# Practical and Efficient Quantum Error Correction

*Oscar Higgott*

A dissertation submitted in partial fulfillment

of the requirements for the degree of

**Doctor of Philosophy**

of

**University College London**.

Department of Physics and Astronomy

University College London

Thesis submission date: October 1, 2023

I, Oscar Higgott, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

# Abstract

Building a scalable quantum computer requires the use of quantum error correction, which protects components and quantum operations from noise and imperfections that would otherwise corrupt the computation. However, quantum error correcting codes add significant redundancy, with a large number of physical qubits used to encode each logical qubit. Furthermore, the control software used to operate a quantum error correcting code, called the decoder, must be fast enough to keep up with the hardware and accurate enough to identify which errors occurred with high probability.

In Part I of this thesis, we focus on methods for decoding topological quantum codes including the surface code, which currently forms the basis of many experimental efforts to build a quantum computer. We also introduce optimal local unitary circuits for encoding unknown states in surface codes. We present sparse blossom, a decoder which can process data fast enough to keep up with superconducting quantum computers in a regime of practical interest. We also introduce belief-matching, a computationally efficient decoder which has improved accuracy, and schedule-induced gauge fixing, a technique for decoding subsystem surface codes more effectively by improving the circuits used for their implementation.

In Part II we construct quantum error correcting codes derived from tilings of negatively curved surfaces. These constructions exploit properties of hyperbolic geometry to improve the encoding efficiency. By reducing the size of operators that must be measured, we find efficient parallelised quantum circuits implementing our constructions, which we show outperform the surface code for practical physical error rates.

# Impact Statement

Quantum computers exploit the laws of quantum mechanics to solve some computational problems much faster than is known to be possible using conventional computers. Solving these problems could help design new batteries, photovoltaics and pharmaceuticals, and will impact public key cryptosystems. Quantum computers are extremely fragile, and for most applications they will require the use of quantum error correction to protect them from noise that would otherwise destroy the computation. However, quantum error correction adds a large amount of redundancy: recent estimates suggest that millions of qubits will be needed to solve useful problems on a quantum computer, with the vast majority of these resources attributable to the cost of quantum error correction. Furthermore, the control software used to operate a quantum error correction scheme, called the decoder, must accurately process measurement data from the device at a very high rate.

In Part I of this thesis, we present faster and more accurate decoders for a family of quantum error correcting codes that includes the surface code, which is the code most widely considered for experimental realisation. It is important that decoders are fast enough to keep up with practical experimental realisations of quantum computers, to prevent a backlog of measurement data that grows exponentially in the size of the computation. A more accurate decoder enables a quantum computer with some target logical performance to be built using fewer physical components, or using components that are noisier and therefore easier to build. The belief-matching decoder we present has already been use in an experiment to demonstrate the suppression of quantum errors using the surface code [5]. The decoders we introduce have been made publicly available as open-source software packages, and have been

used by many research groups in academia and industry to carry out experimental and theoretical quantum error correction research. UCL Business has filed a patent application for our schedule-induced gauge fixing (SIGF) technique [109], which enables a broad family of error correcting codes (subsystem codes) to tolerate higher component error rates.

The quantum error correcting codes and circuits we present in Part II demonstrate that the resource overhead of quantum error correction can be greatly reduced using architectures where quantum operations are not geometrically local on a 2D Euclidean chip. The quantum error correction protocols we construct can be implemented using a qubit connectivity graph that has low degree (each qubit interacts with at most three or four neighbours) and are amenable to fast decoding algorithms. For experimental implementations that can connect small modules of qubits with long-range couplers, such as trapped-ion systems, our quantum error correcting codes could enable more logical qubits to be protected with a given system size.

# Acknowledgements

Firstly, I would like to thank my supervisor, Nikolas Breuckmann, for all his guidance and support throughout my PhD, and for teaching me so much about quantum LDPC codes and fault-tolerance. I am also very grateful to Dan Browne, who also supervised me and provided a great deal of advice and expertise.

To all the current and former members of Dan's group: thank you for all the interesting discussions, coffee/lunch breaks and pub trips. Although I won't list everyone here, I would like to mention Asmae Benhemou, Simon Burton, Kyriakos Georgiades, Sam Griffiths, Lingling Lao, Avinash Mocherla, Andrew Patterson, Arthur Pesah, Hasan Sayginel, Tom Scruby, James Seddon, Shashvat Shukla, George Umbrarescu and Mike Vasmer. I was also fortunate to be part of UCL's Centre for Doctoral Training (CDT) in Quantum Technologies, and am grateful for all the staff who helped run the CDT and taught us in the MRes year. A special thanks to Paul Warburton, Lopa Murgai, Abbie Bray, Anthony Grout and Mariana Iossifova-Kelly for all their help over the years. I'd also like to thank all my fellow PhD students in the CDT, especially Cohort 5, for their camaraderie throughout my time at UCL.

I would like to thank Earl Campbell for supervising my internship project with the AWS Centre for Quantum Computing (CQC). I'm grateful to Earl as well as my other colleagues and collaborators there, including Thom Bohdanowicz, Steve Flammia and Alex Kubica, for everything I learnt during my time at AWS.

I'd like to thank the Google Quantum AI team, especially Cody Jones, Craig Gidney and Mike Newman, for hosting me as a student researcher in Santa Barbara. I'd especially like to thank Craig, whom I learnt a great deal from, for proposing and supervising our sparse blossom project. Additionally, the tools and techniques

introduced in Stim were important for much of the work in this thesis. I'd also like to thank my other colleagues and friends in Santa Barbara who also helped make my time there so memorable. I am also grateful to Cody for the advice and support he provided as my Google PhD Fellowship mentor.

I had a wonderful time in Sydney during and after the Coogee 2023 workshop, and I would like to thank Dominic Williamson, Stephen Bartlett and others from the University of Sydney for inviting me and hosting me. I had many interesting discussions both with group members and fellow visitors during my time there.

I would like to thank my housemates in Brixton and Golders Green, as well as my friends in London and beyond, who made my time outside my studies so enjoyable. I am thankful for my nephew and niece, Max and Orla, who were born around the time I begun my PhD and have been a source of joy throughout my studies. Finally, I would also like to thank my parents, Gordon and Suzanne, my grandmother, Valerie, my brothers, Daniel and Joshua, as well as extended family, for all their support throughout my life.

# Contents

# List of Figures

# List of Tables

# Part I

# Surface code decoders

# Chapter 1

# Introduction

Quantum computers harness the laws of quantum mechanics to solve some problems much faster than is thought possible using conventional computers. Some problems, such as integer factorisation [172] and the simulation of quantum systems [142], admit an exponential speedup over the most efficient known classical algorithms. Solving these problems could help design new batteries, photovoltaics and pharmaceuticals, and will impact public key cryptosystems.

However, the realization of scalable quantum computing depends on our ability to correct errors which arise due to inevitable interactions between the device and the environment. Useful quantum algorithms typically require billions, or even trillions, of quantum operations, yet physical operations in state-of-the-art quantum devices have error rates of around 0.1% to 1%. Reducing error rates in operations from around 0.1% to the required $\approx 10^{-12}$, requires the use of *quantum error correction*. Quantum error correcting codes correct errors by introducing redundancy, encoding $k$ logical qubits into a larger number $n \geq k$ of physical qubits.

The most widely studied quantum error correcting code, both theoretically and experimentally, is the surface code [64], which has a high tolerance for realistic noise and uses four-qubit measurements that are geometrically local in two dimensions. The surface code is therefore especially amenable to implementation in a quantum computer chip (e.g. a solid state device), and the suppression of errors has already been demonstrated for small system sizes using superconducting devices [5].

However, many challenges remain for building a large-scale quantum computer

using surface codes. The chapters in this first part of the thesis focus on the classical control software used to operate surface codes, called a *decoder*. We also introduce optimal local unitary circuits for encoding an unknown state in surface codes in Section 1.3. A decoder determines where errors have occurred using measurement data from the device. It is important that a decoder is fast enough to keep up with the clock speed of the quantum hardware to prevent a backlog of measurement data that grows exponentially in the $T$-gate depth of the computation [186]. For superconducting quantum computers, each round of syndrome extraction has a duration of around one microsecond [145, 133, 5]. The decoder should also be accurate enough to allow logical errors to be corrected with high probability. In Chapter 2 we introduce an algorithm for the minimum-weight perfect matching decoder that can process data as fast as it would be generated by a superconducting quantum computer for reasonable system sizes (distance-17) using a single CPU core. In Chapter 3 we introduce an efficient decoder, belief-matching, which is more accurate than state-of-the-art efficient surface code decoders and therefore allows higher rates to be tolerated, easing hardware requirements. Finally, we conclude Part I in Chapter 4 by introducing schedule-induced gauge-fixing (SIGF), a technique for improving the circuits used to implement a class of quantum error correcting codes called subsystem codes. We show how SIGF can be used to improve the accuracy of decoding for the subsystem surface code, a slight variant of the traditional surface code that has improved (lower degree) qubit connectivity than is required for standard surface code circuits.

## 1.1 Quantum error correction

In this section we will introduce some definitions and background that will be used throughout the thesis. We will describe the state of a quantum computer as a system of $n$ qubits, where each qubit is a two-level quantum system in a state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ in two-dimensional Hilbert space, where $\alpha, \beta \in \mathbb{C}$ and $|\alpha|^2 + |\beta|^2 = 1$. The state space of an $n$-qubit system is the $n$-fold tensor product of single-qubit state spaces.

We define the Hermitian and unitary $2 \times 2$ matrices

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{1.1}$$

where here $X$, $Y$ and $Z$ are together known as the Pauli matrices. These four matrices form a basis for $2 \times 2$ complex matrices. An $n$-qubit Pauli operator $P = \alpha P_n$ is an $n$-fold tensor product $P_n \in \{I, X, Y, Z\}^{\otimes n}$ with the coefficient $\alpha \in \{\pm 1, \pm i\}$. We sometimes omit $\otimes$ signs when describing Pauli operators, e.g. we may denote the operator $X \otimes I \otimes X \otimes Z \otimes Y$ simply by $XIXZY$. We also sometimes omit identity operators and give the indices of non-trivial tensor factors, e.g. the operator $IIIZIYI$ could be denoted $Z_4Y_6$ ($Z$ acting on qubit 4 and $Y$ acting on qubit 6).

The set of all $n$-qubit Pauli operators forms the $n$-qubit Pauli group $\mathcal{P}_n$, and the elements of $\mathcal{P}_n$ form a basis for $2^n \times 2^n$ matrices. We say that the *weight* $\mathrm{wt}(P)$ of a Pauli operator $P \in \mathcal{P}_n$ is the number of qubits on which it acts non-trivially as $X$, $Y$ or $Z$. Note that any two Pauli operators commute if an even number of their tensor factors commute, and anti-commute otherwise.

If the state of $n$ qubits is not fully known, it can be represented using a density operator $\rho = \sum_i p_i |\psi_i\rangle \langle \psi_i|$, which defines the state to be $|\psi_i\rangle \in \mathbb{C}^{2^n}$ with probability $p_i$. Noise acting on a quantum system can be represented using the quantum channel formalism as $\mathcal{E}(\rho) = \sum_k E_k \rho E_k^\dagger$ [154]. Here the $E_k$ are known as Kraus operators which together satisfy the completeness relation $\sum_k E_k^\dagger E_k = I$ if the operation $\mathcal{E}$ is trace-preserving. A common noise model is the single-qubit depolarising channel, which is defined as

$$\mathcal{E}(\rho) = (1-p)\rho + \frac{p}{3}(X\rho X + Y\rho Y + Z\rho Z) \tag{1.2}$$

and which can be interpreted as leaving the state unchanged with probability $1-p$ and applying the operators $X$, $Y$ and $Z$ each with probability $p/3$. The two-qubit

depolarising channel is defined as

$$\mathcal{E}(\rho) = (1-p)\rho + \frac{p}{15}\left(\sum_{E\in\{I,X,Y,Z\}^{\otimes 2}\setminus I\otimes I} E\rho E\right) \tag{1.3}$$

and can be interpreted as leaving the state invariant with probability $1-p$ and otherwise applying one of the 15 non-trivial two-qubit Pauli operators, chosen uniformly at random.

Before we consider concrete examples of quantum error correction, let us first define a quantum error correcting code to be a subspace $\mathcal{C}$ of some larger Hilbert space, and further define a quantum error correcting procedure abstractly by a trace preserving quantum operation $\mathcal{R}$. For some noise channel $\mathcal{E}$ and state $\rho$, our quantum error correction procedure is successful if $(\mathcal{R}\circ\mathcal{E})(\rho) \propto \rho$.

We now briefly review the quantum error correction conditions [16, 129], which give necessary and sufficient conditions for a quantum error correction procedure to be successful. Let $P$ be a projector onto the code $\mathcal{C}$ and further suppose that $\mathcal{E}$ is a noisy quantum channel represented using Kraus operators $\{E_i\}$. A necessary and sufficient condition for the existence of a recovery operation $\mathcal{R}$ that successfully corrects the error channel $\mathcal{E}$ on $\mathcal{C}$ is that

$$PE_i^\dagger E_j P = \alpha_{ij} P \tag{1.4}$$

holds for all $i$ and $j$ and where $\alpha_{ij}$ are elements of a Hermitian matrix $\alpha$ (see Refs. [129, 154] for a proof). If $\mathcal{F}$ is a noisy quantum channel with Kraus operators $\{F_j\}$, where each Kraus operator $F_j$ is a linear combination of the operators $\{E_i\}$, i.e. $F_j = \sum_i m_{ji} E_i$ where $m_{ji} \in \mathbb{C}$, then it can be shown that $\mathcal{R}$ *also* successfully corrects the noisy channel $\mathcal{F}$ acting on $\mathcal{C}$ (see Theorem 10.2 in Ref. [154]). Since $n$-qubit Pauli operators form a basis for linear operators acting on $n$ qubits, this shows that if an error correction procedure is capable of correcting all $n$-qubit Pauli operators acting on $n$ qubits, it is also capable of correcting *any* arbitrary quantum channel acting on the same $n$ qubits. Throughout the rest of this thesis we will restrict

our attention to errors that are Pauli operators (*Pauli errors*), however the quantum error correction conditions show that this is sufficient for a quantum error correcting code to correct more general errors.

### 1.1.1 Stabiliser codes

A broad class of quantum error correcting codes are stabiliser codes [98]. Stabiliser codes are defined in terms of a stabiliser group $\mathcal{S}$, which is an abelian subgroup of $\mathcal{P}_n$ that does not contain the element $-I$. Elements of a stabiliser group are called *stabilisers*. Since every stabiliser group is abelian and Pauli operators have the eigenvalues $\pm 1$, there is a joint $+1$-eigenspace of every stabiliser group, which defines the stabiliser code,

$$\mathcal{C} = \{|\psi\rangle \ : \ s|\psi\rangle = |\psi\rangle \quad \forall s \in \mathcal{S}\}. \tag{1.5}$$

We can define a stabiliser code using a set of generators of its stabiliser group $\mathcal{S} = \langle g_1, g_2, \ldots, g_r \rangle$. A stabilizer code is called a Calderbank-Shor-Steane (CSS) code if its stabilizer group admits a set of generators $g_1, g_2, \ldots, g_r$ such that each generator is either $X$-type or $Z$-type, $g_i \in \{I,X\}^{\otimes n} \cup \{I,Z\}^{\otimes n}$ [47, 176]. If a stabiliser group has $r$ *independent* generators, then the corresponding stabiliser code on $n$ physical qubits encodes $k = n - r$ logical qubits.

The generators of $\mathcal{S}$ all measure $+1$ if the state is uncorrupted, however any generator $g_i$ that anticommutes with an error $E$ will measure -1 (since $g_i E|\psi\rangle = -E g_i|\psi\rangle = -E|\psi\rangle$). If we measure each stabiliser generator after an error $E \in \mathcal{P}_n$ occurs we obtain a list of the measured eigenvalues of the generators of $\mathcal{S}$, known as the *syndrome* $\sigma(E)$. The centraliser $C(\mathcal{S})$ of $\mathcal{S}$ in $\mathcal{P}_n$ is the set of Pauli operators which commute with every stabiliser. If an error $E \in C(\mathcal{S})$ occurs, it will be undetectable. If $E \in \mathcal{S}$, then it acts trivially on the codespace, and no correction is required. However if $E \in C(\mathcal{S}) \setminus \mathcal{S}$, then an undetectable logical error has occurred. The distance $d$ of a stabiliser code is the smallest weight of any nontrivial logical operator,

$$d = \min_{E \in C(\mathcal{S}) \setminus \mathcal{S}} \mathrm{wt}(E). \tag{1.6}$$

We sometimes use the notation $[[n,k,d]]$ to denote the parameters of a code, where $n$ is the number of physical qubits, $k$ is the number of encoded logical qubits and $d$ is its distance.

Given the syndrome and a known noise model, a *decoder* makes a prediction $C \in \mathcal{P}_n$ of which error occurred. The decoder should choose a correction $C$ with a syndrome consistent with the error, $\sigma(C) = \sigma(E)$, such that applying the correction leaves a trivial (all $+1$) syndrome $\sigma(EC)$. If $EC \in \mathcal{S}$ then the decoder has succeeded in correcting the error, whereas if $EC \in C(\mathcal{S}) \setminus \mathcal{S}$ then a logical error has occurred. We will review the decoding problem in more detail in Section 1.4.

Ignoring the phase $\alpha \in \{\pm 1, \pm i\}$, we can use a map $r : \mathcal{P}_n \to \mathbb{F}_2^{2n}$ to represent an $n$-qubit Pauli operator with a binary vector in $\mathbb{F}_2^{2n}$:

$$\alpha \bigotimes_{i=1}^{n} X^{x_i} Z^{z_i} \to (x_1, \ldots, x_n | z_1, \ldots, z_n). \tag{1.7}$$

Using this correspondence, we can conveniently represent the generators of a stabiliser group $\mathcal{S} = \langle g_1, \ldots, g_r \rangle$ using a binary *check matrix* $H$ with $r$ rows and $2n$ columns, where the $i$th row of $H$ is $r(g_i)$. The check matrix is of the form

$$H = (H_X \mid H_Z) \tag{1.8}$$

where the $i$th row and $j$th column of $H_X$ is 1 if generator $i$ of $\mathcal{S}$ acts non-trivially with $X$ or $Y$ on qubit $j$ and is 0 otherwise. Similarly, the $i$th row and $j$th column of $H_Z$ is 1 if generator $i$ acts non-trivially with $Z$ or $Y$ on qubit $j$ and is 0 otherwise. Since by definition $-I \notin \mathcal{S}$ each generator has a phase $\alpha \in \{\pm 1\}$ which we could represent by an additional sign bit (adding an extra column to $H$). Indeed this is additional column for the sign bit is used for determining measurement outcomes in stabiliser circuit simulations [98]. However, the phase does not impact the error correcting properties of a stabiliser code since it has no effect on whether a generator $g_i$ commutes or anti-commutes with a given error $E$, and so it can usually be safely ignored.

Throughout this thesis, arithmetic involving binary vectors is always assumed

to be done modulo 2 (i.e. in $\mathbb{F}_2^{2n}$). Adding rows of $H$ is equivalent to multiplying the corresponding Pauli operators, i.e. $r(g_i)+r(g_j)=r(g_ig_j)$. Two generators $g_i$ and $g_j$ anti-commute if and only if $r(g_i)\Lambda r(g_j)^T=1$, where $\Lambda$ is the $2n\times 2n$ block matrix

$$\Lambda := \begin{pmatrix} 0 & I \\ I & 0 \end{pmatrix} \tag{1.9}$$

and where each block has dimensions $n\times n$. The requirement that $\mathcal{S}$ is abelian corresponds to the constraint $H\Lambda H^T=0$, i.e. $H_XH_Z^T+H_ZH_X^T=0$. Furthermore, $g_i$ and $g_j$ are independent if and only if $r(g_i)$ and $r(g_j)$ are linearly independent. Hence we can find the number of independent generators of $\mathcal{S}$ from the rank of its check matrix $H$, i.e. $k=n-\mathrm{rk}(H)$.

As an example, consider the five-qubit code, which has stabiliser generators $\mathcal{S}=\langle XZZXI, IXZZX, XIXZZ, ZXIXZ\rangle$ and can be represented by the check matrix

$$H = \left(\begin{array}{ccccc|ccccc} 1&0&0&1&0&0&1&1&0&0\\ 0&1&0&0&1&0&0&1&1&0\\ 1&0&1&0&0&0&0&0&1&1\\ 0&1&0&1&0&1&0&0&0&1 \end{array}\right). \tag{1.10}$$

Since the check matrix has rank $r=\mathrm{rk}(H)=4$ we see that the five qubit code encodes $k=n-r=1$ logical qubit. It can also be verified that the code has distance 3, i.e. it is a $[[5,1,3]]$ code.

For any stabiliser code with $n$ physical qubits encoding $k$ logical qubits, we can always find a basis of logical operators of the form $\bar{X}_1,\bar{Z}_1,\bar{X}_2,\bar{Z}_2,\ldots,\bar{X}_k,\bar{Z}_k\in\mathcal{P}_n$. Here $\bar{X}_i$ and $\bar{Z}_i$ act as logical $X$ and $Z$ operators on logical qubit $i$, respectively. These logical operators satisfy the expected commutativity conditions, namely $\bar{X}_i$ anti-commutes with $\bar{Z}_i$ but commutes with all other $\bar{Z}_j$ (for $i\neq j$). The logical operators are in $C(\mathcal{S})\setminus\mathcal{S}$. For example, a valid choice of logical operators for the five-qubit code is $\bar{X}_1=XXXXX$ and $\bar{Z}_1=ZZZZZ$. One way to see why a basis of this form can be constructed for *any* stabiliser code is by using row operations and qubit permutations to put the check matrix into standard form (see Section 10.5.7 of

Ref. [154]), which can be written as the following block matrix:

$$H = \left(\begin{array}{ccc|ccc} I & A & B & C & 0 & D \\ 0 & 0 & 0 & E & I & F \end{array}\right). \tag{1.11}$$

where here both the left and right sides of the check matrix have their columns partitioned into three blocks of widths $s$, $n-k-s$ and $k$ columns from left to right, respectively. The top sub-matrices (including $A$, $B$, $C$ and $D$) have $s$ rows, and the bottom sub-matrices (including $E$ and $F$) have $n-k-s$ rows. Using this standard form, we can construct a check matrix $G_X$ representing the $\bar{X}_i$ operators and check matrix $G_Z$ representing the $\bar{Z}_i$ operators as:

$$G_X = \left(\begin{array}{ccc|ccc} 0 & F^T & I & D^T & 0 & 0 \end{array}\right) \tag{1.12}$$

$$G_Z = \left(\begin{array}{ccc|ccc} 0 & 0 & 0 & B^T & 0 & I \end{array}\right). \tag{1.13}$$

We see that these logical operators have the required commutativity from $G_X \Lambda G_Z^T = I$ and it is also straightforward to verify that they commute with the stabiliser generators, i.e. $G_X \Lambda H^T = 0$ and $G_Z \Lambda H^T = 0$.

The check matrix for any CSS code can be written in the form

$$H = \begin{pmatrix} H_X & 0 \\ 0 & H_Z \end{pmatrix} \tag{1.14}$$

where $H_X$ and $H_Z$ are now redefined to be $n \times n$ matrices defining the $X$-type stabiliser generators and $Z$-type stabiliser generators, respectively. The commutativity condition is now given by the constraint $H_X H_Z^T = 0$. A CSS code encodes $k = n - \mathrm{rk}(H_X) - \mathrm{rk}(H_Z)$ logical qubits.

### 1.1.2 Stabiliser circuits

The stabiliser formalism can be used to analyse and efficiently simulate an important class of quantum circuits called stabiliser circuits. Stabiliser circuits consist of preparation and measurement of qubits in the $Z$ basis, the Hadamard gate $H$, the

phase gate $S$ and the controlled-NOT gate $CNOT$. The $H$, $S$ and $CNOT$ gates together generate the *Clifford* group of unitary operators and have matrix representations

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}. \tag{1.15}$$

Elements of the Clifford group map Pauli operators to Pauli operators, i.e. a unitary $U$ is a Clifford operator if $UPU^\dagger \in \mathcal{P}_n \; \forall P \in \mathcal{P}_n$. The $CNOT$ gate, acting by conjugation, maps Pauli $X$ and $Z$ operators as follows:

$$XI \to XX, \quad IX \to IX, \quad ZI \to ZI, \quad IZ \to ZZ, \tag{1.16}$$

the $H$ gate maps $X \to Z$ and $Z \to X$ under conjugation and the $S$ gate maps $X \to Y$ and $Z \to Z$ under conjugation.

Applying a unitary $U$ to an eigenstate $|\psi\rangle$ of an operator $S$ (with eigenvalue $s$) gives $US|\psi\rangle = sU|\psi\rangle = USU^\dagger U|\psi\rangle$: an eigenstate of $S$ becomes an eigenstate of $USU^\dagger$. Therefore, if we start in a state on $n$ qubits stabilised by $n$ independent stabiliser generators (a stabiliser state such as the $|0\rangle^{\otimes n}$ state) and apply a Clifford circuit, we can compute the stabiliser group of the final stabiliser state by tracking how each gate maps the stabiliser generators under conjugation.

