Bachelor Thesis
Decoding the Color Code

Clemens Schumann,
Advised by Peter-Jan Derks

March 27, 2023

Contents

1 Overview of QEC algebra

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 controlled EM waves, i.e. lasers.
In this thesis, we will focus on *qubit*-based systems, i.e. two-level quantum systems as base units of computation.

In the following, we will analyze quantum circuit diagrams using the different pictures of quantum mechanics. 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 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:
  $(X \otimes \mathbb{I} \otimes \mathbb{I})(|\psi_1\rangle \otimes |\psi_2\rangle \otimes |\psi_3\rangle)$

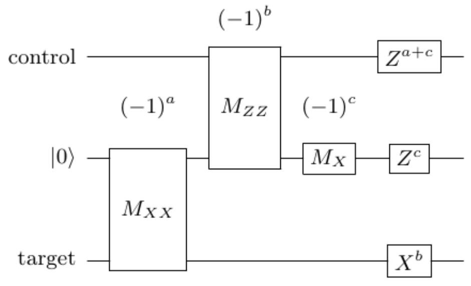


Figure 1: A Quantum Circuit to implement a measurement based Controlled-$X_{|\psi\rangle_{control} \to |\psi\rangle_{target}}$ Gate, where $|0\rangle$ is the +1 eigenstate in σ_z-basis.

1.1 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 qubits 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 Appendix 6.1 , 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 [1], finding that it results in an equal output.

1.2 Heisenberg picture and stabilizer formalism

1.2.1 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 Figureit 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:

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

1.2.2 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 $S'_G = \langle AS_0A^\dagger, ..., AS_nA^\dagger \rangle$.

1.2.3 Effect of measurements on stabilizers

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. for $N \in S_G \cup \overline{X} \cup \overline{Z}$: if $\{N, M\} = 0$: replace N with SN

where $\overline{X}$ and $\overline{Z}$ are the sets of logical X and Z operators respectively. (a logical operator is an operator which acts on a systems metastructure which can be treated as its own qubit)

1.2.4 Circuit Analysis in Stabilizer formalism

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 becomes $\mathbb{I}^{\otimes n} + M$, in the second it becomes $\mathbb{I}^{\otimes n} - M$. Therefore, in the circuit shown in figure 1 , the measurements project onto:

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

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

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

In the following stabilizers will be written without the tensor product symbols, so in our case the stabilizer is initially: $S_G^0 = \langle IZI \rangle$, the logical $\overline{X}$ operator is XXX and the logical $\overline{Z}$ operator is ZIZ.

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).

For the logical operators, if $\overline{X}$ or $\overline{Z}$ do not commute with the measurement operator, we know that their product with another anticommuting operator from the previous stabilizer must then commute with the measurement operator: $[S_{prev}\overline{X}M, MS_{prev}\overline{X}] = 0$ (recall the previous statement that all Paulis and their tensor products must either commute or anti-commute).

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 $\cdot$ 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 $\overline{Z} : ZIZ \rightarrow IIZ$.

Since CNOT maps: $Z \otimes Z \mapsto I \otimes Z$, this implements a logical CNOT from the first to the third qubit.

1.3 The Clifford Gates

It has been proven in *reference source* that operators that map a state stabilized by some member of the Pauli-Group to a state stabilized by another member of the Pauli Group can be simulated efficiently on a classical computer. The Group of operators that satisfy this condition is called the Clifford Group.

For the decoder we wish to implement in this thesis, it therefore makes sense to focus on those first and foremost, as applying corrective gates is a computationally/ experimentally expensive task that should be put off until the latest possible moment, and the propagation of an error until then can be simulated efficiently. Also, the pauli-stabilizers utilized to construct our encoding schemes only stabilize clifford and pauli gates, therefore these encoding schemes *can't* protect against non-clifford/pauli operator errors.

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 Error detection and correction

2.1 Classical codes

The concept of (classical) error-correcting codes (ECC) was introduced by Claude Shannon in 1948. 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).

Two such classical ECCs are the repetition and the ring code. We call these codes linear because their graph representations are linear, or one-dimensional. 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. d arbitrary individual errors being corrigible. In the following, I will refer to linear or 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.

