GNARSIL: SPLITTING STABILIZERS INTO GAUGES

by

Oskar Novak

A Thesis Submitted to the Faculty of the

WYANT COLLEGE OF OPTICAL SCIENCES

In Partial Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

In the Graduate College

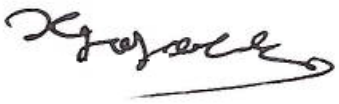
THE UNIVERSITY OF ARIZONA

2024

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Master's Committee, we certify that we have read the thesis prepared by **Oskar Alejandro Novak**, titled ***GNarsil: Splitting Stabilizers into Gauges,*** and recommend that it be accepted as fulfilling the dissertation requirement for the Master's Degree.

_____ _____ Date: __8/2/2024____
Professor Narayanan Rengaswamy



_____ _____ Date: 8/2/2024_____
Professor Christos Gagatsos

_____ _____ Date: _7/31/24_____
Professor Bane V. Vasic

Final approval and acceptance of this thesis is contingent upon the candidate's submission of the final copies of the thesis to the Graduate College.

I hereby certify that I have read this thesis prepared under my direction and recommend that it be accepted as fulfilling the Master's requirement.

_____ _____ Date: _8/2/2024_____
Professor Narayanan Rengaswamy
Master's Thesis Committee Chair
Wyant College of Optical Sciences

ACKNOWLEDGMENTS

Thank God for getting me through this research paper and my parents and friends for their support. Without their guidance and care, I would not have made it through this degree. I also want to thank my advisor, Narayanan Rengaswamy, for his incredible wisdom throughout writing this paper and his patience as I tried out a new idea of my own. I thank Nithin Raveendran and Bane Vasić at the University of Arizona for their insights into LP and quasi-cyclic (QC) LDPC codes, respectively, especially about finding circulant form generator matrices for QC-LDPC codes. I also thank Christos Gagatsos for agreeing to join my thesis committee along with Narayanan and Bane. Finally, I thank M. Sohaib Alam of the Quantum Artificial Intelligence Laboratory at NASA Ames Research Center for his insights on subsystem codes and our algorithms.

TABLE OF CONTENTS

ABSTRACT

Quantum subsystem codes have been shown to improve error-correction performance, ease the implementation of logical operations on codes, and make stabilizer measurements easier by decomposing stabilizers into smaller-weight gauge operators. In this paper, we present two algorithms that produce new subsystem codes from a "seed" CSS code. They replace some stabilizers of a given CSS code with smaller-weight gauge operators that split the remaining stabilizers, while being compatible with the logical Pauli operators of the code. The algorithms recover the well-known Bacon-Shor code computationally as well as produce a new $[\![9, 1, 2, 2]\!]$ rotated surface subsystem code with weight-3 gauges and weight-4 stabilizers. We illustrate using a $[\![100, 25, 3]\!]$ subsystem hypergraph product (SHP) code that the algorithms can produce more efficient gauge operators than the closed-form expressions of the SHP construction. However, we observe that the stabilizers of the lifted product quantum LDPC codes are more challenging to split into small-weight gauge operators. Hence, we introduce the subsystem lifted product (SLP) code construction and develop a new $[\![775, 124, 20]\!]$ code from Tanner's classical quasi-cyclic LDPC code. The code has high-weight stabilizers but all gauge operators that split stabilizers have weight 5, except one. In contrast, the LP stabilizer code from Tanner's code has parameters $[\![1054, 124, 20]\!]$. This serves as a novel example of new subsystem codes that outperform stabilizer versions of them. Finally, based on our experiments, we share some general insights about non-locality's effects on the performance of splitting stabilizers into small-weight gauges.

Chapter 1

Introduction

Quantum error correction is vital for quantum computers to achieve their full potential. The technique requires identifying the location of errors on a quantum computer without disturbing the delicate superposition states of the qubits involved in the computation. This is done by measuring stabilizers, which are quantum parity checks on different subsets of qubits, that help elucidate the locations of errors through an error syndrome. However, if measuring the stabilizers involves a high number of qubits, then the entangling measurements of the process pose the risk of creating additional errors.

Subsystem codes may help alleviate this issue Kribs et al. (2005); Bacon (2006). These are codes with gauge operators or operators on virtual qubits that do not carry quantum information. If designed well, then the eigenvalue of a high-weight stabilizer can be obtained as a product of the eigenvalues of several lower-weight gauge operators that can be measured more easily. Generally, these gauge operators form a non-abelian group, meaning that the order of measurement matters. The ordering can be chosen carefully during the process of *gauge fixing*, where changes to the code can be made mid-computation to help ease measuring stabilizers or even implementing logical operations Breuckmann (2011). Recent work has even translated these gauge-fixing insights of subsystem codes into the ZX calculus Huang et al. (2023). There are several examples of subsystem codes in the literature. The prototypical example of a subsystem code is the Bacon-Shor code Bacon (2006). Most constructions have topological or geometrically local properties, which makes finding the gauge group an intuitive process Higgott and Breuckmann (2021). Closed-form expressions exist

for constructing generalized Bravyi-Bacon-Shor or Subsystem Hypergraph Product (SHP) Codes Li and Yoder (2020). There are also some computational methods for forming a set of gauge generators out of a set of Pauli operators via Gram-Schmidt orthogonalization Wilde (2009), and computational search methods for finding the optimal subsystem code from a set of two-qubit measurement operators Crosswhite and Bacon (2011).

However, there does not exist an algorithm that allows one to input a stabilizer code and derive a subsystem code directly from it, especially one that determines gauge operators that compose to form stabilizers of the input code. In recent years, there has been tremendous progress in constructing quantum low-density parity-check (QLDPC) stabilizer codes with optimal code parameters. Such an algorithm can leverage these advances in stabilizer codes and potentially decompose their stabilizers into smaller-weight *local* gauge checks. For example, a good Lifted Product (LP) code can have stabilizer weights of 8 or even larger than 10, which involve long-range connections between qubits. Hence, the LDPC property alone does not make these constructions practical.

In this work, we present two algorithms capable of deriving subsystem codes from a "seed" CSS stabilizer code and present non-trivial examples of subsystem codes found by our algorithms. The algorithms identify a $[\![9, 1, 2, 2]\!]$ rotated surface subsystem code whose stabilizer weights are still 4 but gauge weights are 3, albeit with some non-locality. In contrast, the subsystem surface code known in the literature Bravyi et al. (2013) uses more qubits and has stabilizers of weight 6. Next, we demonstrate a modified SHP code that reduces the weight of gauge operators needed to produce a stabilizer compared to the closed-form expressions in Li and Yoder (2020). We observe that it is in general difficult to decompose stabilizers of the LP construction into small-weight gauge operators. Hence, we introduce an extension to the SHP construction that we call the *Subsystem Lifted-Product (SLP)* codes, which can have superior parameters compared to the Lifted Product stabilizer code constructed from the same base matrix. As an illustrative example, we produce a $[\![775, 124, 20]\!]$ SLP code from Tanner's classical quasi-cyclic LDPC code, whereas the corresponding LP code has

parameters $[\![1054, 124, 20]\!]$. The code has high-weight stabilizers but the gauge operators to produce the stabilizer are weight-5, except one of high weight. It remains to be seen if the high-weight nature of stabilizers or some of the gauges is intrinsic to the SLP construction.

Chapter 2

Background

2.1 Essentials of Quantum Mechanics

In quantum mechanics, the main objects of the theory are quantum states. In quantum computing, the states we care about are referred to as *qubits*, or two level quantum states. A qubit can be defined as a vector in $\mathbb{C}^2$, the two-dimensional complex vector space Rengaswamy (2020):

$$
\begin{aligned}
|\psi\rangle &\coloneqq \alpha\,|0\rangle + \beta\,|1\rangle\,, \\
|0\rangle &\coloneqq \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \\
|1\rangle &\coloneqq \begin{bmatrix} 0 \\ 1 \end{bmatrix},
\end{aligned}
\tag{2.1}
$$

where α and β are complex numbers. Here $\langle\psi|\psi\rangle = 1$, where $\langle\psi| \coloneqq |\psi\rangle^\dagger$. This implies that $|\alpha|^2 + |\beta|^2 = 1$. Here $\langle\psi|\phi\rangle$ is the inner product between two states, where $\langle\psi|\phi\rangle = \langle\phi|\psi\rangle^*$. We can extend this notion of a single qubit to multi-qubit quantum states, by using the $\mathbb{C}^N$, $N \coloneqq 2^n$ *computational basis*, $\{|v\rangle = |v_1\rangle \otimes |v_2\rangle \otimes ... |v_N\rangle \in \mathbb{C}^N, v_i \in \{0,1\}\}$, where $\otimes$ is the *Kronecker product*. Thus we can write an arbitrary N qubit quantum state as:

$$
|\psi\rangle = \textstyle\sum_{v\in\mathbb{Z}_2^n} \alpha_n\,|v\rangle\,, \qquad \alpha_n \in \mathbb{C}, \qquad \langle\psi|\psi\rangle = \textstyle\sum_{v\in\mathbb{Z}_2^n} |\alpha_n|^2 = 1
\tag{2.2}
$$

If a state is not factorizable into a Kronecker product of two states, it is *entangled* Rengaswamy (2020).

We now generalize our notion of quantum states from our previous section. The previous states are known as *pure states*, which have a well defined coherence, or phase. This extra information allows states with well defined phases to interfere with each other. For example, let $|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ and $|\phi\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$. Thus $|\psi\rangle + |\phi\rangle = |0\rangle$. We can therefore interpret the complex coefficients of our quantum states as complex amplitudes of a wave function that allows for interference between states. However, we may not always know the exact pure state of the system, and instead our knowledge may be limited to a statistical mixture of multiple pure states $\{|\psi_i\rangle, i \in \{1, N\}\}$. We can represent this more general quantum state as a density operator Rengaswamy (2020):

$$\rho = \sum_i p_i |\psi_i\rangle \langle\psi_i| \in \mathbb{C}^{N\times N}, \tag{2.3}$$

where the $\{p_i\}$ are the probabilities of measuring each pure state $\{|\psi_i\rangle\}$. Notice how these are typical probabilities, and not complex amplitudes, so the sum of two density operators will simply dilute the mixture of states, and not behave like a coherent superposition of waves. Our density operator is obviously Hermitian, or $\rho^\dagger = \rho$. Furthermore, $Tr(\rho) = 1$, which implies that the density operator preserves probability. However, $Tr(\rho^2) \leq 1$, which is only an equality when ρ represents a pure state, i.e. when $\rho = |\psi\rangle \langle\psi|$.

