

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 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 ECC codes 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.

An attempt was made at constructing a 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 small 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	4
2	Background	5
2.1	Schroedinger picture	6
2.2	Heisenberg picture and stabilizer formalism	7
2.2.1	Stabilizer group	7
2.2.2	Effect of gates on stabilizers	8
2.2.3	Effect of measurements on stabilizers	8
2.2.4	Circuit Analysis in Stabilizer formalism	9
3	Error detection and correction	11
3.1	Classical codes	11
3.1.1	Repetition code	11
3.1.2	Ringcode	13
3.2	Quantum Error Model	14
3.3	Topological codes	14
3.3.1	Surface code	15
3.3.2	Toric code	16
3.3.3	Color code	17
4	Decoding Schemes	18
4.1	Decoders for Surface/Toric codes	18
4.1.1	MWPM decoding	18
4.1.2	Union-Find decoder	19
4.2	Color code decoders	19
4.2.1	Lookup table decoding	20
4.2.2	Lifting decoder	21
5	Thresholds	22
6	Conclusion	25
7	Appendix	28
7.1	Schroedinger picture calculation of CNOT circuit	28
7.2	Lookup table decoding	30
7.2.1	Table generation	30
7.2.2	Thresholding	31

7.3	Lifting Decoder	33
7.4	Thresholds	44
7.4.1	Surface/Toric code thresholds	44
7.4.2	Color code thresholds	49

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 can therefore with high confidence be said 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 wide 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.

2 Background

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 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:

$$\begin{aligned} &\alpha|00\rangle + \beta|01\rangle + \delta|10\rangle + \gamma|11\rangle \\ &\mapsto \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle \end{aligned}$$

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:

$$|\psi\rangle = |\psi(t)\rangle \quad (1)$$

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

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

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:

$$M_Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad (3)$$

And that measurement would perform a projection P_Z :

$$P_Z^+ = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \text{ or } P_Z^- = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \quad (4)$$

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

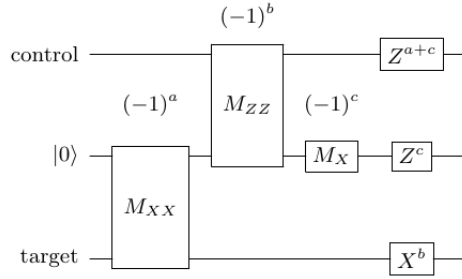


Figure 1: A Quantum Circuit to implement a measurement based Controlled- $X_{|\psi\rangle_{control} \rightarrow |\psi\rangle_{target}}$ Gate, where $|0\rangle$ is the $+1$ eigenstate in σ_z -basis.

can be seen explicitly calculated in the Schroedinger picture in 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:

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

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

$$S_G = \langle S_0, \dots, S_n \rangle : S|\psi_{in}\rangle = |\psi_{in}\rangle \forall S \in S_G \quad (6)$$

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:

$$\forall A, B \in S : AB|\psi\rangle = BA|\psi\rangle = |\psi\rangle \Rightarrow [A, B]|\psi\rangle = 0 \quad (7)$$

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:

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

So we now know that the post-gate state is an eigenstate of S' .

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

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 acts on a systems metastructure that can be treated as its own qubit.

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: $S_G^0 = \langle IZI \rangle$, the logical \overline{X} operator is XXX and the logical \overline{Z} operator is ZIZ.

In the circuit shown in Figure 1, the measurements project onto:

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

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

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

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 :

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

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 \bar{X} and \bar{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 \bar{Z} operator anticommutes with the measurement, so is replaced by $\bar{Z}_3=ZZI \cdot ZIZ = IIZ$. The logical \bar{X} is unaffected since it commutes with the measurement operator.

The stabilizer for the control and target qubit is still identity, and logical $\bar{Z} : ZIZ \rightarrow IIZ$.

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

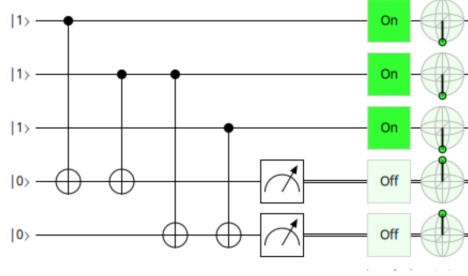


Figure 2: Bitflip Syndrome extractor for $[[3,1,\frac{1}{2}]]$ repetition code
+1 measurement result on first ancilla indicates a bitflip error on qubits 1 or 2, +1 result on second ancilla indicates bitflip on second or third qubit

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:

$$M_{pc} \cdot \vec{v}_{error} = \vec{v}_{syndrome} \quad (13)$$

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

$$M_{pc3} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \quad (14)$$

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:

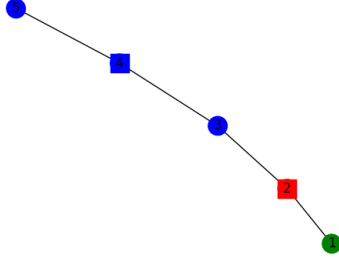


Figure 3: Graph for $[[3,1,\frac{1}{2}]]$ repetition code with error on node 1 marked in green and resulting syndrome marked red. Squares represent ancilla qubits and circles represent data qubits.

3.1.2 Ringcode

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 n th row represents which data qubit is connected to the n th ancilla qubit is the following:

$$M_{pc3} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \quad (15)$$

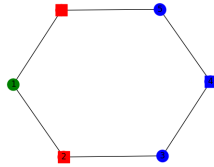


Figure 4: Graph for $[[3,1,\frac{1}{2}]]$ ring code with error on node 1 marked in green and resulting syndrome marked in red. Squares represent ancilla qubits and circles represent data qubits.

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:

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

With an appropriate 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:

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

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 once, and therefore takes the message back to the codespace.

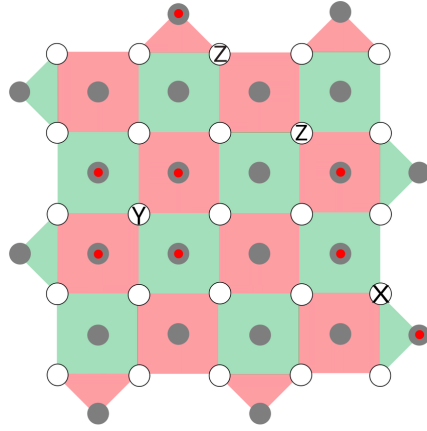


Figure 5: Distance 5 Surface code with data qubits in white and ancilla qubits in grey. Green Faces represent Z stabilizers and Red faces represent X stabilizers. Errors on data qubits are marked by respective Pauli names and violated stabilizers are marked in red.

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.

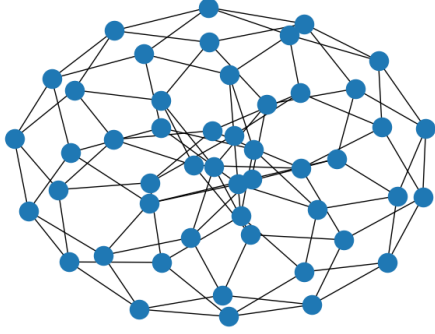


Figure 6: Tanner graph for $[[49,1,7]]$ toric code.

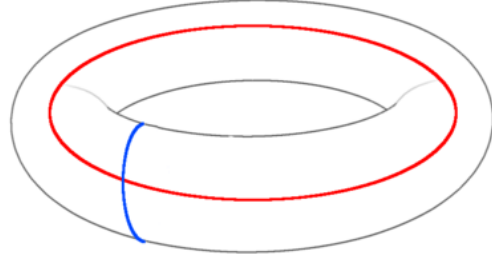


Figure 7: Logical \bar{X} and \bar{Z} operators on toric code Tanner graph. Image courtesy of James Wooton's contribution to Wikipedia.

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 [8].

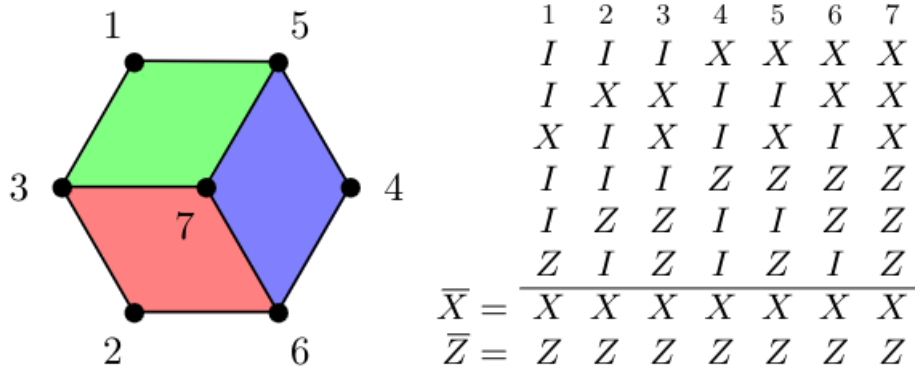


Figure 8: Graph for the $[[7,1,3]]$ color code, also known as the Steane code, and its stabilizers. Figure from [8].

