Bachelor Thesis
Decoding the Color Code

Clemens Schumann,
Advised by Peter-Jan Derks

April 3, 2023

Abstract

The study of Quantum Error Correction (QEC) is essential to the development of quantum computers, as it provides a way to protect quantum information from errors that can occur in a real-world setting subject to electromagnetic/thermal and other noise.
In this thesis, we will give an overview of quantum error correction codes (ECCs) and introduce decoding schemes for the color code, a QEC code that uses three colorable three-regular graph configurations of stabilizers to perform quantum error correction.
We also compare the thresholding performance of various ECCs and decoding schemes, finding a pseudo-threshold of 10^{-3}% for the Steane color code using a lookup table decoder and around 16% for the MWPM Surface/Toric/Cylindric codes.
While unable to determine these Thresholds more precisely due to computational limitations, the author believes that upon further calculation the Cylindric code could be found to have a threshold that lies between the higher surface code threshold and the lower toric code threshold.
We also present a prototype lifting decoder for a toric hexagonal honeycomb lattice color code and a step-by-step guide to this construction is included, however this decoder is incomplete and only works for a subset of possible errors (individually occurring ones) due to a bug in the lifting procedure and is therefore not included in the thresholding comparison.

Contents

1 Introduction

In the last few years, quantum computers have been the focus of intense research since they are expected to be able to solve problems that are intractable for classical computers. Quantum computers employ principles of quantum mechanics, whereby states can exist in superpositions of multiple states, and can be entangled, i.e. correlated in order to perform computation.

One area where quantum computers are expected to be able to outperform classical computers is decrypting RSA encryption by efficiently factoring large numbers. This has recently been shown by researchers at the Beijing Academy of Quantum Information Sciences to require merely 10 error-corrected qubits [1] to efficiently factor a 40-bit length number, and is estimated to require merely 372 error-corrected qubits to efficiently decrypt 2048-bit RSA encryption. It is therefore likely that within the next decade RSA encryption will no longer be viable for protecting sensitive data.

Others include simulations of quantum systems, which can be of great use in medical research and quantum chemistry, as well as optimization problems, which are of great use in logistics and scheduling. Further, the Quantum Fourier Transform, which is a quantum algorithm that can be used to efficiently compute the Discrete Fourier Transform, can be used for things like computing ideal signal output from 5G towers to minimize interference. While providing significantly less advantage over classical computers than the aforementioned applications, the quantum search algorithm also provides a square-root improvement in the time complexity of searching for a specific item in a database, which could also have widespread applications.

In order to be able to use quantum computers for these applications, we need to ensure their resiliency towards errors introduced by thermal, electromagnetic and other noise. This can be done via Quantum Error Correction (QEC) codes, which we will introduce and discuss in this thesis.

A quantum computer operates on so-called *qudits*, which can be any multi-level quantum system. Physical implementations of these include particles with spin as well as optical implementations. In this thesis, we will focus on *qubit*-based systems, i.e. two-level quantum systems as base units of computation.

In Chapter 2 we will introduce the basic QEC terminology and pictures with which to analyze quantum circuits.

In Chapter 3 we will introduce the different types of QEC codes, namely the Surface, Toric and Color codes as well as their construction via Hypergraph products of classical codes.

In Chapter 4 we will go over the different decoding schemes for QEC codes, such as the MWPM, Union-Find and Lifting decoders.

In Chapter 5 we will discuss and compare the simulated thresholds for these codes using introduced decoding schemes.

2 Background

In this chapter, we will analyze a quantum circuit diagram using different pictures of quantum mechanics, namely the Schroedinger and the Heisenberg picture. A quantum circuit diagram is a visual representation of the computation done in a quantum computer, whereby:

- States progress in time along horizontal parallel lines
- Time goes from left to right
- Gates are unitary matrix operators
- Gates denoted X, Y, Z are the single qubit Pauli operators $\sigma_x, \sigma_y, \sigma_z$
- Gates can act on one or multiple qubits, whereby an X gate on qubit 1 in a 3-qubit system should be interpreted as $(X \otimes \mathbb{I} \otimes \mathbb{I})|\psi_{1,2,3}\rangle$
- $M_{\{X,Y,Z\}^n}$ denotes an n qubit measurement of $\{X, Y, Z\}$

In classical computation, a *complete logical signature* is a group of operators, which can be successively applied to express any general boolean computation. One example of such a signature is $\{\neg, \wedge\}$. A quantum equivalent of this is the Pauli Group amended by the *Clifford group*, whereby the Clifford group is the group of operators that project eigenstates of a Pauli group operator onto an eigenstate of a Pauli group operator.

While not enabling universal computation (e.g. the phase estimation in Shor's algorithm [2] would require an additional T gate), the union of Clifford and Pauli group *is* a complete logical signature for those quantum operations that can be simulated efficiently on a classical computer [3]. This is relevant for quantum error correction, as applying corrective gates after an error is computationally and experimentally expensive and should therefore be put off until the first non-Clifford gate is encountered in the program. Until that point the propagation of the error through the circuit can be simulated efficiently.

The Clifford Group can be generated by:

- The Hadamard-Gate H, which performs single qubit basis changes from eigenstates of X to eigenstates of Z and vice-versa:

  $H|+\rangle = |0\rangle$, $H|0\rangle = |+\rangle$, $H|-\rangle = |1\rangle$, $H|1\rangle = |-\rangle$

- The Phase-Gate P, which performs single qubit sign flips on the state parts which are $|1\rangle$ in the computational basis:

  $P(\alpha|0\rangle \pm \beta|1\rangle) = \alpha|0\rangle \mp \beta|1\rangle$

- The CNOT-Gate, which on a two qubit system performs an X gate on the second qubit if the first qubit is $|1\rangle$, so maps:

  $\alpha|00\rangle + \beta|01\rangle + \delta|10\rangle + \gamma|11\rangle$
  $\mapsto \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$

In the σ_z-basis their matrix representations are:

- $H = \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$; $P = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$
- $CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

2.1 Schroedinger picture

In the Schroedinger picture, we focus on the time evolution of qubit states:

$$|\psi\rangle = |\psi(t)\rangle \tag{1}$$

Measurements project these states onto eigenstates of the measurement operators via a projection P, so:

$$P_M^{\pm}|\psi\rangle = \frac{(M \pm \mathbb{I})|\psi\rangle}{2} \tag{2}$$

Where M is a matrix representation of the physical observable to be measured. For example, a measurement of a single qubit's spin along the z-axis would be represented as:

$$M_Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{3}$$

And that measurement would perform a projection P_Z:

$$P_Z^+ = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} or P_Z^- = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \tag{4}$$

on the state, depending on whether the measurement result yielded $+1$ or -1. Therefore, to calculate the output of a quantum circuit in the Schroedinger picture, simply apply the measurements and gates on the input states. As can be seen explicitly calculated in the Schroedinger picture in

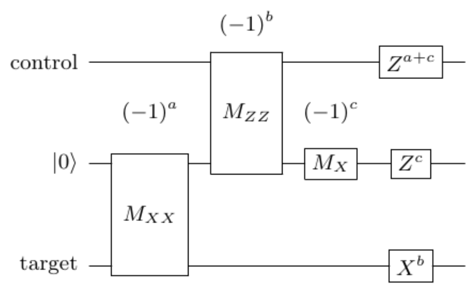


Figure 1: A Quantum Circuit to implement a measurement based Controlled-$X_{|\psi\rangle_{control} \to |\psi\rangle_{target}}$ Gate, where $|0\rangle$ is the $+1$ eigenstate in σ_z-basis.

Appendix 7.1, the circuit from Figure 1 implements a CNOT gate from the control qubit to the target qubit.

We will now analyze this circuit in the Heisenberg picture [4], finding that it results in an equal output.

2.2 Heisenberg picture and stabilizer formalism

2.2.1 Stabilizer group

We call an operator/gate S, to which the input state is an eigenvector ($S|\psi\rangle = |\psi\rangle$), a *stabilizer* of that input state. For n-qubit systems, we write these stabilizers as n-tensor-products of pauli operators $P \in P_G$, where P_G is the group generated by the Pauli operators and the Pauli operators are the operators on $\mathbb{F}_2$ such that:

$$\forall P \in P_G : P^2 = \mathbb{I}. \tag{5}$$

In the Heisenberg picture, stabilizers are tracked instead of states. The stabilizer group S_G is the group generated by the set of stabilizers:

$$S_G = \langle S_0, .., S_n \rangle : S|\psi_{in}\rangle = |\psi_{in}\rangle \forall S \in S_G \tag{6}$$

So for the example in Figure 1 it is the group of operators to whom $|\psi_{control}\rangle \otimes |0\rangle \otimes |\psi_{target}\rangle$ is an eigenstate, namely $\mathbb{I} \otimes Z \otimes \mathbb{I}$ (and trivially $\mathbb{I} \otimes \mathbb{I} \otimes \mathbb{I}$, which we choose to ignore as a stabilizer since any three-qubit state is stabilized by it, and it can be generated by squaring any stabilizer constructed through tensor products of Pauli matrices).

A stabilizer group is always an abelian group i.e. its elements commute, since if:

$$\forall A, B \in S : AB|\psi\rangle = BA|\psi\rangle = |\psi\rangle \Rightarrow [A, B]|\psi\rangle = 0 \tag{7}$$

2.2.2 Effect of gates on stabilizers

To determine the effect a gate operation A has on a stabilizer, consider the following. If $S|\psi\rangle = |\psi\rangle$ then:

$$A|\psi\rangle = AS|\psi\rangle = AS\mathbb{I}|\psi\rangle = \underbrace{ASA^\dagger}_{=S'} A|\psi\rangle \tag{8}$$

So we now know that the post-gate state is an eigenstate of S'. Therefore $S'_G = \langle AS_0A^\dagger, ..., AS_nA^\dagger \rangle$.

2.2.3 Effect of measurements on stabilizers

After a measurement M, an n qubit input state will always collapse into either the +1 or the -1 eigenstate of the measurement operator. In the first case the acting measurement operator was $\mathbb{I}^{\otimes n} + M$, in the second it was $\mathbb{I}^{\otimes n} - M$.