2.2 Time Evolution of Quantum States

In quantum mechanics, the time evolution of a pure state is given by the Schrödinger Equation Sakurai and Napolitano (2020),

$$i\hbar \frac{\partial}{\partial t} |\psi\rangle = \hat{H} |\psi\rangle, \tag{2.4}$$

where $\hat{H}$ is the Hamiltonian operator, derived by promoting the classical Hamiltonian functional density $\mathcal{H} \coloneqq \sum_i p_i \frac{\partial}{\partial t} q_i - \mathcal{L}$, to a Hermitian quantum operator, where p_i, $\frac{\partial}{\partial t} q_i$ are the classical canonical momenta and particle velocities respectively, and $\mathcal{L}$ is the Lagrangian density, given by $\mathcal{L} \coloneqq T - V$. Solutions to (2.4) where $\hat{H}$ is time-independent have the form $|\psi(t)\rangle = e^{\frac{-i}{\hbar}\hat{H}t} |\psi\rangle$, where $i \coloneqq \sqrt{-1}$. If we define $U(t) \coloneqq e^{\frac{-i}{\hbar}\hat{H}t}$, we see that $U(t)U^\dagger(t) = I$. Thus, the time evolution of pure states is given by *Unitary operators*, or operators U such that $UU^\dagger = U^\dagger U = I$. Since $|\det(U)| = 1$, we can see these operators preserve the information of the quantum system, and are akin to a change of basis over time Sakurai and Napolitano (2020). Given a density operator formed as $\rho = |\psi\rangle \langle\psi|$ we see that $\rho \to U\rho U^\dagger$. For a general Unitary operator $U(t)$ acting on an initial pure state $|\psi\rangle$, we can approximate $U(t)$ as a discrete sequence of Unitary operators at discrete time steps. These operators in the sequence can further be approximated by a set of unitaries called *quantum gates*, which are elements of $\mathbb{U}_N \coloneqq \{U \in \mathbb{C}^{N\times N} : UU^\dagger = U^\dagger U = I_N\}$, where $\mathbb{U}_N$ is the *unitary group* over N qubits and I_N is the N-dimensional identity matrix Rengaswamy (2020). Hence, a quantum computation can be seen as the discrete time evolution of an initial state to a final state. For the purposes of this thesis, we will focus only on the following single-qubit quantum gates known as the *Pauli gates*, given by the following Hermitian Unitary matrices:

$$I_2 \coloneqq \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad X \coloneqq \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Z \coloneqq \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad Y \coloneqq iXZ = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}. \tag{2.5}$$

2.3 Representing Pauli Operators as Binary Vectors

Given the vectors $a = [a_1, ..., a_n]$, $b = [b_1, ..., b_n] \in \mathbb{F}_2^n$, we define the Hermitian operator

$$E(a, b) \coloneqq i^{a_1 b_1} X^{a_1} Z^{b_1} \otimes ... \otimes i^{a_n b_n} X^{a_n} Z^{b_n}, \tag{2.6}$$

where X and Z are the standard Pauli operators. We can define the n-qubit Pauli group $\mathcal{P}_n$ as

$$\mathcal{P}_n \coloneqq \langle i^\alpha E(a,b) \ : \ a,b \in \mathbb{F}_2^n; \alpha \in \{0,1,2,3\} \rangle \, . \tag{2.7}$$

This is also known as the Heisenberg-Weyl Group $HW_N, \ N = 2^n$. The standard symplectic inner product in $\mathbb{F}_2^{2n}$ is defined as Rengaswamy et al. (2020):

$$\begin{aligned} \langle [a,b],[a',b'] \rangle_s &\coloneqq a'b^T + b'a^T \pmod 2 \\ &= [a,b] \ \mathbf{\Omega} \ [a',b']^T \pmod 2, \end{aligned} \tag{2.8}$$

where the matrix $\mathbf{\Omega} = \begin{bmatrix} \mathbf{0} & \boldsymbol{I}_n \\ -\boldsymbol{I}_n & \mathbf{0} \end{bmatrix}$ is the symplectic form in $\mathbb{F}_2^{2n}$. We notice that two operators $E(a,b)$, $E(a',b')$ commute if and only if $\langle [a,b],[a',b'] \rangle_s = 0 \pmod 2$. Thus, we see that a vector $[a,b] \in \mathbb{F}_2^{2n}$ is isomorphic to an operator $E(a,b)$ via the map $\gamma : \mathcal{P}_n / \langle i^\alpha \boldsymbol{I}_{2^n} \rangle \to \mathbb{F}_2^{2n}$ defined by

$$\gamma(E(a,b)) \coloneqq [a,b]. \tag{2.9}$$

Thus, without loss of generality, we can represent n-qubit Pauli operators as binary vectors. The *weight* of an n-qubit Pauli operator is the number of qubits on which it applies a non-identity Pauli operator, e.g., the operator $X_1 \otimes X_2 \otimes I_3$ can be described as $[\,1\ 1\ 0,\ 0\ 0\ 0\,]$, and its weight is 2.

2.4 Introduction to Quantum Error Correction

Unfortunately, quantum computers are not closed systems. They are in contact with an environment which can cause errors in our computation. Not only does this environment cause errors in our computation, but we typically have no knowledge of its configuration. Thus we must find a way to protect our computation from errors caused from an environment of which we only know of its existence. We thus posit that our joint computer environment

system can be described by the joint density operator $\rho = \rho_c \otimes \sigma$, where $\sigma = \sum_\alpha c_\alpha |\epsilon_\alpha\rangle \langle\epsilon_\alpha|$, where $\{|\epsilon_\alpha\rangle\}$ forms an orthonormal basis for the Hilbert space of our environment, and ρ_c is our state on the quantum computer. We model an error by a unitary operator acting on the joint Hilbert space of the system $\rho \to U\rho_c \otimes \sigma U^\dagger = \rho'$. However, all we known about the environment is that it exists. Thus, we must see the action of this error on our computation space when the environment is discarded, or marginalized. This can be done by tracing out, or taking the *partial trace* of the density operator with respect to the Hilbert space of the environment Nielsen and Chuang (2010), which is equivalent to subjecting the environment to a series of Von-Neumann measurements, which we then discard. We can model these as a set of projectors $P_\beta = \sum_{\beta j} |f_{\beta j}\rangle \langle f_{\beta j}|$, $\sum_\beta P_\beta = I_\sigma$, where $\{|f_{\beta j}\rangle\}$ forms an orthonormal basis for the environment. Thus we get:

$$\mathcal{E}(\rho_c)_\beta \coloneqq \mathrm{Tr}_\sigma \left(P_\beta U \rho_c \otimes \sigma U^\dagger P_\beta\right) = \mathrm{Tr}_\sigma \left(P_\beta U \rho_c \otimes \sigma U^\dagger\right), \tag{2.10}$$

where $\mathcal{E}(\rho_c)_\beta$ is a linear operator called a *quantum channel* or *super operator* describing the evolution of the state of the quantum computer with no knowledge of the environment. In general, quantum channel evolution is not unitary. We can write (2.10) more explicitly as:

$$\mathcal{E}(\rho_c)_\beta = \sum_{j,\alpha} \sqrt{c_\alpha} \langle f_{\beta j}| U |\epsilon_\alpha\rangle \rho_c \langle\epsilon_\alpha| U^\dagger |f_{\beta j}\rangle \sqrt{c_\alpha} = \sum_{j,\alpha} E_{\beta j\alpha} \rho_c E_{\beta j\alpha}^\dagger, \tag{2.11}$$

where $E_{\beta j\alpha} \coloneqq \sqrt{c_\alpha} \langle f_{\beta j}| U |\epsilon_\alpha\rangle$, which are known as Kraus operators. Thus we can decompose the action of our channel in terms of Kraus operators. For this particular β we can see that $\sum_{j,\alpha} E_{\beta j\alpha}^\dagger E_{\beta j\alpha} = \mathrm{Tr}_\sigma \left(U^\dagger U \sigma\right) = I$. We can now course grain our representation and sum over all β giving us:

$$\mathcal{E}(\rho_c) \coloneqq \sum_\beta \mathcal{E}(\rho_c)_\beta = \sum_\beta E_\beta \rho_c E_\beta^\dagger. \tag{2.12}$$

Thus for each β the probability for us to measure the outcome $\mathcal{E}(\rho_c)_\beta$ is $p_\beta = \mathrm{Tr}\left(\mathcal{E}(\rho_c)_\beta\right)$, with the post measurement state collapsing to:

$$\rho_\beta = \frac{\mathcal{E}(\rho_c)_\beta}{\mathrm{Tr}\left(\mathcal{E}(\rho_c)_\beta\right)}. \tag{2.13}$$

In general, $\sum_\beta E_\beta^\dagger E_\beta \leq 1$, but we will restrict ourselves to the equality case. Finally, we also restrict ourselves to the case where the Kraus operators in (2.12) are described by single qubit Pauli operators. This implies that that the original unitary U on the joint state is of the form $U = U_c \otimes U_\sigma$, which in most cases is the most likely error Nielsen and Chuang (2010).