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 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.

4.1.1 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, i.e. it 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 [9].

4.1.2 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 [10]. 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 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 optimal three-colored graph matching is a confirmed NP-hard problem [11].

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.

```
(.venv) [eterasch@redora coding]$ ./home/ete
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, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)
(1, 1, 1, 0, 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)
(0, 1, 0, 0, 0, 1, 0, 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, 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

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.

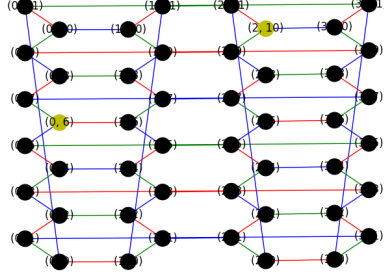
The lookup table will return a set of many possible errors resulting in that syndrome, but simply choosing the one with the least number of errors (minimum weight) gives the correct error prediction.

4.2.2 Lifting decoder

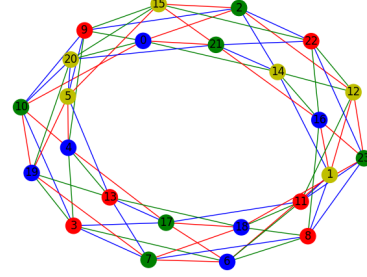
The Lifting decoder works as follows:

- Create dual of color code 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
- If there are remaining edges not forming a loop, abort the lifting procedure
- All nodes bounded by the faces that are elements of the shortest-length loop sets are error nodes.

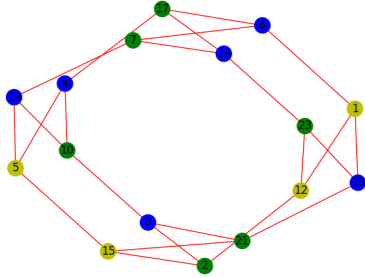
By sub-tiling the graph into smaller subgraphs, we can reduce the problem of decoding e.g. a honeycomb lattice toric color code to a set of MWPM-decodable toric graphs that merely need to be "lifted" into a combination of subgraph decodings to decode the original color code graph [11]. This decoding 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.



(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.

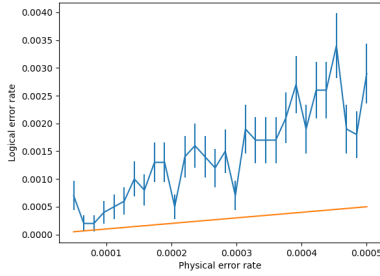


(c) Red subgraph to be decoded via MWPM

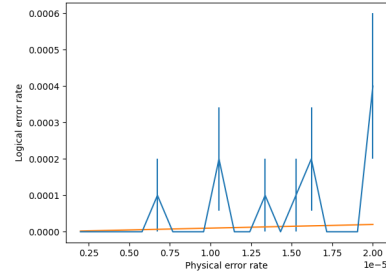
```
(.venv) [clerusch@fedora coding]$ /home/clerusch/Tinkering/colorcode/python/hexcolor.py
hyp_edge_cycles are: [[12, 14, 1], [15, 20, 12], [1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18], [19, 20, 21], [22, 23, 24], [25, 26, 27], [28, 29, 30], [31, 32, 33], [34, 35, 36], [37, 38, 39], [40, 41, 42], [43, 44, 45], [46, 47, 48], [49, 50, 51], [52, 53, 54], [55, 56, 57], [58, 59, 60], [61, 62, 63], [64, 65, 66], [67, 68, 69], [70, 71, 72], [73, 74, 75], [76, 77, 78], [79, 80, 81], [82, 83, 84], [85, 86, 87], [88, 89, 90], [91, 92, 93], [94, 95, 96], [97, 98, 99], [100, 101, 102], [103, 104, 105], [106, 107, 108], [109, 110, 111], [112, 113, 114], [115, 116, 117], [118, 119, 120], [121, 122, 123], [124, 125, 126], [127, 128, 129], [130, 131, 132], [133, 134, 135], [136, 137, 138], [139, 140, 141], [142, 143, 144], [145, 146, 147], [148, 149, 150], [151, 152, 153], [154, 155, 156], [157, 158, 159], [160, 161, 162], [163, 164, 165], [166, 167, 168], [169, 170, 171], [172, 173, 174], [175, 176, 177], [178, 179, 180], [181, 182, 183], [184, 185, 186], [187, 188, 189], [190, 191, 192], [193, 194, 195], [196, 197, 198], [199, 200, 201], [202, 203, 204], [205, 206, 207], [208, 209, 210], [211, 212, 213], [214, 215, 216], [217, 218, 219], [220, 221, 222], [223, 224, 225], [226, 227, 228], [229, 230, 231], [232, 233, 234], [235, 236, 237], [238, 239, 240], [241, 242, 243], [244, 245, 246], [247, 248, 249], [250, 251, 252], [253, 254, 255], [256, 257, 258], [259, 260, 261], [262, 263, 264], [265, 266, 267], [268, 269, 270], [271, 272, 273], [274, 275, 276], [277, 278, 279], [280, 281, 282], [283, 284, 285], [286, 287, 288], [289, 290, 291], [292, 293, 294], [295, 296, 297], [298, 299, 300], [301, 302, 303], [304, 305, 306], [307, 308, 309], [310, 311, 312], [313, 314, 315], [316, 317, 318], [319, 320, 321], [322, 323, 324], [325, 326, 327], [328, 329, 330], [331, 332, 333], [334, 335, 336], [337, 338, 339], [340, 341, 342], [343, 344, 345], [346, 347, 348], [349, 350, 351], [352, 353, 354], [355, 356, 357], [358, 359, 360], [361, 362, 363], [364, 365, 366], [367, 368, 369], [370, 371, 372], [373, 374, 375], [376, 377, 378], [379, 380, 381], [382, 383, 384], [385, 386, 387], [388, 389, 390], [391, 392, 393], [394, 395, 396], [397, 398, 399], [400, 401, 402], [403, 404, 405], [406, 407, 408], [409, 410, 411], [412, 413, 414], [415, 416, 417], [418, 419, 420], [421, 422, 423], [424, 425, 426], [427, 428, 429], [430, 431, 432], [433, 434, 435], [436, 437, 438], [439, 440, 441], [442, 443, 444], [445, 446, 447], [448, 449, 450], [451, 452, 453], [454, 455, 456], [457, 458, 459], [460, 461, 462], [463, 464, 465], [466, 467, 468], [469, 470, 471], [472, 473, 474], [475, 476, 477], [478, 479, 480], [481, 482, 483], [484, 485, 486], [487, 488, 489], [490, 491, 492], [493, 494, 495], [496, 497, 498], [499, 500, 501], [502, 503, 504], [505, 506, 507], [508, 509, 510], [511, 512, 513], [514, 515, 516], [517, 518, 519], [520, 521, 522], [523, 524, 525], [526, 527, 528], [529, 530, 531], [532, 533, 534], [535, 536, 537], [538, 539, 540], [541, 542, 543], [544, 545, 546], [547, 548, 549], [550, 551, 552], [553, 554, 555], [556, 557, 558], [559, 560, 561], [562, 563, 564], [565, 566, 567], [568, 569, 570], [571, 572, 573], [574, 575, 576], [577, 578, 579], [580, 581, 582], [583, 584, 585], [586, 587, 588], [589, 590, 591], [592, 593, 594], [595, 596, 597], [598, 599, 600], [601, 602, 603], [604, 605, 606], [607, 608, 609], [610, 611, 612], [613, 614, 615], [616, 617, 618], [619, 620, 621], [622, 623, 624], [625, 626, 627], [628, 629, 630], [631, 632, 633], [634, 635, 636], [637, 638, 639], [640, 641, 642], [643, 644, 645], [646, 647, 648], [649, 650, 651], [652, 653, 654], [655, 656, 657], [658, 659, 660], [661, 662, 663], [664, 665, 666], [667, 668, 669], [670, 671, 672], [673, 674, 675], [676, 677, 678], [679, 680, 681], [682, 683, 684], [685, 686, 687], [688, 689, 690], [691, 692, 693], [694, 695, 696], [697, 698, 699], [700, 701, 702], [703, 704, 705], [706, 707, 708], [709, 710, 711], [712, 713, 714], [715, 716, 717], [718, 719, 720], [721, 722, 723], [724, 725, 726], [727, 728, 729], [730, 731, 732], [733, 734, 735], [736, 737, 738], [739, 740, 741], [742, 743, 744], [745, 746, 747], [748, 749, 750], [751, 752, 753], [754, 755, 756], [757, 758, 759], [760, 761, 762], [763, 764, 765], [766, 767, 768], [769, 770, 771], [772, 773, 774], [775, 776, 777], [778, 779, 780], [781, 782, 783], [784, 785, 786], [787, 788, 789], [790, 791, 792], [793, 794, 795], [796, 797, 798], [799, 800, 801], [802, 803, 804], [805, 806, 807], [808, 809, 810], [811, 812, 813], [814, 815, 816], [817, 818, 819], [820, 821, 822], [823, 824, 825], [826, 827, 828], [829, 830, 831], [832, 833, 834], [835, 836, 837], [838, 839, 840], [841, 842, 843], [844, 845, 846], [847, 848, 849], [850, 851, 852], [853, 854, 855], [856, 857, 858], [859, 860, 861], [862, 863, 864], [865, 866, 867], [868, 869, 870], [871, 872, 873], [874, 875, 876], [877, 878, 879], [880, 881, 882], [883, 884, 885], [886, 887, 888], [889, 890, 891], [892, 893, 894], [895, 896, 897], [898, 899, 900], [901, 902, 903], [904, 905, 906], [907, 908, 909], [910, 911, 912], [913, 914, 915], [916, 917, 918], [919, 920, 921], [922, 923, 924], [925, 926, 927], [928, 929, 930], [931, 932, 933], [934, 935, 936], [937, 938, 939], [940, 941, 942], [943, 944, 945], [946, 947, 948], [949, 950, 951], [952, 953, 954], [955, 956, 957], [958, 959, 960], [961, 962, 963], [964, 965, 966], [967, 968, 969], [970, 971, 972], [973, 974, 975], [976, 977, 978], [979, 980, 981], [982, 983, 984], [985, 986, 987], [988, 989, 990], [991, 992, 993], [994, 995, 996], [997, 998, 999], [1000, 1001, 1002], [1003, 1004, 1005], [1006, 1007, 1008], [1009, 1010, 1011], [1012, 1013, 1014], [1015, 1016, 1017], [1018, 1019, 1020], [1021, 1022, 1023], [1024, 1025, 1026], [1027, 1028, 1029], [1030, 1031, 1032], [1033, 1034, 1035], [1036, 1037, 1038], [1039, 1040, 1041], [1042, 1043, 1044], [1045, 1046, 1047], [1048, 1049, 1050], [1051, 1052, 1053], [1054, 1055, 1056], [1057, 1058, 1059], [1060, 1061, 1062], [1063, 1064, 1065], [1066, 1067, 1068], [1069, 1070, 1071], [1072, 1073, 1074], [1075, 1076, 1077], [1078, 1079, 1080], [1081, 1082, 1083], [1084, 1085, 1086], [1087, 1088, 1089], [1090, 1091, 1092], [1093, 1094, 1095], [1096, 1097, 1098], [1099, 1100, 1101], [1102, 1103, 1104], [1105, 1106, 1107], [1108, 1109, 1110], [1111, 1112, 1113], [1114, 1115, 1116], [1117, 1118, 1119], [1120, 1121, 1122], [1123, 1124, 1125], [1126, 1127, 1128], [1129, 1130, 1131], [1132, 1133, 1134], [1135, 1136, 1137], [1138, 1139, 1140], [1141, 1142, 1143], [1144, 1145, 1146], [1147, 1148, 1149], [1150, 1151, 1152], [1153, 1154, 1155], [1156, 1157, 1158], [1159, 1160, 1161], [1162, 1163, 1164], [1165, 1166, 1167], [1168, 1169, 1170], [1171, 1172, 1173], [1174, 1175, 1176], [1177, 1178, 1179], [1180, 1181, 1182], [1183, 1184, 1185], [1186, 1187, 1188], [1189, 1190, 