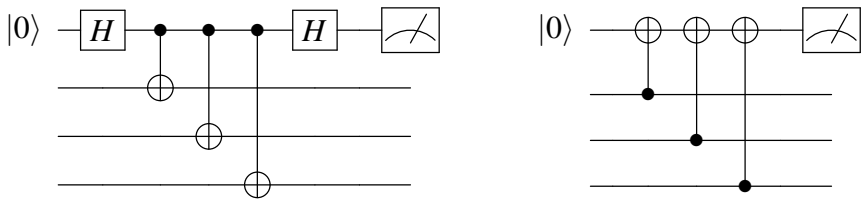
Furthermore, measuring a Pauli operator on a stabiliser state with stabiliser group $\mathcal{S}$ can also be done efficiently. If we measure a Pauli operator $P$ that commutes with all elements of $\mathcal{S}$ then either $P \in \mathcal{S}$ or $-P \in \mathcal{S}$; the measurement outcome is deterministic and is $+1$ if $P \in \mathcal{S}$ and is $-1$ if $-P \in \mathcal{S}$. The state is unchanged after measurement. If instead we measure a Pauli operator $P$ that anti-commutes with any element of $\mathcal{S}$ then the measurement outcome is uniformly random and the post-measurement state is stabilised by $\langle \mathcal{S}_0, \pm P \rangle$, where $\mathcal{S}_0$ is the subgroup of $\mathcal{S}$ that commutes with $P$ and the sign of $P$ is determined by the measurement outcome. Using these principles, it is possible to simulate any stabiliser circuit efficiently

(in polynomial time) and this result is known as the Gottesman-Knill theorem, see Refs. [98, 154] for more details. See Ref. [1] for a more efficient algorithm for simulating stabiliser circuits that can be used to simulate each $CNOT$, $H$ or $S$ gate in $O(n)$ time and each measurement in $O(n^2)$ time, where $n$ is the number of qubits. A further improvement can be obtained using the simulation method in Stim [91], which instead takes $O(n)$ time for deterministic measurements. A standard choice of *universal* gate set is the "Clifford + $T$" gate set, which consists of $CNOT$, $H$ and $S$ gates, along with the non-Clifford

$$T = \sqrt{S} = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$$

gate, as well as state preparation and measurement in the computational ($Z$) basis.

Stabiliser circuits are important in quantum error correction; indeed, *all* the quantum error correction circuits we consider in this thesis are stabiliser circuits, and the Gottesman-Knill theorem is therefore useful both for understanding these circuits and for simulating them efficiently. A common example of a stabiliser circuit used in quantum error correction (QEC) is a circuit for measuring a Pauli operator. For example, the following two circuits use an ancilla qubit to measure a $XXX$ operator (left) and an $ZZZ$ operator (right):




Using knowledge of the way $CNOT$ and $H$ gates map Paulis under conjugations, we see that the measurement of $ZIII$ on the four qubits using the ancilla at the end of each circuit is equivalent to measuring $ZXXX$ at the start of the left circuit or $ZZZZ$ at the start of the right circuit. Since the ancilla is initially stabilised by $Z$, these are equivalent to measuring $XXX$ or $ZZZ$ on the bottom three qubits, respectively. Circuits similar to these can be used to measure check operators (such

as stabiliser generators) in QEC circuits. As we will discuss later in this chapter, it is important that errors that can occur in the measurement circuits themselves can also be corrected. This can achieved by repeating the measurement circuit, however we note that measuring larger Pauli operators can require deeper measurement circuits, which can lead to more error locations and therefore a lower tolerance to errors.

### 1.1.3 Subsystem codes

Subsystem codes are a slight generalisation of stabiliser codes, which we will also consider in this thesis. Subsystem codes are stabiliser codes where only a subset of the available encoded degrees of freedom are used [163]. They can simplify quantum error correction circuits by reducing the weight of the Pauli operators measured when producing the syndrome [6]. Subsystem codes also allow for a procedure called *gauge fixing* which is useful to manipulate the encoded quantum information. Gauge fixing effectively allows us to change the code mid-computation, and in Ref. [155] the authors exploit this to switch between codes which have complementary sets of logical operations. These advantages have motivated experimentalists to pursue subsystem codes for implementing fault-tolerant quantum computation. This includes IBM, who proposed implementing the heavy-hexagon subsystem code [48] to reduce frequency collisions in their superconducting quantum processors [124]. The Bacon-Shor subsystem code has been implemented experimentally in a trapped-ion architecture [75], where the fidelity of the encoded logical operations exceeded that of the entangling physical operations used to implement them.

A subsystem code is a stabiliser code in which a subset of logical operators are chosen not to store information [163]. In a subsystem code, the overall Hilbert space $\mathcal{H}$ can be decomposed as

$$\mathcal{H} = (\mathcal{H}_{\mathcal{L}} \otimes \mathcal{H}_{\mathcal{G}}) \oplus \mathcal{C}^{\perp} \tag{1.17}$$

where only $\mathcal{H}_{\mathcal{L}}$ stores information and any operations applied only on $\mathcal{H}_{\mathcal{G}}$ are ignored. The Pauli operators that act trivially on $\mathcal{H}_{\mathcal{L}}$ form the *gauge group* $\mathcal{G}$ of the code. The stabiliser group $\mathcal{S}$ is the center of $\mathcal{G}$ up to phase factors, $\langle iI, \mathcal{S} \rangle = Z(\mathcal{G}) \coloneqq C(\mathcal{G}) \cap \mathcal{G}$.

Hence, up to phase factors, operators from $\mathcal{G}$ are either stabilisers (acting trivially on $\mathcal{H}_\mathcal{L} \otimes \mathcal{H}_\mathcal{G}$), or act non-trivially on $\mathcal{H}_\mathcal{G}$ only. Logical operators that act non-trivially only on $\mathcal{H}_\mathcal{L}$ are called *bare* logical operators $\mathcal{L}_{bare}$, and are given by $C(\mathcal{G}) \setminus \mathcal{G}$. The *dressed* logical operators $\mathcal{L}_{dressed} = C(\mathcal{S}) \setminus \mathcal{G}$ act non-trivially on both $\mathcal{H}_\mathcal{L}$ and $\mathcal{H}_\mathcal{G}$. A dressed logical operator is a bare logical operator multiplied by a gauge operator in $\mathcal{G} \setminus \mathcal{S}$. The distance $d$ of a subsystem code is the weight of the minimum-weight dressed logical operator, $d = \min_{P \in C(\mathcal{S}) \setminus \mathcal{G}} |P|$. The number of physical qubits $n$, logical qubits $k$, independent stabiliser generators $r$ and gauge qubits $g$ are related as

$$n - k = r + g. \tag{1.18}$$

One advantage of introducing gauge qubits is that they can enable simpler stabiliser measurements if the generators of the gauge group $\mathcal{G}$ (the *gauge generators*) have a lower weight than the generators of the stabiliser group $\mathcal{S}$. Since $\mathcal{S} \subseteq \mathcal{G}$ the outcomes of the gauge generator measurements can be used to infer the eigenvalues of the stabilisers, provided the gauge generators are measured in the appropriate order (since $\mathcal{G}$ is generally not abelian) [181, 186]. We will sometimes refer to standard stabiliser codes, where all logical qubits are used to store quantum information, as *subspace* codes, to distinguish them from subsystem codes.

The technique called *gauge fixing*, applied to a subsystem code, consists of adding an element $g \in \mathcal{G}$ into the stabiliser group $\mathcal{S}$, as well as removing every element $h \in \mathcal{G}$ that anticommutes with $g$. Gauge fixing was introduced by Paetznick and Reichardt and can be useful for performing logical operations [155, 28, 209], including for code deformation and lattice surgery [200]. Gauge fixing can also be used for constructing codes: both the surface code and the heavy-hexagon code [48] are gauge fixings of the Bacon-Shor subsystem code [9], all belonging to the larger family of 2D compass codes [138]. These constructions are static, as the fixed gauge stays the same over time. In Chapter 4, we will show how a *dynamical approach* to gauge fixing can be used to improve the quantum error correcting performance of subsystem codes.

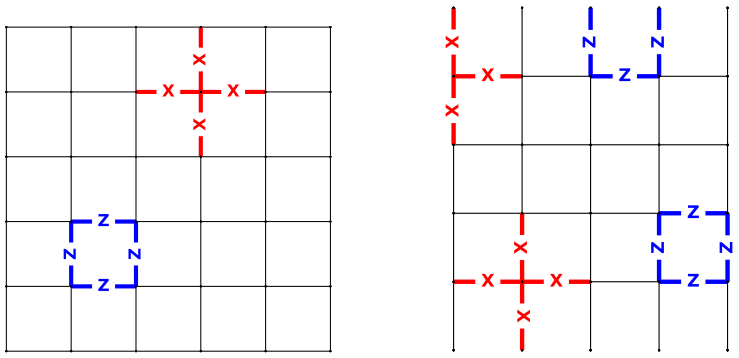



**Figure 1.1:** The check operators for (a) the toric code and (b) the planar code. Opposite edges in (a) are identified and each edge corresponds to a qubit.

## 1.2 Surface codes

### 1.2.1 The toric and planar surface code

The surface code is a CSS code introduced by Kitaev [127, 64], which has stabiliser generators defined on a square lattice embedded in a two-dimensional surface. We associate a qubit with each edge of the lattice. Each *site* or *vertex* check operator is a Pauli operator in $\{I,X\}^n$ which only acts non-trivially on the edges adjacent to a vertex of the lattice. Each *plaquette* check operator is a Pauli operator in $\{I,Z\}^n$ which only acts non-trivially on the edges adjacent to a face of the lattice. In the toric code, the square lattice is embedded in a torus, whereas in the planar code the lattice is embedded in a plane, without periodic boundary conditions (see Figure 1.1). These site and plaquette operators together generate the stabiliser group of the code. While the toric code encodes two logical qubits and has parameters $[[2d^2,2,d]]$, the surface code encodes a single logical qubit and has parameters $[[d^2+(d-1)^2,1,d]]$.

### 1.2.2 The rotated surface code

The *rotated* surface code is a slight variant of the original (or "unrotated") surface code which we just reviewed in Section 1.2.1, and is essentially obtained by cutting a diamond-shaped section out of the unrotated planar code [24]. The rotated surface code is more efficient than the planar code introduced in the previous section, since the diamond-shaped cut does not reduce the distance but reduces the number of

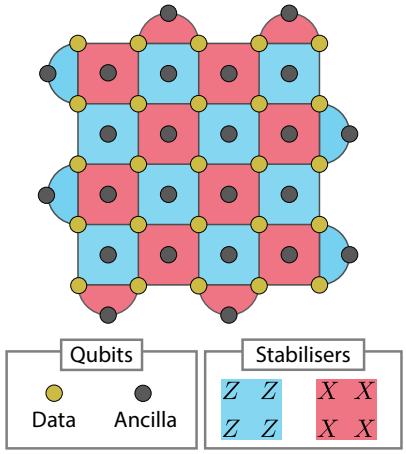



**Figure 1.2:** The stabiliser generators, data qubits and ancilla qubits of the rotated surface code. Data qubits are placed on the vertices of a square lattice. Stabiliser generators are weight-four operators defined on faces in the bulk of the lattice, and boundary stabilisers are weight-two.

physical qubits by almost a factor of 2. The rotated surface code is usually depicted by placing data qubits on the *vertices* of a $d \times d$ square lattice coloured in a red and blue chequerboard pattern, see Figure 1.2. The $X$ stabiliser generators are defined on red faces and the $Z$ stabiliser generators on blue faces. The rotated surface code uses around $2\times$ fewer qubits to achieve the same distance as the unrotated surface code and has parameters $[[d^2, 1, d]]$. In addition to the $d^2$ data qubits, an additional $d^2 - 1$ ancilla qubits are used to measure the stabiliser generators. A logical $\bar{Z}$ operator can be defined as a $Z^{\otimes d}$ operator acting non-trivially on any row of data qubits. Similarly, a logical $\bar{X}$ operator can be defined as an $X^{\otimes d}$ operator acting non-trivially on any column of data qubits.

The most common circuit used for measuring the stabilisers of the rotated surface code is shown in Figure 1.3. As we will explain in Section 1.4, this cycle of stabiliser measurements is repeated in order to handle errors that can occur in the measurement circuit. For a noise model in which two-qubit depolarising noise occurs after each CNOT with probability $p$, qubits suffer single-qubit depolarising noise with strength $p$ when idling, and qubits are initialised or measured in orthogonal states with probability $p$, the circuit of Figure 1.3 as a threshold of around $\approx 0.7\%$. Below a noise strength of $\approx 0.7\%$, increasing the code size suppresses the logical

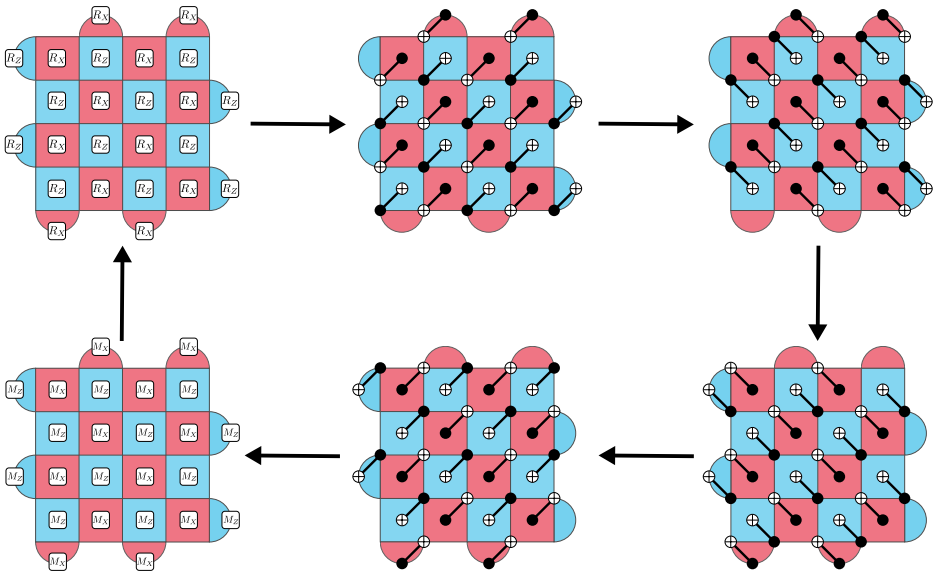


**Figure 1.3:** The main cycle of a syndrome extraction circuit for the rotated surface code. The gates labelled $R_X$ and $R_Z$ are state preparation in the $X$ and $Z$ basis, respectively. The gates labelled $M_X$ and $M_Z$ are measurement in the $X$ and $Z$ basis, respectively.

error rate exponentially in $d$.

### 1.2.3 The subsystem surface code

The subsystem surface code, introduced in Ref. [36], is a subsystem code with gauge operators and stabiliser generators that are geometrically local on a 2D surface. It is closely related to the surface code, and a constant-depth unitary mapping the surface code to the subsystem surface code is also given in Ref. [36]. The threshold of the subsystem surface code for circuit-level noise is around 0.6%, which is similar to (but slightly below) that of the surface code. When the subsystem surface code is defined on a surface with periodic boundary conditions we refer to it as the subsystem toric code.

The stabilisers and gauge operators of the subsystem toric code are shown in Figure 1.4. We place a qubit on the middle of each edge and on each vertex of a square tiling. Each gauge operator is represented by a triangle, with a qubit associated with each of its vertices. Borrowing terminology from Ref. [36], each gauge generator is referred to as a *triangle operator*, and consists of a Pauli operator

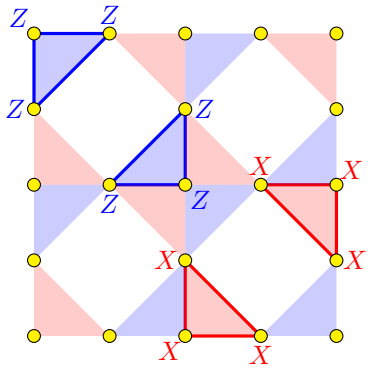



**Figure 1.4:** The subsystem toric code of Ref. [36]. Data qubits (yellow filled circles) are placed in the middle of each edge and on each vertex of a square lattice of the toric code. Opposite sides are identified. The gauge group is generated by three-qubit *triangle operators*. The two $Z$ triangle operators in the top left face are outlined with a blue border, and their product forms a 6-qubit $Z$ stabiliser. Similarly, in the bottom right face, two $X$ triangle operators are outlined with a red border, and their product is a 6-qubit $X$ stabiliser.

acting nontrivially on its three qubits. There are four types of triangle operator in each face of the square lattice: two $Z$-type triangle operators defined in the north-west and south-east corners, and two $X$-type operators defined in the north-east and south-west corners. These four types of triangle operators are highlighted in Figure 1.4 for the $d = 2$ subsystem toric code. Within each face, the product of each pair of $Z$-type triangle operators forms a 6-qubit $Z$ stabiliser, and the product of each pair of $X$-type triangle operators forms a 6-qubit $X$-type stabiliser. The subsystem toric code has $3d^2$ data qubits (there are $d^2$ vertices and $2d^2$ edges of the square lattice) and $2(d^2-1)$ independent stabiliser generators, forming a stabiliser code with $d^2+2$ logical qubits, $d^2$ of which are gauge qubits, with the remaining two logical qubits encoding quantum information. It can be verified that all triangle operators commute with the stabilisers and are therefore logical operators for the stabiliser code (since they are not stabilisers). The logical $\bar{Z}$ and $\bar{X}$ operators for each gauge qubit are chosen to be the north-west and north-east triangle operators of each face respectively. The remaining two pairs of logical operators are the same as for the toric code, each acting non-trivially only on data qubits lying on a (horizontal or vertical) homologically nontrivial loop of the torus. In [36] it was shown that the subsystem toric code has parameters $[[3d^2, 2, d]]$.

In [36] a planar subsystem surface code was also introduced (with two qubit stabilisers on the boundary), which has code parameters $[[3d^2-2d,1,d]]$, and in Ref. [46] a planar rotated subsystem code was introduced (with three-qubit stabilisers on the boundary) which (for odd $d$) has parameters $[[\frac{3}{2}d^2-d+\frac{1}{2},1,d]]$.

Syndrome extraction can be done by measuring only the three-qubit triangle operators, by placing an ancilla qubit in the middle of each triangle operator. As a result, the ancilla qubits only require degree-three connectivity (they interact with three neighbouring qubits), whereas all the qubits in the standard surface code circuit require degree-four connectivity. Lower qubit connectivity can help reduce crosstalk that arises from frequency collisions in some architectures [48].

## 1.3 Preparing a state in the codespace

Several methods can be used to prepare a state in the codespace of a stabiliser code. One method is to use a unitary encoding circuit that maps a product state $|\phi_0\rangle\otimes\ldots\otimes|\phi_{k-1}\rangle\otimes|0\rangle^{\otimes(n-k)}$ of $k$ physical qubits in unknown states (along with ancillas) to the state of $k$ logical qubits encoded in a stabiliser code with $n$ physical qubits. Labelling the ancillas in the initial state $k,k+1,\ldots,n-1$, the initial product state is a $+1$-eigenstate of the stabiliser generators $Z_k,Z_{k+1},\ldots,Z_{n-1}$. A unitary encoding circuit maps the stabilisers $Z_k,Z_{k+1},\ldots,Z_{n-1}$ of the product state to a generating set for the stabiliser group $\mathcal{S}$ of the code. The circuit also maps the logical operators $Z_0,Z_1,\ldots,Z_{k-1}$ and $X_0,X_1,\ldots,X_{k-1}$ of the physical qubits to the corresponding logical operators $\bar{Z}_0,\bar{Z}_1,\ldots,\bar{Z}_{k-1}$ and $\bar{X}_0,\bar{X}_1,\ldots,\bar{X}_{k-1}$ of the encoded qubits (up to stabilisers). For any stabiliser code, it is efficient to construct a unitary encoding circuit that uses at most $O(n(n-k))$ *CNOT*, $H$ or $S$ gates, see Refs. [52, 98].

For specific families of stabiliser codes, more efficient circuits can be found. For the toric code, Aguado and Vidal [3] introduced a Renormalisation Group (RG) unitary encoding circuit for preparing an unknown state that has $O(\log d)$ circuit depth to encode a distance $d$ code. The RG encoder uses gates that are not geometrically local on the torus, and has an inductive step that doubles the code

distance using a constant number of layers of *CNOT* gates. Dennis *et al.* [64] gave an encoding circuit for encoding a distance $d$ planar surface code that uses $O(n) = O(d^2)$ gates, where every gate is geometrically local on the surface. In terms of circuit depth (time steps), the inductive step in their method requires $\Omega(d)$ time steps and encodes a distance $d+1$ planar code from a distance $d$ code by turning smooth edges into rough edges and vice versa. As a result encoding a distance $d$ planar code from an unencoded qubit requires $\Omega(d^2)$ time steps. This is quadratically slower than the $\Omega(d)$ circuit depth lower bound given by Bravyi *et al.* [31]. In Section 1.3.1, we present local unitary encoding circuits for the surface code and toric code that match Bravyi *et al.*'s lower bound and are therefore asymptotically optimal. The original work presented in this section was previously published in Ref. [116].

Encoding circuits can be useful for preparing states in the codespace without the need for ancilla-assisted stabiliser measurements and feedback. Without the need for ancillas, a larger distance code can be prepared using the same number of physical qubits. This can be useful for preparing states in stabiliser based fermion-to-qubit mappings [171], an important component of quantum simulation algorithms, since some mappings introduce stabilisers in order to enforce locality in the transformed fermionic operators [37, 197, 179, 105], or to mitigate errors [123]. For example, the compact mapping of Ref. [65] is closely related to the surface code, so an encoding circuit for the surface code can be used to prepare a state in the compact mapping [116]. In this context of simulation algorithms, the gates in the circuit can usually be assumed to already be reliable or error corrected.

### 1.3.1 Optimal local unitary encoding circuits for the surface code

Our local unitary encoding circuit for the planar code that requires only $2d$ time steps to encode a distance $d$ planar code. The inductive step in our method, shown in Figure 1.5 for $d = 4$, encodes a distance $d+2$ planar code from a distance $d$ planar code using 4 time steps, and does not rotate the code. This inductive step can then be used recursively to encode an unencoded qubit into a distance $d$ planar code using $2d$ time steps. As a base case we can use a distance 3 or 4 planar code, encoding circuits

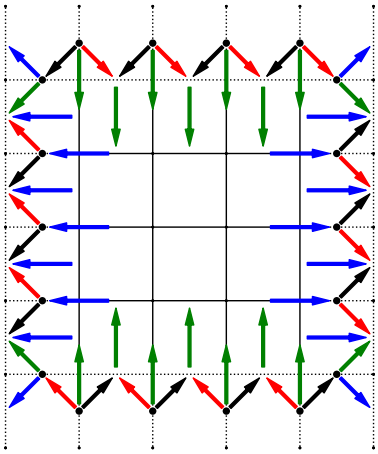


**Figure 1.5:** Circuit to encode a distance 6 planar code from a distance 4 planar code. Each edge corresponds to a qubit. Each arrow denotes a CNOT gate, pointing from control to target. Filled black circles (centred on edges) denote Hadamard gates, which are applied at the beginning of the circuit. The colour of each CNOT gate (arrow) denotes the time step in which it is applied. The first, second, third and fourth time steps correspond to the blue, green, red and black CNOT gates respectively. Solid edges correspond to qubits originally encoded in the $d = 4$ planar code, whereas dotted edges correspond to additional qubits that are encoded in the $d = 6$ planar code.

for which are given in Appendix A.1. Our encoding circuit therefore matches the $\Omega(d)$ lower bound provided by Bravyi *et al.* [31].

We can verify its correctness by checking that stabiliser generators and logicals of the distance $d$ surface code are mapped to stabiliser generators and logicals of the distance $d+2$ surface code. We show how each type of site and plaquette stabiliser generator is mapped by the inductive step of the encoding circuit in Figure 1.6. Note that the site stabiliser generator labelled c (red) is mapped to a weight 7 stabiliser in the $d = 6$ planar code: this is still a valid generator of stabiliser group, and the standard weight four generator can be obtained by multiplication with a site of type b. Similarly, the plaquette stabiliser generator labelled c becomes weight 7, but a weight four generator is recovered from multiplication by a plaquette of type a. Therefore, the stabiliser group of the $d = 4$ planar code is mapped correctly to that of the $d = 6$ planar code, even though minimum-weight generators are not mapped explicitly to minimum-weight generators. It is straightforward to verify that the $X$ and $Z$ logical operators of the $d = 4$ planar code are also mapped to the $X$ and $Z$

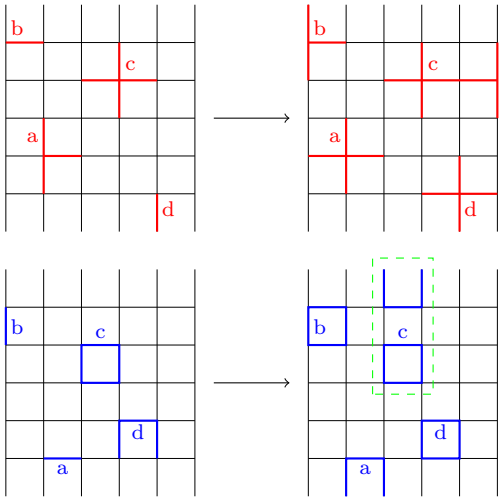



**Figure 1.6:** The transformation of the stabiliser generators of the $d = 4$ planar surface code when the circuit in Figure 1.5 is applied. Top: the four main types of site stabilisers acted on nontrivially by the encoding circuit (labelled a-d) are shown in red before (left) and after (right) the encoding circuit is applied. On the left we assume that the ancillas have already been initialised in the $|+\rangle$ state ($H$ applied). Bottom: the four main types of plaquette stabilisers (also labelled a-d) are shown in blue before (left) and after (right) the encoding circuit is applied. Plaquette c has two connected components after the circuit is applied (right), and is enclosed by a green dashed line for clarity.

logicals of the $L = 6$ planar code by the inductive step.

We can also encode rectangular planar codes and rotated surface codes using similar methods, and encoding circuits for these are given in Appendix A.1 and Appendix A.2.