A Pauli measurement operator M can either commute with all stabilizer operators, in which case M itself is a stabilizer already. In this case the measurement has no effect on the state, since the measurement of a stabilizer projects onto identity. Otherwise it can anticommute with at least one operator in S_G, since Pauli operators as well as their tensor products can only commute or anti-commute with each other. The product of two operators that both anticommute with another operator will then commute with that operator. So in order to obtain the new stabilizers S'_G:

1. Identify $S \in S_G : \{S, M\} = 0$
2. Remove S from S_G
3. Add M to S_G
4. replace each $N \in S_G \cup \overline{X} \cup \overline{Z}$ with SN if $\{N, M\} = 0$

where $\overline{X}$ and $\overline{Z}$ are the sets of logical X and Z operators respectively. A logical operator is an operator which maps an eigenstate of a systems stabilizers to another eigenstate of those stabilizers.

2.2.4 Circuit Analysis in Stabilizer formalism

In the following, stabilizers will be written without the tensor product symbols, so in our case the stabilizer is initially: $S^0_G = \langle IZI \rangle$, the logical $\overline{X}$ operator is IXI and the logical $\overline{Z}$ operator is ZIZ. In the circuit shown in Figure 1, the measurements project onto:

$$P_1^{\pm} = \frac{1}{2}\left(\mathbb{I}^{\otimes 3} \pm \mathbb{I} \otimes X \otimes X\right) \tag{9}$$

$$P_2^{\pm} = \frac{1}{2}\left(\mathbb{I}^{\otimes 3} \pm X \otimes X \otimes \mathbb{I}\right) \tag{10}$$

$$P_3^{\pm} = \frac{1}{2}\left(\mathbb{I}^{\otimes 3} \pm \mathbb{I} \otimes X \otimes \mathbb{I}\right) \tag{11}$$

After the first measurement, the state is stabilized by IXX, since it collapses into an eigenstate of the measurement operator. Notably, if the measurement operator M anticommutes with some element of the stabilizer S:

$$SP_{-}S^{\dagger} = \frac{1}{2}S\left(\mathbb{I}^{\otimes 3} - M\right)S^{\dagger} = \frac{1}{2}\left(\mathbb{I}^{\otimes 3} + M\right)SS^{\dagger} = P_{+} \tag{12}$$

So by applying an anticommuting previous stabilizer operator after the measurement one can ensure that the state is in the P_+ projected state $P_+|\psi_{init}\rangle$ (in short, +1 and -1 eigenstates have the same stabilizers if we add conditional gates accordingly).

In our case, IZI and IXX anticommute, so now the state is stabilized by $S_G^1 = \langle IXX \rangle$. Both initial logical operators commute with the first measurement operator, so they are left unchanged.

After the second measurement M_2=ZZI, since this measurement anticommutes with the IXX stabilizer, the new stabilizers are: $S_G^2 = \langle ZZI \rangle$. The logical $\overline{X}$ and $\overline{Z}$ operators are unaffected , since they commute with the measurement operator.

After the third measurement M_3 =IXI, since this measurement anticommutes with the stabilizer, the new stabilizers are: $S_G^3 = \langle IXI \rangle$. The logical $\overline{Z}$ operator anticommutes with the measurement, so is replaced by $\overline{Z_3}$=ZZI · ZIZ = IIZ. The logical $\overline{X}$ is unaffected since it commutes with the measurement operator.

The stabilizer for the control and target qubit is still identity, and logical $\overline{Z} : ZIZ \to IIZ$. Since this circuit maps $Z_{control} \otimes Z_{target} \mapsto I_{control} \otimes Z_{target}$, and via a similar analysis it can be shown that it also maps $I \otimes Z \mapsto Z \otimes Z$, $Z \otimes I \mapsto Z \otimes I$, $X \otimes I \mapsto X \otimes X$ and $I \otimes X \mapsto I \otimes X$, this circuit implements a logical CNOT from the first to the third qubit.

3 Error detection and correction

The concept of (classical) *error-correcting codes* (ECC) was introduced by Claude Shannon in 1948 [5]. Fundamentally, an ECC encodes *logical* information within a large superset of basic information carriers.

In the case of a classical computer, this means encoding a bitstring within a system containing more physical bits than the length of the encoded message, with the goal of message transmission being resilient to some bits being faulty or subject to interference (i.e. EM-interference). Analogously, in the case of a quantum computer this means encoding a *logical* qubit within a system of multiple qubits, with a similar goal of resilience towards errors caused by external influences.

In this chapter, we will give an overview of different quantum error correction codes, starting with adaptations of classical codes.

3.1 Classical codes

Two known classical ECCs are the repetition and the ring code. In Quantum error correction, we speak of $[[n, k, d]]$ stabilizer codes if an encoding scheme allows for n physical qubits to encode k logical qubits to an error distance of d, i.e. $\lfloor \frac{d-1}{2} \rfloor$ arbitrary individual errors being corrigible.

In the following, I will refer to the classical codes as having a distance of $\frac{1}{2}$, to indicate that they do not protect against an arbitrary single-qubit error but only against flips in one specific eigenbasis.

3.1.1 Repetition code

For this error code information is encoded by repeating the intended message some amount of times, and then decoding it by performing a majority vote on the transmitted message.

A quantum equivalent of the 3-bit repetition code performed on the message $|1\rangle$ is the $[[3,1,\frac{1}{2}]]$ repetition code depicted in Figure 2 , including so-called *syndrome extraction*. A syndrome is a stabilizer that can be measured to detect whether and where an error has occurred in a multi-qubit system. It is crucial that the measurement of such syndromes occurs without harming the actual quantum information stored in the $data - qubits$. Therefore two additional *ancilla-qubits* (both initialized to $|0\rangle$) are attached to the circuit via CNOTs. This circuit is stabilized by IZZ and ZZI, measured by ancilla

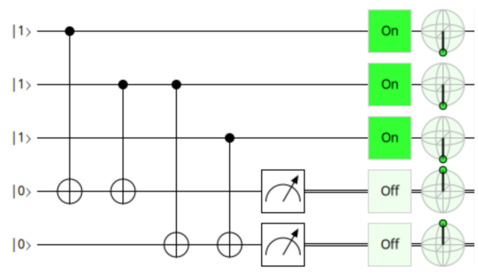


Figure 2: Bitflip Syndrome extractor for [[3,1,$\frac{1}{2}$]] repetition code
+1 measurement result on first ancilla indicates a bitflip error on qubits 1 or 2, +1 result on second ancilla indicates bitflip on second or third qubit

1 and 2. The measurement result will therefore be a vector of length two, with each entry either being +1 or -1. To simplify the algebra this will be changed to the binary representation of 0 for +1 and 1 for -1.

To represent the code, stabilizers can be stacked together to a so-called parity-check-matrix, which satisfies:

$$M_{pc} \cdot \vec{v}_{error} = \vec{v}_{syndrome} \tag{13}$$

So e.g. the parity check matrix for the $[3, 1, \frac{1}{2}]$ repetition code would be:

$$M_{pc3} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \tag{14}$$

And the syndrome for an X error on the first qubit would be $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$.

If we draw a graph to represent this code, with here square nodes being ancilla qubits and round nodes being data qubits, we obtain the following:

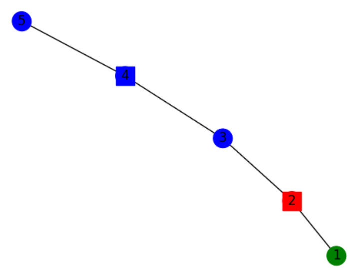


Figure 3: Graph for [[3,1,$\frac{1}{2}$]] repetition code with error on node 1 marked in green and resulting syndrome marked red. Squares represent ancilla qubits and circles represent data qubits.

3.1.2 Ring code

The ring code's graph essentially simply loops around at the repetition code's single-edged ancilla nodes via an additional ancilla. It's edge matrix where the nth row represents which data qubit is connected to the nth ancilla qubit is the following:

$$M_{pc3} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \tag{15}$$

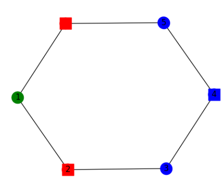


Figure 4: Graph for [[3,1,$\frac{1}{2}$]] ring code with error on node 1 marked in green and resulting syndrome marked in red. Squares represent ancilla qubits and circles represent data qubits.

3.2 Quantum Error Model

This way of encoding information however leaves a notable issue: It only detects bitflip, or Pauli-X, errors occurring on the stored quantum information. While using Hadamard gates one could trivially adapt this code to instead detect Pauli-Z errors, it is not possible to use linear codes like the repetition code to *simultaneously* detect Pauli-X and Pauli-Z errors occurring.

Unlike classical computers, on a quantum computer the type of error is not limited to a bitflip. Even for single-qubit states there exists an infinite amount of differing possible errors, since when representing a single qubit state as a vector on a Bloch sphere it immediately becomes apparent that there are an infinite number of vectors on that sphere which are different from it. It turns out though, that the change from one normalized state to another is merely a sum of rotations.

Noise can therefore be modeled as a sum of Pauli gates. Any single qubit error operator matrix E can be written as:

$$E = \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \alpha\mathbb{I} + \beta X + \delta Y + \gamma Z \tag{16}$$

With an apropriate choice of $\alpha, \beta, \gamma, \delta$. In effect, this means that with probability α, the effect of the error $E|\psi\rangle$ will be $\mathbb{I}$; with probability β its effect will be X, and so on.

It is hence sufficient to determine which of these errors $\mathbb{I}$, X, Y or Z has occurred, and we can apply the appropriate operator to return to the initial state. Since an identity noise occurring is irrelevant to us, and XY as well as ZY (anti-) commute, we need only detect for X and Z errors occurring in order to detect any single qubit errors. (because of the commutation relation between {X,Y} and {Y,Z} a Y error will appear as both an X and Z error).

3.3 Topological codes

Hypergraph product codes, introduced by Tillich and Zémor [6], provide a toolset for generating valid codes from existing encoding schemes. A hypergraph product code of two existing codes will always remain a valid detection code.