1191], [1192, 1193, 1194], [1195, 1196, 1197], [1198, 1199, 1200], [1201, 1202, 1203], [1204, 1205, 1206], [1207, 1208, 1209], [1210, 1211, 1212], [1213, 1214, 1215], [1216, 1217, 1218], [1219, 1220, 1221], [1222, 1223, 1224], [1225, 1226, 1227], [1228, 1229, 1230], [1231, 1232, 1233], [1234, 1235, 1236], [1237, 1238, 1239], [1240, 1241, 1242], [1243, 1244, 1245], [1246, 1247, 1248], [1249, 1250, 1251], [1252, 1253, 1254], [1255, 1256, 1257], [1258, 1259, 1260], [1261, 1262, 1263], [1264, 1265, 1266], [1267, 1268, 1269], [1270, 1271, 1272], [1273, 1274, 1275], [1276, 1277, 1278], [1279, 1280, 1281], [1282, 1283, 1284], [1285, 1286, 1287], [1288, 1289, 1290], [1291, 1292, 1293], [1294, 1295, 1296], [1297, 1298, 1299], [1300, 1301, 1302], [1303, 1304, 1305], [1306, 1307, 1308], [1309, 1310, 1311], [1312, 1313, 1314], [1315, 1316, 1317], [1318, 1319, 1320], [1321, 1322, 1323], [1324, 1325, 1326], [1327, 1328, 1329], [1330, 1331, 1332], [1333, 1334, 1335], [1336, 1337, 1338], [1339, 1340, 1341], [1342, 1343, 1344], [1345, 1346, 1347], [1348, 1349, 1350], [1351, 1352, 1353], [1354, 1355, 1356], [1357, 1358, 1359], [1360, 1361, 1362], [1363, 1364, 1365], [1366, 1367, 1368], [1369, 1370, 1371], [1372, 1373, 1374], [1375, 1376, 1377], [1378, 1379, 1380], [1381, 1382, 1383], [1384, 1385, 1386], [1387, 1388, 1389], [1390, 1391, 1392], [1393, 1394, 1395], [1396, 1397, 1398], [1399, 1400, 1401], [1402, 1403, 1404], [1405, 1406, 1407], [1408, 1409, 1410], [1411, 1412, 1413], [1414, 1415, 1416], [1417, 1418, 1419], [1420, 1421, 1422], [1423, 1424, 1425], [1426, 1427, 1428], [1429, 1430, 1431], [1432, 1433, 1434], [1435, 1436, 1437], [1438, 1439, 1440], [1441, 1442, 1443], [1444, 1445, 1446], [1447, 1448, 1449], [1450, 1451, 1452], [1453, 1454, 1455], [1456, 1457, 1458], [1459, 1460, 1461], [1462, 1463, 1464], [1465, 1466, 1467], [1468, 1469, 1470], [1471, 1472, 1473], [1474, 1475, 1476], [1477, 1478, 1479], [1480, 1481, 1482], [1483, 1484, 1485], [1486, 1487, 1488], [1489, 1490, 1491], [1492, 1493, 1494], [1495, 1496, 1497], [1498, 1499, 1500], [1501, 1502, 1503], [1504, 1505, 1506], [1507, 1508, 1509], [1510, 1511, 1512], [1513, 1514, 1515], [1516, 1517, 1518], [1519, 1520, 1521], [1522, 1523, 1524], [1525, 1526, 1527], [1528, 1529, 1530], [1531, 1532, 1533], [1534, 1535, 1536], [1537, 1538, 1539], [1540, 1541, 1542], [1543, 1544, 1545], [1546, 1547, 1548], [1549, 1550, 1551], [1552, 1553, 1554], [1555, 1556, 1557], [1558, 1559, 1560], [1561, 1562, 1563], [1564, 1565, 1566], [1567, 1568, 1569], [1570, 1571, 1572], [1573, 1574, 1575], [1576, 1577, 1578], [1579, 1580, 1581], [1582, 1583, 1584], [1585, 1586, 1587], [1588, 1589, 1590], [1591, 1592, 1593], [1594, 1595, 1596], [1597, 1598, 1599], [1600, 1601, 1602], [1603, 1604, 1605], [1606, 1607, 1608], [1609, 1610, 1611], [1612, 1613, 1614], [1615, 1616, 1617], [1618, 1619, 1620], [1621, 1622, 1623], [1624, 1625, 1626], [1627, 1628, 1629], [1630, 1631, 1632], [1633, 1634, 1635], [1636, 1637, 1638], [1639, 1640, 1641], [1642, 1643, 1644], [1645, 1646, 1647], [1648, 1649, 1650], [1651, 1652, 1653], [1654, 1655, 1656], [1657, 1658, 1659], [1660, 1661, 1662], [1663, 1664, 1665], [1666, 1667, 1668], [1669, 1670, 1671], [1672, 1673, 1674], [1675, 1676, 1677], [1678, 1679, 1680], [1681, 1682, 1683], [1684, 1685, 1686], [1687, 1688, 1689], [1690, 1691, 1692], [1693, 1694, 1695], [1696, 1697, 1698], [1699, 1700, 1701], [1702, 1703, 1704], [1705, 1706, 1707], [1708, 1709, 1710], [1711, 1712, 1713], [1714, 1715, 1716], [1717, 1718, 1719], [1720, 1721, 1722], [1723, 1724, 1725], [1726, 1727, 1728], [1729, 1730, 1731], [1732, 1733, 1734], [1735, 1736, 1737], [1738, 1739, 1740], [1741, 1742, 1743], [1744, 1745, 1746], [1747, 1748, 1749], [1750, 1751, 1752], [1753, 1754, 1755], [1756, 1757, 1758], [1759, 1760, 1761], [1762, 1763, 1764], [1765, 1766, 1767], [1768, 1769, 1770], [1771, 1772, 1773], [1774, 1775, 1776], [1777, 1778, 1779], [1780, 1781, 1782], [1783, 1784, 1785], [1786, 1787, 1788], [1789, 1790, 1791], [1792, 1793, 1794], [1795, 1796, 1797], [1798, 1799, 1800], [1801, 1802, 1803], [1804, 1805, 1806], [1807, 1808, 1809], [1810, 1811, 1812], [1813, 1814, 1815], [1816, 1817, 1818], [1819, 1820, 1821], [1822, 1823, 1824], [1825, 1826, 1827], [1828, 1829, 1830], [1831, 1832, 1833], [1834, 1835, 1836], [1837, 1838, 1839], [1840, 1841, 1842], [1843, 1844, 1845], [1846, 1847, 1848], [1849, 1850, 1851], [1852, 1853, 1854], [1855, 1856, 1857], [1858, 1859, 1860], [1861, 1862, 1863], [1864, 1865, 1866], [1867, 1868, 1869], [1870, 1871, 1872], [1873, 1874, 1875], [1876, 1877, 1878], [1879, 1880, 1881], [1882, 1883, 1884], [1885, 1886, 1887], [1888, 1889, 1890], [1891, 1892, 1893], [1894, 1895, 1896], [1897, 1898, 1899], [1900, 1901, 1902], [1903, 1904, 1905], [1906, 1907, 1908], [1909, 1910, 1911], [1912, 1913, 1914], [1915, 1916, 1917], [1918, 1919, 1920], [1921, 1922, 1923], [1924, 1925, 1926], [1927, 1928, 1929], [1930, 1931, 1932], [1933, 1934, 1935], [1936, 1937, 1938], [1939, 1940, 1941], [1942, 1943, 1944], [1945, 1946, 1947], [1948, 1949, 1950], [1951, 1952, 1953], [1954, 1955, 1956], [1957, 1958, 1959], [1960, 1961, 1962], [1963, 1964, 1965], [1966, 1967, 1968], [1969, 1970, 1971], [1972, 1973, 1974], [1975, 1976, 1977], [1978, 1979, 1980], [1981, 1982, 1983], [1984, 1985, 1986], [1987, 1988, 1989], [1990, 1991, 1992], [1993, 1994, 1995], [1996, 1997, 1998], [1999, 2000, 2001], [2002, 2003, 2004], [2005, 2006, 2007], [2008, 2009, 2010], [2011, 2012, 2013], [2014, 2015, 2016], [2017, 2018, 2019], [2020, 2021, 2022], [2023, 2024, 2025], [2026, 2027, 2028], [2029, 2030, 2031], [2032, 2033, 2034], [2035, 2036, 2037], [2038, 2039, 2040], [2041, 2042, 2043], [2044, 2045, 2046], [2047, 2048, 2049], [2050, 2051, 2052], [2053, 2054, 2055], [2056, 2057, 2058], [2059, 2060, 2061], [2062, 2063, 2064], [2065, 2066, 2067], [2068, 2069, 2070], [2071, 2072, 2073], [2074, 2075, 2076], [2077, 2078, 2079], [2080, 2081, 2082], [2083, 2084, 2085], [2086, 2087, 2088], [2089, 2090, 2091], [2092, 2093, 2094], [2095, 2096, 2097], [2098, 2099, 2100], [2101, 2102, 2103], [2104, 2105, 2106], [2107, 2108, 2109], [2110, 2111, 2112], [2113, 2114, 2115], [2116, 2117, 2118], [2119, 2120, 2121], [2122, 2123, 2124], [2125, 2126, 2127], [2128, 2129, 2130], [2131, 2132, 2133], [2134, 2135, 2136], [2137, 2138, 2139], [2140, 2141, 2142], [2143, 2144, 2145], [2146, 2147, 2148], [2149, 2150, 2151], [2152, 2153, 2154], [2155, 2156, 2157], [2158, 2159, 2160], [2161, 2162, 2163], [2164, 2165, 216
```

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 $(1 \pm 0.5)10^{-5}$.



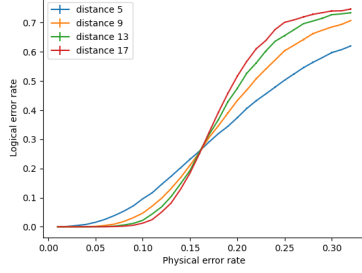
(a) Lookup table Steane code threshold



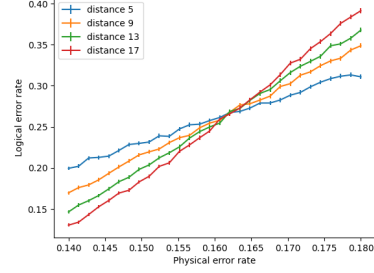
(b) Detailed view at around $per = 10^{-5}$

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

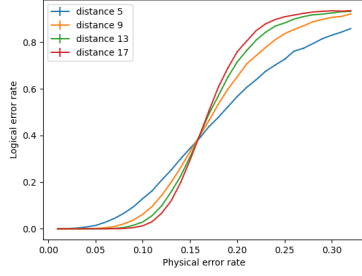
Unfortunately, a threshold for the hexagonal toric color code using the lifting decoder could not be found due to a lifting bug resulting in false error predictions for certain error patterns.



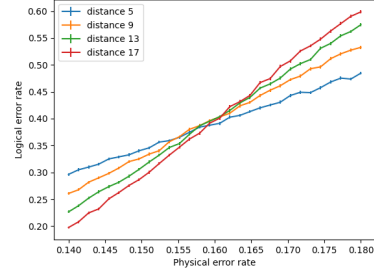
(a) Surface code MWPM thresholding overview



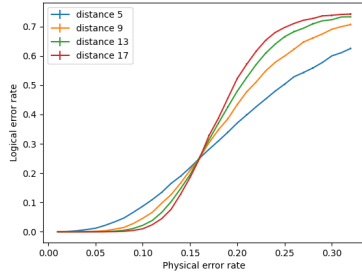
(b) Detailed view for precise threshold determination of surface code



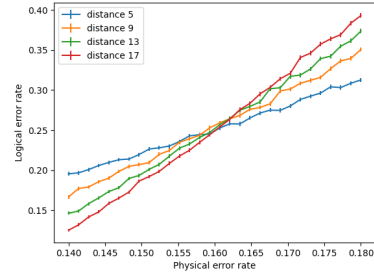
(c) Toric code MWPM thresholding overview



(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 [9] 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 [13]. 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 [14]. 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.

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 $n^{1/2}$,” 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] 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>
- [9] 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>
- [10] 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>
- [11] N. Delfosse, “Decoding color codes by projection onto surface codes,” pp. 12–15, 2018. [Online]. Available: <https://arxiv.org/pdf/1308.6207.pdf>

- [12] C. Schumann, “Decoding the color code,” Git repository, 2023, accessed: March 31, 2023. [Online]. Available: <https://github.com/clerusch/coding>
- [13] 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>
- [14] 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>

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:

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

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

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

if the result is -1, it becomes:

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

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

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

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

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

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

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

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

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

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

