX,-XIXIXIX,-IIIZZZZ,-IZZIIZZ,-ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"

def zero_has_no_z_error(q):

        if len(q) == 7:
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,ZZZZZZZ,IIIXXXX,IXXIIXX,-XIXIXIX,IIIZZZZ,IZZIIZZ,ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,ZZZZZZZ,IIIXXXX,-IXXIIXX,XIXIXIX,IIIZZZZ,IZZIIZZ,ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
```

```
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,ZZZZZZZ,IIIXXXX,-IXXIIXX,-XIXIXIX,
                IIIZZZZ,IZZIIZZ,ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,ZZZZZZZ,-IIIXXXX,IXXIIXX,XIXIXIX,
                IIIZZZZ,IZZIIZZ,ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,ZZZZZZZ,-IIIXXXX,IXXIIXX,-XIXIXIX,
                IIIZZZZ,IZZIIZZ,ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,ZZZZZZZ,-IIIXXXX,-IXXIIXX,XIXIXIX,
                IIIZZZZ,IZZIIZZ,ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"
            print "\tsubset\tmagic,7,%s,%s,%s,%s,%s,%s,%s,ZZZZZZZ,-IIIXXXX,-IXXIIXX,-XIXIXIX,
                IIIZZZZ,IZZIIZZ,ZIZIZIZ"%(q[0],q[1],q[2],q[3],q[4],q[5],q[6])
            print "\tif\tmagic"
            print "\thalt"



##################################################################################3
#MAIN PROGRAM

if len(sys.argv) < 6:

        print "nft.py gate L s s_prime t [noise]"
        print ""
        print "Generate an ARQ source file for an M_L simulated identity gate using"
        print "the Steane [[7,1,3]] code. The preparation, verification, and recovery"
        print "mirrors the nonlocal model given in quant-ph/0410047 by Svore, Terhal,"
        print "and DiVincenzo."
        print ""
        print "gate\t one of id, h, cx, hm (hadamards followed by a measure), or wl."
        print "L\tcreates an M_L simulating circuit"
        print "s\tnumber of syndromes collected prior to recovery"
        print "s_prime\tnumber of syndromes that must agree"
        print "t\tnumber of identity gates to apply"
        print "noise\tdefaults to preparation as a single qubit replacement rule"
        print ""
        print "Assume that enough ancilla are prepared in parallel before the beginning of"
        print "error correction. Assume these are prepared during the previous error correction,"
        print "so they do not contribute to the data wait time."
        sys.exit(1)

gate = sys.argv[1]
L = int(sys.argv[2])
s = int(sys.argv[3])
s_prime = int(sys.argv[4])
t = int(sys.argv[5])
if len(sys.argv) == 6:
        nType = "NFT"
else:
        nType = sys.argv[6]

if L < 1:
        print "Oops, L must be greater than 0!"
```

```
        sys.exit(1)

if t < 1:
        print "Oops, t must be greater than 0!"
        sys.exit(1)



print "# %s"%version_info
print "# %s"%time.ctime()
print "# L=%d, s=%d, s_prime=%d, t=%d"%(L,s,s_prime,t)
print "#"

if gate == "id":
        data = declare_data_qubit("0")
        ancilla,verify1,verify2,verify3,verify4,vbits,xsyndrome0,xsyndrome1,xsyndrome2,zsyndrome0,
            zsyndrome1,zsyndrome2,meas,meas_H = declare_all_but_data()
        print "# prepare the data qubits"
        G(split_qubit(data),"none")
        print "# apply %d identity gates"%t
        counter_top(t,"identity_gate_count")
        print "# prepare up to %d ancilla states"%s
        recover(data,nType)
        identity(data)
        counter_bottom(t,"identity_gate_count")
        s = 1
        s_prime = 1
        recover(data,"none")
        print "\tnoise\toff"
        #is_logical_zero(data)
        zero_has_no_x_error(data)

elif gate == "h":
        data = declare_data_qubit("0")
        ancilla,verify1,verify2,verify3,verify4,vbits,xsyndrome0,xsyndrome1,xsyndrome2,zsyndrome0,
            zsyndrome1,zsyndrome2,meas,meas_H = declare_all_but_data()
        print "# prepare the data qubits"
        G(split_qubit(data),"none")
        print "# apply %d hadamard gates"%t
        counter_top(t,"hadamard_gate_count")
        print "# prepare up to %d ancilla states"%s
        recover(data,nType)
        h(data)
        counter_bottom(t,"hadamard_gate_count")
        s = 1
        s_prime = 1
        recover(data,"none")
        print "\tnoise\toff"
        if (t%2==1):
                h(data)
        is_logical_zero(data)

elif gate == "cx":
        data_c = declare_data_qubit("c")
        data_t = declare_data_qubit("t")
        ancilla,verify1,verify2,verify3,verify4,vbits,xsyndrome0,xsyndrome1,xsyndrome2,zsyndrome0,
            zsyndrome1,zsyndrome2,meas,meas_H = declare_all_but_data()
        print "# prepare the data qubits"
        G(split_qubit(data_c),"none")
        G(split_qubit(data_t),"none")
        print "# apply %d cx gates"%t
```

```
        counter_top(t,"cx_gate_count")
        print "# prepare up to %d ancilla states"%s
        recover(data_c,nType)
        recover(data_t,nType)
        cnot(data_c,data_t)
        counter_bottom(t,"cx_gate_count")
        s = 1
        s_prime = 1
        recover(data_c,"none")
        recover(data_t,"none")
        print "\tnoise\toff"
        cnot(data_c,data_t)
        is_logical_zero(data_c)
        is_logical_zero(data_t)

elif gate == "hm":
        data = declare_data_qubit("0")
        ancilla,verify1,verify2,verify3,verify4,vbits,xsyndrome0,xsyndrome1,xsyndrome2,zsyndrome0,
            zsyndrome1,zsyndrome2,meas,meas_H = declare_all_but_data()
        print "# prepare the data qubits"
        G(split_qubit(data),"none")
        print "# apply %d hm gates"%t
        counter_top(t,"hm_gate_count")
        print "# prepare up to %d ancilla states"%s
        recover(data,nType)
        h(data)
        counter_bottom(t,"hm_gate_count")
        print "\tnoise\toff"
        if (t%2==1):
                h(data)
        print "\tnoise\ton"
        measure(split_qubit(data),"magic")
        print "\tif\tmagic"
        print "\thalt"

elif gate == "w1":
        data = declare_data_qubit("0")
        ancilla,verify1,verify2,verify3,verify4,vbits,xsyndrome0,xsyndrome1,xsyndrome2,zsyndrome0,
            zsyndrome1,zsyndrome2,meas,meas_H = declare_all_but_data()
        print "# prepare the data qubits"
        G(split_qubit(data),"none")
        print "# apply %d wait gates"%t
        counter_top(t,"wait_gate_count")
        print "# prepare up to %d ancilla states"%s
        recover(data,nType)
        print "\tnoise\ton"
        wait(data)
        counter_bottom(t,"wait_gate_count")
        s = 1
        s_prime = 1
        recover(data,"none")
        print "\tnoise\toff"
        is_logical_zero(data)

else:
        print "Oops. Gate must be one of id,h,cx,hm,w1."
        sys.exit(1)

print "# EOF"
```

Appendix E

Sample ARQ Code

In this Appendix we present some sample ARQ code. The sample code simulates six repetitions of the error-correction circuit and then checks whether there is a single Z error on the data using stabilizers. The circuit is exactly the same as described in Chapter 3. Note that wait gates have to be explicitly called.

The language specifications for ARQ are given in Tables E.1, E.2, and E.3. Table E.1 lists the defined classical computer instructions and Tables E.2 and E.3 list the defined quantum computer instructions.

Opcode	Arguments	Description
nop	none	No operation, do nothing
bit	bit_name, ..., bit_name	Create new named classical bits
label	label_name	Create a new jump target
jump	label_name	Set the instruction pointer to the location of label_name
and	tgt_bit, left_bit, right_bit	Store (left_bit & right_bit) in tgt_bit. Either or both of left_bit, right_bit can be binary constants.
xor	tgt_bit, left_bit, right_bit	Store (left_bit r̂ight_bit) in tgt_bit. Either or both of left_bit, right_bit can be binary constants.
or	tgt_bit, left_bit, right_bit	Store (left_bit — right_bit) in tgt_bit. Either or both of left_bit, right_bit can be binary constants.
if	bit_name, ..., bit_name	Execute the next instruction only if each argument bit is 1.
set	tgt_bit, src_bit	Set tgt_bit to the value of src_bit. The src_bit can be a binary constant.
halt	none	Causes the virtual machine to throw an exception. The virtual machine will abort with a "FAIL" result. Use this to signal that the program state has become corrupted.

Table E.1: The classical instructions defined in ARQ.

Opcode	Arguments	Description
qubit	qubit_name, ..., qubit_name	Create new named quantum bits
measure	bit_name, qubit_name	Projectively measure the qubit named qubit_name in the computational basis. Store the result in the classical bit named bit_name.
x	qubit_name	Apply the Pauli X gate (bit-flip) to the qubit named qubit_name
y	qubit_name	Apply the Pauli Y gate (bit-phase-flip) to the qubit named qubit_name
z	qubit_name	Apply the Pauli Z gate (phase-flip) to the qubit named qubit_name
id	qubit_name	Apply the Pauli identity gate to the qubit named qubit_name
w1	qubit_name	Apply a single qubit wait gate to the qubit named qubit_name
h	qubit_name	Apply the Hadamard gate to the qubit named qubit_name. This gate maps $\|0\rangle$ to $\|0\rangle+\|1\rangle$ and $\|1\rangle$ to $\|0\rangle-\|1\rangle$ (normalization factor omitted).
s	qubit_name	Apply the pi/4 gate to the qubit named qubit_name. This gate maps $\|0\rangle$ to $\|0\rangle$ and $\|1\rangle$ to $i\|1\rangle$.

Table E.2: The quantum instructions defined in ARQ.

Opcode	Arguments	Description
cnot	control_qubit, target_qubit	Apply the controlled-NOT gate using control_qubit as the control and target_qubit as the target. The controlled-NOT gate flips the target qubit if the control qubit is 1.
cz	control_qubit, target_qubit	Apply the controlled-phase gate using control_qubit as the control and target_qubit as the target. The controlled-phase gate flips the phase of the target qubit if the control qubit is 1.
subset	bit_name, integer_N, qubit_1, ..., qubit_N, generator_1, ..., generator_N	Compares the requested subset of N qubits to the given stabilizer state (specified by N stabilizer generators).

Table E.3: More quantum instructions defined in ARQ.