### 1.3.2 Encoding a toric code from a planar code

While the method in section 1.3.1 is only suitable for encoding planar codes, we will now show how we can encode a distance $d$ toric code from a distance $d$ planar code using only local unitary operations. Starting with a distance $d$ planar code, $2(d-1)$ ancillas each in a $|0\rangle$ state, and an additional unencoded logical qubit, the circuit in Figure 1.7 encodes a distance $d$ toric code using $d+2$ time steps. The correctness of this step can be verified by seeing how the stabiliser generators are mapped: each ancilla initialised as $|0\rangle$ (stabilised by $Z$) is mapped to a plaquette present in the toric code but not the planar code. Likewise, each ancilla initialised in $|+\rangle$ using an $H$ gate (stabilised by $X$) is mapped to a site generator in the toric code but not the planar

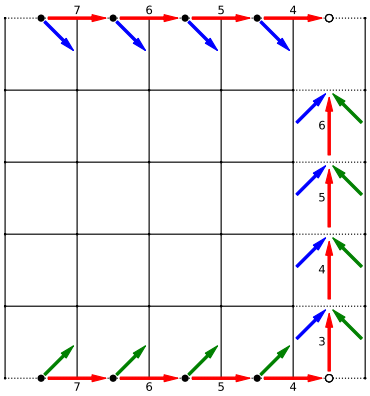


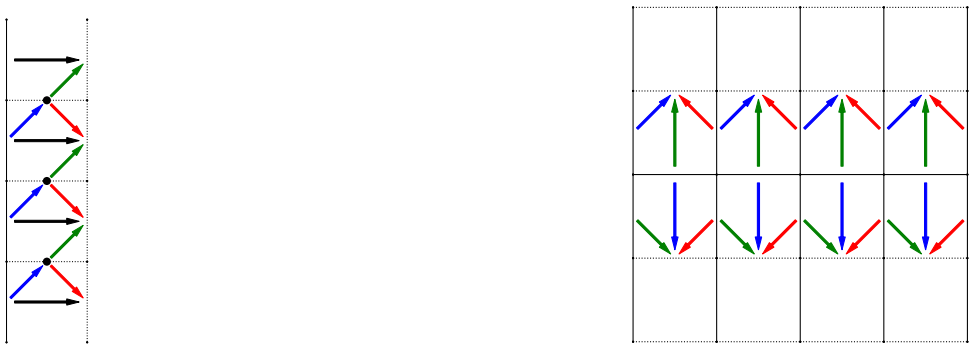

**Figure 1.7:** Circuit to encode a distance 5 toric code from a distance 5 planar code. Solid edges correspond to qubits in the original planar code and dotted edges correspond to qubits added for the toric code. Opposite edges are identified. Arrows denote CNOT gates, and filled black circles denote Hadamard gates applied at the beginning of the circuit. Blue and green CNOT gates correspond to those applied in the first and second time step respectively. Red CNOTs are applied in the time step that they are numbered with. The hollow circles denote the unencoded qubit that is to be encoded into the toric code.

code. The weight-three site and plaquette stabilisers on the boundary of the planar code are also mapped to weight four stabilisers in the toric code. Finally, we see that $X$ and $Z$ operators for the unencoded qubit (the hollow circle in Figure 1.7) are mapped to the second pair of $X$ and $Z$ logicals in the toric code by the circuit, leaving the other pair of $X$ and $Z$ logicals already present from the planar code unaffected.

Therefore, encoding two unencoded qubits in a toric code can be achieved using $3d+2$ time steps using the circuits given in this section and in Section 1.3.1.

### 1.3.3 Encoding a 3D Surface Code

Similar techniques can also be used to encode a distance $d$ 3D surface code using $O(d)$ time steps. We first encode a distance $d$ planar code using the method given in section 1.3.1. This planar code now forms a single layer in the $xy$-plane of a 3D surface code (where the $y$-axis is defined to be aligned with a $Z$-logical in the original planar code). Using the circuit given in Figure 1.8(a), we encode each column of qubits corresponding to a $Z$ logical in the planar code into a layer of the 3D surface code in the $yz$-plane (which has the same stabiliser structure as a planar code if the rest of the $x$-axis is excluded). Since each layer in the $yz$-plane can be encoded in parallel, this stage can also be done in $O(d)$ time steps. If we encode each layer in



(a) (b)

**Figure 1.8:** (a) Circuit to encode a $4 \times 2$ planar code from a four qubit repetition code (where adjacent qubits in the repetition code are stabilised by $XX$). Applied to a column of qubits corresponding to a surface code $\bar{Z}$, this encodes a layer in the $yz$-plane of a 3D surface code. (b) Circuit to encode the $xz$-plane of a 3D surface code once the $yz$-plane layers and a layer in the $xy$-plane have been encoded. Arrows denote CNOT gates pointing from control to target, and blue, green, red and black CNOT gates correspond to the first, second, third and fourth time steps respectively. Solid and dotted edges correspond to qubits that are initially entangled and in a product state respectively.

the $yz$-plane such that the original planar code intersects the middle of each layer in the $yz$-plane, then each layer in the $xz$-plane now has the stabiliser structure shown in Figure 1.8(b). Using the circuit in Figure 1.8(b) repeatedly, all layers in the $xz$-plane can be encoded in parallel in $O(d)$ time steps. Therefore, a single unknown qubit can be encoded into a distance $d$ 3D surface code in $O(d)$ time steps.

### 1.3.4 Fault-tolerant initialisation

While unitary encoding circuits can be convenient in some contexts, they are in general not fault-tolerant: an error occurring during the circuit can propagate to a high-weight error at the end of the circuit. In QEC, a state in the codespace of a stabiliser code is typically prepared by measuring the stabiliser generators of the code instead. For example, to prepare a logical $|\bar{0}\rangle^{\otimes k}$ state of a stabiliser code, we first prepare the product state $|0\rangle^{\otimes n}$ and then measure the stabiliser generators. If we choose the basis for $\bar{Z}$ logicals given in Equation (1.13), then our $\bar{Z}$ logical operators are all $Z$-type Pauli operators. Hence, our initial product state is already in a $+1$-eigenstate of the $\bar{Z}$ logicals. Each of our stabiliser generator measurements will either

be deterministically $+1$ if the generator commutes with $Z^{\otimes n}$, or uniformly random otherwise. However, if a stabiliser generator $g_i$ measures $-1$ we can simply redefine $g_i \coloneqq -g_i$ in software, since the phase of the generator does not impact the error correcting performance of the code. Note that we remain in the $+1$-eigenstate of the $\bar{Z}$ logicals after measuring the stabiliser generators, since the stabiliser generators must commute with the $\bar{Z}$ logicals.

In some cases, we may wish to encode an unknown state in a code fault-tolerantly, for performing fault-tolerant logic using state injection [32]. For the surface code, this can be done using the method of Ref. [139]. More efficient protocols for the surface code have since been found, including the method of Ref. [89], which can prepare logical states in the $XY$ or $YZ$ plane of the Bloch sphere (rather than general unknown states as done in Ref. [139]).

## 1.4 Decoding

Decoders are algorithms running on a classical computer that use measurement results from a quantum error correction circuit, as well as a model of the noise, to determine where errors occurred. In Part I of this thesis, we focus on decoding algorithms for surface codes, as well as variants including subsystem surface codes.

Two important characteristics of a decoder are its accuracy and its speed. The accuracy of a decoder quantifies how good it is at correctly determining where errors occurred. A more accurate decoder can increase the value of the threshold for a quantum error correcting (QEC) code, as well as reducing the number of physical qubits required to achieve a desired logical error rate below threshold. Improving the accuracy of decoders can therefore lead to less demanding hardware requirements. The value of the threshold depends both on the QEC code used and the decoder, with more accurate decoders leading to a higher threshold, and therefore less demanding hardware requirements.

However, the speed of a decoder is also important. A surface code superconducting quantum computer with a million physical qubits will generate measurement data at a rate of around 1 terabit per second. A decoder must process this data at least

as fast as it is generated, to prevent a backlog of data that grows exponentially in the $T$-gate depth of the computation [186]. In order to achieve these speeds at scale, a decoding algorithm should ideally have an expected running time that scales linearly or almost-linearly with the size of the problem. Furthermore, constant factors are also important, and details of the decoder implementation, possibly in hardware (e.g. an FPGA [141] or ASIC [13]), will also be important to achieve the required speeds.

Fast decoders are important not only for operating a quantum computer, but also as a software tool for researching quantum error correction protocols. Estimating the resource requirements of a quantum error correction protocol can require the use of direct Monte Carlo sampling to estimate the probability of extremely rare events. These analyses can be prohibitively expensive without access to a fast decoder, capable of processing millions of shots from circuits containing $\approx 10^5$ gates in a reasonable time-frame.

### 1.4.1 Decoding with perfect syndrome measurements

In Section 1.1.1 we briefly introduced the decoding problem for the context where stabiliser measurements are assumed to be perfect. We defined the syndrome $\sigma(E)$ of some error $E \in \mathcal{P}_n$ as a list of measurement outcomes of the stabiliser generators $g_i$ of the stabiliser group $\mathcal{S} = \langle g_1, \ldots, g_r \rangle$. Recall that, with perfect syndrome measurements, a generator $g_i$ measures $+1$ if it commutes with $E$, $g_i E = E g_i$ and measures $-1$ otherwise. A decoder in this context chooses some correction $C \in \mathcal{P}_n$ and succeeds if $EC \in \mathcal{S}$.

Let us now use the check matrix representation of the Pauli operators to define the decoding problem. We represent an error $E \in \mathcal{P}_n$ as a binary vector $\mathbf{e} := \Lambda r(E)^T \in \mathbb{F}_2^{2n}$. Note that the $X$ and $Z$ components of the binary representation of $\mathbf{e}$ are swapped relative to the check matrix (the first half of $\mathbf{e}$ represents the $Z$ component, the second half represents the $X$ component). The syndrome $\mathbf{s} \in \mathbb{F}_2^r$ is then represented as the binary vector

$$\mathbf{s} := H\mathbf{e} \tag{1.19}$$

where $H$ is the check matrix of Equation (1.11). Here $\mathbf{s}[i]$ is 0 if the error $E$ commutes with stabiliser generator $g_i$ and is 1 if $E$ anti-commutes with $g_i$. The decoder takes as input the syndrome $\mathbf{s}$ as well as a model of the noise, which assigns some probability $\mathbb{P}(\mathbf{e}) \in [0,1]$ to each possible error $\mathbf{e} \in \mathbb{F}_2^{2n}$, and outputs some correction $\mathbf{c} \in \mathbb{F}_2^{2n}$ satisfying $H\mathbf{c} = \mathbf{s}$. We define a *logicals matrix* $L$, using $G_X$ and $G_Z$ from Equation (1.12) and Equation (1.13), as

$$L := \begin{pmatrix} G_X \\ G_Z \end{pmatrix}. \tag{1.20}$$

Applying the correction will remove the syndrome, $H(\mathbf{c}+\mathbf{e}) = 0$, and will be successful if $L(\mathbf{c}+\mathbf{e}) = 0$ (i.e. if the error and correction are in the stabiliser group), whereas we have a logical failure if $L\mathbf{c} \neq L\mathbf{e}$.

Before introducing algorithms that solve this decoding problem, we will first generalise the decoding problem to the case where stabiliser measurement circuits themselves are noisy. As we will see, we can do so by generalising our definitions of the check matrix, logicals matrix and binary error vector, and by defining detectors and logical observables. This definition of the decoding problem for circuit-level noise using detectors and logical observables follows the approach taken by Gidney in Stim [91]. Furthermore, the detector check matrix and logical observables matrix we will define are matrix representations of the detector error model data structure in Stim.

### 1.4.2 Detectors and logical observables

A *detector* can be understood as a generalisation of a stabiliser measurement that is useful for defining and constructing the decoding problem for quantum error correction circuits, in which the quantum gates themselves can be noisy. A detector is defined to be a parity of measurement outcome bits in a quantum error correction circuit that is deterministic in the absence of errors. The outcome of a detector measurement is 1 if the observed parity differs from the expected parity for a noiseless circuit, and is 0 otherwise. We say that a Pauli error $P$ *flips* detector $D$ if including $P$ in the circuit changes the outcome of $D$, and a *detection event* is a

detector with outcome 1.

We define a *logical observable* to be a linear combination of measurement bits, whose outcome instead corresponds to the measurement of a logical Pauli operator. A logical observable measurement is in general not deterministic in the absence of noise. In all cases we consider however, such as in a memory experiment, logical observables *are* deterministic, since the logical qubit is prepared in a known logical basis state so that a quantum error correction protocol can be benchmarked. However, unlike a detector measurement, a logical observable measurement outcome is not given to a decoder, since it corresponds to a measurement of a logical qubit that should be protected by the quantum error correction procedure, and in general the outcome is apriori unknown.

### 1.4.3 Detector check matrix and logical observable matrix

With detectors and observables now defined, we can represent this more general noise model and decoding problem using two binary parity check matrices: a *detector check matrix* $H \in \mathbb{F}_2^{n_d \times m}$ and a *logical observable matrix* $L \in \mathbb{F}_2^{n_l \times m}$, where here $n_d$ is the number of detectors and $n_l$ is the number of logical observables. Each column of these matrices corresponds to an error mechanism, and therefore $m$ denotes the number of these error mechanisms. We set $H[i, j] = 1$ if detector $i$ is flipped by error mechanism $j$, and $H[i, j] = 0$ otherwise. Likewise, we set $L[i, j] = 1$ if logical observable $i$ is flipped by error mechanism $j$, and $L[i, j] = 0$ otherwise. By describing the error model this way, each error mechanism is defined by which detectors and observables it flips, rather than by its Pauli type and location in the circuit. We can then represent an error (a set of error mechanisms) by a vector $\mathbf{e} \in \mathbb{F}_2^m$. The syndrome of $\mathbf{e}$ is the outcome of detector measurements, given by $\mathbf{s} = H\mathbf{e}$. We define the set of *detection events* to be the subset of detectors corresponding to the non-zero elements of $\mathbf{s}$. An undetectable logical error is an error in $B \coloneqq \{\mathbf{e} \in \mathbb{F}_2^m \mid \mathbf{e} \in \ker H, \mathbf{e} \notin \ker L\}$.

We now discuss the possible error mechanisms (columns of $H$ and $L$ in more detail). A model of the noise assigns some probability $\mathbb{P}(\mathbf{e}) \in [0, 1]$ to each error $\mathbf{e} \in \mathbb{F}_2^m$. In this thesis we only consider independent noise models, where each error mechanism occurs (flips bit $\mathbf{e}[i]$) independently with probability $\mathbf{p}[i]$, flippping some

set of detectors and observables. Here $\mathbf{p} \in \mathbb{R}^m$ is a vector of priors, defining the independent noise model. Each column in $H$ or $L$ will always correspond to an independent error mechanism. For an independent noise model, the prior probability $\mathbb{P}(\mathbf{e})$ of an error $\mathbf{e}$ is

$$\mathbb{P}(\mathbf{e}) = \prod_i (1-\mathbf{p}[i])^{(1-\mathbf{e}[i])} \mathbf{p}[i]^{\mathbf{e}[i]} = \prod_i (1-\mathbf{p}[i]) \prod_i \left( \frac{\mathbf{p}[i]}{1-\mathbf{p}[i]} \right)^{\mathbf{e}[i]}. \tag{1.21}$$

The distance of the circuit for an independent noise model is $d = \min_{\mathbf{e} \in B} |\mathbf{e}|$, where $|\mathbf{e}|$ is the Hamming weight of $\mathbf{e}$.

An independent error model can be used to represent circuit-level depolarising noise exactly, and is a good approximation of many commonly considered error models, including general stochastic Pauli noise models [49, 90]. From a stabilizer circuit and Pauli noise model, we can construct $H$ and $L$ efficiently by propagating Pauli errors through the circuit to see which detectors and observables they flip. Each prior $\mathbf{p}[i]$ is then computed by summing over the probabilities of all the Pauli errors that flip the same set of detectors or observables (or more precisely, these equivalent error mechanisms are independent, and we compute the probability that an odd number occurred). This is essentially what the error analysis tools do in the Stim software package, where a "detector error model" (automatically constructed from a Stim circuit) captures the information contained in $H$, $L$ and $\mathbf{p}$ [92].

In Section 1.4.6, we show how the detectors, observables and the matrices $H$ and $L$ are defined for a small example of a distance 2 repetition code circuit.

Note that, from the perspective of the decoder, these matrices $H$ and $L$ are treated the same as the matrices with the same names defined in Section 1.4.1, and for any stabiliser code we can construct a circuit with independent $X$ and $Z$ errors such that the detector check matrices and logical observable matrices coincide with those of Section 1.4.1.

### 1.4.4 Tanner graphs

A Tanner graph is a bipartite graph that can be used to represent any check matrix $H$. The check matrix $H$ is the biadjacency matrix of the corresponding Tanner graph

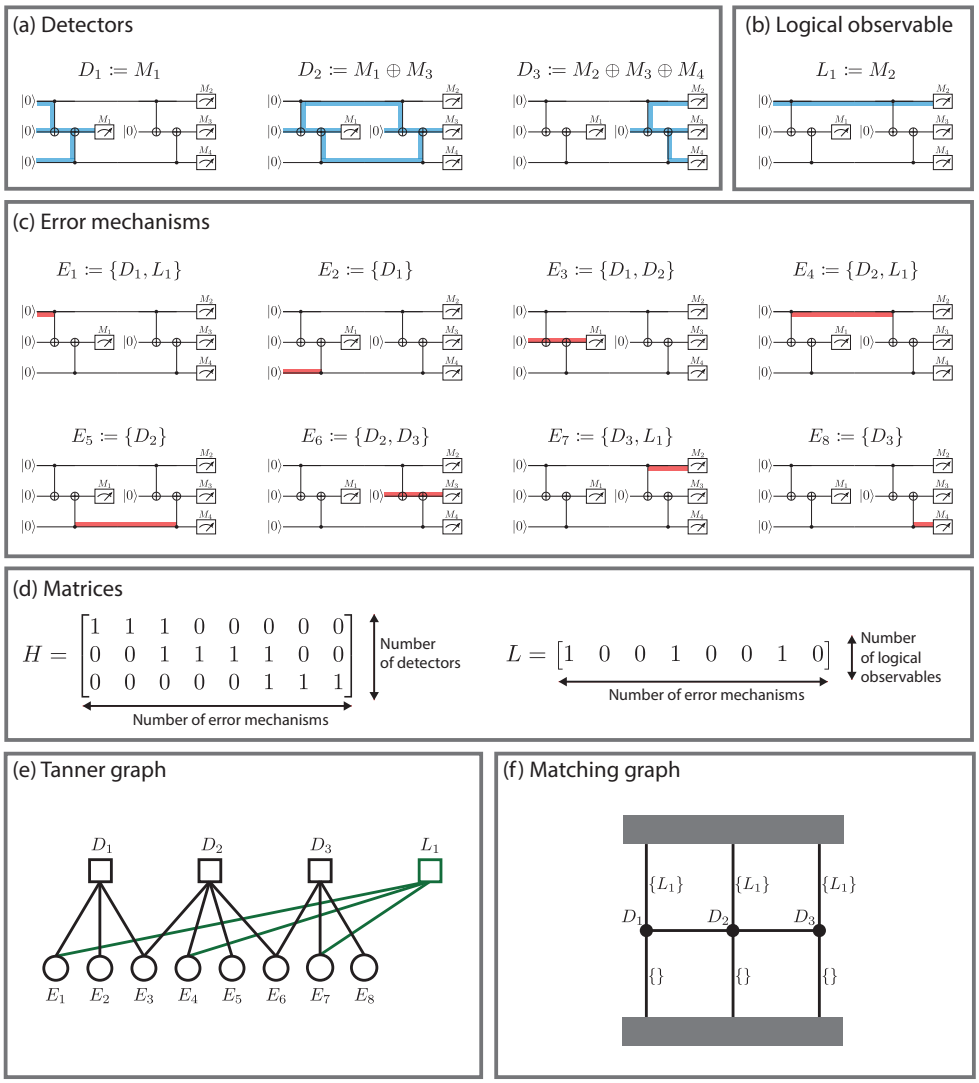



**Figure 1.9:** Representations of (a) detectors, (b) logical observables and (c) error mechanisms for the circuit corresponding to a [[2,1,2]] repetition code memory experiment with two rounds of syndrome extraction. (d) The detector check matrix $H$ and logical observables matrix $L$. (e) The Tanner graph representation of $H$ (and $L$ in green). (f) The matching graph representation of $H$ (which can be used when $H$ has column weight at most 2). Each edge is annotated with the set of logical observables it flips (with $\{\}$ denoting the empty set). The grey regions here represent the boundary. Note that here there are two parallel half-edges adjacent to each node; this is a symptom of the fact that the code has distance 2, and therefore has distinct error mechanisms that flip the same set of detectors but different sets of logical observables.

$\mathcal{T}(H)$.

We define a Tanner graph describing a stabiliser circuit and stochastic Pauli noise model $\mathcal{T}(H) = (V, C, E)$, where $V$ is a set of *variable nodes*, $C$ is a set of *check nodes* and $E$ is the edge set. Recall that since $\mathcal{T}$ is bipartite, for each edge $(v_j, c_i) \in E$ we have $v_j \in V$ and $c_i \in C$. Each check node $c_i \in C$ corresponds to a detector (a row of $H$) and each variable node $v_j \in V$ corresponds to an independent error mechanism (Pauli error) that can occur in the circuit. There is an edge $(v_j, c_i) \in E$ if and only if the error mechanism corresponding to $v_j \in V$ flips the detector corresponding to $c_i \in C$ (i.e. if and only if $H[i, j] = 1$). The Tanner graph representation of the check matrix is used to define the belief propagation decoder, a message-passing algorithm used in Chapter 3, in which messages are sent along the edges of the Tanner graph.

### 1.4.5 Matching graphs

When the check matrix $H$ has column-weight at most two we can represent it graphically using a different type of graph called a *matching graph*, which can be used to construct efficient decoding algorithms, including for the minimum-weight decoding problem we will review later in this chapter, and which is the subject of Chapter 2. Matching graphs are also called matching graphs or decoding graphs in the literature. We sometimes refer to this type of error model, where each error mechanism flips at most two detectors, as a *graphlike error model*. Graphlike error models can be used to approximate common noise models for many important classes of quantum error correction codes including surface codes [64], for which $X$-type and $Z$-type Pauli errors are both graphlike, as we will explain.

In a matching graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, each node $v \in \mathcal{V}$ corresponds to a detector (a detector node, a row of $H$). Each edge $e \in \mathcal{E}$ is a set of detector nodes of cardinality one or two representing an error mechanism that flips this set of detectors (a column of $H$). We can decompose the edge set as $\mathcal{E} = \mathcal{E}_1 \cup \mathcal{E}_2$ where $\forall e \in \mathcal{E}_1 : |e| = 1$ and $\forall e \in \mathcal{E}_2 : |e| = 2$. A regular edge $(u, v) \in \mathcal{E}_2$ flips a pair of detectors $u, v \in \mathcal{V}$, whereas a *half-edge* $(u,) \in \mathcal{E}_1$ flips a single detector $u \in \mathcal{V}$. For a half-edge $(u,) \in \mathcal{E}_1$ we sometimes say that $u$ is connected to the boundary and use the notation $(u, v_b)$, where $v_b$ is a virtual boundary node (which does not correspond to any detector).

Therefore, when we refer to an edge $(u,v) \in \mathcal{E}$ it is assumed that $u$ is a node and $v$ is either a node or the boundary node $v_b$. Each edge $e_i \in \mathcal{E}$ is assigned a weight $w(e_i) = \log((1-\mathbf{p}[i])/\mathbf{p}[i])$, and recall that $\mathbf{p}[i]$ is the probability that error mechanism $i$ occurs. We also define an edge weights vector $\mathbf{w} \in \mathbb{R}^{|\mathcal{E}|}$ for which $\mathbf{w}[i] = w(e_i)$. We also label each edge $e_i = (u,v) \in \mathcal{E}$ with the set of logical observables that are flipped by the error mechanism, which we denote either by $l(e_i)$ or $l(u,v)$. We use $x \oplus y$ to denote the symmetric difference of sets $x$ and $y$. For example, $l(e_1) \oplus l(e_2)$ is the set of logical observables flipped when the error mechanisms 1 and 2 are both flipped. We define the distance $D(u,v)$ between two nodes $u$ and $v$ in the matching graph to be the length of the shortest path between them. We give an example of a matching graph $\mathcal{G}$ for a repetition code circuit in Section 1.4.6.

### 1.4.6 Example of detectors and observables for a repetition code

In Figure 1.9 we show the detectors, observables, matrices $H$ and $L$, the Tanner graph and the matching graph $\mathcal{G}$ for the circuit corresponding to a memory experiment using a [[2,1,2]] bit-flip repetition code with two rounds of syndrome extraction. We use a bit-flip code (with stabilizer group $\langle ZZ \rangle$) and implement transversal initialisation and measurement in the $Z$ basis. The circuit has three detectors (Figure 1.9(a)). The blue highlighted regions are the corresponding $Z$-type detecting regions [147] (an error within this region which anti-commutes with its type will cause the corresponding detector to flip). Figure 1.9(b) shows the logical $Z$ observable. Here, the blue highlighted region is the $Z$-type sensitivity region corresponding to the logical $Z$ observable - errors that anti-commute with $Z$ in this region will flip the outcome of the corresponding logical measurement $L_1$. Given a stochastic Pauli noise model in the circuit, we can characterise errors based on the set of detectors and logical observables they flip (Figure 1.9(c)). For a standard circuit-level depolarising noise model, there are eight different classes of error mechanism in this circuit, when classified this way. For each error mechanism, we highlight in red a region of the circuit where a single-qubit $X$ error would flip the same detectors and observables. Note that these single-qubit $X$ errors are just *examples* of Pauli errors contributing

to the error mechanisms; for example, another Pauli error contributing to $E_1$ would be a two-qubit $YX$ error after the first CNOT. Figure 1.9 also shows representations of this noise model as a detector check matrix $H$, observable matrix $L$, as a Tanner graph and as a matching graph.

Note that for a surface code memory experiment, the circuit and matching graph are instead three-dimensional. In this case, the sensitivity region corresponding to a logical $Z$ observable measurement forms a 2D sheet in spacetime. We denote this logical observable measurement $L_Z$. This observable $L_Z$ is included in the set $l(e)$ of an edge $e \in \mathcal{E}$ in the matching graph if the error mechanism associated with $e$ flips $L_Z$. For this to happen, the edge $e$ must pierce the 2D sheet (sensitivity region) and have $Z$-type detectors (detectors that are parities of $Z$ measurements) at its endpoints.