The parity check matrix H of a hypergraph product code is generated by two m by n parity check matrices of valid codes in the following way:

$$M_{PC_{Hypergraph}} = \begin{pmatrix} \left(M_{pc1} \otimes \mathbb{I}_{n_2} | \mathbb{I}_{m_1} \otimes M_{pc2}^T\right) & 0 \\ 0 & \left(\mathbb{I}_{n_1} \otimes M_{pc2} | M_{pc1}^T \otimes \mathbb{I}_{m_2}\right) \end{pmatrix} \tag{17}$$

3.3.1 Surface code

We can therefore form a hypergraph product code of two repetition codes to obtain the [[d^2,1,d]] "Surface-Code" which can detect up to d of *both* X and Z errors, and therefore any error happening [7]. We can draw this code as a graph, whereby the code's stabilizers are understood as an adjacency Matrix of data to ancilla qubits. Like the repetition code, the Surface code is a code that is regular until its boundary nodes. The logical operators on the surface code are lines that go from one boundary to another that lies across, as this triggers every ancilla along the way twice, thus nonce, and therefore takes the message back to the codespace.

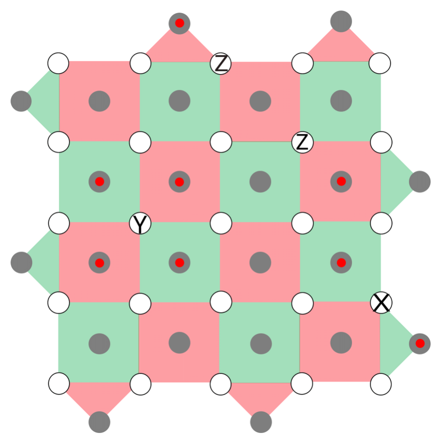


Figure 5: Distance 5 Surface code with data qubits in white and ancilla qubits in grey. Green Faces represent Z stabilizers and Red faces represent X stabilizers. Errors on data qubits are marked by respective Pauli names and violated stabilizers are marked in red. Figure base lattice taken from [8].

3.3.2 Toric code

Similarly, a hypergraph product code of two ring codes can be generated.

Unlike in Figure 5, it is also possible to draw topological ECC graphs without colored plaquettes, by drawing it such that the data qubits are on edges of the graph and the ancilla qubits for Z-checks are on faces while the ancilla qubits for X-checks lie on nodes. This representation is called a Tanner graph [6] and is used in Figure 6.

Since the resulting Tanner graph forms a torus, we call this code the "Toric code".

The logical operators on the toric code are loops, so a circle of 'errors' on nodes is a logical X operator, and a circle of 'errors' on faces is a logical Z operator.

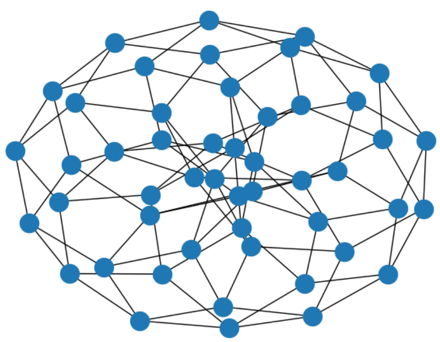


Figure 6: Tanner graph for [[49,1,7]] toric code.

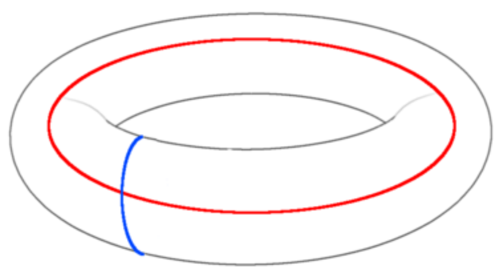
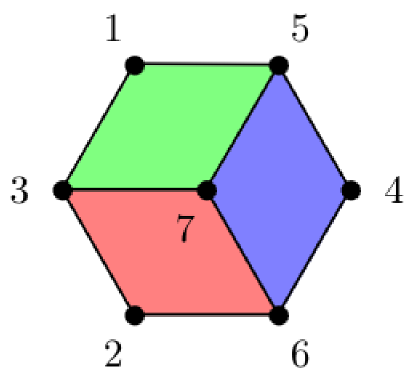


Figure 7: Logical $\overline{X}$ and $\overline{Z}$ operators on toric code Tanner graph. Image courtesy of James Wooton's contribution to Wikipedia.

3.3.3 Color code

The color code's parity-check-matrix's rows are both the code's X stabilizers and Z stabilizers. Any three-colorable and three-valent graph represents a valid color code. On the color code, an error is bounded by syndromic faces of all colors. The simplest color code is the [[7,1,3]] Steane code [9].



$$
\begin{array}{r|ccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
 & I & I & I & X & X & X & X \\
 & I & X & X & I & I & X & X \\
 & X & I & X & I & X & I & X \\
 & I & I & I & Z & Z & Z & Z \\
 & I & Z & Z & I & I & Z & Z \\
 & Z & I & Z & I & Z & I & Z \\
\hline
\overline{X} = & X & X & X & X & X & X & X \\
\overline{Z} = & Z & Z & Z & Z & Z & Z & Z
\end{array}
$$

Figure 8: Graph for the [[7,1,3]] color code, also known as the Steane code , and its stabilizers. Figure from [9].

4 Decoding Schemes

An important task towards achieving fault-tolerant quantum computation is finding efficient decoding schemes. Since error propagation on non-clifford gates cannot be simulated efficiently [3], and we are only given syndromes by our ECC, the decoding scheme must be able to compute occurred errors from syndromes in time before the quantum algorithm our computer intends to calculate reaches a non-Clifford operation. This ideally requires very fast classical computation of the syndrome decoding.

In this chapter, we will introduce some of the main decoding schemes for varying types of quantum error correction codes.

4.1 Decoders for Surface/Toric codes

Syndromes on the surface/toric code are a set of nodes and faces on the code's Tanner graph. The node ancilla syndromes correspond to Z errors, while the face ancilla syndromes correspond to X errors. Since neighboring errors will trigger an ancilla that is between both errors twice, a chain of errors will only appear as two ancilla syndrome bits being flipped at its borders. The task of a decoding scheme for a surface/toric code is thus to find the shortest paths between node pairs/face pairs, since the most likely chain of errors to occur given a $< 50\%$ physical error rate is the shortest one.

In practice, decoders for surface/toric codes only need to be able to match nodes, since the matching of faces is just matching nodes on the dual graphs and the resulting data qubit errors can just be joined (i.e. if an edge is found to have an error on both the X graph as well as the dual Z graph, we know a Y error has occurred on that edge/data-qubit). An example of a distance 5 surface code with two Z errors, one X error and one Y error is shown in Figure 5. As with the ring code, the decoding problem can be seen as either the solution of Equation 13 for a minimum weight $\vec{v}_{error}$ or as a graph-matching problem.

4.1.1 MWPM decoding

The *Minimum-Weight Perfect Matching* algorithm is a variant of Dijkstra's algorithm that can be used to find the shortest vector of edges that are bounded by the input syndrome nodes. The algorithm is as follows:

1. Find a set of unmatched nodes that can be reached from the Matching[1] by alternating between matched and unmatched edges. Call these nodes "augmenting nodes".

2. Find an augmenting path starting from each augmenting node, i.e. a path that starts and ends with an unmatched node, and alternates between matched and unmatched edges.

3. If such a path is found, flip all edges along it from matched to unmatched, and vice versa.

4. Repeat until no augmenting path is found.

This decoding scheme has the advantage of being guaranteed to find a global optimum of decoding edge paths, i.e. it always finds the shortest vector of edges that are bounded by the syndrome nodes. Under the assumption of high error rates and/or large decoding graphs, this scheme also requires significantly less computational memory overhead than the union-find scheme [10].

4.1.2 Union-Find decoder

The union-find decoder is a greedy algorithm that can be used to perform node matching along edges of a graph. It is a variant of Kruskal's algorithm. The algorithm is as follows:

1. Initialize a cluster set for each syndrome node

2. Grow each cluster by one edge in each direction

3. Merge all clusters that share a node

4. For all clusters with an even amount of syndrome nodes, perform MWPM within that cluster. Pop the found error edges from the graph.

5. Repeat until all clusters are merged/discarded.

[1] In graph theory, a matching is a subset of graph edges such that no two edges share a common vertex. The Goal of the MWPM algorithm is to find a Matching with minimum weight, i.e. a shortest vector of edges

While the union-find decoder is faster for small to medium sized graphs and relatively simple to implement, it is not guaranteed to find a global optimum and its performance degrades significantly for large graphs and high error rates [11]. For this reason, a MWPM algorithm was chosen for decoding the toric subgraphs of the color code in our lifting decoder thresholding in Chapter 4.2.2.

4.2 Color code decoders

Unlike the surface and toric codes, in the color code the data qubits sit on the graph's nodes, and the ancillas on the graph's faces. Decoding the color code entails matching three differently colored faces to their enclosed nodes. This is a significantly more challenging task than decoding the surface or toric code, since optimal three-colored graph hyperface matching is an NP-hard problem [12].

4.2.1 Lookup table decoding

A lookup table decoder works by generating the syndromes for the entire set of possible input errors, thus creating a table holding possible errors responsible for each possible syndrome. The decoding then consists of merely assuming the minimum weight error that leads to the known syndrome, since given low physical error rate, the least amount of errors leading to an error is the most probable event.

This decoding scheme is particularly useful for small codes, as well as non-topological (random) LDPC (Low-Density-Parity-Check) codes, since these cannot be decoded using graph theory. A big issue with this decoding scheme is that generating lookup tables is extremely computationally expensive ($O(2^n)$, since a syndrome must be computed and stored for every possible error vector, having length n and 2 possibilities per entry).

This renders it practically unfeasible to generate lookup tables for codes with a larger number of total data qubits.

In Figure 9 is an example of the lookup table result for an X error on qubit 7 (the central qubit) on the Steane code. The resulting syndrome is (1,1,1,0,0,0), with the first three bits indicating the steane code faces X checks, and the second three bits indicating the Steane code faces Z checks. The lookup table will return a set of many possible errors resulting in that

```
The syndrome [1 1 1 0 0 0]
 can be caused by the following errors:
(0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
(0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0)
(1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
(1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)
(0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)
The most likely cause of this syndrome is
 (0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0)
```

Figure 9: Lookup table for an X error on the central qubit of a Steane code (qubit 7), generating code can be found in Appendix 7.2.1

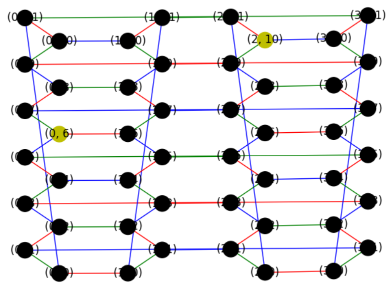
syndrome, but simply choosing the one with the least number of errors (minimum weight) gives the correct error prediction.