```
# nft.py version 1.1 amorten@mit.edu, last
    modified 25 August 2005
# Thurs August 25 21:15:54 2005
# L=1, s=3, s_prime=2, t=6
#
# data qubits and ancilla qubits (2 logical
    qubits @ L=1)
        qubit   qd_0_0
        qubit   qd_0_1
        qubit   qd_0_2
        qubit   qd_0_3
        qubit   qd_0_4
        qubit   qd_0_5
        qubit   qd_0_6
#-----------------------------------------------
# declare(L=1,s=3)
#
# measurement bits(4+1x7 cbits)
        bit     meas_H_0
        bit     meas_H_1
        bit     meas_H_2
        bit     meas_H_3
        bit     meas_1_0
        bit     meas_1_1
        bit     meas_1_2
        bit     meas_1_3
        bit     meas_1_4
        bit     meas_1_5
        bit     meas_1_6
# temporary cbits used in measurement (3 cbits)
        bit     temp0
        bit     temp1
        bit     temp2
        bit     temp3
# temporary cbits used in syndrome extraction
     (7 cbits)
        bit     se0
        bit     se1
        bit     se2
        bit     se3
        bit     se4
        bit     se5
        bit     se6
# qubits and cbits that must be passed into G, V
    , S, etc
# verify cbit
        bit     v
# syndome bits, 2x3x3 of them
        bit     xs00
        bit     xs10
        bit     xs20
        bit     zs00
        bit     zs10
        bit     zs20
        bit     xs01
        bit     xs11
        bit     xs21
        bit     zs01
        bit     zs11
        bit     zs21
        bit     xs02
        bit     xs12
        bit     xs22
        bit     zs02
        bit     zs12
        bit     zs22
#ancilla qubits (1 logical qubit @ L=1)
        qubit   qa1_0
        qubit   qa1_1
        qubit   qa1_2
        qubit   qa1_3
        qubit   qa1_4
        qubit   qa1_5
        qubit   qa1_6
# verification qubits (3 logical qubits @ L-1=0)
        qubit   v0_1_0
        qubit   v1_1_0
        qubit   v2_1_0
        qubit   v3_1_0
# other initialization code
#-----------------------------------------------
# full_add_init
        bit     full_add_xorab
        bit     full_add_andab
        bit     full_add_andcxorab
        set     full_add_xorab,0
        set     full_add_andab,0
        set     full_add_andcxorab,0
# find_best_syndrome_init
        bit     not_guessed_0
        bit     not_guessed_1
        bit     not_guessed_2
        bit     guess_s0
        bit     guess_s1
        bit     guess_s2
        bit     number_of_matches_1
        bit     number_of_matches_2
        bit     match
        bit     match_temp
# other
        noise   depolarize
        bit     magic
# prepare the data qubits
#G ACTING ON [['qd_0_0'], ['qd_0_1'], ['qd_0_2
    '], ['qd_0_3'], ['qd_0_4'], ['qd_0_5'], ['
    qd_0_6']]
#-----------------------------------------------
# G preparation network, noiseType = none
# acting on a 1-block of M_1
#G ACTING ON ['qd_0_0']
#-----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,qd_0_0
        if      temp0
        x       qd_0_0
        noise   on
#G ACTING ON ['qd_0_1']
#-----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
```

```
        noise    off
        measure  temp0,qd_0_1
        if       temp0
        x        qd_0_1
        noise    on
#G ACTING ON ['qd_0_2']
#------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qd_0_2
        if       temp0
        x        qd_0_2
        noise    on
#G ACTING ON ['qd_0_3']
#------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qd_0_3
        if       temp0
        x        qd_0_3
        noise    on
#G ACTING ON ['qd_0_4']
#------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qd_0_4
        if       temp0
        x        qd_0_4
        noise    on
#G ACTING ON ['qd_0_5']
#------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qd_0_5
        if       temp0
        x        qd_0_5
        noise    on
#G ACTING ON ['qd_0_6']
#------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qd_0_6
        if       temp0
        x        qd_0_6
        noise    on
        noise    off
        h        qd_0_6
        h        qd_0_5
        h        qd_0_4
        cnot     qd_0_4,qd_0_3
        cnot     qd_0_5,qd_0_2
        cnot     qd_0_6,qd_0_1
        cnot     qd_0_4,qd_0_2
        cnot     qd_0_5,qd_0_0
        cnot     qd_0_6,qd_0_3
        cnot     qd_0_4,qd_0_1
```

```
        cnot     qd_0_5,qd_0_3
        cnot     qd_0_6,qd_0_0
# apply 6 identity gates
#------------------------------------------------
# counter_head_id_gate_count: do 6 times
        bit      count_id_gate_count_0
        set      count_id_gate_count_0,0
        bit      count_id_gate_count_1
        set      count_id_gate_count_1,0
        bit      count_id_gate_count_2
        set      count_id_gate_count_2,0
        bit      countercondition_id_gate_count
        bit      countertemp_id_gate_count
        label    countertop_id_gate_count
        set      countercondition_id_gate_count,1
        and      countercondition_id_gate_count,
            countercondition_id_gate_count,
            count_id_gate_count_0
        and      countercondition_id_gate_count,
            countercondition_id_gate_count,
            count_id_gate_count_1
        xor      countertemp_id_gate_count,
            count_id_gate_count_2,1
        and      countercondition_id_gate_count,
            countercondition_id_gate_count,
            countertemp_id_gate_count
        if       countercondition_id_gate_count
        jump     counterbottom_id_gate_count
# prepare up to 3 ancilla states
#RECOVER ACTING ON ['qd_0_0', 'qd_0_1', 'qd_0_2
    ', 'qd_0_3', 'qd_0_4', 'qd_0_5', 'qd_0_6
    '] ['qa1_0', 'qa1_1', 'qa1_2', 'qa1_3', '
    qa1_4', 'qa1_5', 'qa1_6'] ['v0_1_0'] ['
    v1_1_0'] ['v2_1_0']
#------------------------------------------------
        label    prepare_until_0
#PREPARE UNTIL PASS ['qa1_0', 'qa1_1', 'qa1_2
    ', 'qa1_3', 'qa1_4', 'qa1_5', 'qa1_6'] ['
    v0_1_0'] ['v1_1_0'] ['v2_1_0'] ['v3_1_0']
#G ACTING ON [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']]
#------------------------------------------------
# G preparation network, noiseType = NFT
# acting on a 1-block of M_1
#G ACTING ON ['qa1_0']
#------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_0
        if       temp0
        x        qa1_0
        noise    on
        id       qa1_0
#G ACTING ON ['qa1_1']
#------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_1
```

```
        if       temp0
        x        qa1_1
        noise    on
        id       qa1_1
#G ACTING ON ['qa1_2']
#----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_2
        if       temp0
        x        qa1_2
        noise    on
        id       qa1_2
#G ACTING ON ['qa1_3']
#----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_3
        if       temp0
        x        qa1_3
        noise    on
        id       qa1_3
#G ACTING ON ['qa1_4']
#----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_4
        if       temp0
        x        qa1_4
        noise    on
        id       qa1_4
#G ACTING ON ['qa1_5']
#----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_5
        if       temp0
        x        qa1_5
        noise    on
        id       qa1_5
#G ACTING ON ['qa1_6']
#----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_6
        if       temp0
        x        qa1_6
        noise    on
        id       qa1_6
        h        qa1_6
        h        qa1_5
        h        qa1_4
        w1       qa1_3
        w1       qa1_2
        w1       qa1_1
        w1       qa1_0
        cnot     qa1_4,qa1_3
        cnot     qa1_5,qa1_2
        cnot     qa1_6,qa1_1
        w1       qa1_0
        cnot     qa1_4,qa1_2
        cnot     qa1_5,qa1_0
        cnot     qa1_6,qa1_3
        w1       qa1_1
        cnot     qa1_4,qa1_1
        cnot     qa1_5,qa1_3
        cnot     qa1_6,qa1_0
        w1       qa1_2
#V ACTING ON [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']] ['v0_1_0'] ['v1_1_0'] ['v2_1_0']
#----------------------------------------------
# V verification network, noiseType = NFT
# acting on a 1-block of M_1
#G ACTING ON ['v0_1_0']
#----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,v0_1_0
        if       temp0
        x        v0_1_0
        noise    on
#G ACTING ON ['v1_1_0']
#----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,v1_1_0
        if       temp0
        x        v1_1_0
        noise    on
#G ACTING ON ['v2_1_0']
#----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,v2_1_0
        if       temp0
        x        v2_1_0
        noise    on
#G ACTING ON ['v3_1_0']
#----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,v3_1_0
        if       temp0
        x        v3_1_0
        noise    on
        id       v0_1_0
        id       v1_1_0
        id       v2_1_0
        id       v3_1_0
        h        v0_1_0
        h        v1_1_0
        h        v2_1_0
```