### 1.4.7 Maximum likelihood decoding

Given the detector check matrix $H$, the logical observables matrix $L$, the syndrome $\mathbf{s} = H\mathbf{e}$ of some unknown error $\mathbf{e} \in \mathbb{F}_2^m$ and a probability distribution modelling the noise $\mathbb{P} : \mathbb{F}_2^m \to \mathbb{R}$, a *maximum likelihood* (ML) decoder returns a correction $\mathbf{c} \in \mathbb{F}_2^m$ that is most likely to successfully correct the error $\mathbf{e}$, such that $L\mathbf{c} = L\mathbf{e}$.

A maximum likelihood decoder therefore predicts the most probable *logical* error (which logical measurements were most likely to have been flipped), which corresponds to the following maximisation problem:

$$\max_{\mathbf{r} \in \mathrm{im}(L)} \left[ \sum_{\mathbf{x} \in \mathbb{F}_2^m \,:\, L\mathbf{x} = \mathbf{r}, H\mathbf{x} = \mathbf{s}} \mathbb{P}(\mathbf{x}) \right]. \tag{1.22}$$

This predicted logical correction $\mathbf{r}$ is by definition the most likely to correct the actual logical error $L\mathbf{e}$. It is usually sufficient to correct the logical measurement with $\mathbf{r}$ directly, but if a physical correction is required, an ML decoder can return any $\mathbf{c} \in \mathbb{F}_2^m$ satisfying $L\mathbf{c} = \mathbf{r}$ and $H\mathbf{c} = \mathbf{s}$. In general, the ML decoding problem is not efficient to solve. Intuitively this is not surprising since the sum in Equation (1.22) contains a number of terms exponential in the number of error mechanisms $m$, and furthermore we are maximising over a number of possible logical errors $\mathbf{r} \in \mathrm{im}(L)$ that is exponential in the number of logical observables $n_l$. Indeed, the general

quantum ML decoding problem for stabiliser codes was shown to be #P-hard in Ref. [122]. Since the problem of decoding stabiliser codes is a special case of decoding quantum circuits with general stochastic Pauli noise, the problem as we have formulated it in this section is also #P-hard. However, for some specific codes and noise models, ML decoding can be implemented efficiently [34], and tensor network methods, although still computationally expensive, can be used to achieve a good approximation of ML decoding for some noise models and codes using modest system sizes [34, 114, 5].

### 1.4.8 Minimum-weight decoding

A *minimum-weight* decoder instead predicts the most probable *physical* error, rather than the most probable logical error. This corresponds to finding a correction $\mathbf{c} \in \mathbb{F}_2^m$ that maximises

$$\max_{\mathbf{c}\in\mathbb{F}_2^m \,:\, H\mathbf{c}=\mathbf{s}} \mathbb{P}(\mathbf{c}). \tag{1.23}$$

A minimum-weight decoder does not use the logical observables matrix $L$.

If we have an independent noise model defined by a vector $\mathbf{p}$, as given in Equation (1.21), then minimum-weight decoding can be equivalently defined as maximising $\log(\mathbb{P}(\mathbf{c})) = C - \sum_i \mathbf{w}[i]\mathbf{c}[i]$, where here $C := \sum_i \log(1-\mathbf{p}[i])$ is a constant and $\mathbf{w}[i] := \log((1-\mathbf{p}[i])/\mathbf{p}[i])$ is a log-likelihood ratio (or "weight") associated with error mechanism $i$. In other words the minimum-weight decoder solves the minimisation problem:

$$\min_{\mathbf{c}\in\mathbb{F}_2^m \,:\, H\mathbf{c}=\mathbf{s}} \left[ \sum_i \mathbf{w}[i]\mathbf{c}[i] \right]. \tag{1.24}$$

If we further have a *uniform* prior, $\mathbf{p}[i] = p \;\forall i \in [1,m]$, and if $p$ satisfies $p < 0.5$, then minimum weight decoding finds an error $\mathbf{c}$ consistent with the syndrome $H\mathbf{c} = \mathbf{s}$ that minimises the Hamming weight $|\mathbf{c}|$. The minimum-weight decoding problem for stabiliser codes was shown to be NP-hard in Refs. [118, 134], via a reduction to the analogous problem for classical linear codes, for which the associated decision problem was shown to be NP-complete in Ref. [18]. As we will see in Section 1.4.10, however, the minimum-weight decoding problem can be solved in polynomial time in some cases, specifically when the column weight of $H$ is at most 2.

### 1.4.9 Connection to the decoding problem for binary linear codes

The circuit decoding problem can be seen as a slight generalisation of the syndrome decoding problem for classical binary linear codes [143]. A binary linear code is a $k$-dimensional subspace $\mathbb{F}_2^k$ of an $n$-dimensional binary vector space $\mathbb{F}_2^n$ defined as the kernel of a binary parity check matrix $H$ with $n$ columns and rank $n-k$. Given some error $\mathbf{e}$ applied to a binary vector encoded in the linear code, the syndrome decoding problem corresponds to using a syndrome vector $\mathbf{s} = H\mathbf{e}$ to infer a correction $\mathbf{c}$ satisfying $H\mathbf{c} = \mathbf{s}$. Here the decoder succeeds only if $\mathbf{e} = \mathbf{c}$. For any linear code we can always construct an equivalent circuit-level decoding problem by using the check matrix of the linear code to define $Z$ stabilisers, which are measured perfectly after independent $X$ errors are applied to the data qubits.

The quantum circuit decoding problem is slightly more general since we only seek to protect certain parities of measurement bits (logical observables) defined by the logicals matrix $L$. Hence the difference lies in the definition of success for the decoding problem: for the quantum circuit decoding problem we succeed if $L\mathbf{e} = L\mathbf{c}$, whereas for decoding a linear code we do not have a concept of a logical observables matrix, and so decoding a linear code is equivalent to the quantum circuit decoding problem where $L$ is the identity matrix.

This connection to linear codes can also be understood from the fact that the set of possible measurement outcome bits of a noiseless Clifford circuit defines a linear code provided (without loss of generality) that the sign of Pauli operators measured are chosen appropriately. See Corollary 2 of Ref. [61] for a proof, where this linear code is referred to as the *outcome code*. The detectors correspond to parity checks of this linear code, constraining allowed measurement outcomes. If the logical observables are deterministic, they also correspond to parity checks of this linear code. It is important to stress that here we are referring to parities of measurement outcomes. In contrast each column of our detector check matrix $H$ (and logicals matrix $L$) used for decoding corresponds to an independent error mechanism that in general flips a *set* of measurement outcomes.

### 1.4.10 The minimum-weight perfect matching decoder

When an independent error model $(H, L, \mathbf{p})$ is graphlike and can be represented by a matching graph, the minimum-weight decoding problem can be solved efficiently using the minimum-weight perfect matching (MWPM) decoder [64, 80]. In this section, we will assume that we have such an independent graphlike error model with detector check matrix $H$, logical observables matrix $L$, priors vector $\mathbf{p}$, as well as the corresponding matching graph $\mathcal{G}$ with edge weights vector $\mathbf{w}$. Given some error $\mathbf{e} \in \mathbb{F}_2^m$ sampled from the graphlike error model, with syndrome $\mathbf{s} = H\mathbf{e}$, the MWPM decoder solves the minimisation problem defined in Equation (1.24).

We now restate the MWPM decoding problem in terms of the matching graph. Given a matching graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with vertex set $\mathcal{V}$, the syndrome $\mathbf{s}$ defines a set of detection events (highlighted nodes) $\mathcal{D} \subseteq \mathcal{V}$. Specifically, a detector node $v_i \in \mathcal{V}$ is included in $\mathcal{D}$ if and only if $\mathbf{s}[i] = 1$. A MWPM decoder finds a set of edges $M_E \subseteq \mathcal{E}$ of minimum weight $\sum_{e \in M_E} w(e)$ such that every node in $\mathcal{D}$ is incident to an *odd* number of edges in $M_E$, and every node in $\mathcal{V} \setminus D$ is incident to an *even* number of edges in $M_E$. For a normal graph that does not contain half-edges, this problem is known as the minimum-weight $T$-join problem in the field of combinatorial optimisation (for $T \equiv \mathcal{D}$) [74, 132]. In this thesis we will refer to this graph theory problem as the minimum-weight embedded matching (MWEM) problem.

#### 1.4.10.1 Example: Code capacity surface code decoding

An example of a MWPM decoding problem is that of decoding depolarising noise in the surface code in the "code capacity" setting, where syndrome measurements are assumed to be perfect. Here, we assume that we are already in the code space and measure each stabiliser generator once, defining an $X$ detector as the (noiseless) measurement of each $X$ stabiliser generator measurement and similarly define a $Z$ detector for each $Z$ stabiliser generator measurement.

We see that each single qubit $X$ error flips two $Z$ detectors (detectors for $Z$ stabiliser generators), and similarly each $Z$ error flips two $X$ detectors. The matching graph therefore has two connected components: an $X$ matching graph and a $Z$ matching graph, see Figure 1.10. For a noise model in which $X$ and $Z$ errors occur

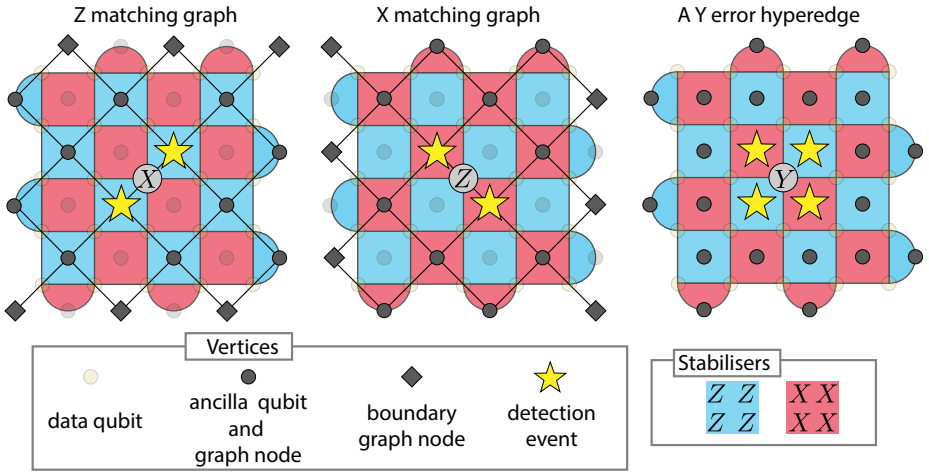



**Figure 1.10:** The MWPM decoding problem for a distance 5 surface code. Left: the $X$ error matching graph, where we associate a node with each stabiliser and an edge $(u, v)$ with each $X$ error, where $u$ and $v$ are the stabilisers that the error anti-commutes with. If an $X$ error anti-commutes with a single stabiliser $u$ we represent it with an edge $(u, v_b)$ between $u$ and a *boundary* node (each a square node in the diagram). Middle: the $Z$ error matching graph is defined similarly, but with an edge for each $Z$ error. Right: A $Y$ error anti-commutes with four stabilizers, so would need to be represented by a hyperedge, and therefore not included in the matching graphs. The $Y$ error induces correlations between the $X$ and $Z$ matching graphs which are ignored by a MWPM decoder.

independently, the matching graph accurately models the noise. For depolarising noise, however, $Y$ errors can also occur with probability $O(p)$ and each single qubit $Y$ error flips *four* detectors in the bulk, which would correspond to a *hyperedge* if it were to be included in the matching graph. The detector check matrix for this problem has the following form:

$$H = \begin{pmatrix} H_X & 0 & H_X \\ 0 & H_Z & H_Z \end{pmatrix} \tag{1.25}$$

where the left, middle and right blocks of columns correspond to single qubit $Z$ errors, $X$ errors and $Y$ errors, respectively. The $H_X$ and $H_Z$ matrices which define the $X$ stabilisers and $Z$ stabilisers, respectively, both have column weight at most 2, but the right hand block of columns in $H$ has column weights of up to 4. Since each $Y$ error can be decomposed into an $X$ and a $Z$ error on the same qubit, it is not necessary to handle $Y$ errors separately; instead it is sufficient to decode the $X$ and

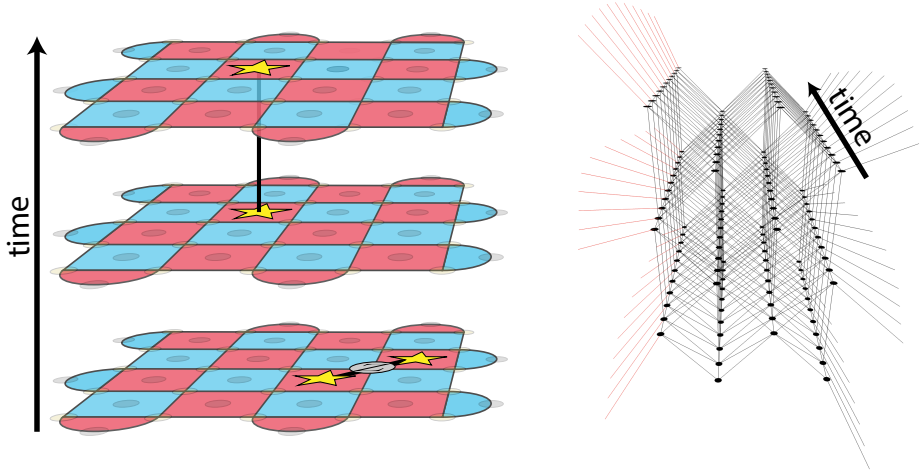



**Figure 1.11:** Left: Three rounds of a surface code memory experiment. The edges and detection events corresponding to a $Z$ error before the first round and an $X$ stabiliser measurement error during the second round are drawn. Right: The full $X$ matching graph for a distance-5, 10-round surface code memory experiment is shown (with the logical qubit prepared and measured in the $X$ basis). The graph (and diagram) is generated using Stim for the circuit of Figure 1.3 with circuit-level depolarising noise. The boundary edges drawn in red are errors that flip the outcome of the logical $X$ observable measurement.

$Z$ matching graphs as separate connected components. Indeed, for any CSS code, provided a decoder can correct at least $t$ $X$ errors and at least $t$ $Z$ errors, it can also correct at least $t$ $Y$ errors.

Although a MWPM decoder can correct up to the full code distance for this problem and has good performance, it is incorrectly modelling $Y$ errors as occurring with probability $O(p^2)$ for depolarising noise, when in fact they occur with probability $O(p)$. This is detrimental to the accuracy of the MWPM decoder, and in Chapter 3 we show how all the information in the Tanner graph can be exploited by combining the MWPM decoder with the belief propagation decoder, leading to even higher accuracy.

#### 1.4.10.2 Example: Surface code memory experiment

A common MWPM decoding problem, which we will use throughout this thesis to benchmark quantum error correcting codes and decoders, is that of decoding a memory experiment. A memory experiment benchmarks how well a quantum error

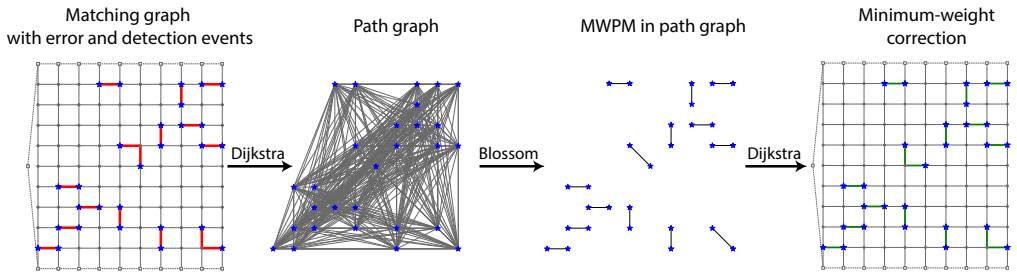



**Figure 1.12:** Solving the MWPM decoding problem via a reduction to the MWPM graph theory problem.

correcting code can preserve a logical observable through time. A memory experiment of a quantum code in the basis $\bar{P} \in \{\bar{X}, \bar{Z}\}$ consists of transversal initialisation of the logical qubit(s) in the $\bar{P}$ basis, repeating $r$ rounds of syndrome measurement and then transversal measurement of the logical qubit(s) in the $\bar{P}$ basis.

We summarise the MWPM decoding problem for an $\bar{X}$ basis surface code memory experiment in Figure 1.11, which uses the circuit in Figure 1.3 for syndrome extraction. The data qubits are initialised in the $X$ basis at the beginning of the circuit and measured in the $X$ basis at the end. In the bulk of the circuit, each detector is the parity of two consecutive measurements of a stabiliser generator. As a result, a single-qubit $Z$ error will only flip the outcome of a pair of $X$ detectors in a single round, since it only causes the $X$ stabiliser measurements to change in a single round. A measurement error now corresponds to a timelike edge: it flips a detector in two consecutive rounds, see Figure 1.11 (left). At the beginning and end of the circuit, the detectors are defined slightly differently. The $X$ detectors in the first round each correspond to a single $X$ stabiliser measurement, since initialisation in the $X$ basis ensures we are already in the $+1$ eigenstate of the $X$ stabilisers. The final $X$ detectors, for each $X$ stabiliser generator $g_i$, are the parity of the $X$ measurements of the data qubits in the support of $g_i$, as well as the final $X$ stabiliser generator ancilla measurement. The parity of the $X$ data qubit measurements in the support of $g_i$ are essentially a noiseless measurement of $g_i$ which can be compared against.

### 1.4.11 Polynomial-time algorithm for solving the MWPM decoding problem

The reason for the MWPM decoder's name is that the MWEM problem can be solved via a reduction to the MWPM problem [74]. We now define the MWPM problem for a graph $G = (V, E)$. Here $G$ is a weighted graph, where each edge $(u, v) \in E$ is a pair of nodes $u, v \in V$ and, unlike matching graphs, there are no half-edges. Each edge is assigned a weight $w(e) \in \mathbb{R}$. A perfect matching $M \subseteq E$ is a subset of edges such that each node $u \in V$ is incident to exactly one edge $(u, v) \in M$. For each $(u, v) \in M$ we say that $u$ is matched to $v$, and vice versa. A MWPM is a perfect matching that has minimum weight $\sum_{e \in M} w(e)$. Clearly, not every graph has a perfect matching (a simple necessary condition is that $|V|$ must be even; a necessary *and sufficient* condition is provided by Tutte's theorem [195]), and a graph may have more than one perfect matching of minimum weight.

By using a polynomial-time algorithm for solving the MWPM problem (e.g. the blossom algorithm [73]), we can obtain a polynomial-time algorithm for solving the MWEM problem via a reduction. We will now describe this reduction for the case that the matching graph $\mathcal{G}$ has no boundary, in which case the MWEM problem is equivalent to the minimum-weight $T$-join problem (see [74, 132]). This reduction was used by Edmonds and Johnson for their polynomial-time algorithm for solving the Chinese postman problem [74]. The boundary can also be handled with a small modification (e.g. see [80]).

Given a matching graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with non-negative edge weights (and which, for now, we assume has no boundary, i.e. $\mathcal{E} = \mathcal{E}_2$), and given a set of detection events $\mathcal{D} \subseteq \mathcal{V}$, we define the *path graph* $\bar{\mathcal{G}}[\mathcal{D}] = (\mathcal{D}, \bar{\mathcal{E}})$ to be the complete graph on the vertices $\mathcal{D}$ for which each edge $(u, v) \in \bar{\mathcal{E}}$ is assigned a weight equal to the distance $D(u, v)$ between $u$ and $v$ in $\mathcal{G}$. Here the distance $D(u, v)$ is the length of the shortest path between $u$ and $v$ in $\mathcal{G}$. In other words, the path graph $\bar{\mathcal{G}}[\mathcal{D}]$ is the subgraph of the metric closure of $\mathcal{G}$ induced by the vertices $\mathcal{D}$. A MWEM $M$ of $\mathcal{D}$ in $\mathcal{G}$ can be found efficiently using the following three steps:

1. Construct the path graph $\bar{\mathcal{G}}[\mathcal{D}]$ using Dijkstra's algorithm.

2. Find the minimum-weight perfect matching $\bar{M} \subseteq \bar{\mathcal{E}}$ in $\bar{\mathcal{G}}[\mathcal{D}]$ using the blossom algorithm.

3. Use $\bar{M}$ and Dijkstra's algorithm to construct the MWEM: $M := \bigoplus_{u,v \in \bar{M}} P^{\min}_{u,v}$.

where here $P^{\min}_{u,v} \subseteq \mathcal{E}$ is a minimum-length path between $u$ and $v$ in $\mathcal{G}$. See Theorem 12.10 of [132] for a proof of this reduction, where their minimum-weight $T$-join is our MWEM, and their set $T$ corresponds to our $\mathcal{D}$. See also [12, 19] for alternative reductions and [30] for a recent review. We give an example of this reduction for the surface code in Figure 1.12.

Unfortunately, solving these three steps sequentially is quite computationally expensive; for example, just the cost of enumerating the edges in $\bar{\mathcal{G}}[\mathcal{D}]$ scales quadratically in the number of detection events $|\mathcal{D}|$, whereas we would ideally like a decoder with an expected running time that scales linearly in $|\mathcal{D}|$. This sequential approach has nevertheless been widely used by QEC researchers, despite its performance being very far from optimal.

A significant improvement was introduced by Fowler [80]. A key observation made by Fowler was that, for QEC problems, typically only low-weight edges in $\bar{\mathcal{G}}[\mathcal{D}]$ are actually used by blossom. Fowler's approach exploited this fact by setting an initial exploration radius in the matching graph, within which separate searches were used to construct some of the edges in $\bar{\mathcal{G}}[\mathcal{D}]$. This exploration radius was then adaptively increased as required by the blossom algorithm. In Chapter 2 we introduce an algorithm for solving the MWPM decoding problem, sparse blossom, that is inspired by Fowler's work, but is different in many of the details.

Before continuing, we will point out two additional details regarding the MWPM decoding problem. Firstly, note that the decoder does not need to output the set of edges directly, but rather can output $L\mathbf{c}$, a prediction of which logical observable measurements were flipped. Recall that the decoder has succeeded if it correctly predicts which observables were flipped, i.e. if $L\mathbf{c} = L\mathbf{e}$. In other words, we can apply a correction at the logical level rather than at the physical level, which is equivalent since $L(\mathbf{c} \oplus \mathbf{e}) = L\mathbf{c} \oplus L\mathbf{e}$. This output is generally much more sparse: for example, for a surface code syndrome extraction circuit such as a memory

experiment, this prediction is simply a single bit, predicting whether or not the logical $X$ (or $Z$) observable measurement outcome was flipped by the error. As we will see in Chapter 2, predicting logical observables $L\mathbf{c}$ rather than the full physical error $\mathbf{c}$ leads to some useful optimizations.

Secondly, observe that the edge weight $\mathbf{w}[i] = \log((1-\mathbf{p}[i])/\mathbf{p}[i])$ is negative when $\mathbf{p}[i] > 0.5$. Fortunately, it is straightforward to decode a syndrome $\mathbf{s}$ for a matching graph $\mathcal{G}$ containing negative edge weights by using efficient pre- and post-processing to instead decode a modified syndrome $\mathbf{s}'$ using a set of adjusted edge weights $\mathbf{v}$ containing only non-negative edge weights. This procedure is explained in Appendix B.5.

# Chapter 2

# Sparse blossom

*The source code for our implementation of the sparse blossom algorithm presented in this chapter, released in PyMatching version 2, can be found on GitHub at* `https://github.com/oscarhiggott/PyMatching`. *PyMatching is also available as a Python 3 pypi package installed via "*`pip install pymatching`*".*

## 2.1 Introduction

The minimum-weight perfect matching decoder, which we reviewed in Section 1.4.10, is the oldest and most popular surface code decoder [64]. The MWPM decoder maps the decoding problem onto a graphical problem by decomposing the error model into $X$-type and $Z$-type Pauli errors [64]. This graphical problem can then be solved with the help of Edmonds' blossom algorithm for finding a minimum-weight perfect matching in a graph [73, 72]. A naive implementation of the MWPM decoder has a worst-case complexity in the number of nodes $N$ in the graph of $O(N^3 \log(N))$, with the expected running time for typical instances found empirically to be roughly $O(N^2)$ [107]. Approximations and optimisations of the MWPM decoder have led to significantly improved expected running times [83, 80, 107]. In particular, Fowler proposed a MWPM decoder with average $O(1)$ parallel running time [80]. However, previously published implementations of MWPM decoders have not demonstrated speeds fast enough for real-time decoding at scale.

There have been several alternatives to the MWPM decoder proposed in the literature. The Union-Find decoder has an almost-linear worst-case running

time [60, 119], and fast hardware implementations have been proposed [55] and implemented [141]. The Union-Find decoder is slightly less accurate than, and can be seen as an approximation of, the MWPM decoder [206]. Maximum-likelihood decoders can achieve a higher accuracy than the MWPM decoder [34, 114, 183] but have high computational complexity, rendering them impractical for real-time decoding. Other decoders, such as correlated MWPM [81], belief-matching [114] and neural network [187, 148] decoders can achieve higher accuracy than MWPM with a much more modest increase in running time. While there has been progress in the development of open-source software packages for decoding surface codes [107, 194], these tools are much slower than stabilizer circuit simulators [92], and have therefore been a bottleneck in surface code simulations. This is perhaps one of the reasons why numerical studies of error correcting codes have often focused on estimating *thresholds* (which require decoding fewer shots), instead of resource overheads (which are more practically useful for making comparisons).