Now that we have a mathematical way to describe error channels caused by the environment, we may ask if it is possible to recover the pre-channel state of our computation. Before we begin our computation, we first encode our state ρ onto a larger Hilbert space which we will call the *code space* $\mathcal{C}$, giving us Bacon:

$$\rho_{enc} = P_\mathcal{C} \rho P_\mathcal{C}. \tag{2.14}$$

Now let us assume that there exists a recovery mapping $\mathcal{R}() = \sum_j R_j() R_j^\dagger$ such that:

$$\mathcal{R}\left(\mathcal{E}\left(\rho_{enc}\right)\right) = \sum_k R_k \left(\sum_k E_j \rho_{enc} E_j^\dagger \right) R_k^\dagger = \alpha \rho_{enc}, \alpha \in \mathbb{C}. \tag{2.15}$$

Inserting (2.14) into (2.15) we notice that we can choose some unitary change of basis matrix that sends $\alpha P_\mathcal{C} \to R_j E_k P_\mathcal{C}$:

$$u_{kj} \alpha P_\mathcal{C} \coloneqq R_k E_j P_\mathcal{C}, \tag{2.16}$$

and taking the conjugate transpose of (2.16) $u_{kj}^* \alpha^* P_\mathcal{C} = P_\mathcal{C} E_j^\dagger R_k^\dagger$ and setting $j = i$, we then multiply this to the left of (2.16).

$$u_{ki}^* u_{kj} |\alpha|^2 P_\mathcal{C} = P_\mathcal{C} E_i^\dagger R_k^\dagger R_k E_j P_\mathcal{C}. \tag{2.17}$$

If we sum (2.17) over all k, since (2.15) is a trace preserving map, we get:

$$P_{\mathcal{C}} E_i^\dagger E_j P_{\mathcal{C}} = \sum_k u_{ki}^* u_{kj} |\alpha|^2 P_{\mathcal{C}} \coloneqq C_{ij} P_{\mathcal{C}}, \tag{2.18}$$

where C_{ij} is obviously Hermitian. After relabeling and looking at a set of particular states in the code space $\mathcal{C}$, we show that (2.18) implies that:

$$\langle \phi_i | \, E_k^\dagger E_l \, | \phi_j \rangle = C_{kl} \delta_{ij}. \tag{2.19}$$

We can interpret (2.19) as saying that we can correct errors that preserve the inner product of the codespace, i.e., errors that do not map a code-word to one that it is orthogonal to, and preserve the value of the nonzero inner-products up to a normalization constant. Finally, we point out that since we have assumed that our set of errors $\{E_k\}$ are single-qubit Pauli errors, we can also correct errors that are linear combinations of our $\{E_k\}$. Thus we can correct a continuum of errors by only designing a code that corrects this discrete set of errors. This is referred to as the *digitzation* of quantum noise Bacon.

2.5 Quantum Stabilizer Codes

A stabilizer group $\mathcal{S} \in \mathcal{P}_n$ is an abelian subgroup of the Pauli group that does not contain $-I$. The corresponding *stabilizer code* is defined by:

$$\mathcal{Q} \coloneqq \left\{ |\psi\rangle \in \mathbb{C}^{2^n} : S \, |\psi\rangle = |\psi\rangle \;\; \forall \; S \in \mathcal{S} \right\}. \tag{2.20}$$

A stabilizer code with n physical qubits and m independent stabilizer generators can encode $k = n - m$ logical qubits. The logical Pauli operators of the code come from the normalizer of $\mathcal{S}$ in $\mathcal{P}_n$, which is also its centralizer, defined as

$$\mathcal{N}(\mathcal{S}) \coloneqq \{ U \in \mathcal{P}_n : [U, S] = 0 \; \forall \; S \in \mathcal{S} \}, \tag{2.21}$$

where $[U, S] \coloneqq US - SU$ is the commutator of U and S. Here, notice that $\mathcal{S} \subset \mathcal{N}(\mathcal{S})$. Finally, the *minimum distance*, d, of the code is given by the minimum weight of any Pauli operator in $\mathcal{N}(\mathcal{S}) \setminus \mathcal{S}$, or the lowest weight of any logical operator. Thus, we denote the parameters of a quantum stabilizer code as $[\![n, k, d]\!]$. Finally, a CSS code is a stabilizer code whose stabilizer $\mathcal{S} = \langle \gamma^{-1}(\boldsymbol{H}_X), \gamma^{-1}(\boldsymbol{H}_Z) \rangle$, $\boldsymbol{H}_X \boldsymbol{H}_Z^T = \boldsymbol{0}$, where $\boldsymbol{H}_X$, $\boldsymbol{H}_Z$ are classical binary parity check matrices, and the map γ^{-1} is applied to each row of $\boldsymbol{H}_X$, $\boldsymbol{H}_Z$. From (2.19), we can show that stabilizer codes can correct all errors $\{E_k^\dagger E_l\}$ such that Nielsen and Chuang (2010):

$$E_{correctable} \coloneqq \{E_k^\dagger E_l : E_k^\dagger E_l \notin \mathcal{N}(\mathcal{S}) \setminus \mathcal{S} \vee E_k^\dagger E_l \in \mathcal{S}\}. \tag{2.22}$$

2.6 Ancilla Qubits

To determine where an error has occurred during a computation, we must measure its location without disturbing the state of the computer. Typically we employ *ancilla* qubits to do this, or extra qubits which are not part of the code's set of physical qubits. If we entangle our physical qubits with the ancilla qubits, we may measure the ancilla qubits and measure the outcome of our stabilizer measurements without disturbing the code space. To illustrate this, we use a 3-qubit bit-flip code as an example. This code can only correct one bit-flip error among the three physical qubits. We therefore encode our state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \to \alpha|000\rangle + \beta|111\rangle$. The stabilizers for the 3-qubit code are as follows Nielsen and Chuang (2010): Given a state $|xyz\rangle$, $x, y, z \in \mathbb{F}_2$, the action of S_1 on $|xyz\rangle$ is:

Stabilizers of the 3-qubit bit-flip code	
Stabilizer Identifier	Stabilizer Operator
S_1	$Z_1 Z_2$
S_2	$Z_2 Z_3$

Table 2.1: Here are the non-trivial stabilizer operators of the three-qubit bit-flip code. Note that the identity operator I is implicitly included in the stabilizer group.

$$Z_1 Z_2 \left|xyz\right\rangle = (-1)^{x \oplus y} \left|xyz\right\rangle, \tag{2.23}$$

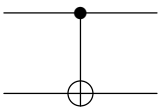
and similarly for S_2, where $x \oplus y$ is addition mod 2 or equivalently the XOR operation between x and y. This means that when x and y are the same, we get a $+1$ eigenvalue and -1 otherwise. To encode this information onto an ancilla qubit, we define a new operation the 2 qubit CNOT gate: We can represent the CNOT gate in circuit and matrix notation as

CNOT Gate	
Control and Target Input	Control and Target Output
$\left\|0\right\rangle_c \left\|0\right\rangle_t$	$\left\|0\right\rangle_c \left\|0\right\rangle_t$
$\left\|0\right\rangle_c \left\|1\right\rangle_t$	$\left\|0\right\rangle_c \left\|1\right\rangle_t$
$\left\|1\right\rangle_c \left\|0\right\rangle_t$	$\left\|1\right\rangle_c \left\|1\right\rangle_t$
$\left\|1\right\rangle_c \left\|1\right\rangle_t$	$\left\|1\right\rangle_c \left\|0\right\rangle_t$

Table 2.2: This table describes the action of the CNOT gate on a pair of qubits, where one acts as a control qubit and the other as a target.

follows Nielsen and Chuang (2010):

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} =$$



(2.24)

thus we can use the CNOT gate to encode the stabilizer information onto the ancillae as follows: thus each pair of CNOTs acting on the last two qubits encodes the stabilizer operations. We can now show that measuring the ancilla qubits can detect a single bit flip error on our code. Letting $\left|\psi\right\rangle_{enc} \rightarrow \sqrt{1-|p|^2}\left(\alpha\left|000\right\rangle + \beta\left|111\right\rangle\right) + p\left(\alpha\left|100\right\rangle + \beta\left|011\right\rangle\right) \coloneqq \left|\epsilon\right\rangle$, we now append to ancilla qubits $\left|00\right\rangle$ to our new state. Thus, following the circuit of our encoding scheme, we get that:

$$\left|\epsilon\right\rangle\left|00\right\rangle \rightarrow \sqrt{1-|p|^2}\left(\alpha\left|000\right\rangle + \beta\left|111\right\rangle\right)\left|00\right\rangle + p\left(\alpha\left|100\right\rangle + \beta\left|011\right\rangle\right)\left|10\right\rangle \tag{2.25}$$

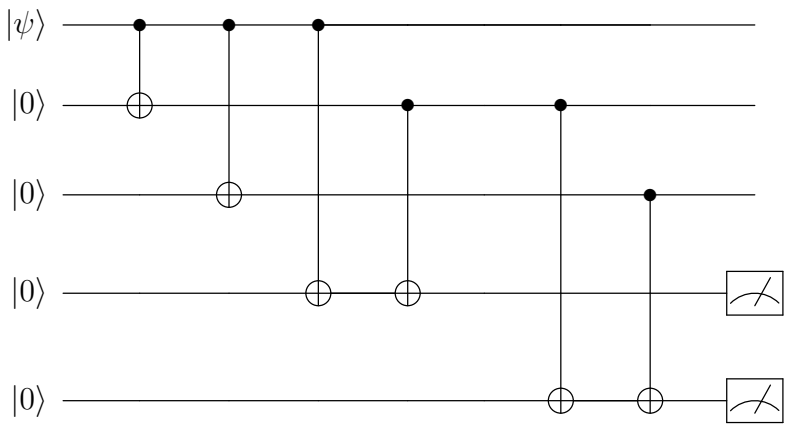


Figure 2.1: Ancilla Circuit for the 3-qubit Code

thus we can measure the error syndrome without disturbing the computation. This generalizes to all other single-bit flip errors in the code. Using ancillas, therefore, we can extract the information the stabilizers gave without disturbing the system.