2.1.1 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 1/2. 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

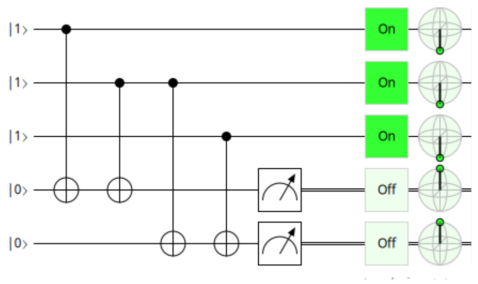


Figure 2: Bitflip Syndrome extractor for [[3,1,$\frac{1}{2}$]] repetition code
+1 measurement result on first ancilla indicates a bitflip error on qubits 1 or 2, +1 result on second ancilla indicates bitflip on second or third qubit

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:

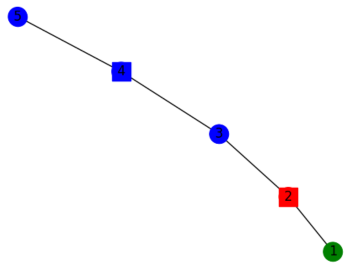


Figure 3: Graph for [[3,1,$\frac{1}{2}$]] repetition code with error on node 1 marked in green and resulting syndrome marked red. Squares represent ancilla qubits and circles represent data qubits.

2.1.2 Ringcode

The ring code's graph essentially just loops around at the repetition code's single-edged ancilla nodes, so for example its edge matrix where the nth row represents which data qubit is connected to the nth ancilla qubit. This matrix for a three-qubit system looks like:

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

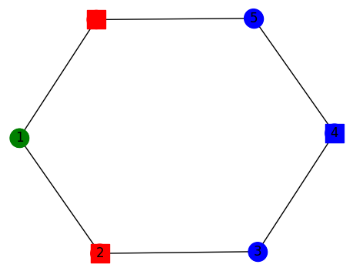


Figure 4: Graph for [[3,1,$\frac{1}{2}$]] ring code with error on node 1 marked in green and resulting syndrome marked in red. Squares represent ancilla qubits and circles represent data qubits.

2.2 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 single qubit states have an infinite amount of differing states to it, 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 in state from one normalized one to another is merely a sum of two rotations.

Fortunately, 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 same operator again 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.

2.3 Topological codes

Previous research in computer science provides a toolset for generating valid codes from existing encoding schemes. Hypergraph product codes, introduced by Tillich and Zémor, 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:

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

2.3.1 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 [2]. 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. Unlike in figure 5, it is possible to draw this graph without colors, 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 wil be done for the following toric code.

INCLUDE LOGICAL OPERATORS

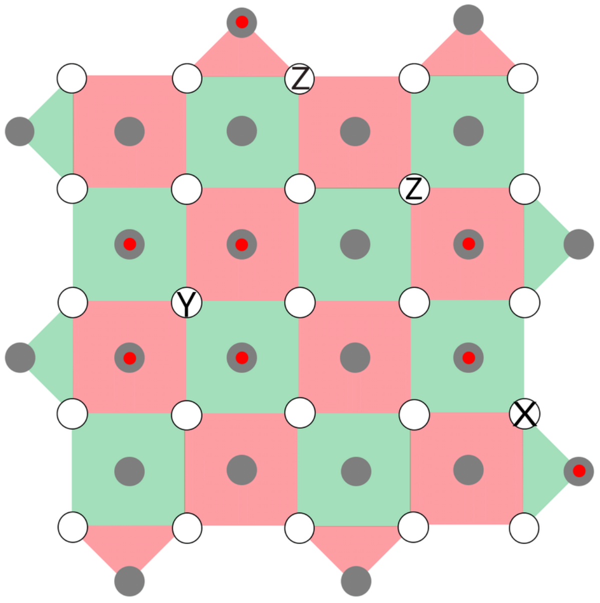


Figure 5: Distance 5 Surface code with data qubits in white and ancilla qubits in grey. Green Faces represent Z stabilizers and Red faces represent X stabilizers. Errors on data qubits are marked by respective Pauli names and violated stabilizers are marked in red.

2.3.2 Toric code

Similarly, a hypergraph product code of two ring codes can be generated. We call this code the "Toric code".