```
        h          v3_1_0
        cz         v0_1_0,qa1_0
        cz         v1_1_0,qa1_1
        cz         v2_1_0,qa1_2
        cz         v3_1_0,qa1_3
        wl         qa1_4
        wl         qa1_5
        wl         qa1_6
        cz         v0_1_0,qa1_5
        cz         v1_1_0,qa1_4
        wl         v2_1_0
        cz         v3_1_0,qa1_6
        wl         qa1_0
        wl         qa1_1
        wl         qa1_2
        wl         qa1_3
        wl         v0_1_0
        cz         v1_1_0,qa1_6
        cz         v2_1_0,qa1_4
        cz         v3_1_0,qa1_5
        wl         qa1_0
        wl         qa1_1
        wl         qa1_2
        wl         qa1_3
        cz         v0_1_0,qa1_6
        wl         v1_1_0
        cz         v2_1_0,qa1_5
        cz         v3_1_0,qa1_4
        wl         qa1_0
        wl         qa1_1
        wl         qa1_2
        wl         qa1_3
        h          v0_1_0
        h          v1_1_0
        h          v2_1_0
        h          v3_1_0
        wl         qa1_0
        wl         qa1_1
        wl         qa1_2
        wl         qa1_3
        wl         qa1_4
        wl         qa1_5
        wl         qa1_6
        measure temp0,v0_1_0
        measure temp1,v1_1_0
        measure temp2,v2_1_0
        measure temp3,v3_1_0
        wl         qa1_0
        wl         qa1_1
        wl         qa1_2
        wl         qa1_3
        wl         qa1_4
        wl         qa1_5
        wl         qa1_6
        set  v,temp0
        or  v,v,temp1
        or  v,v,temp2
        or  v,v,temp3
        if         v
        jump       prepare_until_0
#S ACTING ON [['qd_0_0'], ['qd_0_1'], ['qd_0_2
    '], ['qd_0_3'], ['qd_0_4'], ['qd_0_5'], ['
    qd_0_6']] [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']]
#---------------------------------------------
# S syndrome extraction network
# acting on a 1-block of M_1
        cnot       qa1_0,qd_0_0
        cnot       qa1_1,qd_0_1
        cnot       qa1_2,qd_0_2
        cnot       qa1_3,qd_0_3
        cnot       qa1_4,qd_0_4
        cnot       qa1_5,qd_0_5
        cnot       qa1_6,qd_0_6
        h          qa1_0
        wl         qd_0_0
        h          qa1_1
        wl         qd_0_1
        h          qa1_2
        wl         qd_0_2
        h          qa1_3
        wl         qd_0_3
        h          qa1_4
        wl         qd_0_4
        h          qa1_5
        wl         qd_0_5
        h          qa1_6
        wl         qd_0_6
        measure se0,qa1_0
        wl         qd_0_0
        measure se1,qa1_1
        wl         qd_0_1
        measure se2,qa1_2
        wl         qd_0_2
        measure se3,qa1_3
        wl         qd_0_3
        measure se4,qa1_4
        wl         qd_0_4
        measure se5,qa1_5
        wl         qd_0_5
        measure se6,qa1_6
        wl         qd_0_6
        set        zs00,se0
        xor        zs00,zs00,se2
        xor        zs00,zs00,se4
        xor        zs00,zs00,se6
        set        zs10,se1
        xor        zs10,zs10,se2
        xor        zs10,zs10,se5
        xor        zs10,zs10,se6
        set        zs20,se3
        xor        zs20,zs20,se4
        xor        zs20,zs20,se5
        xor        zs20,zs20,se6
        xor        temp0,1,zs00
        and        temp1,temp0,1
        xor        temp0,1,zs10
        and        temp1,temp1,temp0
        xor        temp0,1,zs20
        and        temp1,temp1,temp0
        if         temp1
```

```
        jump     no_cc_needed_0
        label    prepare_until_1
#PREPARE UNTIL PASS ['qa1_0', 'qa1_1', 'qa1_2
     ', 'qa1_3', 'qa1_4', 'qa1_5', 'qa1_6'] ['
     v0_1_0'] ['v1_1_0'] ['v2_1_0'] ['v3_1_0']
#G ACTING ON [['qa1_0'], ['qa1_1'], ['qa1_2
     '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
     qa1_6']]
#-----------------------------------------------
# G preparation network, noiseType = NFT
# acting on a 1-block of M_1
#G ACTING ON ['qa1_0']
#-----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_0
        if       temp0
        x        qa1_0
        noise    on
        id       qa1_0
#G ACTING ON ['qa1_1']
#-----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_1
        if       temp0
        x        qa1_1
        noise    on
        id       qa1_1
#G ACTING ON ['qa1_2']
#-----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_2
        if       temp0
        x        qa1_2
        noise    on
        id       qa1_2
#G ACTING ON ['qa1_3']
#-----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_3
        if       temp0
        x        qa1_3
        noise    on
        id       qa1_3
#G ACTING ON ['qa1_4']
#-----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_4
        if       temp0
        x        qa1_4
        noise    on
        id       qa1_4
#G ACTING ON ['qa1_5']
#-----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_5
        if       temp0
        x        qa1_5
        noise    on
        id       qa1_5
#G ACTING ON ['qa1_6']
#-----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_6
        if       temp0
        x        qa1_6
        noise    on
        id       qa1_6
        h        qa1_6
        h        qa1_5
        h        qa1_4
        w1       qa1_3
        w1       qa1_2
        w1       qa1_1
        w1       qa1_0
        cnot     qa1_4,qa1_3
        cnot     qa1_5,qa1_2
        cnot     qa1_6,qa1_1
        w1       qa1_0
        cnot     qa1_4,qa1_2
        cnot     qa1_5,qa1_0
        cnot     qa1_6,qa1_3
        w1       qa1_1
        cnot     qa1_4,qa1_1
        cnot     qa1_5,qa1_3
        cnot     qa1_6,qa1_0
        w1       qa1_2
#V ACTING ON [['qa1_0'], ['qa1_1'], ['qa1_2
     '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
     qa1_6']] ['v0_1_0'] ['v1_1_0'] ['v2_1_0']
#-----------------------------------------------
# V verification network, noiseType = NFT
# acting on a 1-block of M_1
#G ACTING ON ['v0_1_0']
#-----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,v0_1_0
        if       temp0
        x        v0_1_0
        noise    on
#G ACTING ON ['v1_1_0']
#-----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,v1_1_0
        if       temp0
```

```
        x       v1_1_0
        noise   on
#G ACTING ON ['v2_1_0']
#------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,v2_1_0
        if      temp0
        x       v2_1_0
        noise   on
#G ACTING ON ['v3_1_0']
#------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,v3_1_0
        if      temp0
        x       v3_1_0
        noise   on
        id      v0_1_0
        id      v1_1_0
        id      v2_1_0
        id      v3_1_0
        h       v0_1_0
        h       v1_1_0
        h       v2_1_0
        h       v3_1_0
        cz      v0_1_0,qa1_0
        cz      v1_1_0,qa1_1
        cz      v2_1_0,qa1_2
        cz      v3_1_0,qa1_3
        w1      qa1_4
        w1      qa1_5
        w1      qa1_6
        cz      v0_1_0,qa1_5
        cz      v1_1_0,qa1_4
        w1      v2_1_0
        cz      v3_1_0,qa1_6
        w1      qa1_0
        w1      qa1_1
        w1      qa1_2
        w1      qa1_3
        w1      v0_1_0
        cz      v1_1_0,qa1_6
        cz      v2_1_0,qa1_4
        cz      v3_1_0,qa1_5
        w1      qa1_0
        w1      qa1_1
        w1      qa1_2
        w1      qa1_3
        cz      v0_1_0,qa1_6
        w1      v1_1_0
        cz      v2_1_0,qa1_5
        cz      v3_1_0,qa1_4
        w1      qa1_0
        w1      qa1_1
        w1      qa1_2
        w1      qa1_3
        h       v0_1_0
        h       v1_1_0
        h       v2_1_0
        h       v3_1_0
        w1      qa1_0
        w1      qa1_1
        w1      qa1_2
        w1      qa1_3
        w1      qa1_4
        w1      qa1_5
        w1      qa1_6
        measure temp0,v0_1_0
        measure temp1,v1_1_0
        measure temp2,v2_1_0
        measure temp3,v3_1_0
        w1      qa1_0
        w1      qa1_1
        w1      qa1_2
        w1      qa1_3
        w1      qa1_4
        w1      qa1_5
        w1      qa1_6
        set  v,temp0
        or   v,v,temp1
        or   v,v,temp2
        or   v,v,temp3
        if      v
        jump    prepare_until_1
        w1      qa1_0
        w1      qa1_1
        w1      qa1_2
        w1      qa1_3
        w1      qa1_4
        w1      qa1_5
        w1      qa1_6
        w1      qa1_0
        w1      qa1_1
        w1      qa1_2
        w1      qa1_3
        w1      qa1_4
        w1      qa1_5
        w1      qa1_6
        w1      qa1_0
        w1      qa1_1
        w1      qa1_2
        w1      qa1_3
        w1      qa1_4
        w1      qa1_5
        w1      qa1_6
#S ACTING ON [['qd_0_0'], ['qd_0_1'], ['qd_0_2
    '], ['qd_0_3'], ['qd_0_4'], ['qd_0_5'], ['
    qd_0_6']] [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']]
#------------------------------------------
# S syndrome extraction network
# acting on a 1-block of M_1
        cnot    qa1_0,qd_0_0
        cnot    qa1_1,qd_0_1
        cnot    qa1_2,qd_0_2
        cnot    qa1_3,qd_0_3
        cnot    qa1_4,qd_0_4
        cnot    qa1_5,qd_0_5
```