2.7 Subsystem Codes

A subsystem code is a quantum error-correcting code that splits the code space $\mathcal{C} = A \otimes B$ into the logical subspace, A, and a gauge subspace, B, that does not carry any logical information Kribs et al. (2005). The Hilbert space of a subsystem code can be written as $\mathcal{H} = \mathcal{C} \oplus \mathcal{C}^{\perp} = A \otimes B \oplus \mathcal{C}^{\perp}$. The gauge subspace is supported on r gauge qubits such that, given n physical qubits and m independent stabilizers, the number of logical qubits is $k = n - m - r$. Thus, a subsystem code with these parameters and code distance d can be written as an $[\![n, k, r, d]\!]$ subsystem code. A subsystem code is defined by its gauge group

$$\mathcal{G} \coloneqq \langle iI, \mathcal{S}, X_1', Z_1', \ldots, X_r', Z_r' \rangle \subset \mathcal{P}_n, \tag{2.26}$$

where X_i', Z_i' are the Pauli operators for the i-th gauge qubit, and $\mathcal{S}$ is the stabilizer group of the code.

We can use $\mathcal{G}$ to define the stabilizers of the code:

$$\mathcal{S} \coloneqq \mathcal{Z}(\mathcal{G}) = \mathcal{C}(\mathcal{G}) \cap \mathcal{G}, \tag{2.27}$$

where $\mathcal{Z}(\mathcal{G})$ is the center of $\mathcal{G}$, and $\mathcal{C}(\mathcal{G})$ is the centralizer of $\mathcal{G}$ in $\mathcal{P}_n$. We define the bare logical operators $\mathcal{L}_b$ as:

$$\mathcal{L}_b \coloneqq \mathcal{C}(\mathcal{G}) \setminus \mathcal{G} = \mathcal{C}(\mathcal{G}) \setminus \mathcal{S}. \tag{2.28}$$

These are generated by k pairs of anti-commuting Pauli operators such that $[\mathcal{G}, \mathcal{L}_b] = 0$. These operators only act non-trivially on the subspace A. We also have the dressed logical operators $\mathcal{L}$, $\mathcal{L}_b \subset \mathcal{L}$, where:

$$\mathcal{L} = \mathcal{C}(\mathcal{S}) \setminus \langle i\mathcal{I} \rangle . \tag{2.29}$$

In other words, the set of dressed logical operators is the set of bare logical operators multiplied by an operator from $\mathcal{G} \setminus \langle i\mathcal{I} \rangle$.

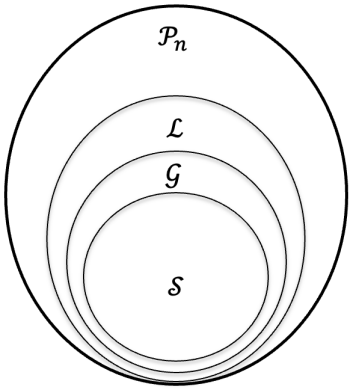


Figure 2.2: A diagram describing the relation of the Gauge group $\mathcal{G}$ to the other important subsets of $\mathcal{P}_n$ for a subsystem code.

Finally, we can define the code distance d such that

$$d \coloneqq \min_{P \in \mathcal{L}} |P|, \tag{2.30}$$

i.e., the minimum weight of a dressed logical operator.

Chapter 3

Binary Description of Subsystem Codes

3.1 Stabilizer Code Construction via Binary Matrices

We can describe a stabilizer code as a $2n \times 2n$ matrix $\boldsymbol{U}$ which has the following form Aaronson and Gottesman (2004); Dehaene and De Moor (2003); Rengaswamy et al. (2020):

$$\boldsymbol{U} = \begin{bmatrix} \mathcal{L}_X \\ \mathcal{S} \\ \mathcal{L}_Z \\ \mathcal{S}' \end{bmatrix}, \tag{3.1}$$

where its rows $\langle \boldsymbol{u}_1, \ldots, \boldsymbol{u}_{2n} \rangle$ span $\mathbb{F}_2^{2n}$. We are overloading notation to represent both the group as well as a binary matrix whose rows represent the generators of the group. Here, $\mathcal{L}_X$ and $\mathcal{L}_Z$ are the binary matrices that represent the generators of the logical Pauli operators. The sub-matrix $\mathcal{S}'$, otherwise known as the "destabilizer" Aaronson and Gottesman (2004), is added so that $\boldsymbol{U}$ has full rank. For a stabilizer code, we choose $\mathcal{S}'$ such that:

$$\boldsymbol{U}\boldsymbol{\Omega}\boldsymbol{U}^T = \boldsymbol{\Omega} \pmod 2. \tag{3.2}$$

This means that each row vector in $\mathcal{S}'$ is chosen so that it anti-commutes with only one stabilizer generator inside $\mathcal{S}$. In other words, $\boldsymbol{U}$ is a binary symplectic matrix Dehaene and De Moor (2003).

Theorem 1. *Let* $\boldsymbol{U}$ *be a binary matrix constructed as in* (3.1)*. If* $\boldsymbol{U}$ *satisfies* (3.2)*, then* $\boldsymbol{U}$ *describes a valid stabilizer code.*

Proof: The L.H.S. of (3.2) is our definition of the symplectic inner product in (2.8) extended to a matrix. Since $\boldsymbol{\Omega}$ has a single non-zero element per row, each row of $\boldsymbol{U}$ anti-commutes with exactly one other row, which is n rows below (or above) it. Such pairs of rows are called *symplectic pairs*. Since an $[\![n, k, d]\!]$ stabilizer code has $n - k$ stabilizers, the symplectic pairs of rows in $\mathcal{L}_X$ must be in the rows of $\mathcal{L}_Z$. This means that the codes' logical operators commute with the operators represented as binary vectors in $\mathcal{S}$, $\mathcal{S}'$. Thus, satisfying (3.2) verifies all necessary conditions for $\boldsymbol{U}$ to represent a valid stabilizer code. ■

3.2 Subsystem Code Construction via Binary Matrices

To construct a subsystem code using the methods of the previous section, we can take a binary matrix $\boldsymbol{U}$ constructed as in (3.1), and choose $2r$ rows of it to become our gauge generators, and $\mathcal{L}_X$, $\mathcal{L}_Z$ become our bare logical operators.

It should be noted that, generally, one can replace these $2r$ rows with other rows and still have a valid subsystem code, as long as $\boldsymbol{U}$ remains symplectic. Also, one is free to choose logical operators to become gauge operators.

Theorem 2. *Let* $\boldsymbol{U}$ *be a* $2n \times 2n$ *binary matrix that has the construction of* (3.1) *except that* $2r$ *of its rows have possibly been altered. We call this altered matrix* $\boldsymbol{U}'$*. Then* $\boldsymbol{U}'$ *represents a valid subsystem code if:*

$$\boldsymbol{U}'\boldsymbol{\Omega}\boldsymbol{U}'^{T} = \boldsymbol{\Omega} \pmod 2. \tag{3.3}$$

Proof: Since only the $2r$ rows that have been altered are those promoted to gauge operators, the rest of the rows of the matrix on the R.H.S. of (3.3) are identical to the rows in the same positions in (3.1). This means that the bare logical operators only anti-commute with their respective symplectic pairs and commute with the operators that generate the gauge group. Furthermore, this also means that since the leftover stabilizers in $\mathcal{S}$ only anti-

commute with the leftover operators in $\mathcal{S}'$, the operators of $\mathcal{S}$ indeed form $\mathcal{Z}(\mathcal{G})$. Finally, as long as the previous conditions are met, then (3.3) guarantees that $\boldsymbol{U}'$ represents a valid subsystem code. ■

Appendix 4.1 proves our algorithm's ability to find all suitable representations for the gauge generators.

Chapter 4

GNarsil Algorithms to Find Gauge Operators

Here we will use *gauge generators* to describe the non-stabilizer generators of the gauge group. In practice, a subsystem code's utility comes from measuring the product of a set of lower-weight gauge operators that yields the same information as measuring a higher-weight stabilizer Higgott and Breuckmann (2021). Using the binary matrix constructions of the previous section, we can find low-weight gauge operators for CSS codes whose composition produces the code's stabilizers up to a remaining gauge operator that is not a gauge generator. Let us call this remaining gauge operator the *residual operator* for that stabilizer. We refer to the minimum (Pauli) weight over all residual operators as the *residual weight*. For a gauge decomposition to be useful, the residual weight must be less than the weight of the decomposed stabilizer. To achieve this, we propose two algorithms Novak and Rengaswamy (2024). The first algorithm finds the set of r gauge generators that decompose stabilizers with the least residual weight and then adds back stabilizers so that $n = k - r - m$ is still satisfied. We describe the first *GNarsil* algorithm, which "cuts" stabilizers into gauges, in Algorithm 1.

We also note that by bypassing the anti-commutation conditions, gauge operators that are not necessarily gauge generators may be found such that they decompose a stabilizer with lower residual weight. We also provide a second *GNarsil* algorithm for this case in Algorithm 2.