Notably, this resembles a donut, or torus. 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.

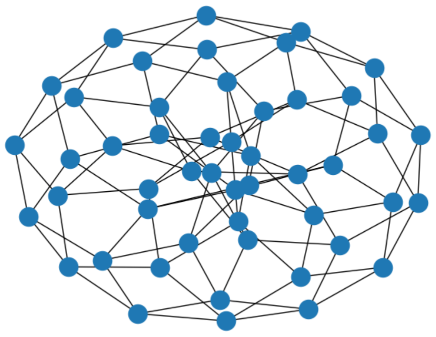
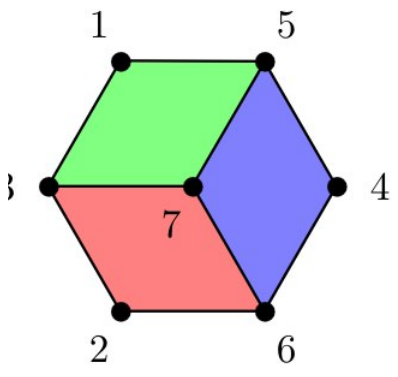


Figure 6: Graph for [[49,1,7]] toric code

2.3.3 Color code

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



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

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

3 Decoding Schemes

3.1 Decoders for Surface/Toric codes

The Syndromes on the surface/toric code are a set of nodes and faces on the code 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 ringcode, 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.

3.1.1 MWPM decoding

1. Find a set of unmatched nodes that can be reached from the matching 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. 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 [4].

3.1.2 Union Find decoder

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.

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 [5]. For this reason, a MWPM algorithm was chosen for decoding the toric subgraphs of the color code in our lifting decoder thresholding in chapter 3.2.2.

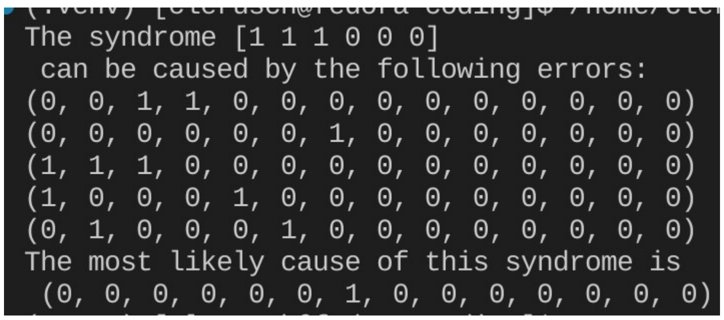
3.2 Color code decoders

Unlike the surface and toric codes, in the color code the data qubits sit on the graphs nodes, and the ancillas on the graphs faces. Decoding the color code entails matching three differently colored faces to its enclosed nodes. This is a significantly more challenging task than decoding the 2D-codes, since three-colored graph matching is a confirmed NP-hard problem. (reference delfosse paper)