```
1 from typing import List
2 from numpy import array, vstack, hstack, zeros, uint8, ones,
  ndarray
3 from itertools import product
4 from random import random
5
6 def genSteaneLookupTable()->dict:
7
8     # Generate Steane parity check matrix from identical
9     # X and Z PCMs
10    H = array([[1, 0, 0, 1, 0, 1, 1],
11              [0, 1, 0, 1, 1, 0, 1],
12              [0, 0, 1, 0, 1, 1, 1]])
13    pcm = vstack((hstack((H, zeros(H.shape))),
14                 hstack((zeros(H.shape), H))))
15
16    # Generate lookup table
17    lookup_table = {}
18    for error in product([0, 1], repeat=14):
19        syndrome = tuple(pcm @ error % 2)
20        if syndrome in lookup_table:
21            lookup_table[syndrome].append(error)
22        else:
23            lookup_table[syndrome] = [error]
24
25    # Remove duplicates from lookup table
26    for key in lookup_table:
27        lookup_table[key] = list(set(lookup_table[key]))
28
29    return lookup_table
30
31 def findMinWeight(predictions)-> ndarray:
32     """
33     Find the minimum weight tuple for a given prediction
34     """
35     curr_pred = ones(14,dtype=uint8)
36     curr_best_weight = 100
37     for pred in predictions:
38         pred = array(pred)
39         yweight = 0
40         for i in range(int(len(pred)/2)):
```