Algorithm 1 *GNarsil* 1: Gauge Generators to Split Stabilizers

1: **Input:** $\boldsymbol{U} = \begin{bmatrix} \mathcal{L}_X \\ \mathcal{S} \\ \mathcal{L}_Z \\ \mathcal{S}' \end{bmatrix}$, $\{i_1, i_2, \ldots, i_r\} \subseteq \{k+1, k+2, \ldots, n\}$: indices of rows of $\boldsymbol{U}$ to be replaced with X-type gauge generators, desired Pauli weight w for gauge generators

2: **Initialization:**
$\mathcal{G}_X \leftarrow \emptyset,\ \mathcal{G}_Z \leftarrow \emptyset$
validXGauges $\leftarrow \emptyset,$ validZGauges $\leftarrow \emptyset$
maxSize $\leftarrow$ max. number of gauge candidates to consider;
$S_{\text{Xtargets}} \leftarrow$ indices of rows of $\boldsymbol{U}$ that are X-stabilizers;
$S_{\text{Ztargets}} \leftarrow$ indices of rows of $\boldsymbol{U}$ that are Z-stabilizers;
gaugesPerStab $\leftarrow$ # of gauge generators per stabilizer;
Xops $\leftarrow$ weight-w (row) vectors in $\{0,1\}^n$ appended with $\mathbf{0}$ at the end to make length $2n$ (for X-gauges);
Zops $\leftarrow$ weight-w (row) vectors in $\{0,1\}^n$ appended with $\mathbf{0}$ at the front to make length $2n$ (for Z-gauges);
$\boldsymbol{V} \leftarrow \boldsymbol{U}(1:(n+k))$ (i.e., remove $\mathcal{S}'$ from $\boldsymbol{U}$)
{Find X, Z gauge generators}
i in 1:size(Xops)
size(validXGauges) $\geq$ maxSize

3: **break**
Xops$(i)\,\boldsymbol{\Omega}\,\boldsymbol{V}^T = \mathbf{0} \pmod 2$ and

4: $\operatorname{rank}\left(\begin{bmatrix} \boldsymbol{V} \\ \text{Xops}(i) \end{bmatrix}\right) > \operatorname{rank}(\boldsymbol{V})$ and

5: $\operatorname{rank}\left(\begin{bmatrix} \text{validXGauges} \\ \text{Xops}(i) \end{bmatrix}\right) > \operatorname{rank}(\text{validXGauges})$

6: validXGauges $\leftarrow$ validXGauges $\cup$ Xops(i)
validXGauges $== \emptyset$

7: $w \leftarrow w + 1$
8: Regenerate Xops and Zops $w \geq n$
9: **stop** {Algorithm Fails}
10: **go** to Line 3
11: XgChoices $\leftarrow$ NCHOOSEK(validXGauges , gaugesPerStab)
{List of all combinations of gaugesPerStab elements in

12: validXGauges}
i in $\mathcal{S}_{\text{Xtargets}}$ j in 1:size(XgChoices)

13: resultantGauge $\leftarrow \boldsymbol{U}(i) + \text{XgChoices}(j) \pmod 2$
14: residualWeight$(j) \leftarrow$ HammingWeight(resultantGauge)
15: minIndex $\leftarrow$ argmin(residualWeight)
16: $\mathcal{G}_X \leftarrow \mathcal{G}_X \cup$ XgChoices(minIndex)
size$(\mathcal{G}_X) \geq r$

17: Drop all rows from $r+1$ (if they exist)
18: **break**

19: Clear the rows $\{i_1, i_2, \ldots, i_r\}$ and $\{n+i_1, n+i_2, \ldots, n+i_r\}$ from $\boldsymbol{U}$ to add gauge generators
20: $\boldsymbol{U}([i_1, i_2, \ldots, i_r]) \leftarrow \mathcal{G}_X$
21: Repeat from Line 3 for Zops by appropriately replacing variables; add Zops(i) to validZGauges only if HammingWeight$\big(\text{Zops}(i)\,\boldsymbol{\Omega}\,\mathcal{G}_X^T\big) = 1$
{Zops(i) anti-commutes with exactly one X-gauge}
22: $\boldsymbol{U}([n+i_1, n+i_2, \ldots, n+i_r]) \leftarrow \mathcal{G}_Z$
23: Replace unused destabilizers in $\mathcal{S}'$ such that $\boldsymbol{U}$ is symplectic
24: **Return:** $\boldsymbol{U}$, codeDistance($\boldsymbol{U}$)

4.1 Proof of Algorithm's ability to find all possible representations of a subsystem code using Binary Symplectic Matrices

Theorem 3. *Given an $[\![n, k, d]\!]$ CSS code with logical operators $\boldsymbol{X_i}, \boldsymbol{Z_i}$ for each logical qubit, the GNarsil algorithms can find all representations for the $2r$ gauge generators of the derived $[\![n, k, r, d]\!]$ subsystem code, where $2n-2r$ of the rows of the matrix $\boldsymbol{U}$ representing the code (3.1) are fixed.*

Proof: Let r be the number of gauge qubits chosen from the input code. The input code is described by a $2n \times 2n$ matrix $\boldsymbol{U}$, a symplectic matrix that describes the code's logical operators and stabilizers. The row vectors $\{\boldsymbol{u}_1, \ldots, \boldsymbol{u}_{2n}\}$ of $\boldsymbol{U}$ then span the space $\mathcal{U} = \mathbb{F}_2^{2n}$. We choose $2r$ vectors $\{\boldsymbol{u}_i : i \in \mathcal{I}\}$, $\mathcal{I} = \{i_1, i_2, \ldots, i_r, n+i_1, n+i_2, \ldots, n+i_r\}$, from $\boldsymbol{U}$ which correspond to a submatrix $\boldsymbol{U}_r$ of $\boldsymbol{U}$. These vectors from $\mathcal{I}$ form a symplectic basis for a 2^{2r}-dimensional subspace $\mathcal{U}_r \subseteq \mathcal{U}$. Since the subspace is symplectic, we see that any vector $\boldsymbol{u}_i, i \in \mathcal{I}$, has a symplectic product $\langle \boldsymbol{u}_i, \boldsymbol{u}_j \rangle_s = 0 \pmod 2$ with any vector $\boldsymbol{u}_j, j \notin \mathcal{I}$.

We now replace the vectors from $\boldsymbol{U}_r$ with vectors $\tilde{\boldsymbol{u}} \in \mathcal{U}_r$ such that $\langle \tilde{\boldsymbol{u}}_i, \tilde{\boldsymbol{u}}_{i+r} \rangle_s = 1 \pmod 2$. To see which pairs of vectors in $\mathcal{U}_r$ form valid symplectic pairs we arrange all of the symplectic products of vectors in $\mathcal{U}_r$ into a matrix $\boldsymbol{P} = \sum_{ij} \langle \boldsymbol{u'}_i, \boldsymbol{u'}_j \rangle_s \, |i\rangle \langle j| \pmod 2$, where $\boldsymbol{u'}_i, \boldsymbol{u'}_j \in \mathcal{U}_r$. We use $\boldsymbol{P}$ to count the number of possible representations generated of symplectic pairs of $\mathcal{U}_r$. The number of representations is defined as the number of unique matrices $\boldsymbol{V}$ built from $\boldsymbol{U}$ by replacing the vectors $\{\boldsymbol{u}_i : i \in \mathcal{I}\}$ with vectors $\tilde{\boldsymbol{u}}$ such that $\boldsymbol{V}\boldsymbol{\Omega}\boldsymbol{V}^T = \boldsymbol{\Omega}$, where $\boldsymbol{\Omega} = \left[\begin{smallmatrix} \boldsymbol{0} & \boldsymbol{I}_n \\ \boldsymbol{I}_n & \boldsymbol{0} \end{smallmatrix}\right]$.

Algorithm 2 *GNarsil* 2: Gauge Operators to Split Stabilizers

1: **Input:** $\boldsymbol{U} = \begin{bmatrix} \mathcal{L}_X \\ \mathcal{S} \\ \mathcal{L}_Z \\ \mathcal{S}' \end{bmatrix}$, $\{i_1, i_2, \ldots, i_r\} \subseteq \{k+1, k+2, \ldots, n\}$: indices of rows of $\boldsymbol{U}$ that will be removed to add gauge operators

2: **Initialization:**
$\mathcal{G}_X \leftarrow \emptyset$, $\mathcal{G}_Z \leftarrow \emptyset$
validXGauges $\leftarrow \emptyset$, validZGauges $\leftarrow \emptyset$
maxSize $\leftarrow$ max. number of gauge candidates to consider;
$S_{\text{Xtargets}} \leftarrow$ indices of rows of $\boldsymbol{U}$ that are X-stabilizers;
$S_{\text{Ztargets}} \leftarrow$ indices of rows of $\boldsymbol{U}$ that are Z-stabilizers;
gaugesPerStab $\leftarrow$ # of gauge operators per stabilizer;
numXGauges $\leftarrow$ gaugesPerStab $\times$ # of X stabilizers;
numZGauges $\leftarrow$ gaugesPerStab$\times$# of Z stabilizers;
Xops $\leftarrow$ weight-w vectors in $\{0,1\}^n$ appended with $\mathbf{0}$ at the end to make length $2n$ (for X-gauges);
Zops $\leftarrow$ weight-w vectors in $\{0,1\}^n$ appended with $\mathbf{0}$ at the beginning to make length $2n$ (for Z-gauges);
$\boldsymbol{V} \leftarrow \boldsymbol{U}(1:(n+k))$ (i.e., remove $\mathcal{S}'$ from $\boldsymbol{U}$)
{Find X, Z gauge operators}
i in 1:size(Xops)
size(validXGauges) $\geq$ maxSize

3: **break**
Xops(i) $\boldsymbol{\Omega}\, \boldsymbol{V}^T = \mathbf{0} \pmod 2$ and

4: Xops(i) $\notin \mathcal{L}_X$

5: validXGauges $\leftarrow$ validXGauges $\cup$ Xops(i)
validXGauges == $\emptyset$

6: $w \leftarrow w + 1$ $w \geq n$

7: **stop** {Algorithm Fails}

8: **go** to Line 3

9: XgChoices $\leftarrow$ NCHOOSEK(validXGauges , gaugesPerStab)
{List of all combinations of gaugesPerStab elements in

10: validXGauges}
i in $\mathcal{S}_{\text{Xtargets}}$ j in 1:size(XgChoices)

11: resultantGauge $\leftarrow \boldsymbol{U}(i)$ + XgChoices(j) (mod 2)

12: residualWeight(j) $\leftarrow$ HammingWeight(resultantGauge)

13: minIndex $\leftarrow$ argmin(residualWeight)

14: $\mathcal{G}_X \leftarrow \mathcal{G}_X \cup$ XgChoices(minIndex)