```
        cnot      qa1_6,qd_0_6
        h         qa1_0
        w1        qd_0_0
        h         qa1_1
        w1        qd_0_1
        h         qa1_2
        w1        qd_0_2
        h         qa1_3
        w1        qd_0_3
        h         qa1_4
        w1        qd_0_4
        h         qa1_5
        w1        qd_0_5
        h         qa1_6
        w1        qd_0_6
        measure   se0,qa1_0
        w1        qd_0_0
        measure   se1,qa1_1
        w1        qd_0_1
        measure   se2,qa1_2
        w1        qd_0_2
        measure   se3,qa1_3
        w1        qd_0_3
        measure   se4,qa1_4
        w1        qd_0_4
        measure   se5,qa1_5
        w1        qd_0_5
        measure   se6,qa1_6
        w1        qd_0_6
        set       zs01,se0
        xor       zs01,zs01,se2
        xor       zs01,zs01,se4
        xor       zs01,zs01,se6
        set       zs11,se1
        xor       zs11,zs11,se2
        xor       zs11,zs11,se5
        xor       zs11,zs11,se6
        set       zs21,se3
        xor       zs21,zs21,se4
        xor       zs21,zs21,se5
        xor       zs21,zs21,se6
        label     prepare_until_2
#PREPARE UNTIL PASS ['qa1_0', 'qa1_1', 'qa1_2
    ', 'qa1_3', 'qa1_4', 'qa1_5', 'qa1_6'] ['
    v0_1_0'] ['v1_1_0'] ['v2_1_0'] ['v3_1_0']
#G ACTING ON [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']]
#------------------------------------------
# G preparation network, noiseType = NFT
# acting on a 1-block of M_1
#G ACTING ON ['qa1_0']
#------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise     off
        measure   temp0,qa1_0
        if        temp0
        x         qa1_0
        noise     on
        id        qa1_0
#G ACTING ON ['qa1_1']
#------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise     off
        measure   temp0,qa1_1
        if        temp0
        x         qa1_1
        noise     on
        id        qa1_1
#G ACTING ON ['qa1_2']
#------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise     off
        measure   temp0,qa1_2
        if        temp0
        x         qa1_2
        noise     on
        id        qa1_2
#G ACTING ON ['qa1_3']
#------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise     off
        measure   temp0,qa1_3
        if        temp0
        x         qa1_3
        noise     on
        id        qa1_3
#G ACTING ON ['qa1_4']
#------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise     off
        measure   temp0,qa1_4
        if        temp0
        x         qa1_4
        noise     on
        id        qa1_4
#G ACTING ON ['qa1_5']
#------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise     off
        measure   temp0,qa1_5
        if        temp0
        x         qa1_5
        noise     on
        id        qa1_5
#G ACTING ON ['qa1_6']
#------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise     off
        measure   temp0,qa1_6
        if        temp0
        x         qa1_6
        noise     on
        id        qa1_6
        h         qa1_6
```

```
        h         qa1_5
        h         qa1_4
        w1        qa1_3
        w1        qa1_2
        w1        qa1_1
        w1        qa1_0
        cnot      qa1_4,qa1_3
        cnot      qa1_5,qa1_2
        cnot      qa1_6,qa1_1
        w1        qa1_0
        cnot      qa1_4,qa1_2
        cnot      qa1_5,qa1_0
        cnot      qa1_6,qa1_3
        w1        qa1_1
        cnot      qa1_4,qa1_1
        cnot      qa1_5,qa1_3
        cnot      qa1_6,qa1_0
        w1        qa1_2
#V ACTING ON [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']] ['v0_1_0'] ['v1_1_0'] ['v2_1_0']
#--------------------------------------------------
# V verification network, noiseType = NFT
# acting on a 1-block of M_1
#G ACTING ON ['v0_1_0']
#--------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise     off
        measure temp0,v0_1_0
        if        temp0
        x         v0_1_0
        noise     on
#G ACTING ON ['v1_1_0']
#--------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise     off
        measure temp0,v1_1_0
        if        temp0
        x         v1_1_0
        noise     on
#G ACTING ON ['v2_1_0']
#--------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise     off
        measure temp0,v2_1_0
        if        temp0
        x         v2_1_0
        noise     on
#G ACTING ON ['v3_1_0']
#--------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise     off
        measure temp0,v3_1_0
        if        temp0
        x         v3_1_0
        noise     on
        id        v0_1_0
        id        v1_1_0
        id        v2_1_0
        id        v3_1_0
        h         v0_1_0
        h         v1_1_0
        h         v2_1_0
        h         v3_1_0
        cz        v0_1_0,qa1_0
        cz        v1_1_0,qa1_1
        cz        v2_1_0,qa1_2
        cz        v3_1_0,qa1_3
        w1        qa1_4
        w1        qa1_5
        w1        qa1_6
        cz        v0_1_0,qa1_5
        cz        v1_1_0,qa1_4
        w1        v2_1_0
        cz        v3_1_0,qa1_6
        w1        qa1_0
        w1        qa1_1
        w1        qa1_2
        w1        qa1_3
        w1        v0_1_0
        cz        v1_1_0,qa1_6
        cz        v2_1_0,qa1_4
        cz        v3_1_0,qa1_5
        w1        qa1_0
        w1        qa1_1
        w1        qa1_2
        w1        qa1_3
        cz        v0_1_0,qa1_6
        w1        v1_1_0
        cz        v2_1_0,qa1_5
        cz        v3_1_0,qa1_4
        w1        qa1_0
        w1        qa1_1
        w1        qa1_2
        w1        qa1_3
        h         v0_1_0
        h         v1_1_0
        h         v2_1_0
        h         v3_1_0
        w1        qa1_0
        w1        qa1_1
        w1        qa1_2
        w1        qa1_3
        w1        qa1_4
        w1        qa1_5
        w1        qa1_6
        measure temp0,v0_1_0
        measure temp1,v1_1_0
        measure temp2,v2_1_0
        measure temp3,v3_1_0
        w1        qa1_0
        w1        qa1_1
        w1        qa1_2
        w1        qa1_3
        w1        qa1_4
        w1        qa1_5
        w1        qa1_6
        set  v,temp0
```

```
        or v,v,temp1
        or v,v,temp2
        or v,v,temp3
        if      v
        jump    prepare_until_2
        wl      qa1_0
        wl      qa1_1
        wl      qa1_2
        wl      qa1_3
        wl      qa1_4
        wl      qa1_5
        wl      qa1_6
        wl      qa1_0
        wl      qa1_1
        wl      qa1_2
        wl      qa1_3
        wl      qa1_4
        wl      qa1_5
        wl      qa1_6
        wl      qa1_0
        wl      qa1_1
        wl      qa1_2
        wl      qa1_3
        wl      qa1_4
        wl      qa1_5
        wl      qa1_6
        wl      qa1_0
        wl      qa1_1
        wl      qa1_2
        wl      qa1_3
        wl      qa1_4
        wl      qa1_5
        wl      qa1_6
        wl      qa1_0
        wl      qa1_1
        wl      qa1_2
        wl      qa1_3
        wl      qa1_4
        wl      qa1_5
        wl      qa1_6
        wl      qa1_0
        wl      qa1_1
        wl      qa1_2
        wl      qa1_3
        wl      qa1_4
        wl      qa1_5
        wl      qa1_6
#S ACTING ON [['qd_0_0'], ['qd_0_1'], ['qd_0_2
    '], ['qd_0_3'], ['qd_0_4'], ['qd_0_5'], ['
    qd_0_6']] [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']]
#-------------------------------------------
# S syndrome extraction network
# acting on a 1-block of M_1
        cnot    qa1_0,qd_0_0
        cnot    qa1_1,qd_0_1
        cnot    qa1_2,qd_0_2
        cnot    qa1_3,qd_0_3
        cnot    qa1_4,qd_0_4
        cnot    qa1_5,qd_0_5
        cnot    qa1_6,qd_0_6
        h       qa1_0
        wl      qd_0_0
        h       qa1_1
        wl      qd_0_1
        h       qa1_2
        wl      qd_0_2
        h       qa1_3
        wl      qd_0_3
        h       qa1_4
        wl      qd_0_4
        h       qa1_5
        wl      qd_0_5
        h       qa1_6
        wl      qd_0_6
        measure se0,qa1_0
        wl      qd_0_0
        measure se1,qa1_1
        wl      qd_0_1
        measure se2,qa1_2
        wl      qd_0_2
        measure se3,qa1_3
        wl      qd_0_3
        measure se4,qa1_4
        wl      qd_0_4
        measure se5,qa1_5
        wl      qd_0_5
        measure se6,qa1_6
        wl      qd_0_6
        set     zs02,se0
        xor     zs02,zs02,se2
        xor     zs02,zs02,se4
        xor     zs02,zs02,se6
        set     zs12,se1
        xor     zs12,zs12,se2
        xor     zs12,zs12,se5
        xor     zs12,zs12,se6
        set     zs22,se3
        xor     zs22,zs22,se4
        xor     zs22,zs22,se5
        xor     zs22,zs22,se6
#FIND BEST SYNDROME
#-------------------------------------------
        set     not_guessed_0,1
        set     not_guessed_1,1
        set     not_guessed_2,1
#GUESSING SYNDROME 0
#-------------------------------------------
        label   guess_0_0
        set     guess_s0,zs00
        set     guess_s1,zs10
        set     guess_s2,zs20
        set     not_guessed_0,0
        set     number_of_matches_1,1
        set     number_of_matches_2,0
        jump    compare_with_all_syndromes_0
#GUESSING SYNDROME 1
#-------------------------------------------
        label   guess_1_0
        set     guess_s0,zs01
        set     guess_s1,zs11
```