4.2.2 Lifting decoder

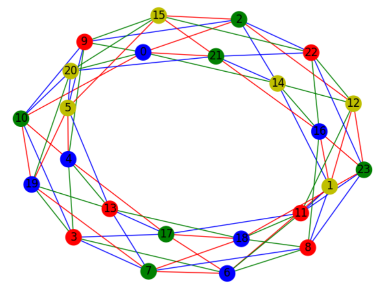
The Lifting decoder works as follows:

1. Create dual of color code graph
2. Generate single-edge-colored subgraphs of the dual
3. Decode subgraphs using MWPM/Union-Find
4. Unify all edges from subgraph corrections
5. Find all shortest-length loops on this union
6. If there are remaining edges not forming a loop, abort the liftin procedure
7. All nodes bounded by the faces that are elements of the shortest-length loop sets are error nodes.

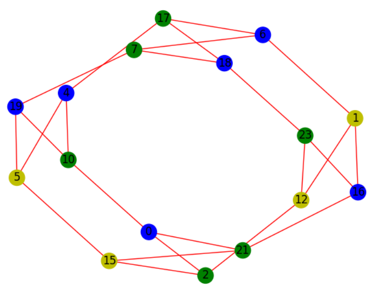
By sub-tiling the graph into smaller subgraphs, we can reduce the problem of decoding e.g. a honeycomb lattice toric color code to a set of MWPM-decodable toric graphs that merely need to be "lifted" into a combination of subgraph decodings to decode the original color code graph [12].



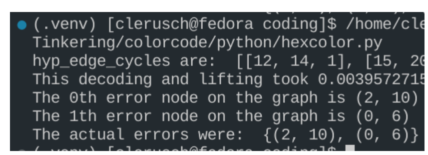
(a) Original toric honeycomb lattice color code. Errors are marked yellow, face colors are implied by opposing colors wrapping them.



(b) Dual of color code lattice. Nodes are faces on the original lattice yellow marked nodes represent syndromes.



(c) Red subgraph to be decoded via MWPM



(d) Correct error prediction output for single distributed error nodes.

Figure 10: Steps in the lifting decoder. Generating code can be found in 7.3 and the entire git repository can be found in [13]

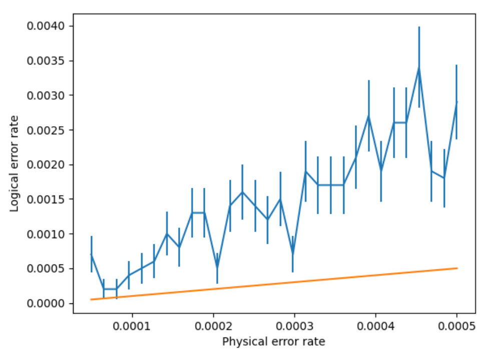
This decoding is not optimal, as it does not take into account the other two colored subgraphs when computing an MWPM edge prediction. The polynomial time complexity of the lifting decoder does not violate the NP-hardness of the 3-color matching problem, since the lifting procedure does not provide an optimal solution. A graphical depiction of the steps of the lifting decoder is shown in Figure 10.

5 Thresholds

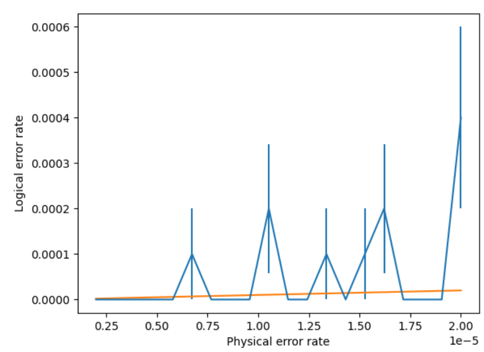
To compare different codes and decoding schemes we introduce the concept of thresholds, whereby the threshold of a specific code of scalable distance with a specific decoding scheme is defined as the physical error rate *per* at which the logical error rate becomes greater than 50% in the limit of infinite distance.
Thresholds can vary depending on the error model, i.e. some codes can have a higher threshold for X than for Z errors. For simplicity's sake in the following, we will assume equal X, Y and Z error rates of $\frac{per}{3}$.

Using this error model, we found a threshold of $16.3 \pm 0.5\%$ for the surface code, $16.0 \pm 0.5\%$ for the toric code and 16.1 ± 0.5 from subfigures b), d) and f) in Figure 12. Their thresholds are within single error margins of each other, and can therefore be called identical.

Since the Steane code for which we generated a lookup table is not a distance-scalable code, only a *pseudo*-threshold can be found here, i.e. the crossing point to worse performance than unencoded information. As can be seen in Figure 11, the pseudo-threshold lies around $(1 \pm 0.5)10^{-5}$.



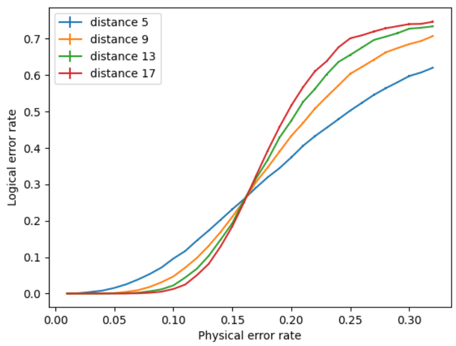
(a) Lookup table Steane code threshold



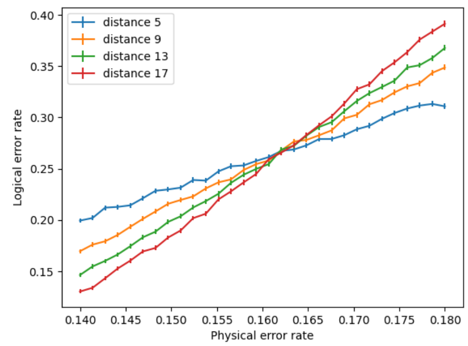
(b) Detailed view at around $per = 10^{-5}$

Figure 11: Lookup table pseudo threshold for the Steane code, generating code can be found in Appendix 7.2.2

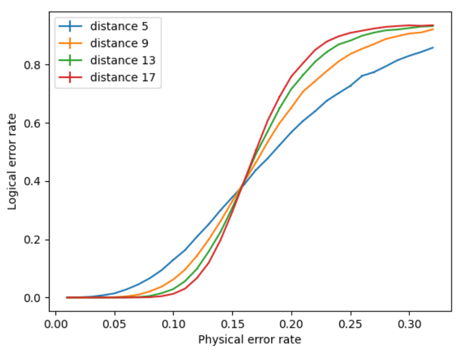
A prototype for a scalable lifting decoder for the hexagonal toric color code that only corrects separate individual errors was implemented in 7.3.



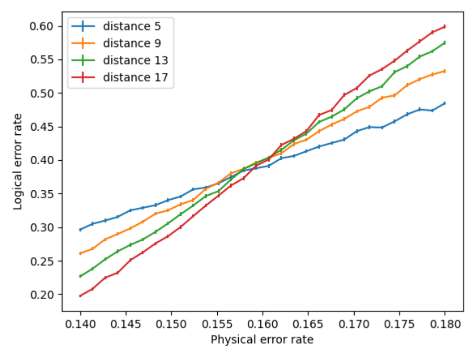
(a) Surface code MWPM thresholding overview



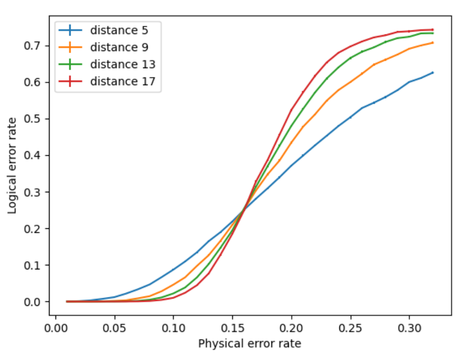
(b) Detailed view for precise threshold determination of surface code



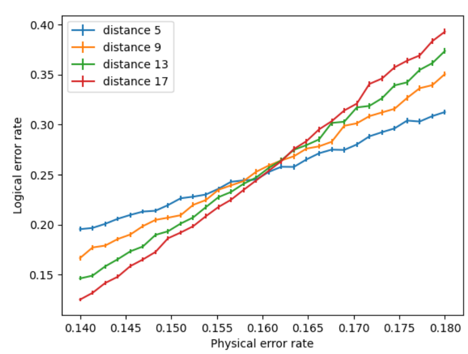
(c) Toric code MWPM thresholding overview



(d) Detailed view for precise threshold determination of toric code



(e) Cylinder code MWPM thresholding overview



(f) Detailed view for precise threshold determination of cylinder code

Figure 12: Thresholding of the surface/toric/cylinder code using the MWPM decoder implemented in the PyMatching [10] library. Generating code can be found in Appendix 7.4.1

6 Conclusion

In this thesis, we gave an overview of existing quantum codes as well as some decoding schemes. The determined thresholds of $16.3 \pm 0.5\%$ for the surface code and $16.0 \pm 0.5\%$ for the toric code were within single and threefold error margin respectively to the literature values [14]. Their thresholds were however not distinguishable with great confidence, and especially for the cylindric code it might be of interest to calculate these thresholds more precisely by using more significant computational resources in future works. The pseudo-threshold for the Steane code was found to be around 10^{-5}, which is the same as in the literature [15]. While the lifting decoder for the hexagonal toric lattice color code did not produce thresholdable output, it did work as a proof-of-concept on smaller error vectors as in 10.

Future work could include adapting a better cycle-finder algorithm for the lifted subgraph in order to obtain a decoder for distance- and logical-qubit-scalable color codes. Constructing such a scalable adaptation would also permit thresholding in order to study the lifting decoder's performance compared to others.

In recent months Floquet codes [16] have also been an area of intense study, and since they are also a honeycomb lattice-based code it might be interesting to analyze the viability of a lifting decoder for them.