15: Remove rows $\{i_1, i_2, \ldots, i_r\}$ and $\{n+i_1, n+i_2, \ldots, n+i_r\}$ from $\boldsymbol{U}$; replace with numXGauges and numZGauges empty rows, respectively

16: $\boldsymbol{U}([i_1, i_2, \ldots, i_{\text{numXGauges}}]) \leftarrow \mathcal{G}_X$

17: Repeat from Line 3 for Zops by appropriately replacing variables; add Zops(i) to validZGauges only if HammingWeight$\big(\text{Zops}(i)\, \boldsymbol{\Omega}\, \mathcal{G}_X^T\big) \geq 1$
{Zops(i) may anti-commute with more than one X-gauge}

18: $\boldsymbol{U}([n+i_1, n+i_2, \ldots, n+i_{\text{numZGauges}}]) \leftarrow \mathcal{G}_Z$

19: **Return:** $\boldsymbol{U}$, codeDistance($\boldsymbol{U}$)

This condition is equivalent to all constraints needed to be satisfied by a valid subsystem code. The number of representations for a given r can be found by examining $\boldsymbol{P}$. For instance, in the case of $r = 2$ we see that:

$$\boldsymbol{P} = \begin{bmatrix} 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&1&1&1&0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1 \\ 0&0&0&0&1&1&1&1&1&1&1&1&0&0&0&0 \\ 0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1 \\ 0&1&0&1&1&0&1&0&0&1&0&1&1&0&1&0 \\ 0&1&0&1&0&1&0&1&1&0&1&0&1&0&1&0 \\ 0&1&0&1&1&0&1&0&1&0&1&0&0&1&0&1 \\ 0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1 \\ 0&0&1&1&1&1&0&0&0&0&1&1&1&1&0&0 \\ 0&0&1&1&0&0&1&1&1&1&0&0&1&1&0&0 \\ 0&0&1&1&1&1&0&0&1&1&0&0&0&0&1&1 \\ 0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0 \\ 0&1&1&0&1&0&0&1&0&1&1&0&1&0&0&1 \\ 0&1&1&0&0&1&1&0&1&0&0&1&1&0&0&1 \\ 0&1&1&0&1&0&0&1&1&0&0&1&0&1&1&0 \end{bmatrix}. \tag{4.1}$$

By examining $\boldsymbol{P}$, we first see that there are $2^{2r} - 1 = 15$ non-zero vectors of $\mathcal{U}_r$ that can be chosen as the first vector. Each of these vectors has $2^{2r-1} = 8$ available symplectic partners. For finding the third vector to add to $\boldsymbol{V}$, out of $2r$ needed vectors, we look for possible candidates by searching for the columns of $\boldsymbol{P}$ where the first two vectors' rows are 0, i.e., the third vector is symplectically orthogonal to the first two vectors added to $\boldsymbol{V}$. We find 4 such columns but note that one is the column corresponding to the trivial zero vector. Therefore, we have $2^{2r-2} - 1 = 3$ possible nontrivial choices for the third vector. Finally, we search for the compatible symplectic pairs of the third vector, which gives us $2^{2r-3} = 2$. Thus, our total number of subsystem codes for $r = 2$ is 720.

However, this number includes the multiplicity of codes due to the possible permutations of the $2r$ rows that still allow $\boldsymbol{V}$ to satisfy $\boldsymbol{V}\boldsymbol{\Omega}\boldsymbol{V}^T = \boldsymbol{\Omega}$. By looking at all possible permutations of the rows such that we preserve the symplectic pairs between the row vectors, we find that the multiplicity is 8 for $r = 2$. Thus, the total number of unique representations for $r = 2$ is 90. We can extrapolate this procedure to arbitrary r and find that the number of representations for a given $r \geq 2$ is

$$\frac{1}{\mathcal{M}} \left(\prod_{l \in \{0,1,2,\ldots,r-1\}} 2^{2r-2l} - 1 \right) \left(\prod_{m \in \{1,3,5,\ldots,2r-1\}} 2^{2r-m} \right), \tag{4.2}$$

where $\mathcal{M}$ is the multiplicity of the $2r$ rows. We conclude that these form all possible representations of the subsystem code as the gauge generators must necessarily come from $\mathcal{U}_r$. ■

Chapter 5

Examples Found by Our Algorithms

5.1 $[\![9, 1, 4, 3]\!]$ Bacon-Shor Code

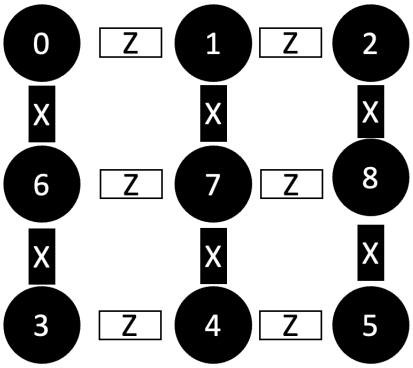


Figure 5.1: The Bacon-Shor code and its weight-2 gauge operators.

Algorithm (1) was able to find the $[\![9, 1, 4, 3]\!]$ Bacon-Shor code from the $[\![9, 1, 3]\!]$ Shor (stabilizer) code. This required first replacing two of the weight-2 Z-stabilizers with linearly independent weight-6 Z-stabilizers from the span of the weight-2 Z-stabilizers before running the algorithm. Finding this prototypical subsystem code example with our algorithm points to its validity. We note here that although six gauge operators are shown in Fig. 5.1, not all are linearly independent. The last two are the product of the linearly independent gauge operators and stabilizers. The specific pre-processing of the stabilizers above is a key step that can naturally be generalized to other CSS concatenated codes. Specifically, the inner code stabilizers can be multiplied to produce large-weight stabilizers, thereby allowing us to make low-weight gauges from the original (inner code) stabilizers.

5.2 $[\![9, 1, 2, 2]\!]$ Rotated Surface Subsystem Code

In this example, we present a novel subsystem version of the $[\![9, 1, 3]\!]$ rotated surface code found by Algorithm (1). The code has weight-4 stabilizer generators and weight-3 gauge operators, as shown in Fig. 5.2. We summarize the set of gauge operators, both generators and dependent ones, in Table 5.1. Using the dependent gauges and gauge generators GX_1, GZ_1, we can measure all of the weight 4 stabilizers shown above at the cost of the dependent gauge operators being not fully local and a small loss of distance. The well-known subsystem version of the surface code in the literature Higgott and Breuckmann (2021); Bravyi et al. (2013) has weight-6 stabilizers instead of the usual weight-4 and requires significantly more qubits, e.g., for lattice size $L = 3$ the code uses $3L^2 + 4L + 1 = 40$ qubits. It is not unreasonable to consider the code in Fig. 5.2, but it will be interesting to explore whether there is an intermediate subsystem surface code between these two solutions that still has distance 3.

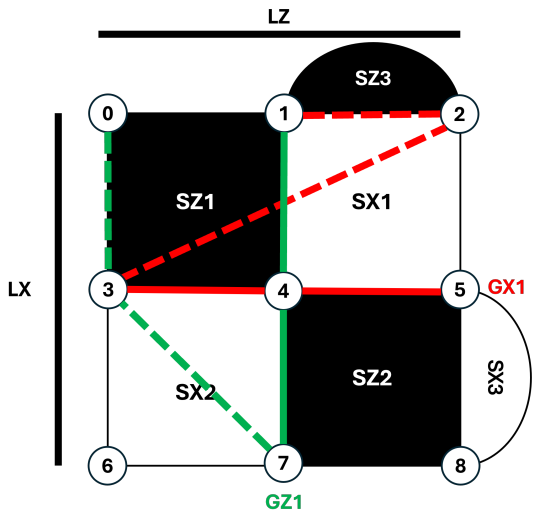


Figure 5.2: The rotated surface subsystem code found by Algorithm 1. The two red (resp. green) operators are the X-gauges (resp. Z-gauges), all weight-3 (also see Table 5.1). The product of the pair of red gauge operators, $X_1X_2X_3$ and $X_3X_4X_5$, gives SX_1, and the product of the pair of green operators gives SZ_1. Stabilizers SX_2, SZ_2 can be obtained by multiplying GX_1, GZ_1 by the respective stabilizers to find the respective dependent gauges. GX_2, GZ_2 are not shown here.

5.3 $[\![100, 25, 3]\!]$ Subsystem Hypergraph Product (SHP) Code

Using Algorithm (2), we present a $[\![100, 25, 3]\!]$ SHP code built from a $[10, 5]$ linear code Ryan et al. (2004) with parity-check matrix

$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}. \tag{5.1}$$

List of $[\![9, 1, 2, 2]\!]$ subsystem code gauge operators	
Gauge Identifier	Gauge Operator
GX_1	$X_3X_4X_5$
GX_2	$X_3X_4X_7$
GX_{1d}	$X_1X_2X_3$
GX_{2d}	$X_5X_6X_7$
GZ_1	$Z_1Z_4Z_7$
GZ_2	$Z_4Z_5Z_8$
GZ_{1d}	$Z_0Z_3Z_7$
GZ_{2d}	$Z_1Z_5Z_8$

Table 5.1: Gauge operators of the $[\![9, 1, 2, 2]\!]$ rotated surface subsystem code. GX_1, GX_2, GZ_1, GZ_2 are the independent gauge generators. The letter 'd' in subscripts indicates dependent gauge operators that are products of the gauge generators and stabilizers. The stabilizers are shown in Fig. 5.2.