```

41         if pred[i] & pred[i+7] == 1:
42             yweight += 1
43             pred[i] = 0
44             pred[i+7] = 0
45         if yweight + sum(pred) < curr_best_weight:
46             curr_pred = pred
47             curr_best_weight = yweight + sum(pred)
48     return curr_pred
49
50 def main():
51     syndrome = array([1,1,1,0,0,0])
52
53     possibles = genSteaneLookupTable()[tuple(syndrome)]
54
55     print(f"The syndrome {syndrome}\n can be caused by the
56 following errors: ")
57
58     print(f"The most likely cause of this syndrome is\n {
59 findMinWeight(possibles)}")
60
61 if __name__=="__main__":
62     main()

```

7.2.2 Thresholding

```

1 from betterlookup import genSteaneLookupTable, findMinWeight,
   findMinWeight
2 from random import random
3 from numpy import zeros, uint8, concatenate, array,\
4 array_equal, linspace, vstack, hstack, zeros, ndarray,
   logspace
5 from matplotlib.pyplot import errorbar, legend, \
6 savefig, xlabel, ylabel, plot
7
8 def genSteaneError(per)->ndarray:
9     """ Generates an error vector on the Steane code"""
10    empty7 = zeros(7, dtype=uint8)
11    xerror = empty7.copy()
12    zerror = empty7.copy()
13    for i in range(len(xerror)):
14        if random()<per:
15            xerror[i] = 1
16    for j in range(len(zerror)):
17        if random()<per:
18            zerror[j] = 1
19    yerror = concatenate((xerror,zerror)) # generating too

```

```

long errors
20 for k, bit in enumerate(yerror[:6]):
21     if random()<per:
22         yerror[k] = (yerror[k] + 1)%2
23         yerror[2*k] = (yerror[2*k]+1)%2
24 # error = (concatenate((xerror, empty7)) + yerror \
25 #         + concatenate((empty7, zerror)))%2
26 return yerror
27
28 def steaneLerCalc(steaneH, nr, per, logicals)->float:
29     """Calculates the logical error rate of the steane
30     code decoded with a lookup table"""
31     numErrors = 0
32     looktable = genSteaneLookupTable()
33     for _ in range(nr):
34         actual_error = genSteaneError(per)
35         syndrome = steaneH@actual_error %2
36         predictions = looktable[tuple(syndrome)]
37         pred = findMinWeight(predictions)
38         pred_L_flips = logicals@pred %2
39         actual_L_flips = logicals@actual_error %2
40         if not array_equal(actual_L_flips, pred_L_flips):
41             numErrors += 1
42     return numErrors/nr
43
44 def makeHgpPcm(Hx, Hz)->ndarray:
45     """
46     Makes a full parity check matrix including x and z
47     checks for a hypergraph product code of two other codes
48     """
49     Hx = hstack((Hx, zeros(Hx.shape, dtype=uint8)))
50     Hz = hstack((zeros(Hz.shape, dtype=uint8), Hz))
51     H = vstack((Hx, Hz))
52     return H
53
54 def main():
55     steanelogicals = \
56         array([\
57             [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
58             [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]])
59     steaneH = array([[1, 0, 0, 1, 0, 1, 1],
60                     [0, 1, 0, 1, 1, 0, 1],
61                     [0, 0, 1, 0, 1, 1, 1]])
62     pers = linspace(2*10**(-6), 2*10**(-5), 20)
63     lers = []

```



```

64     nr = 10000
65     H = makeHgpPcm(steaneH, steaneH)
66     for per in pers:
67         print(f"per={per}")
68         lers.append(\
69             steaneLerCalc(H, nr, per, steaneLogicals))
70     lers = array(lers)
71     std_err = (lers*(1-lers)/nr)**0.5
72     errorbar(pers, lers, yerr=std_err)
73     plot(pers,pers)
74     xlabel("Physical error rate")
75     ylabel("Logical error rate")
76     savefig("img/figures/steaneLookupThreshold.png")
77
78 if __name__ == "__main__":
79     main()

```

7.3 Lifting Decoder

```

1  import networkx as nx
2  from pymatching import Matching
3  from numpy import zeros, uint8, linspace, array
4  from random import random
5  from random import sample
6  from typing import List, FrozenSet, Set
7  from os import makedirs
8  from os.path import exists
9  from time import time
10 from matplotlib.pyplot import figure, savefig, title, show,
    xlabel, ylabel, legend, errorbar, close, plot
11 """
12 The main function of this file will generate a set of images
13 of the pertaining color code graph, its dual, and its
    respective
14 2- colored subgraphs and print an error prediction on the
    subgraphs.
15
16 A folder "img/hexcolor/" will be created to save image files
    if it does not exist.
17 """
18
19 def colorize_graph_black(G: nx.Graph) -> bool:
20     """
21     Args:
22         G(nx.Graph): some graph

```

```

23     Returns:
24         bool: whether it was successful in changing the graph
25         object
26             to all black edges.
27     """
28     for u, v, attr in G.edges(data=True):
29         G[u][v]['color'] = 'black'
30     return True
31
32 def tor_hex48_color_encode(G: nx.Graph, m: int=6, n: int=4) ->
33     bool:
34     """
35     Args:
36         G(nx.Graph): graph we want to encode with three
37         colored faces
38         n,m: how many by how many hexagon, default to 6 and 4
39         like in delfosse
40     Returns:
41         bool: Success of graph object modification procedure
42     """
43     rgb_list = ['r', 'g', 'b']
44     # initialize all edge colors to black
45     for u, v, attr in G.edges(data=True):
46         G[u][v]['color'] = 'black'
47
48     # colorizing algorithm
49
50     # horizontal edges
51     for i in range(int(n/2)):
52         for j in range(m):
53             first_coordinate = (2*i, 2*j)
54             second_coordinate = (2*i+1, 2*j)
55             G[first_coordinate][second_coordinate]['color'] = rgb_list[j%3]
56
57     for i in range(int(n/2)):
58         for j in range(m):
59             first_coordinate = (2*i+1, 2*j+1)
60             second_coordinate = ((2*i+2)%n, 2*j+1)
61             G[first_coordinate][second_coordinate]['color'] =
62             rgb_list[(j-1)%3]
63
64     # left ladder edges
65     for i in range(int(n/2)):
66         for j in range(2*m):
67             first_coordinate = (2*i, j)
68             second_coordinate = (2*i, (j+1)%(2*m))

```

```

62         G[first_coordinate][second_coordinate]['color'] =
        rgb_list[(1-j)%3]
63     # right ladder edges
64     for i in range(int(n/2)):
65         for j in range(2*m):
66             first_coordinate = (2*i+1,j)
67             second_coordinate = (2*i+1,(j+1)%(2*m))
68             G[first_coordinate][second_coordinate]['color'] =
        rgb_list[(1-j)%3]
69     return True
70
71 def make_a_base_graph(m: int=6,n: int=4) -> nx.Graph:
72     """
73     Args:
74         m(int), n(int): desired dimension of faces on graph (
m by n)
75     Returns:
76         G(nx.Graph): A basic color code graph of dimensions m
        by n
77                     with colored edges encircling opposite
        colored faces.
78     """
79     G = nx.hexagonal_lattice_graph(m, n, periodic=True)
80     colorize_graph_black(G)
81     tor_hex48_color_encode(G,m,n)
82     for node in G.nodes:
83         G.nodes[node]['color'] = 'black'
84         G.nodes[node]['fault_ids'] = 0
85     return G
86
87 def draw_graph_with_colored_edges_and_nodes(G: nx.Graph, file
: str=None, name: str=None) -> bool:
88     """
89     Draws a graph who's nodes and edges have colors.
90     Options:
91         filename (str): save file to specified name (will plt
        .show() otherwise)
92         name (str): will create figure with specified name
93     """
94     pos = nx.get_node_attributes(G, 'pos')
95     node_colors = [data['color'] for _, data in G.nodes(data=
        True)]
96     edge_colors = [G[u][v]['color'] for u, v in G.edges()]
97
98     figure()

```