References

[1] Z. T. Bao Yan and S. W. et. al, "Factoring integers with sublinear resources on a superconducting quantum processor," p. 1, 2022. [Online]. Available: https://arxiv.org/pdf/2212.12372.pdf

[2] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," pp. 16–19, 1995. [Online]. Available: https://arxiv.org/pdf/quant-ph/9508027.pdf

[3] D. Gottesman, "A theory of fault-tolerant quantum computation," pp. 8–9, 1998. [Online]. Available: https://arxiv.org/pdf/9702029.pdf

[4] ——, "The heisenberg representation of quantum computers," pp. 5–10, 1998. [Online]. Available: https://arxiv.org/pdf/9807006.pdf

[5] C. Shannon, "A mathematical theory of communication," pp. 22–24, 1948. [Online]. Available: https://people.math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf

[6] J. Tillich and A. Zemor, "Quantum ldpc codes with positive rate and minimum distance proportional to n1/2," pp. 11–14, 2009. [Online]. Available: https://arxiv.org/pdf/0903.0566.pdf

[7] J. Roffe, "Quantum error correction: An introductory guide," pp. 14–18, 2019. [Online]. Available: https://arxiv.org/pdf/1907.11157.pdf

[8] C. Chamberland and P. Ronagh, "Deep neural decoders for near term fault-tolerant experiments," *Quantum Science and Technology*, vol. 3, p. 5, 02 2018.

[9] B. W. Reichardt, "Fault-tolerant quantum error correction for steane‘s seven-qubit color code with few or no extra qubits," p. 1, 2018. [Online]. Available: https://arxiv.org/pdf//1804.06995.pdf

[10] O. Higgott, "Pymatching: A python package for decoding quantum codes with minimum-weight perfect matching," pp. 2–5, 2021. [Online]. Available: https://arxiv.org/pdf/2105.13082.pdf

[11] L. Z. Yue Wu, Namitha Liyanage, "An interpretation of union-find decoder on weighted graphs," pp. 5–6, 2022. [Online]. Available: https://arxiv.org/pdf/2211.03288.pdf

[12] N. Delfosse, "Decoding color codes by projection onto surface codes," pp. 12–15, 2018. [Online]. Available: https://arxiv.org/pdf/1308.6207.pdf

[13] C. Schumann, "Decoding the color code," Git repository, 2023, accessed: March 31, 2023. [Online]. Available: https://github.com/clerusch/coding

[14] D. W. et al., "Threshold error rates for the toric and surface codes," p. 11, 2009. [Online]. Available: https://arxiv.org/pdf/0905.0531.pdf

[15] B. T. Andrew Cross, David DiVincenzo, "A comparative code study for quantum fault tolerance," p. 2, 2009. [Online]. Available: https://arxiv.org/pdf/0711.1556.pdf

[16] M. D. et al., "Floquet codes without parent subsystem codes," 2022. [Online]. Available: https://arxiv.org/pdf/2210.02468.pdf

7 Appendix

7.1 Schroedinger picture calculation of CNOT circuit

In the quantum circuit depicted in figure 1 the input state can be written as $|\psi_{control}\rangle \otimes |0\rangle \otimes |\psi_{target}\rangle$ and the measurement in the first timestep can be expressed as $\mathbb{I} \otimes X \otimes X$.

The initial state $|\phi_{t=0}\rangle = |\psi_{control}\rangle \otimes |\psi_{ancilla}\rangle \otimes |\psi_{target}\rangle$
where
$|\psi_{control}\rangle = \alpha|0\rangle + \beta|1\rangle$
$|\psi_{ancilla}\rangle = |0\rangle$
$|\psi_{target}\rangle = \gamma|0\rangle + \delta|1\rangle$
therefore:

$$|\phi_{t=0}\rangle = \alpha\left(\gamma|000\rangle + \delta|001\rangle\right) + \beta\left(\gamma|100\rangle + \delta|101\rangle\right) \tag{18}$$

If the first measurement result is +1, the state becomes:

$$\begin{aligned}|\phi^{+}_{t=1}\rangle &= \frac{1}{2}\left(\mathbb{I}\otimes\mathbb{I}\otimes\mathbb{I} + \mathbb{I}\otimes X\otimes X\right)|\phi_{t=0}\rangle\\ &= \alpha\left(\gamma\left(|000\rangle + |011\rangle\right) + \delta\left(|001\rangle + |010\rangle\right)\right)\\ &+ \beta\left(\gamma\left(|100\rangle + |111\rangle\right) + \delta\left(|101\rangle + |110\rangle\right)\right)\end{aligned}$$

if the result is -1, it becomes:

$$\begin{aligned}|\phi^{-}_{t=1}\rangle &= \frac{1}{2}\left(\mathbb{I}\otimes\mathbb{I}\otimes\mathbb{I} - \mathbb{I}\otimes X\otimes X\right)|\phi_{t=0}\rangle\\ &= \alpha\left(\gamma\left(|000\rangle - |011\rangle\right) + \delta\left(|001\rangle - |010\rangle\right)\right)\\ &+ \beta\left(\gamma\left(|100\rangle - |111\rangle\right) + \delta\left(|101\rangle - |110\rangle\right)\right)\end{aligned}$$

In the case of the +1 Measurement $\rightarrow$ a=0:

$$\begin{aligned}|\phi^{++}_{t=2}\rangle &= \frac{1}{2}\left(\mathbb{I}\otimes\mathbb{I}\otimes\mathbb{I} + Z\otimes Z\otimes\mathbb{I}\right)|\phi^{+}_{t=1}\rangle\\ &= (|000\rangle\langle 000| + |001\rangle\langle 001| + |110\rangle\langle 110| + |111\rangle\langle 111|)|\phi^{+}_{t=1}\rangle\\ &= \alpha\left(\gamma|000\rangle + \delta|001\rangle\right) + \beta\left(\gamma|111\rangle + \delta|110\rangle\right)\end{aligned}$$

$$\begin{aligned}|\phi^{+-}_{t=2}\rangle &= \frac{1}{2}\left(\mathbb{I}\otimes\mathbb{I}\otimes\mathbb{I} - Z\otimes Z\otimes\mathbb{I}\right)|\phi^{+}_{t=1}\rangle\\ &= (|010\rangle\langle 010| + |011\rangle\langle 011| + |100\rangle\langle 100| + |101\rangle\langle 101|)|\phi^{+}_{t=1}\rangle\\ &= \alpha\left(\gamma|011\rangle + \delta|010\rangle\right) + \beta\left(\gamma|100\rangle + \delta|101\rangle\right)\end{aligned}$$

In the case of the -1 Measurement $\rightarrow$ a=1:

$$
\begin{aligned}
|\phi_{t=2}^{-+}\rangle &= \frac{1}{2}\left(\mathbb{I}\otimes\mathbb{I}\otimes\mathbb{I} + Z\otimes Z\otimes\mathbb{I}\right)|\phi_{t=1}^{-}\rangle \\
&= \alpha\left(\gamma|000\rangle + \delta|001\rangle\right) - \beta\left(\gamma|111\rangle + \delta|110\rangle\right)
\end{aligned}
$$

$$
\begin{aligned}
|\phi_{t=2}^{--}\rangle &= \frac{1}{2}\left(\mathbb{I}\otimes\mathbb{I}\otimes\mathbb{I} - Z\otimes Z\otimes\mathbb{I}\right)|\phi_{t=1}^{-}\rangle \\
&= -\alpha\left(\gamma|011\rangle + \delta|010\rangle\right) + \beta\left(\gamma|100\rangle + \delta|101\rangle\right)
\end{aligned}
$$

Now the applied measurement is $\mathbb{I}\otimes X\otimes\mathbb{I}$, which means:

$$
\begin{aligned}
|\phi_{t=3}^{+++}\rangle &= \frac{1}{2}\left(\mathbb{I}\otimes\mathbb{I}\otimes\mathbb{I} + \mathbb{I}\otimes X\otimes\mathbb{I}\right)|\phi_{t=2}^{++}\rangle \\
&= \frac{1}{2}((|010\rangle + |000\rangle)\langle 000| + (|011\rangle + |001\rangle)\langle 001| \\
&+ (|000\rangle + |010\rangle)\langle 010| + (|001\rangle + |011\rangle)\langle 011| \\
&+ (|110\rangle + |100\rangle)\langle 100| + (|111\rangle + |101\rangle)\langle 101| \\
&+ (|100\rangle + |110\rangle)\langle 110| + (|101\rangle + |111\rangle)\langle 111|)|\phi_{t=2}^{++}\rangle \\
&= \frac{1}{2}(\alpha\left(\gamma(|000\rangle + |010\rangle) + \delta(|011\rangle + |001\rangle)\right) \\
&+ \beta\left(\gamma(|101\rangle + |111\rangle) + \delta(|100\rangle + |110\rangle)\right))
\end{aligned}
$$

In this case, a,b and c would each be zero, therefore no further gate would be applied.
As intended, this state is equivalent to $CNOT_{|\psi_{Control}\rangle\rightarrow|\psi_{Target}\rangle}|\phi_{t=0}\rangle$.
Notably, each measurement sequence has a differing resulting ancilla state, however we do not care since ancillas are meant to be discarded.
Verifying that the other 7 measurement/computation paths also yield a CNOT implementation is left as an exercise to the reader.