In this chapter, we introduce a new algorithm that solves the MWPM decoding problem, and provide an open-source implementation. The algorithm we introduce, sparse blossom, is a variant of the blossom algorithm which is conceptually similar to the approach taken in Refs. [83, 80], in that it solves the MWPM decoding problem directly on the matching graph, rather than naively breaking up the problem into multiple sequential steps and solving the traditional MWPM graph theory problem as a separate subroutine. This avoids the all-to-all Dijkstra searches often used in implementations of the MWPM decoder. Our implementation, which has been released in version 2 of the PyMatching Python package, is orders of magnitude faster than alternative available tools, and can decode both $X$ and $Z$ bases of a distance-17 surface code circuit (for 0.1% circuit-noise) in under one microsecond per round on a single core, matching the rate at which syndrome data is generated on a superconducting quantum processor. At distance 29 with the same noise model (more than sufficient to achieve $10^{-12}$ logical error rates), PyMatching takes 3.5 microseconds per round to decode on a single core. These results suggest that our sparse blossom algorithm is fast enough for real-time decoding of large-scale

superconducting quantum computers, which will generate syndrome data at a rate of around one microsecond per round [145, 133, 5]. A real-time implementation is likely achievable through parallelisation across multiple cores, and by adding support for decoding a stream, rather than a batch, of syndrome data. Our implementation of sparse blossom has been released in version 2 of the PyMatching Python package, and can be combined with Stim [92] to run simulations in minutes on a laptop that previously would have taken hours on a high-performance computing cluster. See Ref. [113] for the preprint describing the original research contained in this chapter.

We would also like to point the reader to impressive independent work by Yue Wu, who also recently developed a new implementation of the blossom algorithm called fusion blossom [207], available at [205]. The conceptual similarity with our approach is that fusion blossom also solves the MWPM decoding problem directly on the matching graph. However, there are many differences in the details of our respective implementations; for example, fusion blossom explores the graph in a similar way to how clusters are grown in union-find, whereas our approach grows exploratory regions uniformly, managed by a global priority queue. While our approach has faster single-core performance, fusion blossom also supports parallel execution of the algorithm itself, which can be used to achieve faster processing speeds for individual decoding instances. When used for error correction simulations, we note that sparse blossom is already trivially parallelisable by splitting the simulation into batches of shots, and processing each batch on a separate core. However, parallelisation of the decoder itself is important for real-time decoding, to prevent an exponentially increasing backlog of data building up within a single computation [186], or to avoid the polynomial slowdown imposed by relying on parallel window decoding instead [175, 184]. Therefore, future work could explore combining sparse blossom with the techniques for parallelisation introduced in fusion blossom.

The chapter is structured as follows. In Section 2.2 we explain our algorithm, sparse blossom, before describing the data structures we use for our implementation in Section 2.3. In Section 2.4 we analyse the running time of sparse blossom, and in

Section 2.5 we benchmark its decoding time, before concluding in Section 2.6.

## 2.2 Sparse Blossom

The blossom algorithm, introduced by Edmonds [73, 72], is an efficient algorithm for solving the minimum-weight perfect matching problem in general graphs. We review the original blossom algorithm in Appendix B.1 for completeness, and explain the connection between concepts in blossom and sparse blossom in Appendix B.2. However this background on the blossom algorithm is not a prerequisite to understanding our approach, since we reintroduce the relevant concepts in this chapter as required.

The variant of the blossom algorithm we introduce, which we call sparse blossom, directly solves the minimum-weight embedded matching problem relevant to quantum error correction. Sparse blossom does not have a separate Dijkstra step for constructing edges in the path graph $\bar{\mathcal{G}}[\mathcal{D}]$. Instead, shortest path information is recovered as part of the blossom algorithm itself. Put another way, we only discover and store an edge $e \in \bar{\mathcal{E}}$ in $\bar{\mathcal{G}}[\mathcal{D}]$ exactly if and when it is needed by the blossom algorithm; the edges that we track at any point in the algorithm correspond exactly to the subset of edges in $\bar{\mathcal{E}}$ being used to represent the core data structures of the blossom algorithm (*tight* edges, each belonging to an alternating tree, a blossom or a match, which we will define). This leads to very large speedups relative to the sequential approach, where all edges in $\bar{\mathcal{E}}$ are found using Dijkstra searches, despite the vast majority never becoming tight edges in the blossom algorithm. We name the algorithm *sparse* blossom, since it exploits the fact that only a small fraction of the detector nodes correspond to detection events for typical QEC problems (and detection events can be paired up locally), and for these problems our approach only ever inspects a small subset of the nodes and edges in the matching graph.

Before explaining sparse blossom and our implementation, we will first introduce and define some concepts.

### 2.2.1 Key concepts

#### 2.2.1.1 Graph fill regions

A *graph fill region* $R$ of radius $y_R$ is an exploratory region of the matching graph. A graph fill region $R$ contains the nodes and edges (or fractions of edges) in the matching graph which are within distance $y_R$ of its *source*. The source of a graph fill region is either a single detection event, or the surface of other graph fill regions forming a *blossom*. We will define blossoms later on, however for the case that the graph fill region $R$ has a single detection event $u$ as its source, every node or fraction of an edge that is within distance $y_R$ of $u$ is contained in $R$. Note that a graph fill region in sparse blossom is analogous to a node or blossom in the standard blossom algorithm, and the graph fill region's radius is analogous to a node or blossom's dual variable in standard blossom. The sparse blossom algorithm proceeds along a timeline (see Section 2.2.1.7), and the radius of each graph fill region can have one of three growth rates: +1 (growing), -1 (shrinking) or 0 (frozen). Therefore, at any time $t$ we can represent the radius of a region using an equation $y_R = mt + c$, where $m \in \{-1, 0, 1\}$. We will at times refer to a graph fill region just as a *region* when it is clear from context.

We denote by $\mathcal{D}(R)$ the set of detection events in a region $R$. When a region contains only a single detection event, $|\mathcal{D}(R)| = 1$, we refer to it as a trivial region. A region can contain multiple detection events if it has a blossom as its source. As well as its radius equation, each graph fill region may also have blossom children and a blossom parent (both defined in Section 2.2.1.6). It also has a *shell area*, stored as a stack. The shell area of a region is the set of detector nodes it contains, excluding the detector nodes contained in its blossom children. We say that a graph fill region is *active* if it does not have a blossom parent. We will let $\mathcal{R}$ denote the set of all graph fill regions.

#### 2.2.1.2 Compressed edges

A *compressed edge* represents a path through the matching graph between two detection events, or between a detection event and the boundary. Given a path

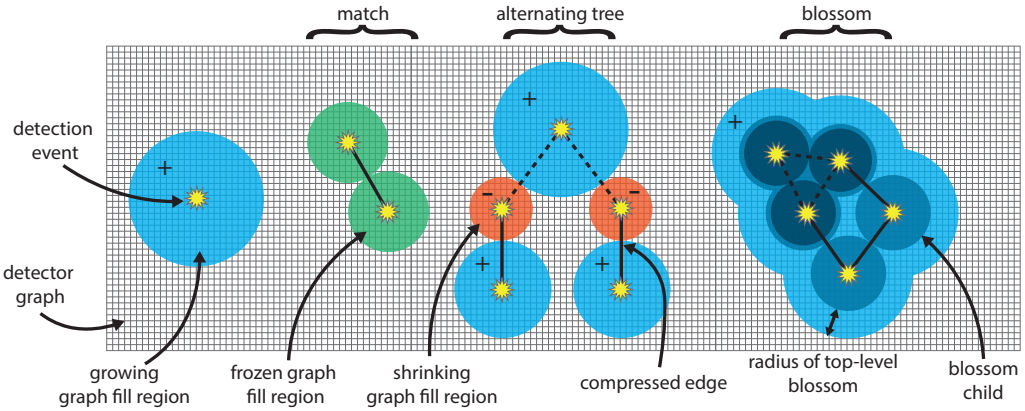



**Figure 2.1:** Key concepts in sparse blossom

$P_{u,v} \subseteq \mathcal{E}$ between $u$ and $v$, where $u$ is a detection event and $v$ is either a detection event or denotes the boundary, the compressed edge $\beta(P_{u,v})$ associated with $P_{u,v}$ is the pair of nodes $(u,v)$ at the endpoints of $P_{u,v}$, as well as the set of logical observables $l(P_{u,v}) \coloneqq \bigoplus_{e \in P_{u,v}} l(e)$ flipped by flipping all edges in $P_{u,v}$. The compressed edge $\beta(P_{u,v})$ is therefore a compressed representation of the path $P_{u,v}$ containing all the information relevant for error correction and, for a given matching graph, can be stored using a constant amount of data (independent of the path length). When the choice of path $P_{u,v} \subseteq \mathcal{E}$ for some given pair of detection events $(u,v)$ is clear from context, we may denote the compressed edge $\beta(P_{u,v})$ instead by $(u,v)$, and may also denote the set of logical observables $l(P_{u,v})$ by $l(u,v)$. We define the *length* of a compressed edge to be the distance $D(u,v)$ between its endpoints $u$ and $v$.

Every compressed edge $\beta(P_{u,v})$ in sparse blossom corresponds to a *shortest* path $P_{u,v}$ between $u$ and $v$. However, we can use a compressed edge to represent any path between $u$ and $v$, and when used in union-find (see Appendix B.3) the path $P_{u,v}$ need not be minimum weight. Compressed edges are used in the representation of several data structures in sparse blossom (alternating trees, matches and blossoms). In particular, each compressed edge $(u,v)$ corresponds to an edge in the path graph $\bar{\mathcal{G}}[\mathcal{D}]$, but unlike in the standard serial approach to implementing the MWPM decoder, we only every discover and construct the small subset of the edges in $\bar{\mathcal{G}}[\mathcal{D}]$ needed by sparse blossom.

We will let $\Gamma$ be the set of all possible compressed edges (e.g. the set $\Gamma$ contains

a compressed edge $(u,v)$ for each edge in $\bar{\mathcal{G}}[\mathcal{D}]$). For a region $R$, we denote by $\delta_\beta(R) \subseteq \Gamma$ the *boundary-compressed-edges* of $R$, defined as

$$\delta_\beta(R) \coloneqq \{(u,v) \in \Gamma \mid u \in \mathcal{D}(R), v \notin \mathcal{D}(R)\}. \tag{2.1}$$

#### 2.2.1.3 Region edges

A *region edge* describes a relationship between two graph fill regions, or between a region and the boundary. We use $(A,B)$ to denote a region edge, where here $A$ and $B$ are both regions. Whenever we describe an edge between two regions, or between a region and the boundary, it is implied that it is a region edge. A region edge $(A,B)$ comprises its endpoints $A$ and $B$ as well as a compressed edge $\beta(P_{u,v})$ representing the shortest path between any detection event in $A$ and any detection event in $B$. We sometimes use the notation $(A_u, B_v)$ for a region edge to explicitly specify the two regions $A$ and $B$ along with the endpoints $(u,v)$ of the compressed edge $\beta(P_{u,v})$ associated with it. More concretely, the compressed edge $\beta(P_{u,v})$ associated with a region edge $(A_u, B_v)$ has endpoints $(u,v)$ defined by

$$(u,v) = \operatorname*{arg\,min}_{x,y \in \mathcal{D}(A) \times \mathcal{D}(B)} D(x,y) \tag{2.2}$$

where here $X \times Y \coloneqq \{(x,y) \mid x \in X, y \in Y\}$ denotes the Cartesian products of sets $X$ and $Y$.

For any region edge that arises in sparse blossom, the following invariants always hold:

1. If $A$ and $B$ are both regions, then either $A$ and $B$ are both active regions (with no blossom-parent), or both have the same blossom-parent.

2. The compressed edge $(u,v)$ associated with a region edge $(A_u, B_v)$ is always *tight* (it would correspond to a tight edge in $\bar{\mathcal{G}}[\mathcal{D}]$ in blossom). More precisely, a compressed edge $(u,v)$ is tight if it satisfies

$$D(u,v) = \sum_{R \in \mathcal{R} \,:\, (u,v) \in \delta_\beta(R)} y_R. \tag{2.3}$$

In other words, if there is a region edge $(A, B)$, then regions $A$ and $B$ must be touching.

#### 2.2.1.4 Matches

We call a pair of regions $A$ and $B$ that are matched to each other a *match*. The matched regions $A$ and $B$ must be touching, assigned a growth rate of zero (they are frozen), and must be joined by a region edge $(A, B)$ which we refer to as a *match-edge*. An example of a match is shown in the middle of Figure 2.1. Initially, all regions are unmatched, and once the algorithm terminates, every region (either a trivial region or a blossom) is matched either to another region or to the boundary.

#### 2.2.1.5 Alternating trees

An alternating tree is a tree where each node corresponds to an active graph fill region and each edge corresponds to a region edge. We refer to each region edge in the alternating tree as a *tree-edge*. Two regions connected by a tree-edge must always be touching (since every tree-edge is a region edge).

An alternating tree contains at least one growing region and can also contain shrinking regions, and always contains exactly one more growing region than shrinking region. Each growing region can have any number of children (each a *tree-child*), all of which must be shrinking regions, and can have a single parent (a *tree-parent*), also a shrinking region. Each shrinking region has a single child, a growing region, as well as a single parent, also a growing region. The leaves of an alternating tree are therefore always growing regions. An example of an alternating tree is shown in Figure 2.1.

#### 2.2.1.6 Blossoms

When two growing regions from within the same alternating tree hit each other they form a *blossom*, which is a region containing an odd-length cycle of regions called a *blossom cycle*. More concretely, we will denote a blossom cycle as an ordered tuple of regions $(R_0, R_1, \ldots, R_{k-1})$ for some odd $k$ and each region $R_i$ in the blossom cycle is connected to each of its two neighbours by a region edge. In other words, the blossom cycle has region edges $\{(R_i, R_{(i+1) \mod k}) | i \in \{0, 1, \ldots, k-1\}\}$ for all

$1 \leq i \leq k$. An example of a blossom is shown on the right side of Figure 2.1.

The regions in the blossom cycle are called the blossom's *blossom-children*. If a blossom $B$ has region $b$ as one of its blossom-children, then we say that $B$ is the *blossom-parent* of $b$. Neighbouring regions in a blossom cycle must be touching (as required by the fact that they are connected by a region edge). Blossoms can also be nested; each blossom-child can itself be a blossom, with its own blossom-children. For example, the top-left blossom-child of the blossom in Figure 2.1 is itself a blossom, with three blossom-children. A blossom *descendant* of a blossom $B$ is a region that is either a blossom child of $B$ or is recursively a descendant of any blossom child of $B$. Similarly, a blossom *ancestor* of $B$ is a region that is either the blossom parent of $B$ or (recursively) an ancestor of the blossom parent of $B$. The radius $y_B$ of a blossom $B$ is the distance it has grown since it formed (the minimum distance between a point on its surface and any point on its source). This is visualised as the distance across the shell of the blossom in Figure 2.1.

We say that a detector node $u$ is *contained* in a region $R$ if $u$ is in the shell area of $R$. A detector node can only be contained in at most one region (shell areas are disjoint). If a detector node is not contained in a region we say that the node is *empty*, and otherwise it is *occupied*. We say that a detector node $u$ is *owned* by a region $R$ either if $u$ is contained in $R$, or if $u$ is contained in a blossom descendant of $R$. The distance $D_S(R,u)$ between a detector node and the surface of a region $R$ is

$$D_S(R,u) = \min_{x \in \mathcal{D}(R)} \left( D(u,x) - \sum_{A \in \mathcal{R}: x \in \mathcal{D}(A), A \leq R} y_A \right) \tag{2.4}$$

where here $A \leq R$ denotes that $A$ is either a descendant of $R$ or $A = R$. If $D_S(R,u) < 0$ then detector node $u$ is owned by region $R$, if $D_S(R,u) > 0$ then detector node $u$ is not owned by $R$, whereas if $D_S(R,u) = 0$ then $u$ may or may not be owned by $R$ (it is on the surface of $R$).

#### 2.2.1.7 The timeline

The algorithm proceeds along a timeline with the time $t$ increasing monotonically as different events occur and are processed. Examples of events include a region

arriving at a node, or colliding with another region. The time that each event occurs is determined based on the radius equations of the regions involved, as well as the structure of the graph. The algorithm terminates when there are no more events left to be processed, which happens when all regions have been matched, and have therefore become frozen.

### 2.2.2 Architecture

Sparse blossom is split into different components: a *matcher*, a *flooder* and a *tracker*. Each component has different responsibilities. The matcher is responsible for managing the structure of the alternating trees and blossoms, without knowledge of the underlying structure of the matching graph. The flooder handles how graph fill regions grow and shrink in the matching graph, as well as noticing when a region collides with another region or the boundary. When the flooder notices a collision involving a region, or when a region reaches zero radius, the flooder notifies the matcher, which is then responsible for modifying the structure of the alternating trees or blossoms. The tracker is responsible for handling *when* the different events occur, and ensures that the flooder handles events in the correct order. The tracker uses a single priority queue, and informs the flooder when every event should be handled.

### 2.2.3 The matcher

At the initialisation stage of the algorithm, every detection event is initialised as the source of a growing region, a trivial alternating tree. As these regions grow and explore the graph, they can hit other (growing or frozen) regions, as well as the boundary, until eventually all regions are matched and the algorithm terminates. A growing region cannot hit a shrinking region, since shrinking regions recede exactly as quickly as growing regions expand.

When the flooder notices that a growing region $R$ has hit another (growing or frozen) region $R'$ or the boundary, it finds the *collision edge* and gives it to the matcher. The collision edge is a region edge between $R$ and $R'$ (or, if $R$ hit the boundary, then between $R$ and the boundary). The collision edge can be constructed by the flooder from local information at the point of collision, as will be explained in

Section 2.2.4, and it is used by the matcher when handling events that change the structure of the alternating tree (which we refer to as *alternating tree events*). The matcher is responsible for handling alternating tree events, as well as for recovering the pairs of matched detection events once all regions have been matched.

#### 2.2.3.1 Alternating tree events

There are seven different types of events that can change the structure of an alternating tree, and which are handled by the matcher, shown in Figure 2.2:

(a) A growing region $R$ in an alternating tree $T$ can hit a region $M_1$ that is matched to another region $M_2$. In this case, $M_1$ becomes a tree-child of $R$ in $T$ (and starts shrinking), and $M_2$ becomes a tree-child of $M_1$ in $T$ (and starts growing). The collision edge $(R, M_1)$ and match-edge $(M_1, M_2)$ both become tree-edges ($R$ is the tree-parent of $M_1$, and $M_1$ is the tree-parent of $M_2$).

(b) A growing region $R$ in an alternating tree $T$ hits a growing region $R'$ in a different alternating tree $T'$. When this happens, $R$ is matched to $R'$ and the remaining regions in $T$ and $T'$ also become matched. The collision edge $(R, R')$ becomes a match-edge, and a subset of the tree-edges also become match-edges. All the regions in $T$ and $T'$ become frozen.

(c) A growing region $R$ in an alternating tree $T$ can hit another growing region $R'$ in the *same* alternating tree $T$. This leads to an odd-length cycle of regions which form the blossom cycle $C$ of a new blossom $B$. The region edges (blossom edges) in the blossom cycle are formed from the collision edge $(R, R')$, as well as the tree-edges along the paths from $R$ and $R'$ to their most recent common ancestor $A$ in $T$. The newly formed blossom becomes a growing node in $T$. When forming the cycle $C$, we define the *orphans* $O$ to be the set of shrinking regions in $T$ but not $C$ that were each a child of a growing region in $C$. The orphans become tree-children of $B$ in $T$. The compressed edge associated with the new tree-edge $(B, T)$ (connecting the new blossom region to its tree-parent $T$) is just the compressed edge that was associated with the old tree-edge $(A, T)$. Similarly, the compressed edges connecting the

each orphan to its alternating tree parent remains unchanged (even though its parent *region* becomes $B$). In other words, if an orphan $O^i$ had been connected to its tree-parent $R^i$ by the tree-edge $(O^i_u, R^i_v)$ before the blossom formed, the new tree-edge connecting it to its new tree-parent $B$ will be $(O^i_u, B^i_v)$ once the blossom $B$ forms and the region $R^i$ becomes part of the blossom cycle $C$.

(d) When a blossom $B$ in an alternating tree $T$ shrinks to a radius of zero, instead of the radius becoming negative the blossom must *shatter*. When the blossom shatters, the odd-length path through its blossom cycle from the tree-child of $B$ to the tree-parent of $B$ is added to $T$ as growing and shrinking regions. The even length path becomes matches. The blossom-edges in the odd length path become tree-edges, and some of the blossom-edges in the even length path become match-edges (the remaining blossom-edges are forgotten). Note that the endpoints of the compressed edges associated with the tree-edges joining $B$ to its tree-parent and tree-child are used to determine where and how the blossom cycle is cut into two paths.

(e) When a trivial region $R$ shrinks to a radius of zero, instead of the radius becoming negative a blossom forms. If $R$ has a child $C$ and parent $P$ in the alternating tree $T$, when $R$ has zero radius it must be that $C$ is touching $P$ (it is as if $C$ has collided with $P$). The newly formed blossom has the blossom cycle $(P,R,C)$. The old tree-edges $(P_u, R_v)$ and $(R_v, C_w)$ become blossom edges in the blossom cycle. The blossom edge connecting $C$ with $P$ in the blossom cycle is computed from edges $(P_u, R_v)$ and $(R_v, C_w)$ and is $(C_w, P_u)$. In other words, its compressed edge has endpoints $(w,u)$ with logical observables $l(u,v) \oplus l(v,w)$.

(f) When a growing region $R$ in an alternating tree $T$ hits the boundary, $R$ matches to the boundary and the collision edge becomes the match-edge. The remaining regions in $T$ also become matches.

(g) When a growing region $R$ in an alternating tree $T$ hits a region $M$ that is matched to the boundary, then $M$ instead becomes matched to $R$ (and the

collision edge becomes the match-edge), and the remaining edges in $T$ also become matches.

Some of these events involve changing the growth rate of regions (for example, two growing regions both become frozen regions when they match to each other). Therefore, when handling each alternating tree event, the matcher informs the flooder of any required changes to region growth.

#### 2.2.3.2 Matched detection events from matched regions

Provided there is a valid solution, eventually all regions become matched to other regions, or to the boundary. However, some of these matched regions may be blossoms, not trivial regions. To extract the compressed edge representing the match for each detection event instead, it is necessary to *shatter* each remaining blossom, and match its blossom children, as shown in in Figure 2.3. Suppose a blossom $B$, with blossom cycle $C$, is matched to some other region $R$ with the match-edge $(B_u, R_v)$, where we recall that $u$ is a detection event in $B$ and $v$ is a detection event in $R$. We find the blossom child $c \in C$ of $B$ which contains the detection event $u$. We shatter $B$ and match $c$ to $R$ with the compressed edge $(u, v)$. The remaining regions in the blossom cycle $C$ are then split into neighbouring pairs, which become matches. This process is repeated recursively until all matched regions are trivial regions.

### 2.2.4 The flooder

The flooder is responsible for managing how graph fill regions grow, shrink or collide in the matching graph, and is not concerned with the structure of the alternating trees and blossoms, which is instead handled by the matcher. We refer to the events handled by the flooder as *flooder events*.

Broadly speaking, we can categorise flooder events into four different types:

1. ARRIVE: A growing region $R$ can grow into an empty detector node $u$.
2. LEAVE: A shrinking region $R$ can leave a detector node $u$.
3. COLLIDE: A growing region can hit another region, or the boundary.
4. IMPLODE: A shrinking region can reach a radius of zero.

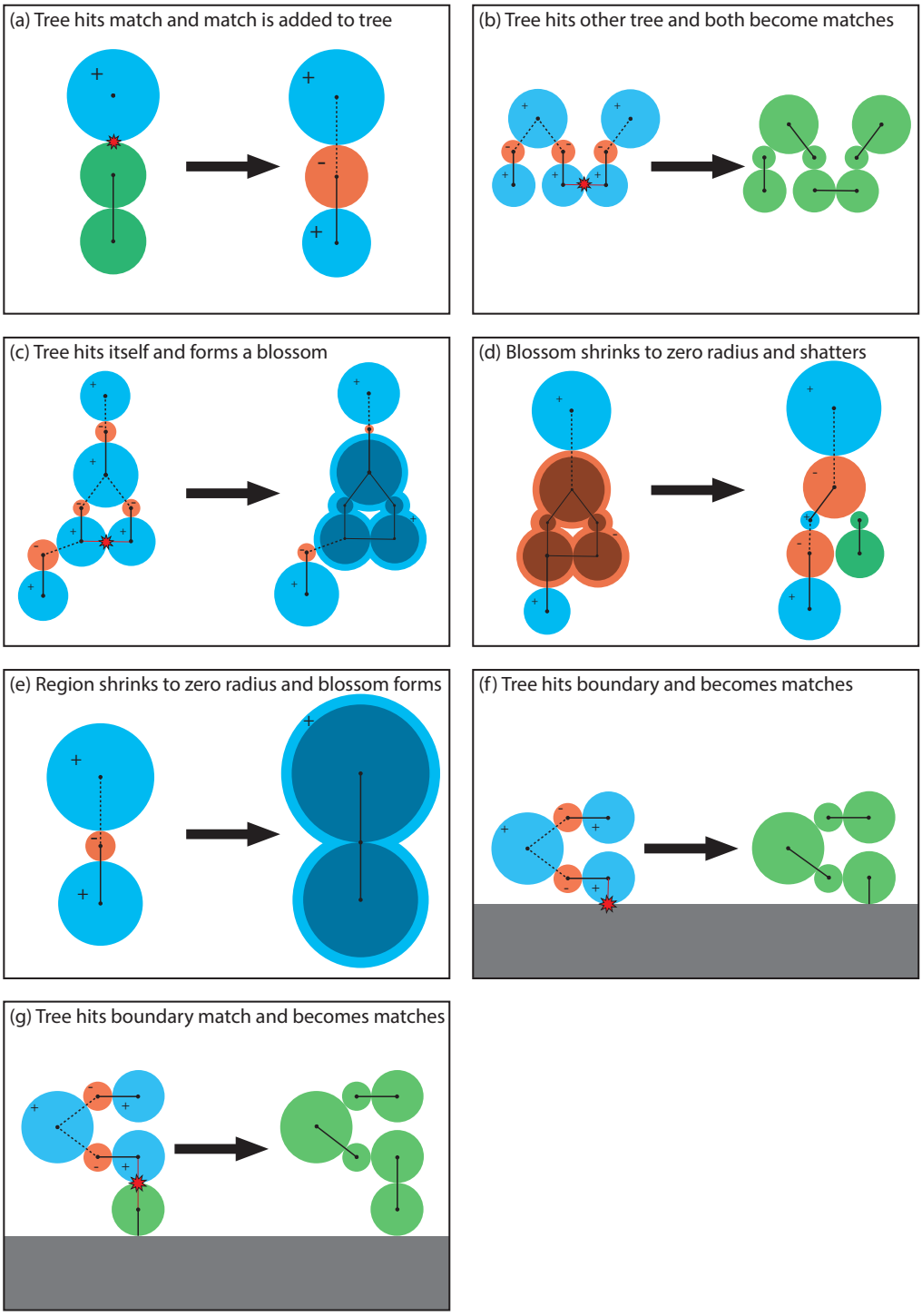



**Figure 2.2:** The main events that change the structure of alternating trees. For clarity, the background matching graph has been omitted. Each node corresponds to a detection event, and each edge corresponds to a compressed edge.