```

99     if name:
100         title(name)
101     if pos:
102         nx.draw(G, pos, with_labels=True, node_color=
node_colors, edge_color=edge_colors)
103     elif not pos:
104         nx.draw(G, with_labels=True, node_color=node_colors,
edge_color=edge_colors)
105     if nx.get_edge_attributes(G, "fault_ids"):
106         nx.draw_networkx_edge_labels(G, pos, edge_labels=nx.
get_edge_attributes(G, "fault_ids"))
107     if file:
108         savefig(file)
109     else:
110         show()
111     close()
112     return True
113
114 def flag_color_graph(graph: nx.Graph, per: float=0.1) -> Set[
any]:
115     """
116     Args:
117         graph(nx.Graph): graph to be altered with errors on
nodes
118         per(float): probability on error occuring on each
node
119     Returns:
120         set(node): Actually occurred errors
121     """
122     error_nodes = set()
123     for node in graph.nodes:
124         if random() < per:
125             graph.nodes[node]['fault_ids'] = 1
126             graph.nodes[node]['color'] = 'y'
127             error_nodes.add(node)
128     return error_nodes
129
130 def find_6_loops(graph: nx.Graph) -> List[FrozenSet[any]]:
131     """
132     Args:
133         nx.Graph: input graph
134     Returns:
135         set[frozenset]: Topology of nodes comprising faces on
input graph """
136     cycles = set()

```

```

137     for node in graph.nodes:
138         for node1 in graph.neighbors(node):
139             for node2 in graph.neighbors(node1):
140                 for node3 in graph.neighbors(node2):
141                     for node4 in graph.neighbors(node3):
142                         for node5 in graph.neighbors(node4):
143                             for node6 in graph.neighbors(
node5):
144                                 if node6 == node:
145                                     cycles.add(frozenset([
node, node1, node2, node3, node4, node5]))
146     faces = [cycle for cycle in cycles if len(cycle) == 6]
147     return faces
148
149 def find_face_color(graph: nx.Graph, face: FrozenSet) -> str:
150     """
151     Args:
152         graph: graph on which face lies
153         face(set[nodes]): face to analyze
154     Returns:
155         color(str): color of face
156     """
157     rgb = set(['r', 'g', 'b'])
158     boundary_colors = set()
159     for node in face:
160         for node2 in graph.neighbors(node):
161             if node2 in face:
162                 boundary_colors.add(graph[node][node2]['color
'])
163     face_color = rgb - boundary_colors
164     face_color = face_color.pop()
165     return face_color
166
167 def dual_of_three_colored_graph(graph: nx.Graph):# -> nx.
Graph:
168     """
169     Args:
170         graph(nx.Graph): graph of which we want the dual
171     Returns:
172         dual_graph(nx.Graph): the dual of that graph
173     """
174     dual_graph = nx.Graph()
175     faces = find_6_loops(graph)
176     # init nodes
177     for i, face in enumerate(faces):

```

```

178         dual_graph.add_node(i, color = 'black')
179         color_of_face = find_face_color(graph, face)
180         dual_graph.nodes[i]['color'] = color_of_face
181         # This is the part for error -> syndrome inheritance
to the dual graph
182         dual_graph.nodes[i]['fault_ids'] = 0
183         for node in face:
184             if graph.nodes[node]['fault_ids'] == 1:
185                 dual_graph.nodes[i]['fault_ids'] = (
dual_graph.nodes[i]['fault_ids']+1)%2
186         # connect nodes
187         for i, face in enumerate(faces):
188             otherfaces = faces[:i]+faces[((i+1)%(len(faces)+1)):]
189             for j, face2 in enumerate(otherfaces):
190                 lap_nodes = set(face & face2)
191                 if lap_nodes:
192                     # A three-colorable graph will only ever have
two nodes between two faces
193                     node1 = lap_nodes.pop()
194                     node2 = lap_nodes.pop()
195                     connecting_color = graph[node1][node2]['color
']
196                     # we are iterating not over the list of faces
, but over the list of otherfaces
197                     # what is j?
198                     second_face_pos = [k for k in range(len(faces
)) if faces[k] == otherfaces[j]].pop()
199                     #
200                     dual_graph.add_edge(i,second_face_pos, color
= connecting_color)
201
202
203         return dual_graph, faces
204
205 def subtile(Graph: nx.Graph, color: str) -> nx.Graph:
206     """
207     Args:
208         Graph(nx.Graph): graph we want to subtile
209         color(str): color in format "r","g", "b" of which all
edges
210
                in the subtiling will be comprised
211     Returns:
212         G(nx.Graph): subtiled graph (does not edit original
object)
213     """

```

```

214     G = Graph.copy()
215     for edge in G.edges:
216         u, v = edge[0], edge[1]
217         if G.edges[u,v]['color'] != color:
218             G.remove_edge(u,v)
219     G.remove_nodes_from(list(nx.isolates(G)))
220     return G
221
222 def decode_subtile(graph: nx.Graph) -> List[any]:
223     """
224     Args:
225         graph(nx.Graph): graph with "fault_ids" property on
226         some nodes
227     Returns:
228         prediction(List[edges]): predicted error edges on
229         graph
230     """
231     # we'll change og and revert this time
232     # renamed_copy = graph.copy()
233     # make renamed_copy usable (hopefully)
234     for i, node in enumerate(graph.nodes):
235         graph.nodes[node]['og_name'] = node
236         graph = nx.relabel_nodes(graph, {node: i})
237     matching = Matching(graph)
238     # generate syndrome on renamed_copy
239     syndrome = zeros(len(graph.nodes), dtype=uint8)
240     for node in graph.nodes:
241         if graph.nodes[node]['fault_ids'] == 1:
242             syndrome[node] = 1
243     # predict edges on the renamed_copy
244     prediction = matching.decode_to_edges_array(syndrome)
245     # rename nodes to be actually usable
246     for edge in prediction:
247         for i in range(len(edge)):
248             edge[i] = graph.nodes[edge[i]]['og_name']
249     # revert the graph back to normal
250     for node in graph.nodes:
251         graph = nx.relabel_nodes(graph, {node: graph.nodes[
252             node]['og_name']})
253
254     return prediction
255
256 def make_a_shower(graph: nx.Graph) -> nx.Graph:
257     """
258     Args:

```

```

256         graph(nx.Graph): graph we want a yellow syndrome
flagged copy of
257     Returns:
258         shower(nx.Graph): graph with yellow marked syndrome
nodes
259     """
260     shower = graph.copy()
261     for node in shower.nodes:
262         if shower.nodes[node]['fault_ids'] == 1:
263             shower.nodes[node]['color'] = 'y'
264     return shower
265
266 def find_hyper_edges(dual_graph: nx.Graph, edges_array_r:
List[any],
267                     edges_array_g: List[any], edges_array_b:
List[any]) -> List[any]:
268     """
269     Takes: a dualgraph and its subgraph matching edges arrays
270     Returns: list of cycles on the dual graph
271     """
272     # generate the surrounding edges of cycles
273     set_of_all_edges_bounding_hyperedge = set()
274     for color in [edges_array_r, edges_array_g, edges_array_b
]:
275         for edge in color:
276             addable_edge = tuple(sorted(edge))
277             set_of_all_edges_bounding_hyperedge.add(
addable_edge)
278     # make a graph of only error cycles
279     error_bound_graph = dual_graph.copy()
280     bad_edges = []
281     # this is necessary because we can't modify edges during
iteration
282     for edge in error_bound_graph.edges:
283         if edge not in set_of_all_edges_bounding_hyperedge:
284             bad_edges.append(edge)
285     error_bound_graph.remove_edges_from(bad_edges)
286     isolates = list(nx.isolates(error_bound_graph))
287     error_bound_graph.remove_nodes_from(isolates)
288     # draw_graph_with_colored_edges_and_nodes(
error_bound_graph, "img/hexcolor/decodergraph.png")
289     cycles = nx.cycle_basis(error_bound_graph)
290     return cycles
291
292 def flag_c_graph_specific(graph: nx.Graph, nodes: List[any])

```