7.2 Lookup table decoding

7.2.1 Table generation

```python
from typing import List
from numpy import array, vstack, hstack, zeros, uint8, ones,
    ndarray
from itertools import product
from random import random

def genSteaneLookupTable()->dict:

    # Generate Steane parity check matrix from identical
    # X and Z PCMs
    H = array([[1, 0, 0, 1, 0, 1, 1],
               [0, 1, 0, 1, 1, 0, 1],
               [0, 0, 1, 0, 1, 1, 1]])
    pcm = vstack((hstack((H, zeros(H.shape))),
                  hstack((zeros(H.shape), H))))

    # Generate lookup table
    lookup_table = {}
    for error in product([0, 1], repeat=14):
        syndrome = tuple(pcm @ error % 2)
        if syndrome in lookup_table:
            lookup_table[syndrome].append(error)
        else:
            lookup_table[syndrome] = [error]

    # Remove duplicates from lookup table
    for key in lookup_table:
        lookup_table[key] = list(set(lookup_table[key]))

    return lookup_table

def findMinWeight(predictions)-> ndarray:
    """
    Find the minimum weight tuple for a given prediction
    """
    curr_pred = ones(14,dtype=uint8)
    curr_best_weight = 100
    for pred in predictions:
        pred = array(pred)
        yweight = 0
        for i in range(int(len(pred)/2)):
```

```
            if pred[i] & pred[i+7] == 1:
                yweight += 1
                pred[i] = 0
                pred[i+7] = 0
        if yweight + sum(pred) < curr_best_weight:
            curr_pred = pred
            curr_best_weight = yweight + sum(pred)
    return curr_pred

def main():
    syndrome = array([1,1,1,0,0,0])

    possibles = genSteaneLookupTable()[tuple(syndrome)]

    print(f"The syndrome {syndrome}\n can be caused by the
   following errors: ")

    print(f"The most likely cause of this syndrome is\n {
   findMinWeight(possibles)}")

if __name__=="__main__":
    main()
```

7.2.2 Thresholding

```
from betterlookup import genSteaneLookupTable, findMinWeight,
    findMinWeight
from random import random
from numpy import zeros, uint8, concatenate, array,\
    array_equal, linspace, vstack, hstack, zeros, ndarray,
   logspace
from matplotlib.pyplot import errorbar, legend, \
    savefig, xlabel, ylabel, plot

def genSteaneError(per)->ndarray:
    """ Generates an error vector on the Steane code"""
    empty7 = zeros(7, dtype=uint8)
    xerror = empty7.copy()
    zerror = empty7.copy()
    for i in range(len(xerror)):
        if random()<per:
            xerror[i] = 1
    for j in range(len(zerror)):
        if random()<per:
            zerror[j] = 1
    yerror = concatenate((xerror,zerror))
```

```
    for k, bit in enumerate(yerror[:6]):
        if random()<per:
            yerror[k] = (yerror[k] + 1)%2
            yerror[2*k] = (yerror[2*k]+1)%2
    return yerror

def steaneLerCalc(steaneH, nr, per, logicals)->float:
    """Calculates the logical error rate of the steane
    code decoded with a lookup table"""
    numErrors = 0
    looktable = genSteaneLookupTable()
    for _ in range(nr):
        actual_error = genSteaneError(per)
        syndrome = steaneH@actual_error %2
        predictions = looktable[tuple(syndrome)]
        pred = findMinWeight(predictions)
        pred_L_flips = logicals@pred %2
        actual_L_flips = logicals@actual_error %2
        if not array_equal(actual_L_flips, pred_L_flips):
            numErrors += 1
    return numErrors/nr

def makeHgpPcm(Hx, Hz)->ndarray:
    """
    Makes a full parity check matrix including x and z
    checks for a hypergraph product code of two other codes
    """
    Hx = hstack((Hx, zeros(Hx.shape, dtype=uint8)))
    Hz = hstack((zeros(Hz.shape, dtype=uint8), Hz))
    H = vstack((Hx, Hz))
    return H

def main():
    steanelogicals = \
        array([\
            [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1]])
    steaneH = array([[1, 0, 0, 1, 0, 1, 1],
                     [0, 1, 0, 1, 1, 0, 1],
                     [0, 0, 1, 0, 1, 1, 1]])
    pers = linspace(2*10**(-6), 2*10**(-5), 20)
    lers = []
    nr = 10000
    H = makeHgpPcm(steaneH, steaneH)
    for per in pers:
```

```
        print(f"per={per}")
        lers.append(\
            steaneLerCalc(H, nr, per, steanelogicals))
    lers = array(lers)
    std_err = (lers*(1-lers)/nr)**0.5
    errorbar(pers, lers, yerr=std_err)
    plot(pers,pers)
    xlabel("Physical error rate")
    ylabel("Logical error rate")
    savefig("img/figures/steaneLookupThreshold.png")

if __name__ == "__main__":
    main()
```