3.2.1 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. A big issue with this decoding scheme is that generating lookup tables is extremely computationally expensive ($O(2^n)$). This renders it practically unfeasible to generate lookup tables for codes with a larger number of total data qubits.



```
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 8: Lookup table for an X error on the central qubit of a Steane code (qubit 7), generating code can be found in Appendix 7.1

In figure 8 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 reaction, and the second three bits indicating the Steane code faces Z reaction. The lookup table will return a set of many possible errors resulting in that syndrome, but simply choosing the one with the least number of errors (minimum weight) gives the correct error prediction.

3.2.2 Lifting decoder

The Lifting decoder works as follows:

- Create dual of graph
- Generate single-edge-colored subgraphs of the dual
- Decode subgraphs using MWPM/Union-Find
- Unify all edges from subgraph corrections
- Find all shortest-length loops on this union

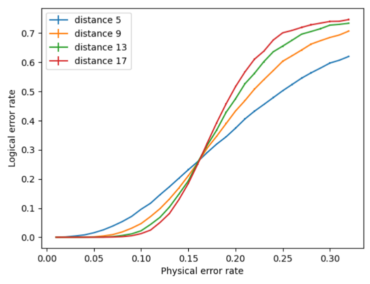
- NOLIFTING CASE ????
- All nodes bounded by the faces that are elements of the shortest-length loop sets are error nodes.

The reason for this seemingly complicated procedure is that, on its own, 3-colored matching would be an NP-hard problem. However, 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 [6]. The polynomial time complexity of the lifting decoder does not violate the NP-hardness of the 3-color matching problem, since the lifting procedure can also fail (NOLIFTING case).

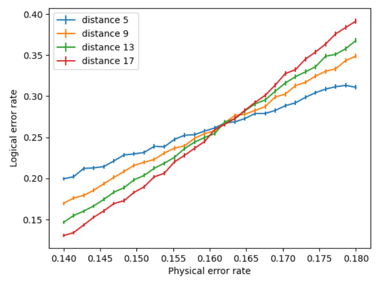
4 Thresholds

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 $16.3 \pm 0.5\%$ from figure 9 and $16.0 \pm 0.5\%$ from figure 10 for the surface and toric code respectively. Their thresholds are within single error margins of each other, and can therefore be called identical.

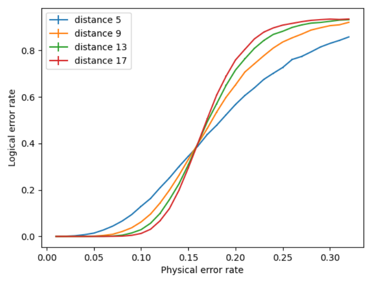


(a) Surface code MWPM thresholding overview

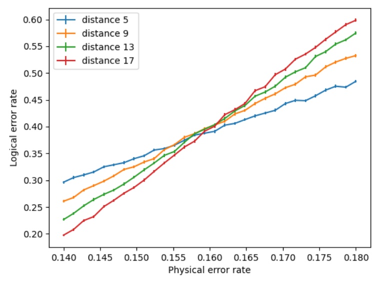


(b) Detailed view for precise threshold determination

Figure 9: Thresholding of the surface code using the MWPM decoder implemented in the PyMatching [4] library. Generating code can be found in Appendix 7.3.1



(a) Toric code MWPM thresholding overview



(b) Detailed view for precise threshold determination

Figure 10: Thresholding of the Toric code using the MWPM decoder implemented in the PyMatching [4] library. Generating code can be found in Appendix 7.3.1

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 is somehow very bad. I do not know why.

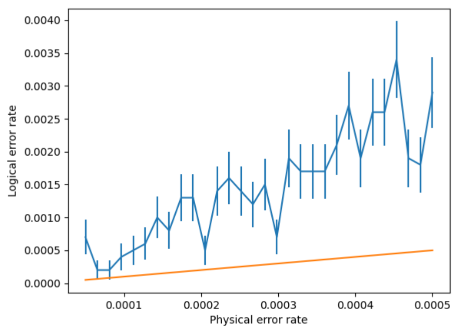


Figure 11: Lookup table pseudo threshold for the Steane code, generating code can be found in Appendix 7.2

On the color code, we found a threshold of $< 0.0001\%$ using the lookup table decoder on the Steane code and could not find a threshold for the scalable hexagonal toric color code, as our code does not work there yet.

5 Conclusion

6 Appendix

6.1 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 $CNOT_{|\psi_{Control}\rangle\rightarrow|\psi_{Target}\rangle}|\phi_{t=0}\rangle$.
If the last measurement result is -1:

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

Notably, each measurement sequence has a differing resulting ancilla state, however we do not care since ancillas are meant to be discarded.
For now, the other 7 final computation steps are left as an exercise to the reader, however I probably will still finish that.

7 Lookup table decoding

7.1 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 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
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()
```

```python
    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
    error = (concatenate((xerror, empty7)) + yerror \
        + concatenate((empty7, zerror)))%2
    return error

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(5/100000, 5/10000, 30)
    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 Thresholds

7.3.1 Surface/Toric code thresholds

```
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

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()
```

7.3.2 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
from matplotlib.pyplot import errorbar, legend, \
    savefig, xlabel, ylabel, plot

```

```python
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
    error = (concatenate((xerror, empty7)) + yerror \
        + concatenate((empty7, zerror)))%2
    return error

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(5/100000, 5/10000, 30)
    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()
```

References

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

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

[3] 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

[4] 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

[5] 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

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