```

293     -> bool:
294         """
295         Flags down specific nodes on a graph from a list of nodes
296         """
297         for node in nodes:
298             graph.nodes[node]['fault_ids'] = 1
299             graph.nodes[node]['color'] = 'y'
300         return True
301
302 def lift(dual_edge_cycles: List[any], faces: List[FrozenSet])
303     -> Set[any]:
304     """
305     Takes: List of dual graph cycles, facenodes to face map
306     Returns: List of enclosed og nodes
307     """
308     ## Strategy: understand and comment the below code
309     enc_nodes = set()
310     for dual_edge_cycle in dual_edge_cycles:
311         face_on_dec = dual_edge_cycle.pop()
312         bounded_nodes = faces[face_on_dec]
313         for face in dual_edge_cycle:
314             bounded_nodes = bounded_nodes & faces[face]
315             bounded_nodes = frozenset(bounded_nodes)
316             enc_nodes.add(bounded_nodes)
317         # clear up empty frozenset and pop the items to a set
318         if frozenset() in enc_nodes:
319             enc_nodes.remove(frozenset())
320         res = set()
321         for enc_node in enc_nodes:
322             res.add(next(iter(set(enc_node))))
323         return res
324
325 def total_decoder(graph: nx.Graph, per: float) -> bool:
326     """
327     Takes: a color code graph and physical error rate
328     Returns: Success of correction operation
329     """
330     actual_errors = flag_color_graph(graph, per)
331     #### dualizing and subtiling
332     dual, faces = dual_of_three_colored_graph(graph)
333     subr, subg, subb = subtile(dual, 'r'), subtile(dual, 'g')
334     , subtile(dual, 'b')
335     #### decoding part
336     pred_r, pred_g, pred_b = decode_subtile(subr),
337     decode_subtile(subg), decode_subtile(subb)

```

```

334     hyper_edge_cycles = find_hyper_edges(dual, pred_r, pred_g
, pred_b)
335     ## get back to og nodes from dual nodes/ faces
336     og_enc_nodes_by_dual_cycles = lift(hyper_edge_cycles,
faces)
337     return og_enc_nodes_by_dual_cycles == actual_errors
338
339 def cc_ler_calc(graph: nx.Graph, per: float, nr: int) ->
float:
340     numErrors = 0
341     for _ in range(nr):
342         if not total_decoder(graph, per):
343             numErrors += 1
344     return numErrors/nr
345
346 def cc_threshold_plotter(dists: List[any], pers: List[float],
nr:int, file=None) -> bool:
347     log_errors_all_dist = []
348     for d in dists:
349         print("Simulating d = {}".format(d))
350         origG = make_a_base_graph(d[0],d[1])
351         lers = []
352         for per in pers:
353             print(f"per={per}")
354             graph = origG.copy()
355             lers.append(cc_ler_calc(graph, per, nr))
356         log_errors_all_dist.append(array(lers))
357     figure()
358     for dist, logical_errors in zip(dists,
log_errors_all_dist):
359         std_err = (logical_errors*(1-logical_errors)/nr)**0.5
360         errorbar(pers, logical_errors, yerr=std_err, label="L
={}".format(dist))
361     plot(pers, pers, label = 'Threshold')
362     xlabel("Physical error rate")
363     ylabel("Logical error rate")
364     legend(loc=0)
365     if file:
366         if not exists("img/hexcolor"):
367             makedirs("img/hexcolor")
368         savefig("img/hexcolor/"+file)
369     else:
370         show()
371     close()
372     return True

```

```

373
374 def main() -> bool:
375
376
377     ##### just making sure image filesaves work
378     if not exists("img/hexcolor"):
379         makedirs("img/hexcolor")
380     ##### initialize color code graph with errors
381     origG = make_a_base_graph()
382     actual_errors = flag_color_graph(origG, 0.05)
383     ## This is for manually setting faults
384     # actual_errors = [(0,0),(1,2)]
385     # flag_c_graph_specific(origG, actual_errors)
386     ##### dualizing and subtiling
387     dual, faces = dual_of_three_colored_graph(origG)
388     subr, subg, subb = subtile(dual, 'r'), subtile(dual, 'g')
389     , subtile(dual, 'b')
390     ##### flag syndromes yellow for better visualizing
391     dual_syn, subr_syn, subg_syn, subb_syn = make_a_shower(
392     dual), make_a_shower(subr), make_a_shower(subg),
393     make_a_shower(subb)
394     ##### decoding part
395     start = time()
396     pred_r, pred_g, pred_b = decode_subtile(subr),
397     decode_subtile(subg), decode_subtile(subb)
398     hyper_edge_cycles = find_hyper_edges(dual, pred_r, pred_g
399     , pred_b)
400     ## get back to og nodes from dual nodes/ faces
401     print("hyp_edge_cycles are: ", hyper_edge_cycles)
402     og_enc_nodes_by_dual_cycles = lift(hyper_edge_cycles,
403     faces)
404     end = time()
405     ##### visualizing part
406     print(f"This decoding and lifting took {end-start}
407     seconds.")
408     draw_graph_with_colored_edges_and_nodes(origG, "img/
409     hexcolor/original.png")
410     draw_graph_with_colored_edges_and_nodes(dual_syn, "img/
411     hexcolor/dual.png")
412     for i, graph in enumerate([subr_syn, subg_syn, subb_syn])
413     :
414         draw_graph_with_colored_edges_and_nodes(graph, f"img/
415         hexcolor/{i}.png")
416     # print("The red prediction is: ", pred_r)
417     # print("The green prediction is: ", pred_g)

```

```

407     # print("The blue prediction is: ", pred_b)
408     # print("Hyperedges are: ", hyper_edges)
409     if og_enc_nodes_by_dual_cycles:
410         for i in range(len(og_enc_nodes_by_dual_cycles)):
411             print(f"The {i}th error node on the graph is {
og_enc_nodes_by_dual_cycles.pop()}")
412     print("The actual errors were: ", actual_errors)
413     return True
414
415     # dists = [(6,4)]#,(12,8),(24,8)]
416     # pers = linspace(0.01, 0.2, 20)
417     # nr = 1000
418     # cc_threshold_plotter(dists, pers, nr, "firstThreshold")
419     # return True
420
421 if __name__ == "__main__":
422     main()

```

7.4 Thresholds

7.4.1 Surface/Toric code thresholds

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from pymatching import Matching
4 from scipy.sparse import hstack, kron, eye, csr_matrix,
    block_diag
5 from ldpc import mod2
6
7 ##### Helper functions
8 #####
9 def genRepPCM(distance):
10     """
11     Generates a repetition code parity-check-matrix
12     Args:
13         distance(Int): distance of the code
14     Returns:
15         pcm(np.array([[[]]]): repetition code parity check
matrix corresponding to distance
16     """
17     nq = distance # number of qubits
18     na = nq - 1 # number of ancillas
19     pcm = np.array([[0 for _ in range(nq)] for _ in range(na)
20 ])
21     for i in range(na):
22         pcm[i][i] = 1

```

```

21         pcm[i][(i+1) % nq] = 1
22     return pcm
23
24 def genRingPCM(distance):
25     """
26     Generates a ring code parity-check-matrix
27
28     Args:
29         distance(Int): distance of
30
31     Returns:
32         pcm(np.array([[[]]])): generated parity check matrix of
33         distance
34     """
35     pcm=np.eye(distance)
36     for i in range(distance):
37         pcm[i][(i+1)%distance] = 1
38     return pcm
39
40 def ring_code(n):
41     """
42     scipy sparse Parity check matrix of a ring code with
43     length n.
44     """
45     return csr_matrix(genRingPCM(n))
46
47 def rep_code(n):
48     """
49     scipy sparse Parity check matrix of a rep code with n
50     qubits
51     """
52     return csr_matrix(genRepPCM(n))
53
54 def genXStabilizers(first_pcm_generator, second_pcm_generator
55 , dist):
56     """
57     check matrix for the X stabilizers of a hypergraph
58     product code of distance dist
59     """
60     H1 = first_pcm_generator(dist)
61     H2 = second_pcm_generator(dist)
62     H = hstack(
63         [kron(H1, eye(H2.shape[1])), kron(eye(H1.shape[0]),
64         H2.T)],
65         dtype=np.uint8

```

```

60     )
61     return H
62
63 def genZStabilizers(first_pcm_generator, second_pcm_generator
64 , dist):
65     """
66     check matrix for the Z stabilizers of a hypergraph
67     product code of distance dist
68     """
69     H1 = first_pcm_generator(dist)
70     H2 = second_pcm_generator(dist)
71     H = hstack(
72         [kron(eye(H1.shape[1]), H2), kron(H1.T, eye(H2.
73 shape[0]))],
74         dtype=np.uint8
75     )
76     return H
77
78 def genHxHz(first_code, second_code, d):
79     """
80     generates Hx and Hz of a hgp code from two codes
81     """
82     Hx = genXStabilizers(first_code, second_code, d).todense()
83     Hz = genZStabilizers(first_code, second_code, d).todense()
84     # Hx = np.hstack((Hx, np.zeros(Hx.shape, dtype=np.uint8)))
85     # Hz = np.hstack((np.zeros(Hz.shape, dtype=np.uint8), Hz))
86
87     return Hx, Hz
88
89 def compute_lz(hx,hz):
90     #lz logical operators
91     #lz\in ker{hx} AND \notin Im(Hz.T)
92     # hx = hx.todense()
93     # hz = hz.todense()
94     ker_hx=mod2.nullspace(hx) #compute the kernel
95     basis of hx
96     im_hzT=mod2.row_basis(hz) #compute the image
97     basis of hz.T
98
99     #in the below we row reduce to find vectors in kx
100     that are not in the image of hz.T.