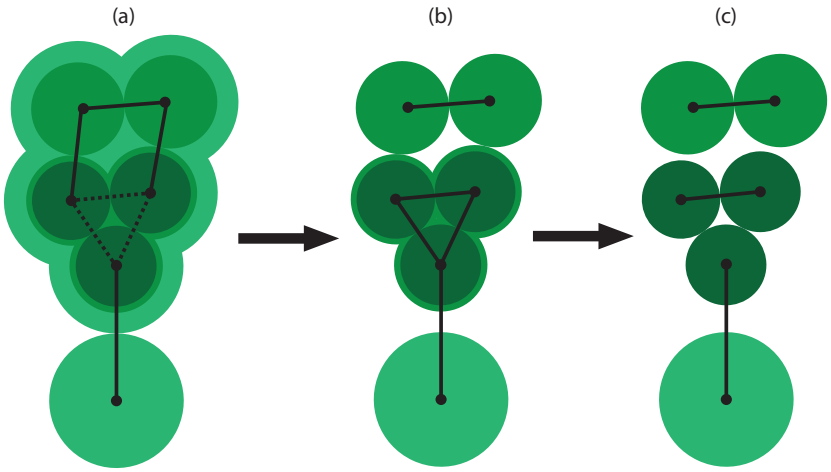



**Figure 2.3:** Shattering a matched blossom. Solid lines within a blossom are edges in the topmost blossom cycle. Dashed lines are edges in the blossom cycle of the blossom-child of the topmost blossom.

Let us first consider what happens for ARRIVE and LEAVE events. Neither of these types of events can change the structure of the alternating trees or blossoms, so the matcher does not need to be notified. Instead, it is the flooder's responsibility to ensure that any new flooder events get scheduled (inserted into the tracker's priority queue) after the events have been processed. When a region grows into a node $u$, the flooder *reschedules* the node $u$, by notifying the tracker of the next flooder event that can occur along an edge adjacent to $u$ (either an ARRIVE or COLLIDE event, see Section 2.2.4.1). When a shrinking region leaves a node, the flooder immediately checks the top of the shell area stack and schedules the next LEAVE or IMPLODE event (see Section 2.2.4.2).

The flooder only needs to notify the matcher of a COLLIDE or IMPLODE event, and when a collision occurs the flooder passes the collision edge to the matcher as well. When either of these types of events occur, the matcher may change the growth rate of some regions when updating the structure of alternating trees or blossoms. The matcher then notifies the flooder of any change of growth rate, which may require the flooder to reschedule some flooder events. For example, if a region $R$ was shrinking, but then becomes frozen or starts growing, the flooder reschedules all nodes contained in $R$ (including blossom children, and their children etc.), to check for new ARRIVE or COLLIDE events. When a region starts shrinking, then

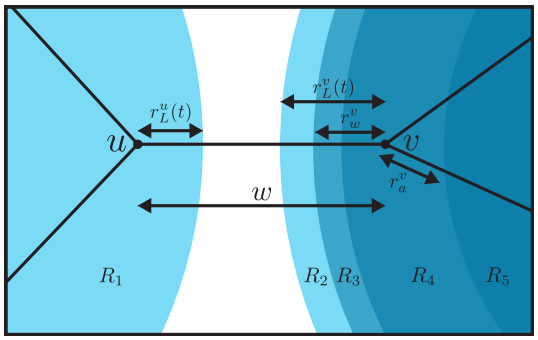



**Figure 2.4:** Two regions $R_1$ and $R_2$ colliding along an edge $(u,v)$. Node $u$ is contained in region $R_1$, which is an active region (no blossom parent). Node $v$ is contained in region $R_4$, and is owned by $R_4$ as well as its blossom ancestors $R_2$ and $R_3$. Region $R_4$ also has $R_5$ as one of its blossom children. We have labelled the local radius $r_L^u(t)$ of node $u$ and the local radius $r_L^v(t)$ of node $v$, as well as the wrapped radius $r_w^v$ of $v$ and the radius of arrival for $v$. The edge weight $w$ of the edge $(u,v)$ is also shown.

the flooder informs the tracker of the next LEAVE or IMPLODE event by checking the top of the shell area stack.

The correct ordering of these flooder events is ensured by the tracker, and we create a growing region for each detection event simultaneously at time $t = 0$. Our implementation therefore uses an alternating tree growth strategy that is analogous to what's described as a "multiple tree approach with fixed $\delta$" in [131].

#### 2.2.4.1 Rescheduling a node

When the flooder *reschedules* a node, it looks for an ARRIVE or COLLIDE event along each neighboring edge. There will be an ARRIVE event along an edge if one node is occupied by a growing region and the other is empty. There will be a COLLIDE event if both nodes are owned by active regions $(R_1, R_2)$ with growth rates $(1,1)$, $(0,1)$ or $(1,0)$, or if one region is growing towards a boundary, for a half-edge.

In order to calculate *when* an ARRIVE or COLLIDE event will occur along an edge, we use the *local radius* of each node. The *local radius* $r_L^v(t)$ of an node $v$ is the amount that regions owning $v$ have grown beyond $v$ (see Figure 2.4). To define the local radius more precisely, we will need some more definitions. The *radius of arrival* $r_a^v$ for an occupied node $v$ contained in a region $R$ is the radius that $R$ had

when it arrived at $v$. We denote the radius of a region $A$ by $y_A(t)$ and we let $\mathcal{O}(v)$ be the set of regions that own a detector node $v$ (the region that $v$ is contained in, as well as its blossom ancestors). The local radius is then defined as

$$r_L^v(t) = -r_a^v + \sum_{A \in \mathcal{O}(v)} y_A(t). \tag{2.5}$$

Both the local radius and radius of arrival of a node $v$ are defined to be zero if $v$ is empty. Therefore, for an edge $(u,v)$ with weight $w$, the time of an ARRIVE or COLLIDE event can be found by solving $r_L^u(t) + r_L^v(t) = w$ for $t$. The only situation in which this involves division is when the local radius of both nodes are growing (have gradient one), in which case the collision occurs at time $t_{\text{collide}} = (w - r_L^u(0) - r_L^v(0))/2$. However, provided all edges are assigned even integer weights all flooder events, including these collisions between growing regions, occur at integer times.

#### 2.2.4.2 Rescheduling a shrinking region

When a region is shrinking, we find the time of the next LEAVE or IMPLODE event by inspecting the shell area stack of the region. If the stack is empty, or if the region has no blossom children and only a single node remains on the stack (the region's source detection event), then the next event is an IMPLODE event, the time of which can be found from the $x$-intercept of the region's radius equation. Otherwise, the next event is a LEAVE event, with the node $u$ at the top of the stack leaving the region. We find the time of this next LEAVE event using the local radius of $u$, by solving $r_L^u(t) = 0$ for $t$. Using this approach, shrinking a region is much cheaper than growing a region, as it doesn't require enumerating edges in the matching graph.

#### 2.2.4.3 An example

We give a small example of how the timeline of the flooder progresses in Figure 2.5. Since regions $r_1$, $r_2$ and $r_3$ are all initialised at $t = 0$, their radius equations are all equal to $t$. Regions $r_1$ and $r_2$ are separated by a single edge with weight 8 and therefore collide at time $t = 4$ (and recall that edge weights are always even integers to ensure collisions occur at integer times). When the matcher is informed of the

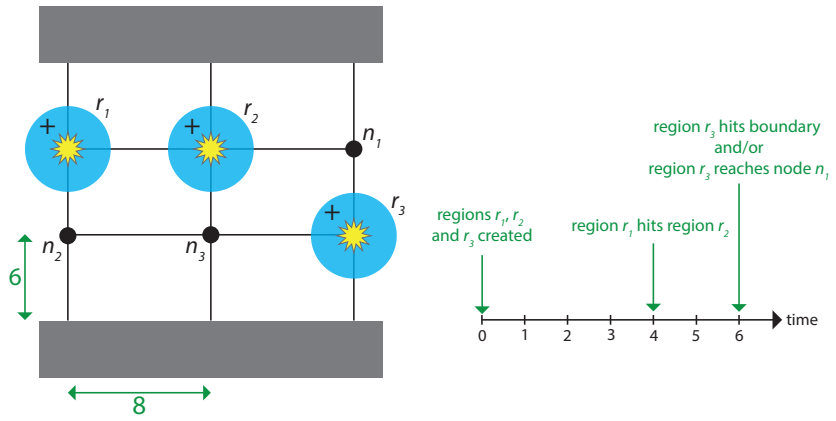



**Figure 2.5:** An example of some flooder events in a matching graph with three growing regions.

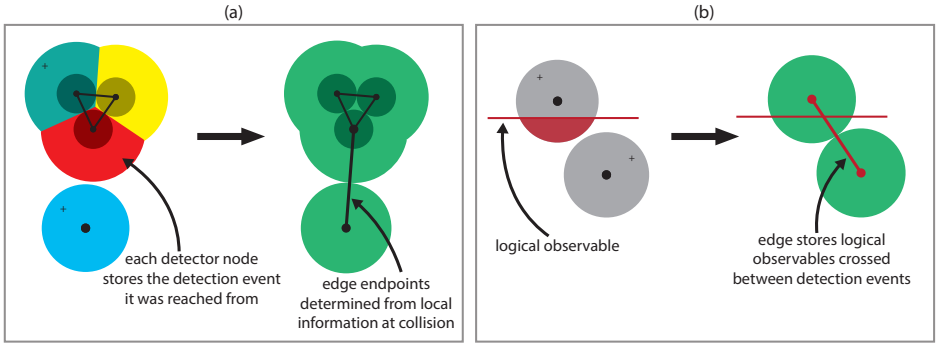



**Figure 2.6:** (a) As a region expands, each detector node it contains stores the detection event it was reached from (visualised by the Voronoi-style colouring of the blossom on the left). When a collision occurs, this allows the endpoints of the corresponding compressed edge (collision edge) to be determined from local information at the point of collision. (b) Each detector node also stores (as a 64-bit bitmask) the observables that were crossed to reach it from the detection event it was reached from. This allows the observables bitmask of the compressed edge to be recovered efficiently, also from local information at the point of collision.

collision, $r_1$ and $r_2$ are matched and become frozen regions. Region $r_3$ reaches empty node $n_1$ and the boundary at the same time ($t = 6$), and so there are two equally valid sequences of events. Either region $r_3$ matches to the boundary, and never reaches $n_1$, or $r_3$ reaches $n_1$ and then matches to the boundary. Clearly the final state of the algorithm is not unique, however there is a unique solution *weight*, and in this instance, both outcomes lead to the same set of compressed edges in the solution.

#### 2.2.4.4 Compressed tracking

Whenever a collision occurs between two regions $A$ and $B$, the flooder constructs a region edge $(A,B)$, which we recall includes a compressed edge corresponding to the shortest path between a detection event in $A$ and a detection event in $B$. By storing relevant information on nodes as regions grow, the compressed edge can be constructed efficiently using local information at the point of collision (e.g. using only information stored on the edge $(u,v)$ that the collision occurs on, and on the nodes $u$ and $v$).

This is explained in Figure 2.6. As a region $R$ reaches an empty node $v$ by growing along a edge $(u,v)$ from a predecessor node $u$, we store on $v$ a pointer to the detection event $\mathcal{S}(v)$ it was reached from (which can simply be copied from $u$). In other words, we set $\mathcal{S}(v) \coloneqq \mathcal{S}(u)$ once $v$ is reached from $u$. Initially, when a search is started from a detection event $w$ (i.e. a trivial growing region is created containing $w$), then we set $\mathcal{S}(w) \coloneqq w$. We refer to $\mathcal{S}(v)$ as the *source detection event* of $v$.

We also store on $v$ the set of observables $l(v)$ crossed during the region growth (e.g. the observables crossed along a shortest path from $\mathcal{S}(v)$ to $v$). This set of crossed observables $l(v)$ can be efficiently computed when $v$ is reached from $u$ along edge $(u,v)$ from $l(v) \coloneqq l(u) \oplus l(u,v)$, where we implement $\oplus$ as a bitwise XOR since $l(u)$ and $l(u,v)$ are stored as bitmasks. Initially, when a trivial growing region is created at a detection event $w$ we set $l(w)$ to the empty set.

Therefore, when a collision occurs between regions $R$ and $R'$ along an edge $(p,q)$, the endpoints $(x,y)$ of the compressed edge associated with the collision edge $(R_x, R'_y)$ can then be determined locally as $x = \mathcal{S}(p)$ and $y = \mathcal{S}(q)$. The observables $l(x,y)$ associated with the collision edge can be computed locally as $l(x,y) \coloneqq l(p) \oplus l(q) \oplus l(p,q)$. Note that compressed tracking can also be used to remove the peeling step of the union-find decoder [60], as we explain in Appendix B.3. We note that it would also be interesting to explore how compressed tracking could be generalised to improve decoding for other families of codes that are not decodable with matching (for which the corresponding error models are not graphlike).

In PyMatching, we only rely on compressed tracking when there are 64 logical

observables or fewer. When there are more than 64 logical observables, we use sparse blossom to find which detection events are matched to each other. Then after sparse blossom has completed, for each matched pair $(u, v)$ we use a bi-directional Dijkstra search (implemented by adapting the flooder and tracker as required) to find the shortest path between $u$ and $v$. If $C \subseteq \mathcal{E}$ is the set of all edges along the shortest paths found this way by the Dijkstra search post-processing, then the solution output by PyMatching is $\bigoplus_{e_i \in C} l(e_i)$. Note that since we are only post-processing with Dijkstra rather than constructing the full path graph, this only adds a small relative overhead (typically less than 50%) to the runtime.

### 2.2.5 Tracker

The tracker is responsible for ensuring flooder events occur in the correct order. A simple approach one *could* take to implement the tracker would just be to place every flooder event in a priority queue. However, many of the potential flooder events are chaff. For example, when a region $R$ is growing, a flooder event would be added to the queue for each of its neighboring edges. We say an edge $(u, v)$ is a neighbor of a region $R$ if $u$ is in $R$ and $v$ is not (or vice versa). Along each neighboring edge, there will be an event either corresponding to the region growing into an empty node, or colliding with another region or boundary. However, if the region becomes frozen or shrinking, then all of these remaining events will be invalidated.

To reduce this chaff, rather than adding every flooder event to a priority queue, the tracker instead adds *look-at-node* and *look-at-region* events to a priority queue. The flooder just finds the time of the *next* event at a node or region, and asks the tracker for a reminder to look back at that time. As a result, at each node, we only need to add the next event to the priority queue. The remaining potential flooder events along neighboring edges will not be added if they have become invalidated.

When the flooder reschedules a node, it finds the time of the next ARRIVE or COLLIDE event along a neighboring edge, and asks the tracker for a reminder to look back at that time (a *look-at-node* event). The flooder finds the time of the next LEAVE or IMPLODE event for a shrinking region by checking the top of the shell area stack, and asks the tracker for a reminder to look back at the region at that time

(a *look-at-region* event). To further reduce chaff, the tracker only adds a *look-at-node* or *look-at-region* event to the priority queue if it will occur at an earlier time than an event already in the queue for the same node or region. Once the tracker reminds the flooder to look back at a node or region, the flooder checks if it is still a valid event by recomputing the next event for the node or region, processing it if so.

## 2.3 Data structures

In this section, we outline the data structures we use in sparse blossom. Each detector node $u$ stores its neighbouring edges in the matching graph as an adjacency list. For each neighbouring edge $(u,v)$, we store its weight $w((u,v))$ as a 32-bit integer, the observables $l(u,v)$ it flips as a 64-bit bitmask, as well as its neighbouring node $v$ as a pointer (or as nullptr for the boundary). Edge weights are discretised as *even* integers, which ensures that all events (including two growing regions colliding) occur at integer times.

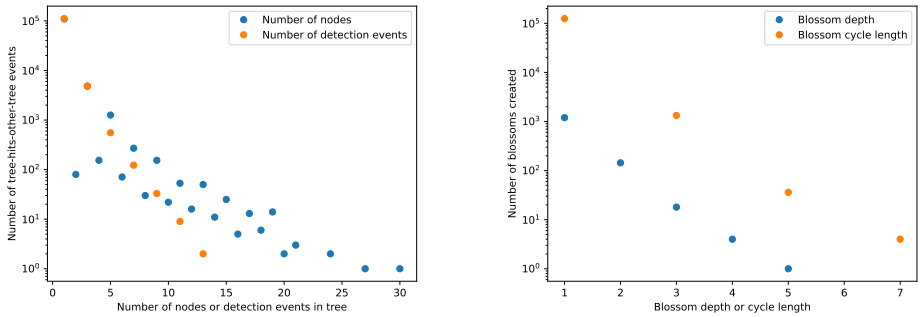
Each *occupied* detector node $v$ stores a pointer to the source detection event it was reached from $\mathcal{S}(v)$, a bitmask of the observables crosssed $l(v)$ along the path from its source detection event and a pointer to the graph fill region $C(v)$ it is contained in (the region that arrived at the node). There are some properties that we cache on each occupied node whenever the blossom structure changes, in order to speed up the process of finding the next COLLIDE or ARRIVE event along an edge. This includes storing a pointer to the *active* region the occupied detector node $v$ is owned by (the topmost blossom ancestor of the region it is contained in) as well as its radius of arrival $r_a^v$, which is the radius that the node's containing region $C(v)$ had when it arrived at $v$ (see Section 2.2.4.1). Additionally, we cache the *wrapped radius* $r_w^v$ of each detector node $v$ whenever its owning regions' blossom structure changes (if a blossom is formed or shattered). The wrapped radius of an occupied node $v$ is the local radius of the node, excluding the (potentially growing or shrinking) radius of the active region it is owned by. If we let $y_A(t)$ be the radius of the active region that owns $v$, we can recover the local radius from the wrapped radius with $r_L^v(t) = r_w^v + y_A(t)$.

Each graph fill region $R$ has a pointer to its blossom parent and its topmost blossom ancestor. Its blossom cycle is stored as an array $L$ of *cycle edges*, where the cycle edge $L[i]$ stores a pointer to the $i$th blossom child of $R$, along with the compressed edge associated with the region edge joining child $i$ to child $i+1 \bmod n_c$, where $n_c$ is the number of blossom children of $R$. Its shell area stack is an array of pointers to the detector nodes it contains (in the order they were added). For its radius $y_R(t) = mt + c$ we use 62 bits to store the $y$-intercept $c$ and with 2 bits used to store the gradient $c \in \{-1, 0, 1\}$. Each region also stores its match as a pointer to the region it is matched to, along with the compressed edge associated with the match-edge. Finally, each growing or shrinking region has a pointer to an AltTreeNode.

An AltTreeNode is used to represent the structure of an alternating tree. Each AltTreeNode corresponds to a growing region in the tree, as well as its shrinking parent region (if it has one). Each AltTreeNode has a pointer to its growing graph fill region and its shrinking graph fill region (if it has one). It also has a pointer to its parent AltTreeNode in the alternating tree (as a pointer and a compressed edge), as well as to its children (as an array of pointers and compressed edges). We also store the compressed edge corresponding to the shortest path between the growing and shrinking region in the AltTreeNode.

Each detector node and graph fill region also has a tracker field, which stores the desired time the node or region should next be looked at, as well as the time (if any) it is already scheduled to be looked at as a result of a *look-at-node* or *look-at-region* event already in the priority queue (called the *queued time*). The tracker therefore only needs to add a new *look-at-node* or *look-at-region* event to the priority queue if its desired time is set to be *earlier* than its current queued time.

A property of the algorithm is that each event dequeued from the priority queue must have a time that is greater than or equal to all previous events dequeued. This allows us to use a radix heap [4], which is an efficient monotone priority queue with good caching performance. Since the next flooder event can never be more than one edge length away from the current time, we use a cyclic time window for the priority,




**(a)** **(b)**

**Figure 2.7:** Distribution of alternating tree and blossom sizes observed in sparse blossom when decoding 1000 shots of distance-11 surface code circuits with $p = 0.3\%$ circuit-level noise. (a) A histogram of the size of alternating trees observed in events where a tree hits another tree, in terms of both the number of detection events and detector nodes contained in each tree. (b) A histogram of the size of blossoms formed, in terms of both the length of each blossom's blossom cycle, as well as its recursive depth. A blossom depth or cycle of size one is a trivial blossom (a graph fill region without blossom children).

rather than the total time. We use 24, 32 and 64 bits of integer precision for the edge weights, flooder event priorities and total time, respectively.

## 2.4 Expected running time

Empirically, we observe an almost-linear running time of our algorithm for surface codes below threshold (see Figure 2.8). We would expect the running time to be linear at low physical error rates, since in this regime a typical error configuration will consist of small isolated clusters of errors. Provided the clusters are sufficiently well separated from one another, each cluster is essentially handled as an independent matching problem by sparse blossom. Furthermore, using results from percolation theory [149], we would expect the number of clusters of a given size to decay exponentially in this size [80]. Since the number of operations required to match a cluster is polynomial in its size, this leads to a constant cost per cluster, and therefore an expected running time that is linear in the size of the graph.

In more detail, suppose we highlight every edge in the matching graph $\mathcal{G}$ ever touched by sparse blossom when decoding a particular syndrome. Now consider the subgraph $\mathcal{F}$ of $\mathcal{G}$ induced by these highlighted edges. This graph will generally have

many connected components, and we will refer to each connected component as a *cluster region*. Clearly, running sparse blossom on each cluster region separately will give an identical solution to running the algorithm on $\mathcal{G}$ as a whole. Let us *assume* that the probability that a detector node in $\mathcal{G}$ is within a cluster region of $n_c$ detector nodes is at most $Ae^{-bn_c}$ for some $b > 0$. Since the worst-case running time of sparse blossom is polynomial in the number of nodes $n$ (at most $O(n^4)$, see Appendix B.4), the expected running time to decode a cluster region (if any) at a given node is at most $\sum_{n_c=1}^{\infty} Ae^{-bn_c} n_c^4 = O(1)$, i.e. constant. Therefore, the running time is linear in the number of nodes. Here we have assumed that the probability of observing a cluster region at a node decays exponentially in its size. However, in [80] it was shown that this is indeed the case at very low error rates. Furthermore, we provide empirical evidence for this exponential decay for error rates of practical interest in Figure 2.7, where we plot the distribution of the sizes of alternating trees and blossoms observed when using sparse blossom to decode surface codes with 0.3% circuit-level noise. Our benchmarks in Figure 2.8 and Figure 2.9 are also consistent with a running time that is roughly linear in the number of nodes, and even above threshold (Figure 2.10) the observed complexity of $O(n^{1.32})$ is only slightly worse than linear.

Note that here we have ignored the fact that our radix heap priority queue is shared shared by all clusters. This does not impact the overall theoretical complexity, since the insert and extract-min operations of the radix heap have an amortized time complexity that is independent of the number of items in the queue. In particular, inserting an element takes $O(1)$ time, and extract-min takes amortized $O(B)$ time, where $B$ is the number of bits used to store the priority (for us $B = 32$).

Nevertheless, our use of the timeline concept and a multiple tree approach (relying on the priority queue) does result in more cache misses for very large problem instances, empirically resulting in an increase in the processing time per detection event. This is because events that are local in time (and therefore processed soon after one another) are in general not local in memory, since they can require inspecting regions or edges that are very far from one another in the matching graph.

In contrast we would expect a single tree approach to have better memory locality for very large problem instances. However, an advantage of the multiple tree approach we have taken is that it "touches" less of the matching graph. For example, consider a simple problem of two isolated detection events in a uniform matching graph, separated by distance $d$. In sparse blossom, since we use a multiple tree approach, these two regions will grow at the same rate and will have explored a region of radius $d/2$ when they collide. In a 3D graph, let's assume that a region of radius $r$ touches $\approx kr^3$ edges for some constant $k$. In contrast, in a single tree approach, one region will grow to radius $d$ and collide with the region of the other detection event, which will still have radius 0. Therefore, the multiple tree approach has touched $\approx 2k(d/2)^3$ edges, $\approx 4\times$ fewer than the $kd^3$ edges touched by the single tree approach. This is essentially the same advantage that a bidirectional Dijkstra search has over a regular Dijkstra search.

## 2.5 Computational results

We benchmarked the running time of our implementation of sparse blossom (PyMatching 2) for decoding surface code memory experiments (see Figure 2.8, Figure 2.9 and Figure 2.10). For 0.1% circuit-level depolarising noise, sparse blossom processes both X and Z bases of distance-17 surface code circuits in less than one microsecond per round of syndrome extraction on a single core, which matches the rate at which syndrome data is generated by superconducting quantum computers.

At low physical error rates (e.g. Figure 2.8), the roughly linear scaling of PyMatching v2 is a quadratic improvement over the empirical scaling of an implementation that constructs the path graph explicitly and solves the traditional MWPM problem as a separate subroutine. For 0.1%-1% physical error rates and distance 29 and larger, PyMatching v2 is $> 100,000\times$ faster than a pure Python implementation that uses the exact reduction to MWPM. Compared to the local matching approximation of the MWPM decoder used in [107], PyMatching v2 has a similar empirical scaling but is around $100\times$ faster near threshold (error rates of 0.5% to 1%) and almost $1000\times$ faster below threshold ($p = 0.1\%$).