7.3 Lifting Decoder

```
import networkx as nx
from pymatching import Matching
from numpy import zeros, uint8, linspace, array
from random import random
from random import sample
from typing import List, FrozenSet, Set
from os import makedirs
from os.path import exists
from time import time
from matplotlib.pyplot import figure, savefig, title, show, xlabel, ylabel, legend, errorbar, close, plot
"""
The main function of this file will generate a set of images
of the pertaining color code graph, its dual, and its respective
2- colored subgraphs and print an error prediction on the subgraphs.

A folder "img/hexcolor/" will be created to save image files if it does not exist.
"""

def colorize_graph_black(G: nx.Graph) -> bool:
    """
    Args:
        G(nx.Graph): some graph
    Returns:
        bool: wether it was successful in changing the graph object
```

```
                to all black edges.
    """
    for u, v, attr in G.edges(data=True):
        G[u][v]['color'] = 'black'
    return True

def tor_hex48_color_encode(G: nx.Graph,m: int=6,n: int=4) -> bool:
    """
    Args:
        G(nx.Graph): graph we want to encode with three colored faces
        n,m: how many by how many hexagon, default to 6 and 4 like in delfosse
    Returns:
        bool: Success of graph object modification procedure
    """
    rgb_list = ['r', 'g', 'b']
    # initialize all edge colors to black
    for u, v, attr in G.edges(data=True):
        G[u][v]['color'] = 'black'

    # colorizing algorithm

    # horizontal edges
    for i in range(int(n/2)):
            for j in range(m):
                first_coordinate = (2*i,2*j)
                second_coordinate = (2*i+1,2*j)
                G[first_coordinate][second_coordinate]['color'] = rgb_list[j%3]
    for i in range(int(n/2)):
        for j in range(m):
            first_coordinate = (2*i+1,2*j+1)
            second_coordinate = ((2*i+2)%n,2*j+1)
            G[first_coordinate][second_coordinate]['color'] = rgb_list[(j-1)%3]
    # left ladder edges
    for i in range(int(n/2)):
        for j in range(2*m):
            first_coordinate = (2*i,j)
            second_coordinate = (2*i,(j+1)%(2*m))
            G[first_coordinate][second_coordinate]['color'] = rgb_list[(1-j)%3]
    # right ladder edges
```

```python
    for i in range(int(n/2)):
        for j in range(2*m):
            first_coordinate = (2*i+1,j)
            second_coordinate = (2*i+1,(j+1)%(2*m))
            G[first_coordinate][second_coordinate]['color'] =
    rgb_list[(1-j)%3]
    return True

def make_a_base_graph(m: int=6,n: int=4) -> nx.Graph:
    """
    Args:
        m(int), n(int): desired dimension of faces on graph (
    m by n)
    Returns:
        G(nx.Graph): A basic color code graph of dimensions m
    by n
                    with colored edges encircling opposite
    colored faces.
    """
    G = nx.hexagonal_lattice_graph(m, n, periodic=True)
    colorize_graph_black(G)
    tor_hex48_color_encode(G,m,n)
    for node in G.nodes:
        G.nodes[node]['color'] = 'black'
        G.nodes[node]['fault_ids'] = 0
    return G

def draw_graph_with_colored_edges_and_nodes(G: nx.Graph, file
    : str=None, name: str=None) -> bool:
    """
    Draws a graph who's nodes and edges have colors.
    Options:
        filename (str): save file to specified name (will plt
    .show() otherwise)
        name (str): will create figure with specified name
    """
    pos = nx.get_node_attributes(G, 'pos')
    node_colors = [data['color'] for _, data in G.nodes(data=
    True)]
    edge_colors = [G[u][v]['color'] for u, v in G.edges()]

    figure()
    if name:
        title(name)
    if pos:
```

```
        nx.draw(G, pos, with_labels=True, node_color=
    node_colors, edge_color=edge_colors)
    elif not pos:
        nx.draw(G, with_labels=True, node_color=node_colors,
    edge_color=edge_colors)
    if nx.get_edge_attributes(G,"fault_ids"):
        nx.draw_networkx_edge_labels(G, pos, edge_labels=nx.
    get_edge_attributes(G, "fault_ids"))
    if file:
        savefig(file)
    else:
        show()
    close()
    return True

def flag_color_graph(graph: nx.Graph, per: float=0.1) -> Set[
    any]:
    """
    Args:
        graph(nx.Graph): graph to be altered with errors on
    nodes
        per(float): probability on error occuring on each
    node
    Returns:
        set(node): Actually occured errors
    """
    error_nodes = set()
    for node in graph.nodes:
        if random()< per:
            graph.nodes[node]['fault_ids'] = 1
            graph.nodes[node]['color'] = 'y'
            error_nodes.add(node)
    return error_nodes

def find_6_loops(graph: nx.Graph) -> List[FrozenSet[any]]:
    """
    Args:
        nx.Graph: input graph
    Returns:
        set[frozenset]: Topology of nodes comprising faces on
     input graph """
    cycles = set()
    for node in graph.nodes:
        for node1 in graph.neighbors(node):
            for node2 in graph.neighbors(node1):
```

```
                for node3 in graph.neighbors(node2):
                    for node4 in graph.neighbors(node3):
                        for node5 in graph.neighbors(node4):
                            for node6 in graph.neighbors(node5):
                                if node6 == node:
                                    cycles.add(frozenset([node, node1, node2, node3, node4, node5]))
    faces = [cycle for cycle in cycles if len(cycle) == 6]
    return faces

def find_face_color(graph: nx.Graph, face: FrozenSet) -> str:
    """
    Args:
        graph: graph on which face lies
        face(set[nodes]): face to analyze
    Returns:
        color(str): color of face
    """
    rgb = set(['r','g','b'])
    boundary_colors = set()
    for node in face:
        for node2 in face:
            if node2 in graph.neighbors(node):
                boundary_colors.add(graph[node][node2]['color'])
    face_color = rgb - boundary_colors
    face_color = face_color.pop()
    return face_color

def dual_of_three_colored_graph(graph: nx.Graph):# -> nx.Graph:
    """
    Args:
        graph(nx.Graph): graph of which we want the dual
    Returns:
        dual_graph(nx.Graph): the dual of that graph
    """
    dual_graph = nx.Graph()
    faces = find_6_loops(graph)
    # init nodes
    for i, face in enumerate(faces):
        dual_graph.add_node(i, color = 'black')
        color_of_face = find_face_color(graph, face)
        dual_graph.nodes[i]['color'] = color_of_face
```

```
        # This is the part for error -> syndrome inheritance to the dual graph
        dual_graph.nodes[i]['fault_ids'] = 0
        for node in face:
            if graph.nodes[node]['fault_ids'] == 1:
                dual_graph.nodes[i]['fault_ids'] = (dual_graph.nodes[i]['fault_ids']+1)%2
    # connect nodes
    for i, face in enumerate(faces):
        otherfaces = faces[:i]+faces[((i+1)%(len(faces)+1)):]
        for j, face2 in enumerate(otherfaces):
            lap_nodes = set(face & face2)
            if lap_nodes:
                # A three-colorable graph will only ever have two nodes between two faces
                node1 = lap_nodes.pop()
                node2 = lap_nodes.pop()
                connecting_color = graph[node1][node2]['color']
                # we are iterating not over the list of faces, but over the list of otherfaces
                # what is j?
                second_face_pos = [k for k in range(len(faces)) if faces[k] == otherfaces[j]].pop()
                #
                dual_graph.add_edge(i,second_face_pos, color = connecting_color)


    return dual_graph, faces

def subtile(Graph: nx.Graph, color: str) -> nx.Graph:
    """
    Args:
        Graph(nx.Graph): graph we want to subtile
        color(str): color in format "r","g", "b" of all edges
                    in the subtiling will be comprised
    Returns:
        G(nx.Graph): subtiled graph (does not edit original object)
    """
    G = Graph.copy()
    for edge in G.edges:
        u, v = edge[0], edge[1]
```

```
        if G.edges[u,v]['color'] != color:
            G.remove_edge(u,v)
    G.remove_nodes_from(list(nx.isolates(G)))
    return G

def decode_subtile(graph: nx.Graph) -> List[any]:
    """
    Args:
        graph(nx.Graph): graph with "fault_ids" property on some nodes
    Returns:
        prediction(List[edges]): predicted error edges on graph
    """
    for i, node in enumerate(graph.nodes):
        graph.nodes[node]['og_name'] = node
        graph = nx.relabel_nodes(graph,{node: i})
    matching = Matching(graph)
    # generate syndrome on renamed_copy
    syndrome = zeros(len(graph.nodes), dtype=uint8)
    for node in graph.nodes:
        if graph.nodes[node]['fault_ids'] == 1:
            syndrome[node] = 1
    # predict edges on the renamed_copy
    prediction = matching.decode_to_edges_array(syndrome)
    # rename nodes to be actually usable
    for edge in prediction:
        for i in range(len(edge)):
            edge[i] = graph.nodes[edge[i]]['og_name']
    # revert the graph back to normal
    for node in graph.nodes:
        graph = nx.relabel_nodes(graph, {node: graph.nodes[node]['og_name']})

    return prediction

def make_a_shower(graph: nx.Graph) -> nx.Graph:
    """
    Args:
        graph(nx.Graph): graph we want a yellow syndrome flagged copy of
    Returns:
        shower(nx.Graph): graph with yellow marked syndrome nodes
    """
```

```
    shower = graph.copy()
    for node in shower.nodes:
        if shower.nodes[node]['fault_ids'] == 1:
            shower.nodes[node]['color'] = 'y'
    return shower

def find_hyper_edges(dual_graph: nx.Graph, edges_array_r: List[any],
                     edges_array_g: List[any], edges_array_b: List[any]) -> List[any]:
    """
    Takes: a dualgraph and its subgraph matching edges arrays
    Returns: list of cycles on the dual graph
    """
    # generate the surrounding edges of cycles
    set_of_all_edges_bounding_hyperedge = set()
    for color in [edges_array_r, edges_array_g, edges_array_b]:
        for edge in color:
            addable_edge = tuple(sorted(edge))
            set_of_all_edges_bounding_hyperedge.add(addable_edge)
    # make a graph of only error cycles
    error_bound_graph = dual_graph.copy()
    bad_edges = []
    # this is necessary because we can't modify edges during iteration
    for edge in error_bound_graph.edges:
        if edge not in set_of_all_edges_bounding_hyperedge:
            bad_edges.append(edge)
    error_bound_graph.remove_edges_from(bad_edges)
    isolates = list(nx.isolates(error_bound_graph))
    error_bound_graph.remove_nodes_from(isolates)
    cycles = nx.cycle_basis(error_bound_graph)
    return cycles

def flag_c_graph_specific(graph: nx.Graph, nodes: List[any]) -> bool:
    """
    Flags down specific nodes on a graph from a list of nodes
    """
    for node in nodes:
        graph.nodes[node]['fault_ids'] = 1
        graph.nodes[node]['color'] = 'y'
    return True
```

```
def lift(dual_edge_cycles: List[any], faces: List[FrozenSet])
     -> Set[any]:
    """
    Takes: List of dual graph cycles, facenodes to face map
    Returns: List of enclosed og nodes
    """
    ## Find corresponding nodes in original graph
    enc_nodes = set()
    for dual_edge_cycle in dual_edge_cycles:
        face_on_dec = dual_edge_cycle.pop()
        bounded_nodes = faces[face_on_dec]
        for face in dual_edge_cycle:
            bounded_nodes = bounded_nodes & faces[face]
        bounded_nodes = frozenset(bounded_nodes)
        enc_nodes.add(bounded_nodes)
    # clear up empty frozenset and pop the items to a set
    if frozenset() in enc_nodes:
        enc_nodes.remove(frozenset())
    res = set()
    for enc_node in enc_nodes:
        res.add(next(iter(set(enc_node))))
    return res

def total_decoder(graph: nx.Graph, per: float) -> bool:
    """
    Takes: a color code graph and physical error rate
    Returns: Success of correction operation
    """
    actual_errors = flag_color_graph(graph, per)
    #### dualizing and subtiling
    dual, faces = dual_of_three_colored_graph(graph)
    subr, subg, subb = subtile(dual, 'r'), subtile(dual, 'g')
    , subtile(dual, 'b')
    #### decoding part
    pred_r, pred_g, pred_b = decode_subtile(subr),
    decode_subtile(subg), decode_subtile(subb)
    hyper_edge_cycles = find_hyper_edges(dual, pred_r, pred_g
    , pred_b)
    ## get back to og nodes from dual nodes/ faces
    og_enc_nodes_by_dual_cycles = lift(hyper_edge_cycles,
    faces)
    return og_enc_nodes_by_dual_cycles == actual_errors

def cc_ler_calc(graph: nx.Graph, per: float, nr: int) ->
```

```
    float:
    numErrors = 0
    for _ in range(nr):
        if not total_decoder(graph, per):
            numErrors += 1
    return numErrors/nr

def cc_threshold_plotter(dists: List[any], pers: List[float], nr:int, file=None) -> bool:
    log_errors_all_dist = []
    for d in dists:
        print("Simulating d = {}".format(d))
        origG = make_a_base_graph(d[0],d[1])
        lers = []
        for per in pers:
            print(f"per={per}")
            graph = origG.copy()
            lers.append(cc_ler_calc(graph, per, nr))
        log_errors_all_dist.append(array(lers))
    figure()
    for dist, logical_errors in zip(dists, log_errors_all_dist):
        std_err = (logical_errors*(1-logical_errors)/nr)**0.5
        errorbar(pers, logical_errors, yerr=std_err, label="L={}".format(dist))
    plot(pers, pers, label = 'Threshold')
    xlabel("Physical error rate")
    ylabel("Logical error rate")
    legend(loc=0)
    if file:
        if not exists("img/hexcolor"):
            makedirs("img/hexcolor")
        savefig("img/hexcolor/"+file)
    else:
        show()
    close()
    return True

def main() -> bool:


    #### just making sure image filesaves work
    if not exists("img/hexcolor"):
        makedirs("img/hexcolor")
    #### initialize color code graph with errors
```

```python
    origG = make_a_base_graph()
    actual_errors = flag_color_graph(origG, 0.05)
    ## This is for manually setting faults
    # actual_errors = [(0,0),(1,2)]
    # flag_c_graph_specific(origG, actual_errors)
    #### dualizing and subtiling
    dual, faces = dual_of_three_colored_graph(origG)
    subr, subg, subb = subtile(dual, 'r'), subtile(dual, 'g'), subtile(dual, 'b')
    #### flag syndromes yellow for better visualizing
    dual_syn, subr_syn, subg_syn, subb_syn = make_a_shower(dual), make_a_shower(subr), make_a_shower(subg), make_a_shower(subb)
    #### decoding part
    start = time()
    pred_r, pred_g, pred_b = decode_subtile(subr), decode_subtile(subg), decode_subtile(subb)
    hyper_edge_cycles = find_hyper_edges(dual, pred_r, pred_g, pred_b)
    ## get back to og nodes from dual nodes/ faces
    print("hyp_edge_cycles are: ", hyper_edge_cycles)
    og_enc_nodes_by_dual_cycles = lift(hyper_edge_cycles, faces)
    end = time()
    #### visualizing part
    print(f"This decoding and lifting took {end-start} seconds.")
    draw_graph_with_colored_edges_and_nodes(origG, "img/hexcolor/original.png")
    draw_graph_with_colored_edges_and_nodes(dual_syn, "img/hexcolor/dual.png")
    for i, graph in enumerate([subr_syn, subg_syn, subb_syn]):
        draw_graph_with_colored_edges_and_nodes(graph, f"img/hexcolor/{i}.png")
    if og_enc_nodes_by_dual_cycles:
        for i in range(len(og_enc_nodes_by_dual_cycles)):
            print(f"The {i}th error node on the graph is {og_enc_nodes_by_dual_cycles.pop()}")
    print("The actual errors were: ", actual_errors)
    return True

if __name__ == "__main__":
    main()
```