We construct the SHP code with the following definitions Li and Yoder (2020):

$$\begin{aligned}
\mathcal{G}_X &\coloneqq (\boldsymbol{H} \otimes \boldsymbol{I}_n)\,, \\
\mathcal{G}_Z &\coloneqq (\boldsymbol{I}_n \otimes \boldsymbol{H})\,, \\
\mathcal{L}_X &\coloneqq (\boldsymbol{I}_n \otimes \boldsymbol{G})\,, \\
\mathcal{L}_Z &\coloneqq (\boldsymbol{G} \otimes \boldsymbol{I}_n)\,, \\
\mathcal{S}_X &\coloneqq (\boldsymbol{H} \otimes \boldsymbol{G})\,, \\
\mathcal{S}_Z &\coloneqq (\boldsymbol{G} \otimes \boldsymbol{H})\,,
\end{aligned} \tag{5.2}$$

where $\boldsymbol{G}$ is the generator matrix of the code defined by the parity check matrix $\boldsymbol{H}$, i.e, $\boldsymbol{G}\boldsymbol{H}^T = 0$. Thus, in general, the SHP construction in this form gives an $[\![n^2, k^2, d]\!]$ code with n, k, d being the respective values for the classical code defined by the parity check matrix $\boldsymbol{H}$. Here, the key observation is that although the stabilizers for this code are weight-12, our algorithm finds a decomposition that only requires weight-4 operators with a residual weight of 0 (resp. 5) for the X-stabilizers (resp. Z-stabilizers). This is in contrast to the gauge operators found from the closed-form expression in (5.2), which are all weight-4, requiring residual weights of 6 and 16 for X- and Z-stabilizers respectively. Thus, not only does our algorithm find more gauge operators than the closed-form expression, it also finds gauge operators that decompose the given stabilizers more efficiently than the closed-form operators. This decomposition makes this high-rate code far easier to implement. We also note that high-weight stabilizers are common for the SHP construction, as shown by the $[\![49, 16, 3]\!]$ SHP code Li and Yoder (2020) built from the $[7, 4, 3]$ Hamming code with stabilizer weights 12 and 16, respectively. The parity check matrix for the $[7, 4, 3]$ Hamming code is given below.

$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix}. \tag{5.3}$$

Hence, our algorithm is likely useful to determine more efficient implementations of SHP codes in general.

Chapter 6

Subsystem Lifted-Product (SLP) Codes

As a natural extension of hypergraph product codes, lifted-product (LP) CSS codes Panteleev and Kalachev (2021) are constructed from the hypergraph product of base parity-check matrices $\boldsymbol{A}$ and $\boldsymbol{A}^*$, which is then lifted over the ring $R = \mathbb{F}_q(G)$, where G is some group. The resulting parity-check matrices are as follows:

$$\begin{aligned} \boldsymbol{H}_X &= [\boldsymbol{A} \otimes \boldsymbol{I}, \boldsymbol{I} \otimes \boldsymbol{A}] , \\ \boldsymbol{H}_Z &= [\boldsymbol{I} \otimes \boldsymbol{A}^*, \boldsymbol{A}^* \otimes \boldsymbol{I}] . \end{aligned} \tag{6.1}$$

where $\boldsymbol{A}^*$ is the conjugate transpose of $\boldsymbol{A}$. This construction is typically to obtain good quantum LDPC codes, but its sparsity ends up producing non-local stabilizer operators. When LDPC codes are input into (6.1), and the resulting $\boldsymbol{H}_X, \boldsymbol{H}_Z$ are used as input for Algorithm 2, the algorithm is able to find several gauge operators for these codes. However, none of them are useful since the residual weight will greatly exceed the weight of the stabilizers themselves. This fact seems to stem from the sparse nature of the stabilizers: in order to commute with them, the gauge operators must also be sparse, which in turn requires a larger amount of residual weight to cover the spread of the stabilizers. The precise understanding of this situation for LP codes is an interesting future direction.

The SLP Construction: To find a lifted code with a better gauge decomposition, one is tempted to use the SHP construction in (5.2) as a natural extension of LP codes. We will refer to this as the SLP construction. By employing the base parity check-matrix $\boldsymbol{A}$ along

with its (base) generator matrix $\boldsymbol{G_A}$ into (5.2) and then lifting the construction by a circulant of size L, we obtain an $[\![Ln^2, Lk^2, D]\!]$ subsystem code, where n, k are the respective values defined by the base parity-check matrix $\boldsymbol{A}$. Here, $\boldsymbol{A}\boldsymbol{G}_{\boldsymbol{A}}^{*} = 0$, which implies that $\mathcal{G}_X\mathcal{S}_Z^{*} = 0$. This lifting procedure yields codes with potentially larger distances than a typical SHP code. We can demonstrate the lifting procedure by lifting the following matrix by $L = 2$ circulant matrices:

$$\begin{bmatrix} 1 & x \\ x & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} \tag{6.2}$$

6.1 Comparing the SHP and SLP Constructions

For the most direct comparison, we choose a binary base matrix:

$$\boldsymbol{A} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}, \tag{6.3}$$

where the generator matrix of (6.3) is given by:

$$\boldsymbol{G_A} = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}. \tag{6.4}$$

Placing into (5.2), this yields a $[\![9, 1, 3]\!]$ SHP code, which is simply the Bacon-Shor code. For the SLP construction, we interpret (6.3) as the matrix of powers of monomials corresponding to $L = 2$ circulant matrices. Thus, we define our new base matrix as:

$$\boldsymbol{B} = \begin{bmatrix} 1 & x & x \\ x & x & 1 \end{bmatrix}. \tag{6.5}$$

Clearly, Eqn. (6.4) interpreted in the same fashion is no longer a valid solution, so we must use another generator matrix. It can be shown that the following matrix is a valid generator matrix for the base matrix $\boldsymbol{B}$:

$$\boldsymbol{G_B} = \begin{bmatrix} 1+x & 1+x & 0 \\ 1+x & 0 & 1+x \\ x & 0 & 1 \end{bmatrix}. \tag{6.6}$$

This yields a $[\![18, 2, 2]\!]$ SLP code. The SLP code here has the same rate as the SHP code but with more physical qubits and a small loss in distance. However, we will see other cases where the distance fairs favorably compared to an LP code with the same base matrix.

6.2 Finding the Generator Matrix of a Base Parity Check Matrix $\boldsymbol{A}$

Again, taking (6.5) as our parity check matrix, we wish to find a systematic way to find (6.6). At first, Gaussian Elimination seems like the obvious answer, but after a set of row operations, we end up with the matrix:

$$\begin{bmatrix} 1 & x & x \\ 0 & 1+x & 0 \end{bmatrix}, \tag{6.7}$$

However, the polynomial $1+x$ is not invertible over the ring we use to lift the matrix. This means our systematic approach via Gaussian elimination must end here, and the result must be intuited. With larger base matrices and higher-order lifts, this becomes a much taller task than in this case. However, Smarandache et al. (2022) proposes a systematic way to find code words for the generator matrix given a quasi-cyclic parity check base matrix of polynomials as we see for the SLP construction. Thus, given an $m \times n$ quasi-cyclic polynomial parity check matrix $\boldsymbol{A}$, we define an subset $\mathcal{M} \in \{1, \cdots, n\}$ of cardinality m. Then we define a

vector $\boldsymbol{c}(x) = [c_1(x), c_2(x), \cdots, c_n(x)]$, where:

$$c_i(x) \coloneqq \begin{cases} \det^* (\boldsymbol{A})_{\mathcal{M}\setminus i} = \Delta_{\mathcal{M}\setminus i}, & i \in \mathcal{M} \\ 0, & \text{else} \end{cases} \tag{6.8}$$

then $\boldsymbol{c}(x)$ is a code word of the code generated by $\boldsymbol{G_A}$. Since we have a method for finding code words of our generator matrix polynomial form, we naively may try to build a matrix $\boldsymbol{G_A}$ by simply putting $n-k$ rows of code words into it in base matrix form. However, this matrix is not guaranteed to have the correct rank once it is lifted to binary. Looking at our previous example in (6.2), we see that although the base matrix has full rank, only two of the four rows in binary are linearly dependent. This is because we are lifting using circular permutation matrices of the identity; thus, it stands to reason that some rows may eventually be duplicated. Thus, we must add extra rows to $\boldsymbol{G_A}$ to compensate for this discrepancy. We notice that for some circulant lift size $L = N$, we can always show that:

$$\sum_{i=0}^{N-1} x^{i+j} = \sum_{i=0}^{N-1} x^i, j \in \{1, \cdots, N-1\} \tag{6.9}$$

Thus, since the matrix given by $\sum_{i=0}^{N-1} x^i$ is the all ones matrix, its row has rank one in binary. We can, therefore add permutations of the length n vector $\boldsymbol{a}(x) = \left[\sum_{i=0}^{N-1} x^i, \cdots, 0, \cdots, \sum_{i=0}^{N-1} x^i\right]$ to our matrix $\boldsymbol{G_A}$ to achieve the correct rank. Finally, for one special case were the matrix $\boldsymbol{A}$ contains a $\sum_{i=0}^{N-1} x^i$ as one of its elements, you may also replace one of the $\boldsymbol{a}(x)$'s for a vector $\boldsymbol{b}(x) = \left[1, \cdots, 0, \cdots, \sum_{i=0}^{N-1} x^i\right]$, where the 1 is located in the column of the $\sum_{i=0}^{N-1} x^i$ in $\boldsymbol{A}$. This reduces the number of rows needed to get to a full rank generator matrix, as the rank of the row $\boldsymbol{b}(x)$ in binary will be L. We now look at a few non-trivial examples of the SLP construction and their performance.

6.3 $[\![27, 12, 2]\!]$ SLP code

Given $L = 3$, the $[\![27, 12, 2]\!]$ SLP code can be constructed by the following base matrix:

$$\boldsymbol{A} = \begin{bmatrix} 1 + x + x^2 & 1 + x & x \end{bmatrix}, \tag{6.10}$$

along with the base generator matrix of the code:

$$\boldsymbol{G_A} = \begin{bmatrix} x^2 & x & 1 \\ x & x^2 & x \\ 1 & 0 & 1 + x + x^2 \end{bmatrix}. \tag{6.11}$$

After inserting these into (5.2) and lifting the construction, we find that the Pauli weight of the code's stabilizers is 18, and the Pauli weight of the gauge operators is 6. We also find that the gauge operators in the closed-form construction decompose the stabilizers with a residual weight of 9, given three weight-6 gauge operators as input. However, our algorithm is able to decompose the stabilizers with residual weights of $4, 6, 8$ for the X- and Z-stabilizers, an improvement over the closed-form gauge operator decomposition. Finally, we also note that the rate of the SLP construction is $\frac{4}{9}$, which is greater than the rate of the $[\![39, 12, 2]\!]$ LP code constructed from the same base matrix, $\frac{4}{13}$. Note that this is also while preserving the distance of the code.