```
        set     guess_s2,zs21
        set     not_guessed_1,0
        set     number_of_matches_1,1
        set     number_of_matches_2,0
        jump    compare_with_all_syndromes_0
#COMPARE GUESS WITH ALL UNGUESSED SYNDROMES
#----------------------------------------------
        label   compare_with_all_syndromes_0
        if      not_guessed_1
        jump    compare_to_1_0
        label   compared_to_1_0
        if      not_guessed_2
        jump    compare_to_2_0
        label   compared_to_2_0
        if      not_guessed_1
        jump    guess_1_0
        jump    error_corrected_0
#COMPARE GUESS TO SYNDROME 1
        label   compare_to_1_0
        set     match,1
        xor     match_temp,guess_s0,zs01
        xor     match_temp,match_temp,1
        and     match,match,match_temp
        xor     match_temp,guess_s1,zs11
        xor     match_temp,match_temp,1
        and     match,match,match_temp
        xor     match_temp,guess_s2,zs21
        xor     match_temp,match_temp,1
        and     match,match,match_temp
        xor     match_temp,match,1
        if      match_temp
        jump    compared_to_1_0
        set     not_guessed_1,0
        and     match_temp,number_of_matches_1,1
        set     number_of_matches_2,match_temp
        and     match_temp,number_of_matches_0,1
        set     number_of_matches_1,match_temp
        if      number_of_matches_2
        jump    found_best_syndrome_0
        jump    compared_to_1_0
#COMPARE GUESS TO SYNDROME 2
        label   compare_to_2_0
        set     match,1
        xor     match_temp,guess_s0,zs02
        xor     match_temp,match_temp,1
        and     match,match,match_temp
        xor     match_temp,guess_s1,zs12
        xor     match_temp,match_temp,1
        and     match,match,match_temp
        xor     match_temp,guess_s2,zs22
        xor     match_temp,match_temp,1
        and     match,match,match_temp
        xor     match_temp,match,1
        if      match_temp
        jump    compared_to_2_0
        set     not_guessed_2,0
        and     match_temp,number_of_matches_1,1
        set     number_of_matches_2,match_temp
        and     match_temp,number_of_matches_0,1
        set     number_of_matches_1,match_temp
        if      number_of_matches_2
        jump    found_best_syndrome_0
        jump    compared_to_2_0
        label   found_best_syndrome_0
#ERROR CORRECTING Z
#----------------------------------------------
        if      guess_s2
        jump    correct_1xx_0
        if      guess_s1
        jump    correct_01x_0
        if      guess_s0
        jump    correct_001_0
        jump    error_corrected_0
        label   correct_1xx_0
        if      guess_s1
        jump    correct_11x_0
        if      guess_s0
        jump    correct_101_0
        z       qd_0_3
        w1      qd_0_0
        w1      qd_0_1
        w1      qd_0_2
        w1      qd_0_4
        w1      qd_0_5
        w1      qd_0_6
        jump    error_corrected_0
        label   correct_01x_0
        if      guess_s0
        jump    correct_011_0
        z       qd_0_1
        w1      qd_0_0
        w1      qd_0_2
        w1      qd_0_3
        w1      qd_0_4
        w1      qd_0_5
        w1      qd_0_6
        jump    error_corrected_0
        label   correct_001_0
        z       qd_0_0
        w1      qd_0_1
        w1      qd_0_2
        w1      qd_0_3
        w1      qd_0_4
        w1      qd_0_5
        w1      qd_0_6
        jump    error_corrected_0
        label   correct_11x_0
        if      guess_s0
        jump    correct_111_0
        z       qd_0_5
        w1      qd_0_0
        w1      qd_0_1
        w1      qd_0_2
        w1      qd_0_3
        w1      qd_0_4
        w1      qd_0_6
        jump    error_corrected_0
        label   correct_101_0
        z       qd_0_4
        w1      qd_0_0
        w1      qd_0_1
        w1      qd_0_2
```

```
        wl      qd_0_3
        wl      qd_0_5
        wl      qd_0_6
        jump    error_corrected_0
        label   correct_011_0
        z       qd_0_2
        wl      qd_0_0
        wl      qd_0_1
        wl      qd_0_3
        wl      qd_0_4
        wl      qd_0_5
        wl      qd_0_6
        jump    error_corrected_0
        label   correct_111_0
        z       qd_0_6
        wl      qd_0_0
        wl      qd_0_1
        wl      qd_0_2
        wl      qd_0_3
        wl      qd_0_4
        wl      qd_0_5
        jump    error_corrected_0
        label   no_ec_needed_0
        wl      qd_0_0
        wl      qd_0_1
        wl      qd_0_2
        wl      qd_0_3
        wl      qd_0_4
        wl      qd_0_5
        wl      qd_0_6
        wl      qd_0_0
        wl      qd_0_1
        wl      qd_0_2
        wl      qd_0_3
        wl      qd_0_4
        wl      qd_0_5
        wl      qd_0_6
        label   error_corrected_0
        label   prepare_until_3
#PREPARE UNTIL PASS ['qa1_0', 'qa1_1', 'qa1_2
    ', 'qa1_3', 'qa1_4', 'qa1_5', 'qa1_6'] ['
    v0_1_0'] ['v1_1_0'] ['v2_1_0'] ['v3_1_0']
#G ACTING ON [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']]
#------------------------------------------------
# G preparation network, noiseType = NFT
# acting on a 1-block of M_1
#G ACTING ON ['qa1_0']
#------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,qa1_0
        if      temp0
        x       qa1_0
        noise   on
        id      qa1_0
#G ACTING ON ['qa1_1']
#------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,qa1_1
        if      temp0
        x       qa1_1
        noise   on
        id      qa1_1
#G ACTING ON ['qa1_2']
#------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,qa1_2
        if      temp0
        x       qa1_2
        noise   on
        id      qa1_2
#G ACTING ON ['qa1_3']
#------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,qa1_3
        if      temp0
        x       qa1_3
        noise   on
        id      qa1_3
#G ACTING ON ['qa1_4']
#------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,qa1_4
        if      temp0
        x       qa1_4
        noise   on
        id      qa1_4
#G ACTING ON ['qa1_5']
#------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,qa1_5
        if      temp0
        x       qa1_5
        noise   on
        id      qa1_5
#G ACTING ON ['qa1_6']
#------------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,qa1_6
        if      temp0
        x       qa1_6
        noise   on
        id      qa1_6
        h       qa1_6
        h       qa1_5
        h       qa1_4
        wl      qa1_3
```

```
        wl      qa1_2
        wl      qa1_1
        wl      qa1_0
        cnot    qa1_4,qa1_3
        cnot    qa1_5,qa1_2
        cnot    qa1_6,qa1_1
        wl      qa1_0
        cnot    qa1_4,qa1_2
        cnot    qa1_5,qa1_0
        cnot    qa1_6,qa1_3
        wl      qa1_1
        cnot    qa1_4,qa1_1
        cnot    qa1_5,qa1_3
        cnot    qa1_6,qa1_0
        wl      qa1_2
#V ACTING ON [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']] ['v0_1_0'] ['v1_1_0'] ['v2_1_0']
#--------------------------------------------
# V verification network, noiseType = NFT
# acting on a 1-block of M_1
#G ACTING ON ['v0_1_0']
#--------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,v0_1_0
        if      temp0
        x       v0_1_0
        noise   on
#G ACTING ON ['v1_1_0']
#--------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,v1_1_0
        if      temp0
        x       v1_1_0
        noise   on
#G ACTING ON ['v2_1_0']
#--------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0.v2_1_0
        if      temp0
        x       v2_1_0
        noise   on
#G ACTING ON ['v3_1_0']
#--------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,v3_1_0
        if      temp0
        x       v3_1_0
        noise   on
        id      v0_1_0
        id      v1_1_0
        id      v2_1_0
        id      v3_1_0
        h       v0_1_0
        h       v1_1_0
        h       v2_1_0
        h       v3_1_0
        cz      v0_1_0,qa1_0
        cz      v1_1_0,qa1_1
        cz      v2_1_0,qa1_2
        cz      v3_1_0,qa1_3
        wl      qa1_4
        wl      qa1_5
        wl      qa1_6
        cz      v0_1_0,qa1_5
        cz      v1_1_0,qa1_4
        wl      v2_1_0
        cz      v3_1_0,qa1_6
        wl      qa1_0
        wl      qa1_1
        wl      qa1_2
        wl      qa1_3
        wl      v0_1_0
        cz      v1_1_0,qa1_6
        cz      v2_1_0,qa1_4
        cz      v3_1_0,qa1_5
        wl      qa1_0
        wl      qa1_1
        wl      qa1_2
        wl      qa1_3
        cz      v0_1_0,qa1_6
        wl      v1_1_0
        cz      v2_1_0,qa1_5
        cz      v3_1_0,qa1_4
        wl      qa1_0
        wl      qa1_1
        wl      qa1_2
        wl      qa1_3
        h       v0_1_0
        h       v1_1_0
        h       v2_1_0
        h       v3_1_0
        wl      qa1_0
        wl      qa1_1
        wl      qa1_2
        wl      qa1_3
        wl      qa1_4
        wl      qa1_5
        wl      qa1_6
        measure temp0,v0_1_0
        measure temp1,v1_1_0
        measure temp2,v2_1_0
        measure temp3,v3_1_0
        wl      qa1_0
        wl      qa1_1
        wl      qa1_2
        wl      qa1_3
        wl      qa1_4
        wl      qa1_5
        wl      qa1_6
        set  v,temp0
        or  v,v,temp1
        or  v,v,temp2
        or  v,v,temp3
```