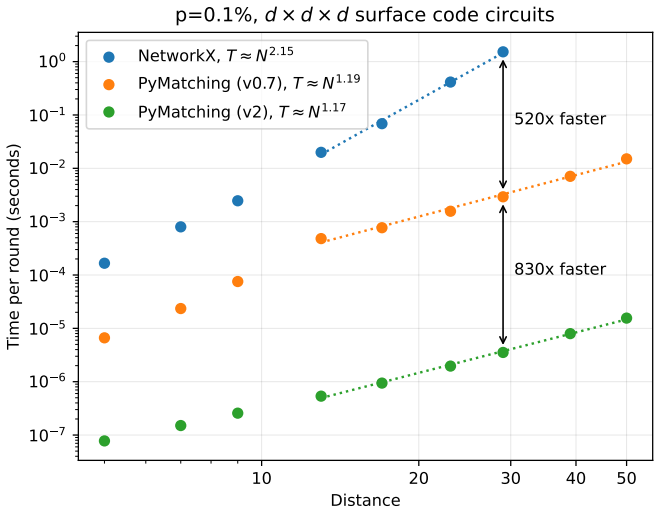



**Figure 2.8:** Decoding time per round for PyMatching v2 (our implementation of sparse blossom), compared to PyMatching v0.7 and a NetworkX implementation. For distance $d$, we find the time to decode $d$ rounds of a distance $d$ surface code circuit and divide this time by $d$ to obtain the time per round. We use a circuit-level depolarising noise model where the probability $p = 0.1\%$ sets the strength of two-qubit depolarising noise after each CNOT gate, the probability that each reset or measurement fails, as well as the strength of single-qubit depolarising noise applied before each round. The threshold for this noise model is around $0.71\%$. PyMatching v0.7 uses a C++ implementation of the local matching algorithm described in [107]. The pure Python NetworkX implementation first constructs a complete graph on the detection events, where each edge $(u, v)$ represents a shortest path between $u$ and $v$, and then uses the standard blossom algorithm on this graph to decode. All three decoders use a single core of an M1 Max processor.

We analysed the distribution of the running time per shot of PyMatching v2 for simulated surface code data, see Figure 2.11 and Figure 2.12. For example, for distance-17 surface code circuits with $p = 0.1\%$ circuit-level noise, we observe a mean running time of 0.62 microseconds per round and find that 97.4% of the million shots were decoded with a running time below 1 microsecond per round. We also plot the relative standard deviation $\sigma/\mu$ of the running time per shot in Figure 2.12 and find that $\sigma/\mu$ decreases as either the distance or error rate is increased.

We also compared the speed of PyMatching v0.7 with that of PyMatching v2 on experimental data, by running both decoders on the full dataset of Google's recent experiment demonstrating the suppression of quantum errors by scaling a surface code logical qubit from distance 3 to distance 5 [5, 185]. On an M2 chip, PyMatching

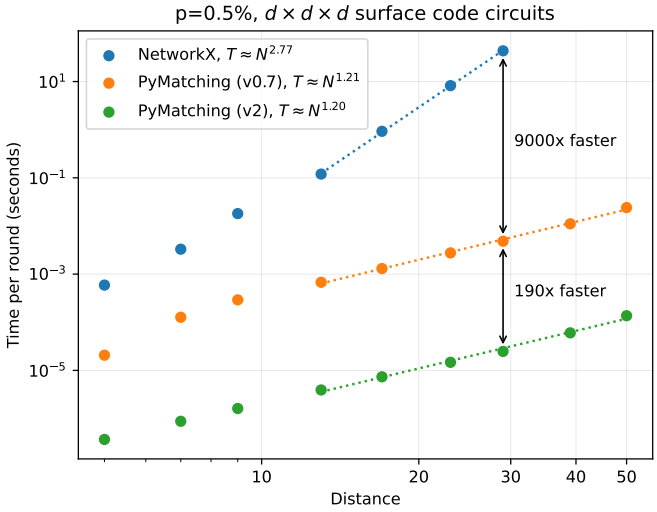



**Figure 2.9:** Decoding time per round for PyMatching v2 (our implementation of sparse blossom), compared to PyMatching v0.7 and a NetworkX implementation. The only difference with Figure 2.8 is that here we set $p = 0.5\%$.

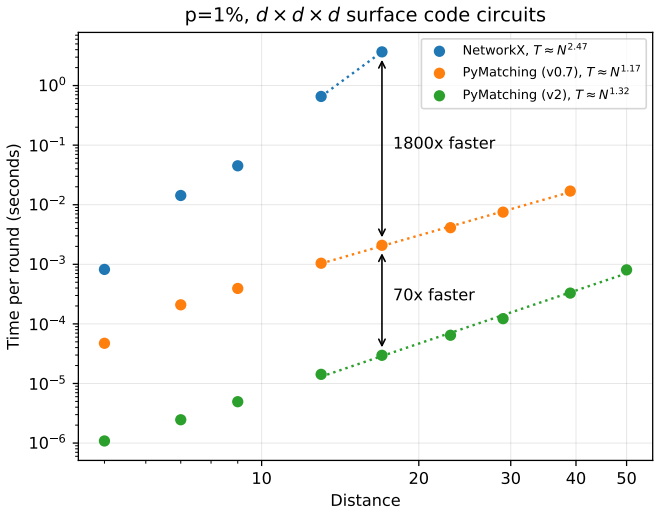
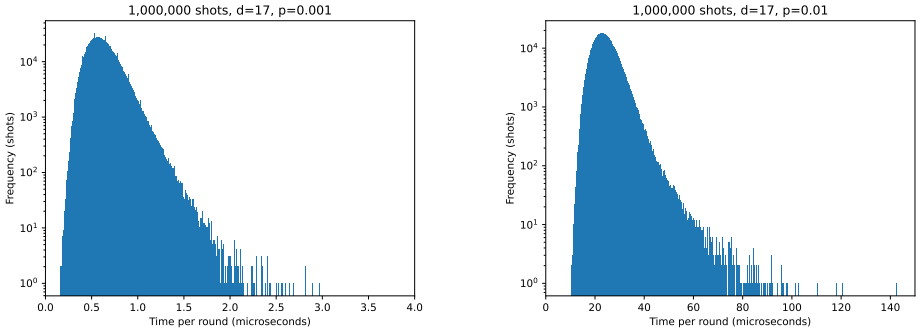



**Figure 2.10:** Decoding time per round for PyMatching v2 (our implementation of sparse blossom), compared to PyMatching v0.7 and a NetworkX implementation. The only difference with Figure 2.8 is that here we set $p = 1\%$.




**(a)** **(b)**

**Figure 2.11:** Histograms showing the distribution of running times of sparse blossom using 17-round, distance-17 surface code circuits and a standard circuit-level depolarising noise model. In (a) we use $p = 0.1\%$ and a histogram bin width of 0.01 microseconds. 97.4% of the shots have a running time per round below 1 microsecond. In (a) we instead use $p = 1\%$ and a histogram bin width of 0.2 microseconds.

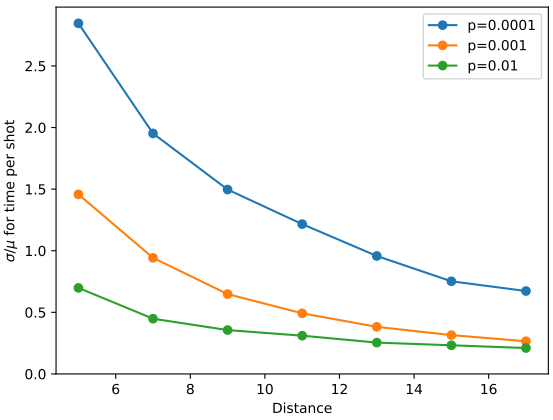



**Figure 2.12:** Relative standard deviation $\sigma/\mu$ of the time per shot for distance-$d$ surface code circuits (with $d$ rounds) and standard circuit-level depolarising noise. Here $\sigma$ and $\mu$ are the standard deviation and mean, respectively, of the time per shot, sampling 1 million shots for each data point.

v0.7 took 3 hours and 43 minutes to decode all 7 million shots in the dataset, whereas PyMatching v2 took 71 seconds.

## 2.6 Conclusion

In this chapter, we have introduced a variant of the blossom algorithm, which we call sparse blossom, that directly solves the minimum-weight perfect matching decoding problem relevant to error correction. Our approach avoids the computationally expensive all-to-all Dijkstra searches often used in implementations of the MWPM decoder, where a reduction to the traditional blossom algorithm is used. Our implementation, available in version 2 of the open-source PyMatching Python package, can process around a million errors per second on a single core. For a distance-17 surface code, it can decode both $X$ and $Z$ bases in under one microsecond per round of error correction, which matches the rate at which syndrome data is generated on a superconducting quantum computer.

Some of the techniques we have introduced can be directly applied to improve the performance of other decoders. For example, we introduced *compressed tracking*, which exploits the fact that the decoder only need to predict which *logical observables* were flipped, rather than the physical errors themselves. This allowed us to use a sparse representation of paths in the matching graph, storing only the endpoints of a path, along with the logical observables it flips (as a bitmask). We showed that compressed tracking can be used to significantly simplify the union-find decoder (see Appendix B.3), leading to a compressed representation of the disjoint-set data structure and eliminating the need to construct a spanning tree in the peeling step of the algorithm.

When used for error correction simulations, our implementation can be trivially parallelised across batches of shots. However, achieving the throughput necessary for real-time decoding at scale motivates the development of a parallelised implementation of sparse blossom. For example, for the practically relevant task of decoding a distance-30 surface code at 0.1% circuit-level noise, the throughput of sparse blossom is around 3.5× slower than the one microsecond per round throughput

desired for a superconducting quantum computer. It would therefore be interesting to investigate whether a multi-core CPU or FPGA-based implementation could achieve the throughput necessary for real-time decoding at scale by adapting techniques in [207] for sparse blossom. A shortcoming of MWPM decoders such as sparse blossom is that they do not exploit hyperedge errors (error mechanisms that flip more than two detectors) that arise in realistic noise models. However, in the next chapter we present an efficient decoder, belief-matching, that improves on the accuracy of MWPM decoders by taking advantage of these hyperedge error mechanisms, but still using a MWPM decoder as a subroutine.

## Chapter 3

# Belief-matching

*The source code for an implementation of the belief-matching decoder presented in this chapter can be found on GitHub at* https://github.com/oscarhiggott/BeliefMatching. *Belief-matching is also available as a Python 3 pypi package installed via "*`pip install beliefmatching`*".*

## 3.1 Introduction

Decoders for the surface code are usually either highly accurate [34, 10, 21] or efficient [64, 80, 60, 119, 113, 207, 141] but not both. Fast decoders for the surface code, including the minimum-weight perfect matching (MWPM) decoder [64, 80, 113, 207], presented in the previous section, and union-find (UF) [60, 119], ignore important error mechanisms that are common in experiments. For example, both MWPM and UF ignore the possibility of $Y$ errors, which cause four detection events in the bulk of the lattice, and therefore cannot be represented by a single edge in a matching graph (they would need to be represented as a *hyperedge*). For this reason, the matching graph is only an approximation of the full error model; if an $X$, $Y$ or $Z$ error each occur with probability $p$ at some location in the circuit, a matching graph approximation of the noise model is forced to model the $Y$ error as occurring with probability $O(p^2)$ (an $X$ and a $Z$ error). As a result of this approximation, MWPM and UF are not as accurate as some other decoders [34, 10, 21]. The decoders we introduce in this chapter exploit $Y$ errors (or, more generally, hyperedge errors) more effectively than prior work, including for circuit-level noise, while remaining

computationally efficient.

Several different approaches have been previously proposed for handling hyperedge error mechanisms more effectively than MWPM or UF [81, 34, 183, 62, 54, 11, 187, 148, 193, 15]. In Ref. [34], a tensor network decoder was introduced that approximates maximum-likelihood decoding for surface codes. However, this approach has high computational complexity and assumes error-free syndrome extraction circuits. In Ref. [193] a decoder was introduced for the surface code tailored to the case where hyperedge error mechanisms dominate over graphlike error mechanisms, finding improved thresholds relative to the MWPM decoder in this noise regime [193]. However, while the performance of the decoder is promising, it is not clear how well suited it is to other noise models, such as depolarising noise or general circuit-level errors in syndrome extraction circuits. In Ref. [54] BP was used, along with multi-path summation, to choose edge weights for a MWPM decoder, finding a threshold of 17.76% for the surface code with depolarising noise and perfect syndrome measurements. However, Ref. [54] did not consider how to generalise the method to handle noisy gates in the syndrome extraction circuit.

The decoders we introduce in this chapter use the belief propagation (BP) algorithm as a subroutine, which updates prior beliefs of the probability of error mechanisms in the circuit using an easily parallelisable message passing algorithm. Our use of BP enables us to exploit all the information present in circuit-level noise models more effectively, handling correlations between the $X$ and $Z$ decoding problems, and thereby achieving higher accuracy than MWPM or UF. While BP is effective at exploiting the full noise information, by itself BP often fails to converge to a valid solution. We show that by combining BP with MWPM or weighted UF we ensure convergence and make full use of all information about the noise model, boosting accuracy.

More precisely, whenever BP fails to converge, we use the updated beliefs output by BP to determine the edge weights in a matching graph. We then decode this re-weighted matching graph either: using MWPM, in which case we refer to the overall decoder as *belief-matching*; or instead using weighted union-find, in which

case we name the decoder *belief-find*. Belief-matching has conceptual similarities to the decoder proposed by Criger and Ashraf [54], which considered a toy noise model with perfect measurement results. A key difference of our approach is applicability to real experimental data and circuit-level noise simulations of experiments. We show that belief-matching and belief-find are the most accurate of all known computationally efficient decoders, and belief-find even has an almost-linear (worst-case) running time. Our numerical simulations show that the high accuracy of our decoders leads to an increase in the surface code threshold with circuit-level noise from 0.82% (for MWPM) to 0.94% (for belief-matching and belief-find). Our belief-matching decoder has also recently been shown to be highly accurate in real devices. In their recent QEC experiment demonstrating the suppression of quantum errors by scaling a surface code logical qubit [5], the Google Quantum AI team found that our belief-matching decoder was the only efficient decoder accurate enough to reduce the logical error rate as the system size was increased from distance 3 to 5.

The structure of this chapter is as follows. In Section 3.2 we review the belief propagation decoder and show how it can be used to decode circuit-level noise. In Section 3.3 we explain how hyperedge error mechanisms can be decomposed into graphlike error mechanisms (edges), before introducing our belief-matching and belief-find decoders in Section 3.4. Section 3.5 analyses the running time of our decoder. We present numerical simulations demonstrating the performance of our decoders in Section 3.6, before concluding in Section 3.7, where we also discuss possible future research directions. We also refer the reader to Ref. [114], where most of the original research described in this chapter was previously presented.

## 3.2 Belief propagation decoding of circuit-level noise

The improved accuracy of our decoder relies on the Tanner graph representation of the circuit-level noise model, which we defined in Section 1.4.4. Recall that the Tanner graph $\mathcal{T}(H) = (V, C, E)$ of a circuit is a bipartite graph, the biadjacency matrix of which is the detector check matrix $H$. Each variable node $v \in V$ corresponds to an error mechanism and each check node $c \in C$ corresponds to a detector. There is an

edge $(v,c) \in E$ if and only if the error mechanism corresponding to $v$ flips detector $c$. Here each error mechanism corresponds to some Pauli error that could occur at any location in the circuit (for example an $XY$ error occurring after a CNOT gate, one of the 15 error mechanisms in a two-qubit depolarising noise channel).

The Tanner graph representation of the classical binary linear codes is used by the belief propagation (BP) decoder, which has been hugely successful for decoding classical LDPC codes [144]. In this chapter we apply BP to the Tanner graph representation of circuit-level noise, which allows us to exploit knowledge of hyperedge error mechanisms that are ignored by MWPM.

BP is a message-passing algorithm, in which messages are passed along the edges of the Tanner graph. It takes as input the syndrome as well as a prior distribution, which associates a log-likelihood ratio (or prior probability) to each variable node, giving the probability that the associated error mechanism would be expected to flip. BP then estimates the marginal probability that each error mechanism has been flipped *given the syndrome*. The algorithm proceeds in iterations and in each iteration there is a "check-to-variable" step and a "variable-to-check" step. In the "check-to-variable" node step, each check node sends a message to its neighbouring variable nodes in the Tanner graph, whereas in the "variable-to-check" step each variable node sends a message to its neighbouring check nodes. After each iteration, the most recent messages can be used to estimate a marginal probability for each error mechanism. See Appendix C for more details on the belief propagation algorithm and its implementation.

If the Tanner graph is a tree, BP is guaranteed to converge to finding the correct marginal probabilities after a certain number of iterations. Unfortunately, Tanner graphs of both classical and quantum LDPC codes contain cycles, which degrade BP's performance. However, for classical LDPC codes, provided the girth of the Tanner graph is not too small (usually a girth of six is sufficient), the marginals output by BP are good enough to achieve good decoding performance.

Applying BP to Tanner graphs of quantum codes or circuits is more challenging for two main reasons [164, 8]. Firstly, due to the requirement that stabilisers must

commute, some noise models inevitably lead to ubiquitous 4-cycles in the Tanner graph. As an example, consider an $X$ stabiliser $S_X$ and a $Z$ stabiliser $S_Z$ with non-trivial support on some qubit $q_i$. Since $S_X$ and $S_Z$ commute, we know that they must also share support on at least one other qubit $q_j$. Now if we represent $Y$ errors as error mechanisms in the Tanner graph, we end up with a 4-cycle $(S_X, Y_{q_i}, S_Z, Y_{q_j}, S_X)$. We can sometimes avoid this problem by decoding $X$ and $Z$ errors separately, ignoring $Y$ errors. However, in this chapter we are specifically using BP to exploit the presence of $Y$ errors, so ignoring them would not be fruitful.

The second problem is that of degeneracy. There are usually many low-weight error mechanisms that neither flip any detectors *nor* flip any logical observables. By definition these are in the kernel of $H$ and $L$, usually corresponding to stabilisers (an example of such an error which is *not* a stabiliser is an $X$ error immediately prior to a transversal $\bar{X}$ measurement in a surface code memory experiment). This does not arise in classical codes, for which we have $L = I$. Consider one such error $\mathbf{b} \in \ker(H) \cap \ker(L)$, and note that $\mathbf{b}$ can have low weight much smaller than the distance, since it is in $\ker(L)$. If two possible errors $\mathbf{c}$ and $\mathbf{c} + \mathbf{b}$ are both (equally) highly probable, satisfying $H\mathbf{c} = H(\mathbf{c} + \mathbf{b}) = \mathbf{s}$, then BP can become *split* between the two potential solutions and unable to recover either of them. This is because BP is solving a *marginalisation* problem, and so if the posterior distribution is not peaked on a single solution with high probability but instead split between multiple solutions, the marginals may not provide enough information to pick any of the (equally valid) individual solutions.

As a result of these issues, we cannot hope to recover a threshold for the surface code with full circuit-level depolarising noise using BP alone. However, we will show that combining BP with the MWPM decoder leads to a very accurate decoder, which improves on both BP and MWPM individually.

## 3.3 Hyperedge error decomposition

An important concept we will use in belief-matching is hyperedge decomposition. For surface codes, we can always decompose each hyperedge error mechanism

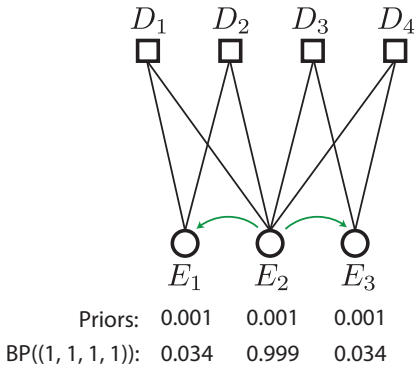



**Figure 3.1:** An example of a hyperedge $E_2 = \{D_1, D_2, D_3, D_4\}$ that can be decomposed into edges $E_1 = \{D_1, D_2\}$ and $E_3 = \{D_3, D_4\}$. Here, the green arrows denote this decomposition $E_2 = E_1 \wedge E_3$. This Tanner graph could be considered a subgraph of a much larger Tanner graph decribing a noisy stabiliser circuit (see Section 1.4.4 and Figure 1.9), where each error mechanism $E_i$ corresponds to a Pauli error (or set of indistinguishable Pauli errors) at some location in the circuit. Each prior $\mathbf{p}[i]$ is the probability of error mechanism $E_i$ given only the noise model. The row $\mathrm{BP}((1,1,1,1))$ gives the marginal probabilities output by BP given the noise model *and* the syndrome $(1,1,1,1)$.

$(D_t, D_u, D_v, D_w)$ into existing edges $(D_t, D_u)$ and $(D_v, D_w)$ in the matching graph (here $D_t$, $D_u$, $D_v$ and $D_w$ are detectors). This is because $Y$ components of errors can be decomposed into $X$ and $Z$ terms, which are always graphlike. For example, at some location in the circuit we may have a $Y$ error that decomposes into an $X$ error and a $Z$ error, or we could have an $XY$ error which decomposes into $XX$ and $IZ$. For the standard CSS surface code (or other CSS variants of the surface code such as subsystem surface codes or hyperbolic surface codes), we could use the Pauli-type to decompose $Y$ components into $X$ and $Z$ as just described. These decompositions provide a means by which information about hyperedge error mechanisms can be incorporated into edges in the matching graph, and therefore used by decoders such as MWPM or UF.

The concept of hyperedge error decomposition generalises to non-CSS codes, such as the XY [191] and XZZX [29] surface codes and Floquet codes [102], amongst others. We denote by $D(E_i)$ the set of detectors flipped by error mechanism $E_i$ and denote by $L(E_i)$ the set of logical operators flipped by $E_i$. We say that error mechanism $E_h$ can be decomposed into graphlike error mechanisms $\{E_h^1, E_h^2, \ldots, E_h^c\}$ provided

that $D(E_h) = \bigoplus_{i=1}^{c} D(E_h^i)$ and $L(E_h) = \bigoplus_{i=1}^{c} L(E_h^i)$, where here $\oplus$ denotes the symmetric distance of sets. We can write this decomposition as $E_h = E_h^1 \wedge E_h^2 \wedge \cdots \wedge E_h^c$. In general there is not always a unique decomposition of each hyperedge into edges and different choices can affect decoder performance. However, several heuristic methods for error decomposition have been developed by Gidney and incorporated into Stim, and these perform well in practice [91]. For our implementation of belief-matching, we use the hyperedge decomposition provided by Stim when constructing a detector error model using error decomposition. A slight technicality is that we merge repeated error mechanisms in the detector error model in Stim, and choosing one of the given decompositions for the merged error mechanism (since the decompositions provided by Stim may not be unique for the same repeated error mechanism). We give an example of a hyperedge decomposition in a Tanner graph in Figure 3.1.

## 3.4 Belief-matching and belief-find

Our belief-matching and belief-find decoders are given a prior distribution of the error model (an assignment of an independent error probability to each of the edge or hyperedge error mechanisms), as well as the observed syndrome from the implemented error correction circuit. Both decoders consist of two stages, illustrated in Figure 3.2 for the more simple case where syndrome measurements are perfect.

In the first stage, we use BP to estimate a posterior distribution of the error model, given the observed syndrome. More specifically, BP is run on the Tanner graph of the circuit, and estimates the marginal probability that each possible error mechanism in the noisy syndrome extraction circuit has occurred. Unlike a conventional MWPM or UF decoder, this stage uses knowledge of the full error model, including the hyperedge error mechanisms. However, as discussed in the previous section, BP is only able to approximate the posterior distribution and does not have a threshold if used on its own.

In the second stage, if BP fails to converge on its own, we use the posterior marginal probabilities estimated by BP to set the edge weights in a matching graph.

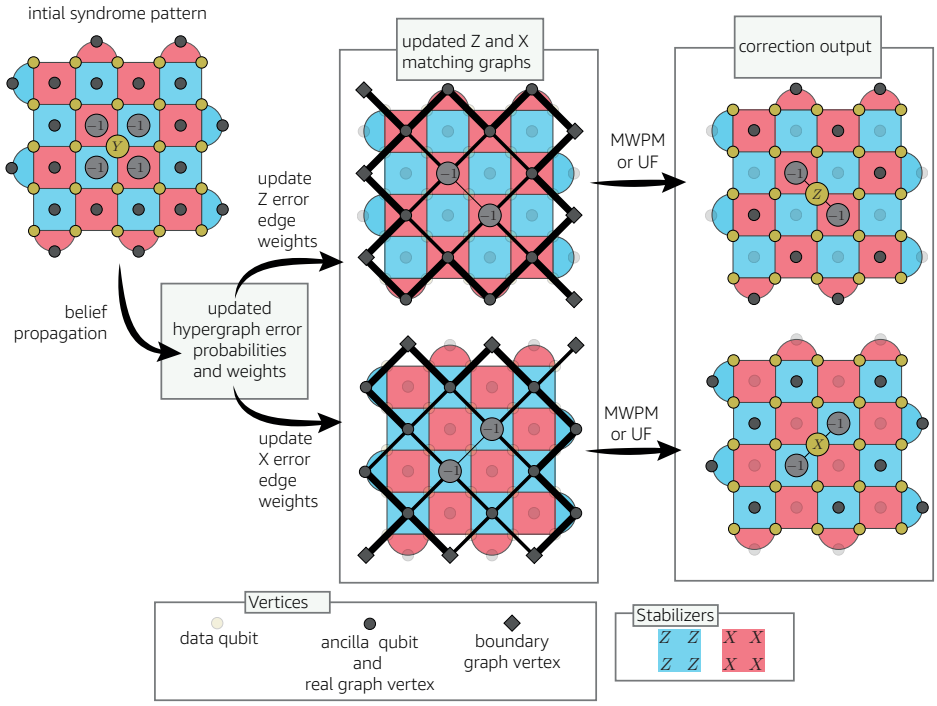



**Figure 3.2:** Illustration of belief-matching and belief-find. Given an observed syndrome and an error model, belief propagation is used to estimate the marginal probability that each error mechanism occurred. These updated error probabilities are used to set edge weights (here, thicker edges correspond to higher edge weights) in the $X$ and $Z$ matching graphs, which are then decoded with MWPM (for belief-matching) or weighted UF (for belief-find). In this figure we consider the decoding problem for perfect syndrome measurements for simplicity (as considered in Ref. [54]), however belief-matching and belief-find can also handle more complicated error models arising from measurements in the syndrome extraction circuit.