6.4 $[\![775, 124, 20]\!]$ SLP code

For our final example, we give the SLP code constructed by Tanner's $(3, 5)$ QC-LDPC code with $L = 31$. Here, we use the following base matrix Raveendran et al. (2022):

$$\boldsymbol{B} = \begin{bmatrix} x & x^2 & x^4 & x^8 & x^{16} \\ x^5 & x^{10} & x^{20} & x^9 & x^{18} \\ x^{25} & x^{19} & x^7 & x^{14} & x^{28} \end{bmatrix}. \tag{6.12}$$

As shown by Smarandache, and then Chimal-Dzul, Lieb, and Rosenthal Smarandache et al. (2022), Chimal-Dzul et al. (2022), a generator matrix for (6.12) can be constructed using the matrix:

$$
\begin{aligned}
\boldsymbol{G_B} &= \begin{bmatrix}
u_{11} & u_{12} & u_{13} & u_{14} & 0 \\
u_{21} & u_{22} & u_{23} & 0 & u_{25} \\
f & f & 0 & 0 & 0 \\
f & 0 & f & 0 & 0 \\
f & 0 & 0 & f & 0 \\
f & 0 & 0 & 0 & f
\end{bmatrix}, \\
u_{11} &= x^{28} + x^{25} + x^{18} + x^{16} + x^5 + x, \\
u_{12} &= x^{23} + x^{22} + x^{20} + x^{17} + x^7 + x^4, \\
u_{13} &= x^{29} + x^{25} + x^{21} + x^{12} + x^5 + x, \\
u_{14} &= u_{25} = x^{28} + x^{18} + x^{16} + x^{14} + x^9 + x^8, \\
u_{21} &= x^{27} + x^{24} + x^{19} + x^{11} + x^{10} + x^2, \\
u_{22} &= x^{30} + x^{28} + x^{26} + x^{18} + x^{16} + x^6, \\
u_{23} &= x^{20} + x^{14} + x^9 + x^8 + x^7 + x^4, \\
f &= \frac{x^{31} - 1}{x - 1} = x^{30} + \cdots + x + 1.
\end{aligned} \tag{6.13}
$$

We find that the code has stabilizer weights of 120, 310, and 465 and gauge operator weights of 5. Due to the large number of physical qubits, this code is beyond the practical scale that our algorithm can handle. Our algorithm is suited for small- to medium-sized codes due to its exponential memory complexity, but this decomposition by GNarsil may be achievable with a sufficiently large amount of memory. Our SLP example demonstrates that the SLP construction can be quite advantageous if a generator matrix can be found for a certain base matrix. For the same base matrix (6.12), an LP($\boldsymbol{B}$,$\boldsymbol{B}^*$) construction yields a $[\![1054, 124, 20]\!]$ code, which trails the SLP version both in rate. If a circulant form generator

matrix is found for a base matrix, then the resulting SLP code will have a higher rate than its LP counterpart with the same code dimension. No relationship between the distance of the two constructions has been formalized, but it seems that the SLP code will at least have the same distance as its LP counterpart. This makes the SLP construction for a given base matrix a very attractive construction whenever possible.

Chapter 7

Conclusion

We have introduced a new set of algorithms, which we call *GNarsil*, for deriving subsystem codes from an input "seed" stabilizer code. We have demonstrated that these algorithms not only recover well-known examples of subsystem codes but also find interesting new ones, such as a novel version of the rotated surface subsystem code and a new SHP code that is more efficient than the closed-form construction in Li and Yoder (2020). We also reported that we could not find useful gauge decompositions of LP codes due to the highly non-local structure of the stabilizers. However, by using our new SLP construction, we can construct codes with excellent parameters that also have promising stabilizer decomposition properties, which our algorithms can improve. As noted, this also depends on finding a generator matrix for the base matrix in circulant form, which is not always guaranteed to exist.

We foresee these algorithms becoming useful tools for finding subsystem versions of stabilizer codes that may prove easier to implement due to the stabilizers' decomposition into smaller-weight gauge operators. However, it is clear that this problem remains complex, as the solution spaces for useful gauge generators are sparse, making it likely that our algorithms are optimal for this case, even though they are exponentially complex in memory. Thus, our algorithms work best for small- to medium-sized codes. Improving our algorithms' performance will most likely require specializing them for certain code constructions. This would allow us to exploit code symmetries to find optimal gauge operators.

We also foresee that the measure of residual weight can be useful for understanding the properties of good subsystem codes and the symmetries that they may possess. Finally, we

wish to extend this work into looking at the relationship between fault-tolerant operations on a stabilizer code and its derived subsystem code(s) by extending tools such as the LCS algorithm Rengaswamy et al. (2020). This can potentially be approached by observing how new gauge operators found by our algorithm change the structure of logical gates for our code from the seed stabilizer code. Insights into this relationship may be useful for the development of logical operations on Floquet codes Hastings and Haah (2021), at least the ones with parent subsystem codes Davydova et al. (2023).

References

S. Aaronson and D. Gottesman. Improved simulation of stabilizer circuits. *Physical Review A*, 70(5):052328, 2004. URL https://arxiv.org/abs/quant-ph/0406196.

D. Bacon. Cse 599d - quantum computing the quantum error correcting criteria.

D. Bacon. Operator quantum error-correcting subsystems for self-correcting quantum memories. *Physical Review A*, 73(1):012340, 2006. URL https://arxiv.org/abs/quant-ph/0506023.

S. Bravyi, G. Duclos-Cianci, D. Poulin, and M. Suchara. Subsystem surface codes with three-qubit check operators. *Quant. Inf. Comp.*, 13:0963–0985, 2013. URL https://arxiv.org/abs/1207.1443.

N. P. Breuckmann. Quantum subsystem codes: Their theory and use, 2011.

H. Chimal-Dzul, J. Lieb, and J. Rosenthal. Generator matrices of quasi-cyclic codes over extension fields obtained from Gröbner basis. *IFAC-PapersOnLine*, 55(30):61–66, 2022.

G. M. Crosswhite and D. Bacon. Automated searching for quantum subsystem codes. *Physical Review A*, 83(2):022307, 2011. URL https://arxiv.org/abs/1009.2203.

M. Davydova, N. Tantivasadakarn, and S. Balasubramanian. Floquet codes without parent subsystem codes. *PRX Quantum*, 4(2):020341, 2023. URL https://arxiv.org/abs/2210.02468.

J. Dehaene and B. De Moor. Clifford group, stabilizer states, and linear and quadratic operations over GF(2). *Physical Review A*, 68(4):042318, 2003. URL https://arxiv.org/abs/quant-ph/0304125.

M. B. Hastings and J. Haah. Dynamically generated logical qubits. *Quantum*, 5:564, 2021. URL https://arxiv.org/abs/2107.02194.

O. Higgott and N. P. Breuckmann. Subsystem codes with high thresholds by gauge fixing and reduced qubit overhead. *Physical Review X*, 11(3):031039, 2021. URL https://arxiv.org/abs/2010.09626.

J. Huang, S. M. Li, L. Yeh, A. Kissinger, M. Mosca, and M. Vasmer. Graphical CSS code transformation using ZX calculus. *arXiv preprint arXiv:2307.02437*, 2023. URL https://arxiv.org/abs/2307.02437.

D. Kribs, R. Laflamme, and D. Poulin. Unified and generalized approach to quantum error correction. *Physical review letters*, 94(18):180501, 2005. URL https://arxiv.org/abs/quant-ph/0412076.

M. Li and T. J. Yoder. A numerical study of Bravyi-Bacon-Shor and subsystem hypergraph product codes. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 109–119. IEEE, 2020. URL https://arxiv.org/abs/2002.06257.

M. A. Nielsen and I. L. Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

O. Novak and N. Rengaswamy. Gnarsil: Splitting stabilizers into gauges. *arXiv preprint arXiv:2404.18302*, 2024.

P. Panteleev and G. Kalachev. Quantum LDPC codes with almost linear minimum distance. *IEEE Transactions on Information Theory*, 68(1):213–229, 2021. URL https://arxiv.org/abs/2012.04068.

N. Raveendran, N. Rengaswamy, F. Rozpędek, A. Raina, L. Jiang, and B. Vasić. Finite rate QLDPC-GKP coding scheme that surpasses the CSS hamming bound. *Quantum*, 6:767, 2022. URL https://arxiv.org/abs/2111.07029.

N. Rengaswamy. *Classical coding approaches to quantum applications*. PhD thesis, Duke University, 2020.

N. Rengaswamy, R. Calderbank, S. Kadhe, and H. D. Pfister. Logical Clifford synthesis for stabilizer codes. *IEEE Transactions on Quantum Engineering*, 1:1–17, 2020. URL https://arxiv.org/abs/1907.00310.

W. E. Ryan et al. An introduction to LDPC codes. *CRC Handbook for Coding and Signal Processing for Recording Systems*, 5(2):1–23, 2004.

J. J. Sakurai and J. Napolitano. *Modern quantum mechanics*. Cambridge University Press, 2020.

R. Smarandache, A. Gómez-Fonseca, and D. G. Mitchell. Using minors to construct generator matrices for quasi-cyclic ldpc codes. In *2022 IEEE International Symposium on Information Theory (ISIT)*, pages 548–553. IEEE, 2022.

M. M. Wilde. Logical operators of quantum codes. *Physical Review A*, 79(6):062322, 2009. URL https://arxiv.org/abs/0903.5256.