```
        if      v
        jump    prepare_until_3
#S ACTING ON [['qd_0_0'], ['qd_0_1'], ['qd_0_2
    '], ['qd_0_3'], ['qd_0_4'], ['qd_0_5'], ['
    qd_0_6']] [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']]
#------------------------------------------
# S syndrome extraction network
# acting on a 1-block of M_1
        cz      qa1_0,qd_0_0
        cz      qa1_1,qd_0_1
        cz      qa1_2,qd_0_2
        cz      qa1_3,qd_0_3
        cz      qa1_4,qd_0_4
        cz      qa1_5,qd_0_5
        cz      qa1_6,qd_0_6
        h       qa1_0
        w1      qd_0_0
        h       qa1_1
        w1      qd_0_1
        h       qa1_2
        w1      qd_0_2
        h       qa1_3
        w1      qd_0_3
        h       qa1_4
        w1      qd_0_4
        h       qa1_5
        w1      qd_0_5
        h       qa1_6
        w1      qd_0_6
        measure se0,qa1_0
        w1      qd_0_0
        measure se1,qa1_1
        w1      qd_0_1
        measure se2,qa1_2
        w1      qd_0_2
        measure se3,qa1_3
        w1      qd_0_3
        measure se4,qa1_4
        w1      qd_0_4
        measure se5,qa1_5
        w1      qd_0_5
        measure se6,qa1_6
        w1      qd_0_6
        set     xs00,se0
        xor     xs00,xs00,se2
        xor     xs00,xs00,se4
        xor     xs00,xs00,se6
        set     xs10,se1
        xor     xs10,xs10,se2
        xor     xs10,xs10,se5
        xor     xs10,xs10,se6
        set     xs20,se3
        xor     xs20,xs20,se4
        xor     xs20,xs20,se5
        xor     xs20,xs20,se6
        xor     temp0,1,xs00
        and     temp1,temp0,1
        xor     temp0,1,xs10
        and     temp1,temp1,temp0
        xor     temp0,1,xs20
        and     temp1,temp1,temp0
        if      temp1
        jump    no_ec_needed_1
        label   prepare_until_4
#PREPARE UNTIL PASS ['qa1_0', 'qa1_1', 'qa1_2
    ', 'qa1_3', 'qa1_4', 'qa1_5', 'qa1_6'] ['
    v0_1_0'] ['v1_1_0'] ['v2_1_0'] ['v3_1_0']
#G ACTING ON [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']]
#------------------------------------------
# G preparation network, noiseType = NFT
# acting on a 1-block of M_1
#G ACTING ON ['qa1_0']
#------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,qa1_0
        if      temp0
        x       qa1_0
        noise   on
        id      qa1_0
#G ACTING ON ['qa1_1']
#------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,qa1_1
        if      temp0
        x       qa1_1
        noise   on
        id      qa1_1
#G ACTING ON ['qa1_2']
#------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,qa1_2
        if      temp0
        x       qa1_2
        noise   on
        id      qa1_2
#G ACTING ON ['qa1_3']
#------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,qa1_3
        if      temp0
        x       qa1_3
        noise   on
        id      qa1_3
#G ACTING ON ['qa1_4']
#------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,qa1_4
        if      temp0
```

```
        x        qa1_4
        noise    on
        id       qa1_4
#G ACTING ON ['qa1_5']
#-----------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_5
        if       temp0
        x        qa1_5
        noise    on
        id       qa1_5
#G ACTING ON ['qa1_6']
#-----------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_6
        if       temp0
        x        qa1_6
        noise    on
        id       qa1_6
        h        qa1_6
        h        qa1_5
        h        qa1_4
        w1       qa1_3
        w1       qa1_2
        w1       qa1_1
        w1       qa1_0
        cnot     qa1_4,qa1_3
        cnot     qa1_5,qa1_2
        cnot     qa1_6,qa1_1
        w1       qa1_0
        cnot     qa1_4,qa1_2
        cnot     qa1_5,qa1_0
        cnot     qa1_6,qa1_3
        w1       qa1_1
        cnot     qa1_4,qa1_1
        cnot     qa1_5,qa1_3
        cnot     qa1_6,qa1_0
        w1       qa1_2
#V ACTING ON [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']] ['v0_1_0'] ['v1_1_0'] ['v2_1_0']
#-----------------------------------------
# V verification network, noiseType = NFT
# acting on a 1-block of M_1
#G ACTING ON ['v0_1_0']
#-----------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,v0_1_0
        if       temp0
        x        v0_1_0
        noise    on
#G ACTING ON ['v1_1_0']
#-----------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,v1_1_0
        if       temp0
        x        v1_1_0
        noise    on
#G ACTING ON ['v2_1_0']
#-----------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,v2_1_0
        if       temp0
        x        v2_1_0
        noise    on
#G ACTING ON ['v3_1_0']
#-----------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,v3_1_0
        if       temp0
        x        v3_1_0
        noise    on
        id       v0_1_0
        id       v1_1_0
        id       v2_1_0
        id       v3_1_0
        h        v0_1_0
        h        v1_1_0
        h        v2_1_0
        h        v3_1_0
        cz       v0_1_0,qa1_0
        cz       v1_1_0,qa1_1
        cz       v2_1_0,qa1_2
        cz       v3_1_0,qa1_3
        w1       qa1_4
        w1       qa1_5
        w1       qa1_6
        cz       v0_1_0,qa1_5
        cz       v1_1_0,qa1_4
        w1       v2_1_0
        cz       v3_1_0,qa1_6
        w1       qa1_0
        w1       qa1_1
        w1       qa1_2
        w1       qa1_3
        w1       v0_1_0
        cz       v1_1_0,qa1_6
        cz       v2_1_0,qa1_4
        cz       v3_1_0,qa1_5
        w1       qa1_0
        w1       qa1_1
        w1       qa1_2
        w1       qa1_3
        cz       v0_1_0,qa1_6
        w1       v1_1_0
        cz       v2_1_0,qa1_5
        cz       v3_1_0,qa1_4
        w1       qa1_0
        w1       qa1_1
        w1       qa1_2
```

```
        w1      qa1_3
        h       v0_1_0
        h       v1_1_0
        h       v2_1_0
        h       v3_1_0
        w1      qa1_0
        w1      qa1_1
        w1      qa1_2
        w1      qa1_3
        w1      qa1_4
        w1      qa1_5
        w1      qa1_6
        measure temp0,v0_1_0
        measure temp1,v1_1_0
        measure temp2,v2_1_0
        measure temp3,v3_1_0
        w1      qa1_0
        w1      qa1_1
        w1      qa1_2
        w1      qa1_3
        w1      qa1_4
        w1      qa1_5
        w1      qa1_6
        set  v,temp0
        or  v,v,temp1
        or  v,v,temp2
        or  v,v,temp3
        if      v
        jump    prepare_until_4
        w1      qa1_0
        w1      qa1_1
        w1      qa1_2
        w1      qa1_3
        w1      qa1_4
        w1      qa1_5
        w1      qa1_6
        w1      qa1_0
        w1      qa1_1
        w1      qa1_2
        w1      qa1_3
        w1      qa1_4
        w1      qa1_5
        w1      qa1_6
        w1      qa1_0
        w1      qa1_1
        w1      qa1_2
        w1      qa1_3
        w1      qa1_4
        w1      qa1_5
        w1      qa1_6
#S ACTING ON [['qd_0_0'], ['qd_0_1'], ['qd_0_2
    '], ['qd_0_3'], ['qd_0_4'], ['qd_0_5'], ['
    qd_0_6']] [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']]
#----------------------------------------------
# S syndrome extraction network
# acting on a 1-block of M_1
        cz      qa1_0,qd_0_0
        cz      qa1_1,qd_0_1
        cz      qa1_2,qd_0_2
        cz      qa1_3,qd_0_3
        cz      qa1_4,qd_0_4
        cz      qa1_5,qd_0_5
        cz      qa1_6,qd_0_6
        h       qa1_0
        w1      qd_0_0
        h       qa1_1
        w1      qd_0_1
        h       qa1_2
        w1      qd_0_2
        h       qa1_3
        w1      qd_0_3
        h       qa1_4
        w1      qd_0_4
        h       qa1_5
        w1      qd_0_5
        h       qa1_6
        w1      qd_0_6
        measure se0,qa1_0
        w1      qd_0_0
        measure se1,qa1_1
        w1      qd_0_1
        measure se2,qa1_2
        w1      qd_0_2
        measure se3,qa1_3
        w1      qd_0_3
        measure se4,qa1_4
        w1      qd_0_4
        measure se5,qa1_5
        w1      qd_0_5
        measure se6,qa1_6
        w1      qd_0_6
        set     xs01,se0
        xor     xs01,xs01,se2
        xor     xs01,xs01,se4
        xor     xs01,xs01,se6
        set     xs11,se1
        xor     xs11,xs11,se2
        xor     xs11,xs11,se5
        xor     xs11,xs11,se6
        set     xs21,se3
        xor     xs21,xs21,se4
        xor     xs21,xs21,se5
        xor     xs21,xs21,se6
        label   prepare_until_5
#PREPARE UNTIL PASS ['qa1_0', 'qa1_1', 'qa1_2
    ', 'qa1_3', 'qa1_4', 'qa1_5', 'qa1_6'] ['
    v0_1_0'] ['v1_1_0'] ['v2_1_0'] ['v3_1_0']
#G ACTING ON [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']]
#----------------------------------------------
# G preparation network, noiseType = NFT
# acting on a 1-block of M_1
#G ACTING ON ['qa1_0']
#----------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise   off
        measure temp0,qa1_0
        if      temp0
```

```
        x        qa1_0
        noise    on
        id       qa1_0
#G ACTING ON ['qa1_1']
#---------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_1
        if       temp0
        x        qa1_1
        noise    on
        id       qa1_1
#G ACTING ON ['qa1_2']
#---------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_2
        if       temp0
        x        qa1_2
        noise    on
        id       qa1_2
#G ACTING ON ['qa1_3']
#---------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_3
        if       temp0
        x        qa1_3
        noise    on
        id       qa1_3
#G ACTING ON ['qa1_4']
#---------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_4
        if       temp0
        x        qa1_4
        noise    on
        id       qa1_4
#G ACTING ON ['qa1_5']
#---------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_5
        if       temp0
        x        qa1_5
        noise    on
        id       qa1_5
#G ACTING ON ['qa1_6']
#---------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,qa1_6
        if       temp0
        x        qa1_6
```