```

```

95         log_stack=np.vstack([im_hzT,ker_hx])
96         pivots=mod2.row_echelon(log_stack.T)[3]
97         log_op_indices=[i for i in range(im_hzT.shape[0],
log_stack.shape[0]) if i in pivots]
98         log_ops=log_stack[log_op_indices]
99         return log_ops
100
101 def calc_logicals(hx, hz):
102     """ calculates actual logical operators from two parity
check matrices of
103     codes generating a hgp code
104     """
105
106     lx = compute_lz(hz, hx)
107     lz = compute_lz(hx, hz)
108     lx=np.vstack((np.zeros(lz.shape, dtype=np.uint8),lx))
109     lz=np.vstack((lz,np.zeros(lx.shape, dtype=np.uint8)))
110     # temp = mod2.inverse(lx@lz.T %2)
111     # lx = temp@lx % 2
112     return np.hstack((lx, lz))
113
114 def makeHgpPcm(Hx, Hz):
115     """
116     Makes a full parity check matrix including x and z checks
for a
117     hypergraph product code of two other codes
118     """
119     # Hx = genXStabilizers(first_code, second_code, d).
todense()
120     # Hz = genZStabilizers(first_code, second_code, d).
todense()
121     Hx = np.hstack((Hx, np.zeros(Hx.shape, dtype=np.uint8)))
122     Hz = np.hstack((np.zeros(Hz.shape, dtype=np.uint8), Hz))
123     H = np.vstack((Hx, Hz))
124     return csr_matrix(H)
125
126 ##### Hotstuff
#####
127 def lerCalc(H, logicals, nr=1000, per = 0.3):
128     "calculates logical error rate assuming a noise model of
p/3 X,Y,Z errors"
129     matching = Matching.from_check_matrix(H)#, faults_matrix=
logicals)
130     numErrors = 0
131     for _ in range(nr):

```

```

132     noise = np.zeros(H.shape[1], dtype=np.uint8)
133     halflength = int(len(noise)/2)
134     for i in range(halflength):
135         # this is physical X errors, editing first half
of entries
136         if np.random.rand() < per/3:
137             noise[i] = (noise[i]+1) % 2
138         # this is physical Z errors, editing second half
of entries
139         if np.random.rand() < per/3:
140             noise[i+halflength] = (noise[i+halflength] +
1) % 2
141         # this is physical Y errors, assuming same
syndrome as X and Z implies same error
142         if np.random.rand() < per/3:
143             noise[i] = (noise[i]+1) % 2
144             noise[i+halflength] = (noise[i+halflength] +
1) % 2
145     noise = csr_matrix(noise)
146     noise = noise.T
147     syndrome = csr_matrix(((H@noise).todense() % 2))
148     prediction = csr_matrix(matching.decode(syndrome.
todense()))).T
149     predicted_flips = (logicals@prediction).todense() % 2
150     actualLflips = (logicals@noise).todense() % 2
151     if not np.array_equal(actualLflips, predicted_flips):
152         numErrors += 1
153     return numErrors/nr
154
155 def thresholdPlotter(dists, pers, nr, first_code, second_code
, codename):
156     """
157     plots logical error rates of a quantum code with a list
of distances
158     and physical error rates
159     """
160     np.random.seed(2)
161     log_errors_all_dist = []
162     for d in dists:
163         print("Simulating d = {}".format(d))
164         Hx, Hz = genHxHz(first_code, second_code, d)
165         H = makeHgpPcm(Hx, Hz)
166         logicals = csr_matrix(calc_logicals(Hx, Hz))
167         lers = []
168         for per in pers:

```



```

169         print(f"per={per}")
170         lers.append(lerCalc(H, logicals, nr, per))
171         log_errors_all_dist.append(np.array(lers))
172     plt.figure()
173     for dist, logical_errors in zip(dists,
log_errors_all_dist):
174         std_err = (logical_errors*(1-logical_errors)/nr)**0.5
175         plt.errorbar(pers, logical_errors, yerr=std_err,
label="distance {}".format(dist))
176     plt.xlabel("Physical error rate")
177     plt.ylabel("Logical error rate")
178     plt.legend(loc=0)
179     plt.savefig(codename)
180
181 def main():
182     dists = range(5,20,4)
183     pers = np.linspace(0.01, 0.32, 32)
184     nr = 30000
185     print("Thresholding the surface code...")
186     thresholdPlotter(dists, pers, nr, rep_code, rep_code, "
surfaceThresholdOverview.png")
187     print("Thresholding the toric code...")
188     thresholdPlotter(dists, pers, nr, ring_code, ring_code, "
toricThresholdOverview.png")
189     print("Thresholding the cylindric code...")
190     thresholdPlotter(dists, pers, nr, rep_code, ring_code, "
cylinderThresholdOverview.png")
191
192 if __name__ == "__main__":
193     main()

```

7.4.2 Color code thresholds

```

1 from betterlookup import genSteaneLookupTable, findMinWeight,
    findMinWeight
2 from random import random
3 from numpy import zeros, uint8, concatenate, array,\
4     array_equal, linspace, vstack, hstack, zeros, ndarray,
    logspace
5 from matplotlib.pyplot import errorbar, legend, \
6     savefig, xlabel, ylabel, plot
7
8 def genSteaneError(per)->ndarray:
9     """ Generates an error vector on the Steane code"""
10    empty7 = zeros(7, dtype=uint8)
11    xerror = empty7.copy()

```

```

12     zerror = empty7.copy()
13     for i in range(len(xerror)):
14         if random() < per:
15             xerror[i] = 1
16     for j in range(len(zerror)):
17         if random() < per:
18             zerror[j] = 1
19     yerror = concatenate((xerror, zerror)) # generating too
long errors
20     for k, bit in enumerate(yerror[:6]):
21         if random() < per:
22             yerror[k] = (yerror[k] + 1)%2
23             yerror[2*k] = (yerror[2*k]+1)%2
24     # error = (concatenate((xerror, empty7)) + yerror \
25     #         + concatenate((empty7, zerror)))%2
26     return yerror
27
28 def steaneLerCalc(steaneH, nr, per, logicals)->float:
29     """Calculates the logical error rate of the steane
30     code decoded with a lookup table"""
31     numErrors = 0
32     looktable = genSteaneLookupTable()
33     for _ in range(nr):
34         actual_error = genSteaneError(per)
35         syndrome = steaneH@actual_error %2
36         predictions = looktable[tuple(syndrome)]
37         pred = findMinWeight(predictions)
38         pred_L_flips = logicals@pred %2
39         actual_L_flips = logicals@actual_error %2
40         if not array_equal(actual_L_flips, pred_L_flips):
41             numErrors += 1
42     return numErrors/nr
43
44 def makeHgpPcm(Hx, Hz)->ndarray:
45     """
46     Makes a full parity check matrix including x and z
47     checks for a hypergraph product code of two other codes
48     """
49     Hx = hstack((Hx, zeros(Hx.shape, dtype=uint8)))
50     Hz = hstack((zeros(Hz.shape, dtype=uint8), Hz))
51     H = vstack((Hx, Hz))
52     return H
53
54 def main():
55     steanelogicals = \

```

```

56         array([\
57             [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
58             [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1]])
59     steaneH = array([[1, 0, 0, 1, 0, 1, 1],
60                     [0, 1, 0, 1, 1, 0, 1],
61                     [0, 0, 1, 0, 1, 1, 1]])
62     pers = linspace(2*10**(-6), 2*10**(-5), 20)
63     lers = []
64     nr = 10000
65     H = makeHgpPcm(steaneH, steaneH)
66     for per in pers:
67         print(f"per={per}")
68         lers.append(\
69             steaneLerCalc(H, nr, per, steanelogicals))
70     lers = array(lers)
71     std_err = (lers*(1-lers)/nr)**0.5
72     errorbar(pers, lers, yerr=std_err)
73     plot(pers, lers)
74     xlabel("Physical error rate")
75     ylabel("Logical error rate")
76     savefig("img/figures/steaneLookupThreshold.png")
77
78 if __name__ == "__main__":
79     main()

```