7.4 Thresholds

7.4.1 Surface/Toric code thresholds

```python
import numpy as np
import matplotlib.pyplot as plt
from pymatching import Matching
from scipy.sparse import hstack, kron, eye, csr_matrix,
    block_diag
from ldpc import mod2

############################ Helper functions
    ######################
def genRepPCM(distance):
    """
    Generates a repetition code parity-check-matrix
    Args:
        distance(Int): distance of the code
    Returns:
        pcm(np.array([[]])): repetition code parity check
    matrix corresponding to distance
    """
    nq = distance   # number of qubits
    na = nq - 1     # number of ancillas
    pcm = np.array([[0 for _ in range(nq)] for _ in range(na)
    ])
    for i in range(na):
        pcm[i][i] = 1
        pcm[i][(i+1) % nq] = 1
    return pcm

def genRingPCM(distance):
    """
    Generates a ring code parity-check-matrix

    Args:
        distance(Int): distance of

    Returns:
        pcm(np.array([[]])): generated parity check matrix of
     distance
    """
    pcm=np.eye(distance)
    for i in range(distance):
        pcm[i][(i+1)%distance] = 1
```

```
    return pcm

def ring_code(n):
    """
    scipy sparse Parity check matrix of a ring code with length n.
    """
    return csr_matrix(genRingPCM(n))

def rep_code(n):
    """
    scipy sparse Parity check matrix of a rep code with n qubits
    """
    return csr_matrix(genRepPCM(n))

def genXStabilizers(first_pcm_generator, second_pcm_generator, dist):
    """
    check matrix for the X stabilizers of a hypergraph product code of distance dist
    """
    H1 = first_pcm_generator(dist)
    H2 = second_pcm_generator(dist)
    H = hstack(
        [kron(H1, eye(H2.shape[1])), kron(eye(H1.shape[0]), H2.T)],
        dtype=np.uint8
    )
    return H

def genZStabilizers(first_pcm_generator, second_pcm_generator, dist):
    """
    check matrix for the Z stabilizers of a hypergraph product code of distance dist
    """
    H1 = first_pcm_generator(dist)
    H2 = second_pcm_generator(dist)
    H = hstack(
            [kron(eye(H1.shape[1]), H2), kron(H1.T, eye(H2.shape[0]))],
            dtype=np.uint8
        )
    return H
```

```python

def genHxHz(first_code, second_code, d):
    """
    generates Hx and Hz of a hgp code from two codes
    """
    Hx = genXStabilizers(first_code, second_code, d).todense
    ()
    Hz = genZStabilizers(first_code, second_code, d).todense
    ()
    # Hx = np.hstack((Hx, np.zeros(Hx.shape, dtype=np.uint8))
    )
    # Hz = np.hstack((np.zeros(Hz.shape, dtype=np.uint8), Hz)
    )

    return Hx, Hz

def compute_lz(hx,hz):
            #lz logical operators
            #lz\in ker{hx} AND \notin Im(Hz.T)
            # hx = hx.todense()
            # hz = hz.todense()
            ker_hx=mod2.nullspace(hx) #compute the kernel
    basis of hx
            im_hzT=mod2.row_basis(hz) #compute the image
    basis of hz.T

            #in the below we row reduce to find vectors in kx
     that are not in the image of hz.T.
            log_stack=np.vstack([im_hzT,ker_hx])
            pivots=mod2.row_echelon(log_stack.T)[3]
            log_op_indices=[i for i in range(im_hzT.shape[0],
    log_stack.shape[0]) if i in pivots]
            log_ops=log_stack[log_op_indices]
            return log_ops

def calc_logicals(hx, hz):
    """ calculates actual logical operators from two parity
    check matrices of
    codes generating a hgp code
    """

    lx = compute_lz(hz, hx)
    lz = compute_lz(hx, hz)
    lx=np.vstack((np.zeros(lz.shape,dtype=np.uint8),lx))
    lz=np.vstack((lz,np.zeros(lz.shape,dtype=np.uint8)))
```

```
    # temp = mod2.inverse(lx@lz.T %2)
    # lx   = temp@lx % 2
    return np.hstack((lx, lz))

def makeHgpPcm(Hx, Hz):
    """
    Makes a full parity check matrix including x and z checks for a
    hypergraph product code of two other codes
    """
    # Hx = genXStabilizers(first_code, second_code, d).todense()
    # Hz = genZStabilizers(first_code, second_code, d).todense()
    Hx = np.hstack((Hx, np.zeros(Hx.shape, dtype=np.uint8)))
    Hz = np.hstack((np.zeros(Hz.shape, dtype=np.uint8), Hz))
    H = np.vstack((Hx, Hz))
    return csr_matrix(H)

############################ Hotstuff ############################
def lerCalc(H, logicals, nr=1000, per = 0.3):
    "calculates logical error rate assuming a noise model of p/3 X,Y,Z errors"
    matching = Matching.from_check_matrix(H)#, faults_matrix=logicals)
    numErrors = 0
    for _ in range(nr):
        noise = np.zeros(H.shape[1], dtype=np.uint8)
        halflength = int(len(noise)/2)
        for i in range(halflength):
            # this is physical X errors, editing first half of entries
            if np.random.rand() < per/3:
                noise[i] = (noise[i]+1) % 2
            # this is physical Z errors, editing second half of entries
            if np.random.rand() < per/3:
                noise[i+halflength] = (noise[i+halflength] + 1) % 2
            # this is physical Y errors, assuming same syndrome as X and Z implies same error
            if np.random.rand() < per/3:
                noise[i] = (noise[i]+1) % 2
                noise[i+halflength] = (noise[i+halflength] +
```

```
1) % 2
        noise = csr_matrix(noise)
        noise = noise.T
        syndrome = csr_matrix(((H@noise).todense() % 2))
        prediction = csr_matrix(matching.decode(syndrome.todense())).T
        predicted_flips = (logicals@prediction).todense() % 2
        actualLflips = (logicals@noise).todense() % 2
        if not np.array_equal(actualLflips, predicted_flips):
            numErrors += 1
    return numErrors/nr

def thresholdPlotter(dists, pers, nr, first_code, second_code, codename):
    """
    plots logical error rates of a quantum code with a list of distances
    and physical error rates
    """
    np.random.seed(2)
    log_errors_all_dist = []
    for d in dists:
        print("Simulating d = {}".format(d))
        Hx, Hz = genHxHz(first_code, second_code, d)
        H = makeHgpPcm(Hx, Hz)
        logicals = csr_matrix(calc_logicals(Hx, Hz))
        lers = []
        for per in pers:
            print(f"per={per}")
            lers.append(lerCalc(H, logicals, nr, per))
        log_errors_all_dist.append(np.array(lers))
    plt.figure()
    for dist, logical_errors in zip(dists, log_errors_all_dist):
        std_err = (logical_errors*(1-logical_errors)/nr)**0.5
        plt.errorbar(pers, logical_errors, yerr=std_err, label="distance {}".format(dist))
    plt.xlabel("Physical error rate")
    plt.ylabel("Logical error rate")
    plt.legend(loc=0)
    plt.savefig(codename)

def main():
    dists = range(5,20,4)
    pers = np.linspace(0.01, 0.32, 32)
```

```
    nr = 30000
    print("Thresholding the surface code...")
    thresholdPlotter(dists, pers, nr, rep_code, rep_code, "
    surfaceThresholdOverview.png")
    print("Thresholding the toric code...")
    thresholdPlotter(dists, pers, nr, ring_code, ring_code, "
    toricThresholdOverview.png")
    print("Thresholding the cylindric code...")
    thresholdPlotter(dists, pers, nr, rep_code, ring_code, "
    cylinderThresholdOverview.png")

if __name__ == "__main__":
    main()
```

7.4.2 Color code thresholds

```
from betterlookup import genSteaneLookupTable, findMinWeight,
     findMinWeight
from random import random
from numpy import zeros, uint8, concatenate, array,\
    array_equal, linspace, vstack, hstack, zeros, ndarray,
    logspace
from matplotlib.pyplot import errorbar, legend, \
    savefig, xlabel, ylabel, plot

def genSteaneError(per)->ndarray:
    """ Generates an error vector on the Steane code"""
    empty7 = zeros(7, dtype=uint8)
    xerror = empty7.copy()
    zerror = empty7.copy()
    for i in range(len(xerror)):
        if random()<per:
            xerror[i] = 1
    for j in range(len(zerror)):
        if random()<per:
            zerror[j] = 1
    yerror = concatenate((xerror,zerror))
    for k, bit in enumerate(yerror[:6]):
        if random()<per:
            yerror[k] = (yerror[k] + 1)%2
            yerror[2*k] = (yerror[2*k]+1)%2
    return yerror

def steaneLerCalc(steaneH, nr, per, logicals)->float:
    """Calculates the logical error rate of the steane
    code decoded with a lookup table"""
```

```
    numErrors = 0
    looktable = genSteaneLookupTable()
    for _ in range(nr):
        actual_error = genSteaneError(per)
        syndrome = steaneH@actual_error %2
        predictions = looktable[tuple(syndrome)]
        pred = findMinWeight(predictions)
        pred_L_flips = logicals@pred %2
        actual_L_flips = logicals@actual_error %2
        if not array_equal(actual_L_flips, pred_L_flips):
            numErrors += 1
    return numErrors/nr

def makeHgpPcm(Hx, Hz)->ndarray:
    """
    Makes a full parity check matrix including x and z
    checks for a hypergraph product code of two other codes
    """
    Hx = hstack((Hx, zeros(Hx.shape, dtype=uint8)))
    Hz = hstack((zeros(Hz.shape, dtype=uint8), Hz))
    H = vstack((Hx, Hz))
    return H

def main():
    steanelogicals = \
        array([\
            [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1]])
    steaneH = array([[1, 0, 0, 1, 0, 1, 1],
                     [0, 1, 0, 1, 1, 0, 1],
                     [0, 0, 1, 0, 1, 1, 1]])
    pers = linspace(2*10**(-6), 2*10**(-5), 20)
    lers = []
    nr = 10000
    H = makeHgpPcm(steaneH, steaneH)
    for per in pers:
        print(f"per={per}")
        lers.append(\
            steaneLerCalc(H, nr, per, steanelogicals))
    lers = array(lers)
    std_err = (lers*(1-lers)/nr)**0.5
    errorbar(pers, lers, yerr=std_err)
    plot(pers,pers)
    xlabel("Physical error rate")
    ylabel("Logical error rate")
```

```python
    savefig("img/figures/steaneLookupThreshold.png")

if __name__ == "__main__":
    main()
```

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Physik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Valinhos, SP, Brasilien, den ____________________ Unterschrift: ______________________