```
        noise    on
        id       qa1_6
        h        qa1_6
        h        qa1_5
        h        qa1_4
        w1       qa1_3
        w1       qa1_2
        w1       qa1_1
        w1       qa1_0
        cnot     qa1_4,qa1_3
        cnot     qa1_5,qa1_2
        cnot     qa1_6,qa1_1
        w1       qa1_0
        cnot     qa1_4,qa1_2
        cnot     qa1_5,qa1_0
        cnot     qa1_6,qa1_3
        w1       qa1_1
        cnot     qa1_4,qa1_1
        cnot     qa1_5,qa1_3
        cnot     qa1_6,qa1_0
        w1       qa1_2
#V ACTING ON [['qa1_0'], ['qa1_1'], ['qa1_2
    ']. ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']] ['v0_1_0'] ['v1_1_0'] ['v2_1_0']
#---------------------------------------------
# V verification network, noiseType = NFT
# acting on a 1-block of M_1
#G ACTING ON ['v0_1_0']
#---------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0,v0_1_0
        if       temp0
        x        v0_1_0
        noise    on
#G ACTING ON ['v1_1_0']
#---------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0.v1_1_0
        if       temp0
        x        v1_1_0
        noise    on
#G ACTING ON ['v2_1_0']
#---------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0.v2_1_0
        if       temp0
        x        v2_1_0
        noise    on
#G ACTING ON ['v3_1_0']
#---------------------------------------------
# G preparation network, noiseType = none
# acting on a 0-block of M_1
        noise    off
        measure  temp0.v3_1_0
        if       temp0
```

```
x        v3_1_0
noise    on
id       v0_1_0
id       v1_1_0
id       v2_1_0
id       v3_1_0
h        v0_1_0
h        v1_1_0
h        v2_1_0
h        v3_1_0
cz       v0_1_0,qa1_0
cz       v1_1_0,qa1_1
cz       v2_1_0,qa1_2
cz       v3_1_0,qa1_3
wl       qa1_4
wl       qa1_5
wl       qa1_6
cz       v0_1_0,qa1_5
cz       v1_1_0,qa1_4
wl       v2_1_0
cz       v3_1_0,qa1_6
wl       qa1_0
wl       qa1_1
wl       qa1_2
wl       qa1_3
wl       v0_1_0
cz       v1_1_0,qa1_6
cz       v2_1_0,qa1_4
cz       v3_1_0,qa1_5
wl       qa1_0
wl       qa1_1
wl       qa1_2
wl       qa1_3
cz       v0_1_0,qa1_6
wl       v1_1_0
cz       v2_1_0,qa1_5
cz       v3_1_0,qa1_4
wl       qa1_0
wl       qa1_1
wl       qa1_2
wl       qa1_3
h        v0_1_0
h        v1_1_0
h        v2_1_0
h        v3_1_0
wl       qa1_0
wl       qa1_1
wl       qa1_2
wl       qa1_3
wl       qa1_4
wl       qa1_5
wl       qa1_6
measure  temp0,v0_1_0
measure  temp1,v1_1_0
measure  temp2,v2_1_0
measure  temp3,v3_1_0
wl       qa1_0
wl       qa1_1
wl       qa1_2
wl       qa1_3
wl       qa1_4
wl       qa1_5
wl       qa1_6
set  v,temp0
or  v,v,temp1
or  v,v,temp2
or  v,v,temp3
if       v
jump     prepare_until_5
wl       qa1_0
wl       qa1_1
wl       qa1_2
wl       qa1_3
wl       qa1_4
wl       qa1_5
wl       qa1_6
wl       qa1_0
wl       qa1_1
wl       qa1_2
wl       qa1_3
wl       qa1_4
wl       qa1_5
wl       qa1_6
wl       qa1_0
wl       qa1_1
wl       qa1_2
wl       qa1_3
wl       qa1_4
wl       qa1_5
wl       qa1_6
wl       qa1_0
wl       qa1_1
wl       qa1_2
wl       qa1_3
wl       qa1_4
wl       qa1_5
wl       qa1_6
wl       qa1_0
wl       qa1_1
wl       qa1_2
wl       qa1_3
wl       qa1_4
wl       qa1_5
wl       qa1_6
wl       qa1_0
wl       qa1_1
wl       qa1_2
wl       qa1_3
wl       qa1_4
wl       qa1_5
wl       qa1_6
#S ACTING ON [['qd_0_0'], ['qd_0_1'], ['qd_0_2
    '], ['qd_0_3'], ['qd_0_4'], ['qd_0_5'], ['
    qd_0_6']] [['qa1_0'], ['qa1_1'], ['qa1_2
    '], ['qa1_3'], ['qa1_4'], ['qa1_5'], ['
    qa1_6']]
#------------------------------------------------
# S syndrome extraction network
# acting on a 1-block of M_1
cz       qa1_0,qd_0_0
cz       qa1_1,qd_0_1
cz       qa1_2,qd_0_2
```

```
        cz        qa1_3,qd_0_3
        cz        qa1_4,qd_0_4
        cz        qa1_5,qd_0_5
        cz        qa1_6,qd_0_6
        h         qa1_0
        wl        qd_0_0
        h         qa1_1
        wl        qd_0_1
        h         qa1_2
        wl        qd_0_2
        h         qa1_3
        wl        qd_0_3
        h         qa1_4
        wl        qd_0_4
        h         qa1_5
        wl        qd_0_5
        h         qa1_6
        wl        qd_0_6
        measure   se0,qa1_0
        wl        qd_0_0
        measure   se1,qa1_1
        wl        qd_0_1
        measure   se2,qa1_2
        wl        qd_0_2
        measure   se3,qa1_3
        wl        qd_0_3
        measure   se4,qa1_4
        wl        qd_0_4
        measure   se5,qa1_5
        wl        qd_0_5
        measure   se6,qa1_6
        wl        qd_0_6
        set       xs02,se0
        xor       xs02,xs02,se2
        xor       xs02,xs02,se4
        xor       xs02,xs02,se6
        set       xs12,se1
        xor       xs12,xs12,se2
        xor       xs12,xs12,se5
        xor       xs12,xs12,se6
        set       xs22,se3
        xor       xs22,xs22,se4
        xor       xs22,xs22,se5
        xor       xs22,xs22,se6
#FIND BEST SYNDROME
#------------------------------------------
        set       not_guessed_0,1
        set       not_guessed_1,1
        set       not_guessed_2,1
#GUESSING SYNDROME 0
#------------------------------------------
        label     guess_0_1
        set       guess_s0,xs00
        set       guess_s1,xs10
        set       guess_s2,xs20
        set       not_guessed_0,0
        set       number_of_matches_1,1
        set       number_of_matches_2,0
        jump      compare_with_all_syndromes_1
#GUESSING SYNDROME 1
#------------------------------------------
        label     guess_1_1
        set       guess_s0,xs01
        set       guess_s1,xs11
        set       guess_s2,xs21
        set       not_guessed_1,0
        set       number_of_matches_1,1
        set       number_of_matches_2,0
        jump      compare_with_all_syndromes_1
#COMPARE GUESS WITH ALL UNGUESSED SYNDROMES
#------------------------------------------
        label     compare_with_all_syndromes_1
        if        not_guessed_1
        jump      compare_to_1_1
        label     compared_to_1_1
        if        not_guessed_2
        jump      compare_to_2_1
        label     compared_to_2_1
        if        not_guessed_1
        jump      guess_1_1
        jump      error_corrected_1
#COMPARE GUESS TO SYNDROME 1
        label     compare_to_1_1
        set       match,1
        xor       match_temp,guess_s0,xs01
        xor       match_temp,match_temp,1
        and       match,match,match_temp
        xor       match_temp,guess_s1,xs11
        xor       match_temp,match_temp,1
        and       match,match,match_temp
        xor       match_temp,guess_s2,xs21
        xor       match_temp,match_temp,1
        and       match,match,match_temp
        xor       match_temp,match,1
        if        match_temp
        jump      compared_to_1_1
        set       not_guessed_1,0
        and       match_temp,number_of_matches_1,1
        set       number_of_matches_2,match_temp
        and       match_temp,number_of_matches_0,1
        set       number_of_matches_1,match_temp
        if        number_of_matches_2
        jump      found_best_syndrome_1
        jump      compared_to_1_1
#COMPARE GUESS TO SYNDROME 2
        label     compare_to_2_1
        set       match,1
        xor       match_temp,guess_s0,xs02
        xor       match_temp,match_temp,1
        and       match,match,match_temp
        xor       match_temp,guess_s1,xs12
        xor       match_temp,match_temp,1
        and       match,match,match_temp
        xor       match_temp,guess_s2,xs22
        xor       match_temp,match_temp,1
        and       match,match,match_temp
        xor       match_temp,match,1
        if        match_temp
        jump      compared_to_2_1
        set       not_guessed_2,0
        and       match_temp,number_of_matches_1,1
        set       number_of_matches_2,match_temp
```

```
        and     match_temp, number_of_matches_0, 1
        set     number_of_matches_1, match_temp
        if      number_of_matches_2
        jump    found_best_syndrome_1
        jump    compared_to_2_1
        label   found_best_syndrome_1
#ERROR CORRECTING X
#------------------------------------------------
        if      guess_s2
        jump    correct_1xx_1
        if      guess_s1
        jump    correct_01x_1
        if      guess_s0
        jump    correct_001_1
        jump    error_corrected_1
        label   correct_1xx_1
        if      guess_s1
        jump    correct_11x_1
        if      guess_s0
        jump    correct_101_1
        x       qd_0_3
        w1      qd_0_0
        w1      qd_0_1
        w1      qd_0_2
        w1      qd_0_4
        w1      qd_0_5
        w1      qd_0_6
        jump    error_corrected_1
        label   correct_01x_1
        if      guess_s0
        jump    correct_011_1
        x       qd_0_1
        w1      qd_0_0
        w1      qd_0_2
        w1      qd_0_3
        w1      qd_0_4
        w1      qd_0_5
        w1      qd_0_6
        jump    error_corrected_1
        label   correct_001_1
        x       qd_0_0
        w1      qd_0_1
        w1      qd_0_2
        w1      qd_0_3
        w1      qd_0_4
        w1      qd_0_5
        w1      qd_0_6
        jump    error_corrected_1
        label   correct_11x_1
        if      guess_s0
        jump    correct_111_1
        x       qd_0_5
        w1      qd_0_0
        w1      qd_0_1
        w1      qd_0_2
        w1      qd_0_3
        w1      qd_0_4
        w1      qd_0_6
        jump    error_corrected_1
        label   correct_101_1
        x       qd_0_4
        w1      qd_0_0
        w1      qd_0_1
        w1      qd_0_2
        w1      qd_0_3
        w1      qd_0_5
        w1      qd_0_6
        jump    error_corrected_1
        label   correct_011_1
        x       qd_0_2
        w1      qd_0_0
        w1      qd_0_1
        w1      qd_0_3
        w1      qd_0_4
        w1      qd_0_5
        w1      qd_0_6
        jump    error_corrected_1
        label   correct_111_1
        x       qd_0_6
        w1      qd_0_0
        w1      qd_0_1
        w1      qd_0_2
        w1      qd_0_3
        w1      qd_0_4
        w1      qd_0_5
        jump    error_corrected_1
        label   no_ec_needed_1
        w1      qd_0_0
        w1      qd_0_1
        w1      qd_0_2
        w1      qd_0_3
        w1      qd_0_4
        w1      qd_0_5
        w1      qd_0_6
        w1      qd_0_0
        w1      qd_0_1
        w1      qd_0_2
        w1      qd_0_3
        w1      qd_0_4
        w1      qd_0_5
        w1      qd_0_6
        w1      qd_0_0
        w1      qd_0_1
        w1      qd_0_2
        w1      qd_0_3
        w1      qd_0_4
        w1      qd_0_5
        w1      qd_0_6
        w1      qd_0_0
        w1      qd_0_1
        w1      qd_0_2
        w1      qd_0_3
        w1      qd_0_4
        w1      qd_0_5
        w1      qd_0_6
        w1      qd_0_0
        w1      qd_0_1
        w1      qd_0_2
        w1      qd_0_3
        w1      qd_0_4
        w1      qd_0_5
        w1      qd_0_6
```

```
        wl      qd_0_0
        wl      qd_0_1
        wl      qd_0_2
        wl      qd_0_3
        wl      qd_0_4
        wl      qd_0_5
        wl      qd_0_6
        label   error_corrected_1
        id      qd_0_0
        id      qd_0_1
        id      qd_0_2
        id      qd_0_3
        id      qd_0_4
        id      qd_0_5
        id      qd_0_6
#------------------------------------------------
# counter_bottom_id_gate_count
        xor     full_add_xorab,1,
            count_id_gate_count_2
        and     full_add_andab,1,
            count_id_gate_count_2
        and     full_add_andcxorab,
            full_add_xorab,0
        xor     count_id_gate_count_2,0,
            full_add_xorab
        or      countertemp_id_gate_count,
            full_add_andab,full_add_andcxorab
        xor     full_add_xorab,0,
            count_id_gate_count_1
        and     full_add_andab,0,
            count_id_gate_count_1
        and     full_add_andcxorab,
            full_add_xorab,
            countertemp_id_gate_count
        xor     count_id_gate_count_1,
            countertemp_id_gate_count,
            full_add_xorab
        or      countertemp_id_gate_count,
            full_add_andab,full_add_andcxorab
        xor     full_add_xorab,0,
            count_id_gate_count_0
        and     full_add_andab,0,
            count_id_gate_count_0
        and     full_add_andcxorab,
            full_add_xorab,
            countertemp_id_gate_count
        xor     count_id_gate_count_0,
            countertemp_id_gate_count,
            full_add_xorab
        or      countertemp_id_gate_count,
            full_add_andab,full_add_andcxorab
        jump    countertop_id_gate_count
        label   counterbottom_id_gate_count
        noise   off
        subset  magic,7,qd_0_0,qd_0_1,qd_0_2,
            qd_0_3,qd_0_4,qd_0_5,qd_0_6,-
            ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,
            IIIZZZZ,IZZIIZZ,-ZIZIZIZ
        if      magic
        halt
        subset  magic,7,qd_0_0,qd_0_1,qd_0_2,
            qd_0_3,qd_0_4,qd_0_5,qd_0_6,-
            ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,
            IIIZZZZ,-IZZIIZZ,ZIZIZIZ
        if      magic
        halt
        subset  magic,7,qd_0_0,qd_0_1,qd_0_2,
            qd_0_3,qd_0_4,qd_0_5,qd_0_6,-
            ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,
            IIIZZZZ,-IZZIIZZ,-ZIZIZIZ
        if      magic
        halt
        subset  magic,7,qd_0_0,qd_0_1,qd_0_2,
            qd_0_3,qd_0_4,qd_0_5,qd_0_6,-
            ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,-
            IIIZZZZ,IZZIIZZ,ZIZIZIZ
        if      magic
        halt
        subset  magic,7,qd_0_0,qd_0_1,qd_0_2,
            qd_0_3,qd_0_4,qd_0_5,qd_0_6,-
            ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,-
            IIIZZZZ,IZZIIZZ,-ZIZIZIZ
        if      magic
        halt
        subset  magic,7,qd_0_0,qd_0_1,qd_0_2,
            qd_0_3,qd_0_4,qd_0_5,qd_0_6,-
            ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,-
            IIIZZZZ,-IZZIIZZ,ZIZIZIZ
        if      magic
        halt
        subset  magic,7,qd_0_0,qd_0_1,qd_0_2,
            qd_0_3,qd_0_4,qd_0_5,qd_0_6,-
            ZZZZZZZ,IIIXXXX,IXXIIXX,XIXIXIX,-
            IIIZZZZ,-IZZIIZZ,-ZIZIZIZ
        if      magic
        halt
# EOF
```

Bibliography

[1] S. Aaronson and D. Gottesman. Improved simulation of stabilizer circuits. *Phys. Rev. A*, 70:052328, 2004. arXive e-print quant-ph/0406196.

[2] D. Aharonov and M. Ben-Or. Fault-tolerant quantum computation with constant error. *Proceedings of the 29th Annual ACM Symposium on the Theory of Computation*, pages 176–188, 1997. arXive e-print quant-ph/9906129.

[3] P. Aliferis, D. Gottesman, and J. Preskill. Quantum accuracy threshold for concatenated distance-3 codes. *Unpublished*, 2005. arXive e-print quant-ph/0504218.

[4] A. R. Calderbank and P. W. Shor. Good quantum error-correcting codes exist. *Phys. Rev. A*, 54(2):1098–1106, 1996. arXive e-print quant-ph/9512032.

[5] A. Cross. Synthesis and evaluation of fault-tolerant quantum computer architectures. Master's thesis, Massachusetts Institute of Technology, 2005.

[6] D. Deutsch. Quantum computational networks. *Proceedings of the Royal Society of London*, 425:73–90, 1989.

[7] E. Farhi, J. Goldstone, S. Gutmann, and M. Sipser. Quantum computation by adiabatic evolution. *Unpublished*, 2000. arXive e-print quant-ph/0001106.

[8] D. Gottesman. *Stabilizer codes and quantum error correction.* PhD dissertation, Caltech, 1997. arXive e-print quant-ph/9705052.

[9] A. Kitaev. Quantum computations: algorithms and error correction. *Russian Math Surveys*, 52:1191–1249, 1997.

[10] E. Knill and R. Laflamme. Concatenated quantum codes. report LAUR-96-2808, LANL, 1996. arXive e-print quant-ph/9608012.

[11] E. Knill, R. Laflamme, and W. Zurek. Resilient quantum computation: Error models and thresholds. *Science*, 279(5349), 1998. arXive e-print quant-ph/9702058.

[12] M. Nielsen and I. Chuang. *Quantum computation and quantum information.* Cambridge University Press, Cambridge, England, 2000.

[13] J. Preskill. Reliable quantum computers. *Proc. Roy. Soc. Lond. A*, 454:385–410, 1998. arXive e-print quant-ph/9705031.

[14] J. Preskill. Fault-tolerant quantum computation. In H. Lo, S. Popescu, and T. Spiller, editors, *Introduction to quantum computation and information.* World Scientific Publishing Company, 2001. arXive e-print quant-ph/9712048.

[15] R. Raussendorf, D. E. Browne, and H. J. Briegel. Measurement-based quantum computation on cluster states. *Phys. Rev. A*, 68:22312, 2003. arXive e-print quant-ph/0301052.

[16] B. Reichardt. Improved ancilla preparation scheme increases fault-tolerant threshold. 2004. preprint arXive e-print quant-ph/0406196.

[17] P. W. Shor. Scheme for reducing decoherence in quantum computer memory. *Physical Review A*, 52(4):2493–2496, 1995.

[18] A. Steane. Error-correcting codes in quantum theory. *Phys. Rev. A*, 52:2493, 1995.

[19] A. Steane. Overhead and noise threshold of fault-tolerant quantum error correction. *Phys. Rev. A*, 68(042322), 2003. arXive e-print quant-ph/0207119.

[20] A. Steane. Fast fault-tolerant filter for quantum codewords. 2004. arXive e-print quant-ph/0202036.

[21] K. Svore, B. Terhal, and D. DiVincenzo. Local fault-tolerant quantum computation. *To appear in Phys. Rev. A*, 2005. arXive e-print quant-ph/0410047.

[22] C. Zalka. Threshold estimate for fault tolerant quantum computing. *Unpublished*, 1996. arXive e-print quant-ph/9612028.