This contrasts to a standard MWPM or UF decoder, where the prior distribution is used to set edge weights instead. We set the edge weights from the BP posteriors using a decomposition of each hyperedge into edges. We add the posterior marginal probability of each hyperedge to the marginal probabilities of the edges in its decomposition when setting edge weights. After updating the edge weights, we decode the matching graph using MWPM [64] (for belief-matching) or weighted UF [60, 119] (for belief-find).

We now elaborate more on how we use the BP marginals to set the edge weights. Let $p_{\mathrm{BP}}(E_i)$ be the marginal posterior probability output by BP for error mechanism $E_i$ (an edge or hyperedge). For each graphlike error mechanism $E_i$, let $\gamma(E_i)$ be the set

of all error mechanisms (generally hyperedges) which have $E_i$ in their decomposition. For each edge, we define an adjusted probability[1]

$$p_{\text{adj}}(E_i) := \min\left[\left(p_{\text{BP}}(E_i) + \sum_{E_h \in \gamma(E_i)} p_{\text{BP}}(E_h)\right), 1\right] \tag{3.1}$$

and assign the weight

$$w(E_i) := -\log\big(p_{adj}(E_i)\big) \tag{3.2}$$

to each edge in the matching graph[2]. Note that we always ensure that each hyperedge has a unique decomposition into edges (if there exists more than one valid decomposition, then we pick one arbitrarily). We give pseudocode for belief-matching and belief-find in Algorithm 1.

**Algorithm 1** Belief-matching/ belief-find

**Require:** The circuit-level Tanner graph $\mathcal{T}(H)$, the priors $p_{\text{prior}}$ and the syndrome $\sigma$

**Ensure:** A correction operator, given as a set of error mechanisms corresponding to variable nodes in $\mathcal{T}(H)$ (columns of the detector check matrix $H$)

1: Compute the marginal posterior probability $p_{\text{BP}}(E_i)$ for each error mechanism $E_i$ by running BP, which takes $\mathcal{T}$, $p_{\text{prior}}$ and $\sigma$ as input.
2: Find a tentative correction $\mathbf{c}'$, which is the set of error mechanisms $E_i$ for which $p_{\text{BP}}(E_i) > 0.5$. We say that BP has *converged* if $\mathbf{c}'$ also has syndrome $\sigma$.
3: **if** BP has converged **then**
4: **return** The set of variable nodes $\mathbf{c}'$
5: **else**
6: Distribute the posterior $p_{\text{BP}}(E_h)$ of each error mechanism with a decomposition (generally a hyperedge) $E_h$ to the edges in its decomposition using Equation (3.1) and compute the edge weights in the matching graph $\mathcal{G}$ using Equation (3.2).
7: Decode $\mathcal{G}$ with syndrome $\sigma$ using MWPM (for belief-matching) or weighted UF (for belief-find), to find a set of graphlike error mechanisms $\mathbf{c}$
8: **return** The error mechanisms $\mathbf{c}$, which correspond to variable nodes in $\mathcal{T}$

[1] Another natural choice of edge weight would be to use $w(E_i) := \log\big((1-p_{adj}(E_i))/p_{adj}(E_i)\big)$ and then handle the negative weights with MWPM or weighted UF using the method in Appendix B.5. However, we find that our choice $w(E_i) := -\log\big(p_{adj}(E_i)\big)$ instead leads to slightly improved decoding performance for both belief-matching and belief-find. This could be due to BP giving overly confident marginals as a result of short loops in the Tanner graph.

[2] Alternatively we could treat $p_{\text{BP}}$ as probabilities of independent events (even though they are not), and take $p_{adj}(E_i)$ to be the probability that an odd number of faults in the set $\{E_h\} \cup \gamma(E_i)$ occurred under this assumption. This alternative approach would be consistent with how edges and hyperedges are usually merged in matching graphs.

## 3.5 Running time

We now consider the running time of belief-matching and belief-find. The worst-case running time of belief-find is almost-linear in the number of error mechanisms, since the weighted UF decoder has almost-linear worst-case running time [60, 119], and BP has linear running time. Furthermore, both weighted UF and the min-sum approximation of BP are comparatively simple decoding algorithms, which are amenable to implementation in hardware [55, 196]. For belief-matching, the worst-case running time is instead dominated by the MWPM step, which has worst-case running time $O(N^3 \log(N))$, where $N$ is the number of nodes in the matching graph [106]. However, the *expected* running time of MWPM has been shown to scale approximately linearly with the number of detection events when below threshold [80, 113, 207], and we have confirmed empirically that the expected running time of belief-matching is approximately linear in the number of error mechanisms in this regime when using sparse blossom for the MWPM subroutine [113]. Furthermore, our numerical results demonstrate that the decoding performance of belief-find is almost identical to that of belief-matching, despite having improved worst-case running time. The BP step, although linear time, can still be quite computationally intensive, since the number of edges in the circuit-level Tanner graph is a constant factor larger than the number of edges in the corresponding matching graph, and running time does not depend strongly on the weight of the syndrome (it is not necessarily faster at low $p$, unlike MWPM or weighted UF, since BP touches the whole Tanner graph). However, we expect these challenges to be overcome since BP is highly parallelisable, and very fast implementations are already widely used for decoding classical LDPC codes.

Since the advantage that belief-matching and belief-find offer over MWPM or weighted UF alone derives from their use of *hyperedges* present in the circuit-level Tanner graph, we expect them to outperform MWPM for most experimentally-relevant circuit-level noise models, for which the characterisation of hyperedge failure mechanisms is crucial to obtain good decoding performance [50].

## 3.6 Numerical simulations

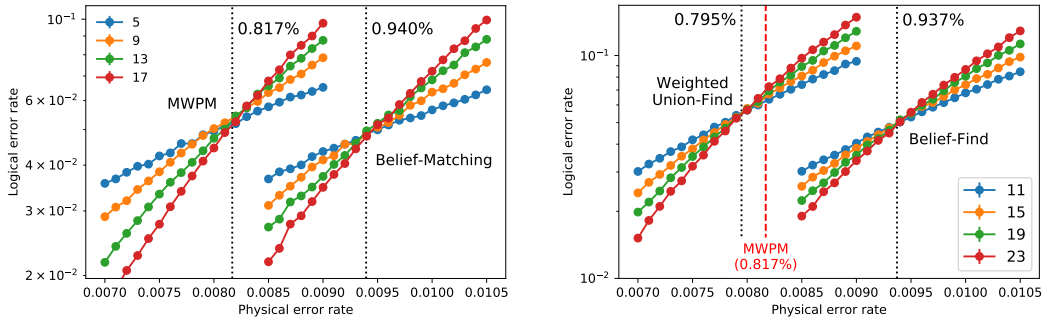
We compared the performance of belief-matching and belief-find to MWPM and union-find decoders through numerical simulations for the rotated surface code, using a standard circuit-level depolarising noise model.

### 3.6.1 Methods

Our noise model is parameterised by a noise strength and is defined as follows:

- Each CNOT gate is followed by a two-qubit depolarising channel of strenth $p$ (Equation (1.3))
- The outcome of each measurement is flipped with probability $2p/3$
- Initialisation in $|0\rangle$ is followed by an $X$ error with probability $2p/3$
- Initialisation in $|+\rangle$ is followed by a $Z$ error with probability $2p/3$
- A single-qubit depolarising noise channel of strength $p$ is applied to data qubits during measurement and reset of ancillas (Equation (1.2))

We used Stim to construct the detector error models, decompose hyperedges into edges and sample from the syndrome extraction circuits [91]. We used PyMatching to decode with MWPM. Our implementation of weighted UF is very similar to the version used in Ref. [161]. As in Ref. [161], we grow clusters on a *split-edge* graph, obtained from $\mathcal{G}$ by adding a node in the middle of each edge. We find that this modification significantly improves decoding performance. Additionally, in each round of growth, we grow smaller odd clusters before larger ones and fuse clusters at the endpoints of an edge (and update their parity) as soon as the edge becomes fully grown. This means we do not grow a cluster if its parity has already changed from odd to even earlier in the same round of growth (unlike in Algorithm 2 of Ref. [60]). Finally, we construct a spanning tree, not a minimum-weight spanning tree, in the peeling decoder stage of weighted UF (here we are consistent with Ref. [60] but not Ref. [119]). None of these modifications affect the asymptotic running time of the algorithm.




**(a)** **(b)**

**Figure 3.3:** Thresholds of our decoders for the rotated surface code using circuit-level depolarising noise. In each plot, the legend gives the code distance. (a) For MWPM and belief-matching we observe thresholds of 0.817(5)% and 0.940(3)%, respectively. (b) For weighted union-find and belief-find we find thresholds of 0.795(1)% and 0.937(2)%, respectively.

We estimate thresholds using the critical exponent method of Ref. [201], with $1\sigma$ uncertainties in the last digit (estimated using jackknife sampling over lattice sizes) given in parentheses.

### 3.6.2 Thresholds

In Figure 3.3a we show the performance of belief-matching for the rotated surface code for circuit-level depolarising noise, and compare its performance to that of a MWPM decoder. The MWPM decoder has previously had the highest reported circuit-level threshold for the surface code, which we find to be 0.817(5)% for our noise model. We find that belief-matching increases the threshold to 0.940(3)%, a $1.15\times$ improvement. This $1.15\times$ improvement can be attributed to belief-find taking advantage of correlations between the X and Z matching graphs due to Y errors.

Figure 3.3b shows thresholds for circuit-level depolarising noise using the weighted UF decoder, as well as belief-find. We find that belief-find also outperforms MWPM, achieving a threshold of 0.937(2)% despite having a worst-case running time almost-linear in $N$. We observe very little difference in decoding performance between weighted UF and MWPM alone, with weighted UF obtaining a threshold of 0.795(1)%, compared to a threshold of 0.817% for MWPM. Furthermore, there is no statistically significant difference between the 0.940(3)% threshold of belief-

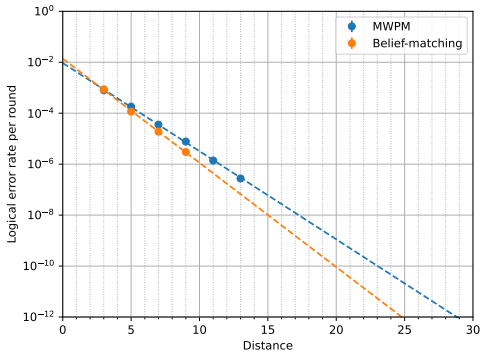



**Figure 3.4:** Logical error rate per round vs. code distance for MWPM and belief-matching, using circuit-level depolarising noise with a noise strength of $p = 0.2\%$. Error bars (95% Clopper-Pearson confidence intervals) are plotted but smaller than the marker size. The dashed lines are least-squares linear fits to the data (taking the logarithm of the logical error rate), extrapolated down to a logical error rate of $10^{-12}$.

matching (see Figure 3.3a) and the 0.937(2)% threshold of belief-find.

### 3.6.3 Qubit overhead below threshold

We also estimate the resources required to achieve a logical error rate of $10^{-12}$ below threshold using belief-matching and MWPM. In Figure 3.4 we plot the logical error rate per round vs. the code distance for MWPM and belief-matching, using $p = 0.2\%$ circuit-level depolarising noise. For a distance $d$ surface code, the logical error rate per round is estimated using a memory experiment with $2d$ rounds, and then dividing by the number of rounds. Using a least-squares fit to the data, we extrapolate down to $10^{-12}$ logical error rates, and estimate that a distance 29 rotated surface code (1,681 physical qubits) is required to survive a trillion rounds using MWPM, whereas only distance 25 (1,249 physical qubits) is needed for belief-matching. Therefore, in a practical regime where we might want to use the surface code for computation, belief-matching reduces the number of physical qubits required by around 25%, compared to the same setup using a MWPM decoder.

## 3.7 Conclusion

In this chapter we introduced new efficient decoders for the surface code, belief-matching and belief-find, which we showed have improved accuracy for decoding

circuit-level noise. Our decoders use knowledge of the full circuit-level noise model, i.e. they consider all possible error mechanisms in the circuit along with their associated error probabilities. By contrast, standard MWPM only considers error mechanisms that are "graphlike" (errors that flip one or two detectors). We therefore expect that belief-matching and belief-find will have good performance for a wide range experimentally-relevant noise models and can use noise models calibrated from experimental data [50]. Indeed, after the pre-print of the work in this chapter was released, our belief-matching decoder was used to experimentally demonstrate the suppression of quantum errors by scaling a surface code logical qubit from distance 3 to 5 [5]. In this surface code experiment, it was shown that belief-matching outperformed both MWPM [64, 80] and the correlated MWPM decoder of [81] for experimental noise [5]. The improved accuracy relative to the correlated MWPM decoder of Ref. [81] can be understood from the fact that belief-matching considers the full circuit-level noise model, whereas correlated MWPM considers each pair of correlated edges in isolation and only updates edge weights in close proximity to an initial (uncorrelated) MWPM solution. Our belief-find decoder has an almost-linear worst-case runtime while having very similar accuracy to belief-matching. This worst-case runtime is an improvement on the worst-case runtimes of MWPM and belief-matching, although these matching decoders can still have a linear expected runtime in practice at low error rates. Future work could explore implementations of belief-matching and belief-find in hardware [55, 196]. Belief-matching and belief-find can be applied to any code for which MWPM can be used, which includes 2D surface codes [64, 192, 29, 44, 67], subsystem surface codes [36, 112] and Floquet codes [102], amongst others [9, 138, 48]. Previous work developing decoders that handle hyperedge error mechanisms in the surface code have mostly assumed perfect syndrome measurements or a phenomenological error model [70, 69, 71, 204, 120, 26, 26, 54]. More generally, we have demonstrated how high performance decoders for classical LDPC codes (such as BP) can be applied directly to infer probable error locations in realistic circuit-level noise models, and expect that the techniques presented in this chapter will inspire the application of

similar techniques to other quantum error correction codes and protocols.

# Chapter 4

# Schedule-induced gauge fixing

In this chapter, we introduce new circuits and decoding methods for subsystem codes. Subsystem codes have usually had lower thresholds, an issue which can be attributed to their higher weight stabilisers. We introduce a technique for constructing syndrome measurement circuits and decoders that we show can lead to subsystem codes achieving high thresholds for realistic circuit-level noise models. The technique we introduce, called schedule-induced gauge fixing, improves the error correcting performance of a wide class of subsystem codes, especially under biased noise models. By changing the order in which check operators are measured, valuable additional information can be gained, and we introduce a new method for decoding which uses this information to improve performance. *Static* gauge-fixing has been used to improve decoding performance before in Ref. [138], where the Bacon-Shor code was used as a template to construct elongated compass codes, which can be tailored to biased noise models. However, this method requires changing interactions at the hardware level, as well as measuring high weight stabilisers directly, since elongated compass codes are not subsystem codes themselves. In contrast, our technique can be implemented entirely in software, and only requires measuring the low-weight gauge operators of the code. In essence, schedule-induced gauge fixing allows us to switch repeatedly between different codes such that more information can be inferred about potential errors.

In Section 4.1 we introduce schedule-induced gauge fixing (SIGF), our technique for improving the quantum error correcting performance of subsystem codes.

We then show how SIGF changes the structure of the matching graph in Section 4.2. We present numerical results in Section 4.3, where we apply SIGF to the subsystem toric code for depolarising and biased circuit-level noise, before concluding in Section 4.4. Most of the original research in this chapter has been previously published in Ref. [112].

## 4.1 Gauge-fixing schedules and circuits

We will now introduce some general techniques that improve the quantum error correcting performance of a wide class of subsystem codes. We will alter the stabiliser measurement procedure *in software*, in such a way that the individual gauge operator measurements themselves yield useful information. This use of individual gauge operator measurements is in contrast to prior work on decoding subsystem codes in the literature, where individual gauge operator measurements themselves are never treated as syndrome bits, and only their products (the stabilisers) are used for decoding.

While we will analyse these techniques numerically using the subsystem surface code (and for the subsystem hyperbolic codes introduced in Chapter 6), the key ideas can be applied to the vast majority of subsystem codes considered in the literature, for which stabiliser eigenvalues can be inferred by measuring gauge operators. In fact, these techniques address one of the main drawbacks of subsystem codes, which is that they typically have lower thresholds. Low thresholds arise partly because stabiliser eigenvalues are determined by combining the outcomes of many gauge operator measurements, each of which may be faulty, making their measurement less reliable. Additionally, these high weight stabilisers provide less information about which qubit has suffered an error, further reducing the threshold. The most dramatic example of this effect is the Bacon-Shor code [9] which, although it has weight-2 check operators, has no threshold without concatenation, as the stabilizer operators grow with system size (although it was recently shown that a threshold can be recovered using concatenation with geometrically local gates [93]). The techniques we introduce can also be used when applying logical operations with *subspace* codes,

as we explain in Appendix D.2.2, since lattice surgery and code deformation for surface codes can be interpreted as gauge fixing of a larger subsystem code [200].

We call the general method *schedule-induced gauge fixing*, since we will be altering the schedule of the stabiliser measurement circuits in such a way that gauge fixing can be used to significantly improve the error correcting performance when decoding. We will refer to it simply as gauge fixing when the meaning is clear from context.

Schedule-induced gauge fixing can be applied to a large class of subsystem codes, for which there are stabilisers $s$ that are the product of gauge operators, $s = g_0 g_1 \dots g_{m-1}, \quad g_i \in \mathcal{G} \setminus \mathcal{S}$. We call these gauge operators *gauge factors* $\mathcal{G}^s$ of $s$,

$$\mathcal{G}^s := \{g_0, \dots, g_{m-1} | g_i \in \mathcal{G} \setminus \mathcal{S}, s = g_0 g_1 \dots g_{m-1}\}, \tag{4.1}$$

and stabilisers which admit such a decomposition will be referred to as *composite stabilisers*. In general there can be more than one such decomposition for a given stabiliser, though we are typically most interested in the *minimum-weight decomposition*, where the average weight of gauge factors $g_i \in \mathcal{G} \setminus \mathcal{S}$ is minimised. For the codes we construct in this work there is a unique minimum-weight decomposition for each stabiliser, though in general there can be more than one [45]. For CSS subsystem stabiliser codes the gauge factors of each stabiliser mutually commute, and can be measured in any relative order. For more general subsystem codes, the order of measurements of gauge factors $g_0 g_1 \dots g_{m-1}$ of each stabiliser $s \in \mathcal{S}$ must be chosen such that each gauge factor measurement $g_i$ commutes with the product $g_0 g_1 \dots g_{i-1}$ of gauge factor measurements before it. In Ref. [181], this condition was shown to be both necessary and sufficient to guarantee that the stabiliser can indeed be recovered from the product of individual measurements. Schedule-induced gauge fixing will typically be most useful for subsystem codes which have at least one composite stabiliser, and for which the weight of each composite stabiliser is greater than the weight of each of its gauge factors. In the case of the subsystem codes studied in this work, the gauge factors of each $Z$ stabiliser associated with a face are the $Z$ triangle operators belonging to that face (and similarly for $X$ stabilisers

and $X$ triangle operators).

When decoding subsystem codes with existing methods, the syndrome used consists of eigenvalues of stabilisers. In other words, where a stabiliser is composite, measured by taking the product of the measurements of its $m$ gauge factors $g_i \in \mathcal{G}^s$, it is the product that is used for decoding, not the result of each gauge factor measurement individually. Therefore, for each stabiliser, we are measuring $m$ bits of information, and only using a single bit (their parity) for decoding. Or more generally, when repeating syndrome measurements and comparing consecutive rounds, each detector is formed as the parity of $2m$ bits. For the most simple stabiliser measurement schedules typically used, this is indeed all the useful information that can be used for decoding. This is because $\mathcal{G}$ is not abelian and, by definition, each gauge factor $g_i \in \mathcal{G}^s$ must anti-commute with at least one other gauge operator $h \in \mathcal{G}$. Once $h$ is measured, either $h$ or $-h$ becomes a stabiliser, and a subsequent measurement of $g_i$ will result in either 1 or $-1$ at random with $P(1) = P(-1) = 0.5$. Consider a schedule $W$ of measurements of check operators $K_0 K_1 \dots K_{N-1}$, chronological order from left to right, where each check operator $K_i$ is either a gauge factor or a stabiliser that is not composite, and where each $K_i$ is measured once. If this measurement schedule $W$ is simply repeated periodically, then every consecutive pair of measurements of any check operator $K_i$ will be separated by one measurement of every other check operator. As a result, if the check operators in $W$ generate $\mathcal{S}$ as required, every measurement of a gauge factor will give a random outcome and will not be useful for decoding, since its eigenvalue will not have been preserved between consecutive measurements. In fact, the eigenvalue of any product of check operators that is not in $\mathcal{S}$ will also not be preserved between consecutive measurements, following similar reasoning.

However, we can instead choose a measurement schedule $W$, again repeated periodically, where some gauge factors $g_i$ are measured multiple times within $W$, with no anti-commuting check operators measured between consecutive measurements of $g_i$ within $W$. In this case, the first measurement of $g_i$ in $W$ will have a uniformly random outcome $c \in \{\pm 1\}$, but will project the state into an eigenstate of the

generator $cg_i$ (the measurement outcome determines the phase of the generator). Borrowing terminology from Ref. [102], $cg_i$ becomes part of the *instantaneous stabiliser group* (ISG), i.e. a part of the stabiliser group of the state at that instant in the circuit, even though it is not in the stabiliser group of the subsystem code itself. The subsequent repeated measurements of $g_i$ will be deterministic, since $g_i$ is now in the ISG (it stabilises the state) and, provided no error has occurred, the measurement outcome will remain $c$.

### 4.1.1 Gauge-fixing for CSS codes

We will now restrict our attention to CSS subsystem codes, for which the gauge group $\mathcal{G}$ can be decomposed into a set of operators each in $\{I,X\}^n$, which we denote $\mathcal{G}_X$, and a set of operators each in $\{I,Z\}^n$, which we denote $\mathcal{G}_Z$, with $\mathcal{G} = \mathcal{G}_X \cup \mathcal{G}_Z$. The stabiliser group can similarly be decomposed into either $X$-type or $Z$-type Pauli operators. For CSS subsystem codes, the most common measurement schedule consists of alternating between measuring all $X$-type and all $Z$-type check operator measurements in a repeating sequence. In other words, the sequence of measurements for measuring the $X$ or $Z$ stabilisers is of the form $(ZX)^r$, where $2r$ is the number of rounds of stabiliser measurements, and the chronological order is from left to right. We call such a sequence of measurements a *homogeneous* schedule, since all stabilisers of the same Pauli-type are given identical measurement schedules. Equivalently, for the subsystem codes we construct, a homogeneous schedule assigns the same schedule to each face of the lattice from which it is derived. We will sometimes denote a schedule just by its longest repeating subsequence if the number of repetitions is not relevant (i.e. denoting the above schedule by $ZX$ rather than $(ZX)^r$).

For the $ZX$ schedule, each $X$ gauge operator measurement comes directly after the measurement of a $Z$ gauge operator that it anti-commutes with (and vice versa), and so the outcome of each individual gauge operator measurement is random. However, by repeating $X$ or $Z$ check operator measurements we can temporarily *fix* some gauge operators as stabilisers. As an example, consider a homogeneous schedule of the form $(Z^2X^2)^r$ and some $X$-type gauge operator $g_j$. The first measurement of $g_j$

in each consecutive pair of $X$ gauge operator measurements will be random, but will project the state into an eigenstate of $g_j$ or $-g_j$. The second measurement will be deterministic and have the same measurement outcome as the first. Clearly the same is true for the first and second $Z$ gauge operator measurement outcomes.

### 4.1.2 Homogeneous stabiliser measurement circuits

In order to measure the triangle operators (and therefore stabilisers) of the subsystem surface code, we require a circuit to measure each triangle operator using an ancilla qubit. We will now show how these circuits can be constructed for *homogeneous schedules*, where the same schedule is applied to each face in the lattice. In each face of the subsystem toric code Each triangle operator consists of three data qubits and at least one ancilla, and can be measured using three CNOT gates, along with state preparation and measurement of an ancilla. A time step is defined as the time taken for a CNOT gate, and we assume that state preparation and measurement combined take a single time step. This assumption is similar to the assumption of non-demolition measurements in Refs. [202, 83], except we will assume both state preparation and measurement errors, rather than just the latter. In Ref. [36] the authors instead assume that state preparation and measurement each take a time step, and use an additional ancilla to parallelise state preparation and measurement into a single time step. The parity check measurement circuit therefore takes four time steps.

The measurement schedules we use are shown in Figure 4.1. The schedule shown on the left of Figure 4.1 is for alternating measurement of the Pauli-$Z$ and Pauli-$X$ operators ($ZX$ schedule), and is the same as that used in Ref. [36]. The right hand diagram in Figure 4.1 shows the schedule for measuring $ZZ$ (blue labels) as well as the schedule for measuring $XX$ (red labels). All three of these schedules have period 4, and so the time steps which each gate is labelled with are given modulo 4. Note that the first half of the $ZZ$ schedule matches the $Z$ component of the $ZX$ schedule, and the first half of the $XX$ schedule matches the $X$ component of the $ZX$ schedule. Therefore, the schedule for *any* homogeneous sequence can be implemented by concatenating these three schedules (or subsets of them). For the

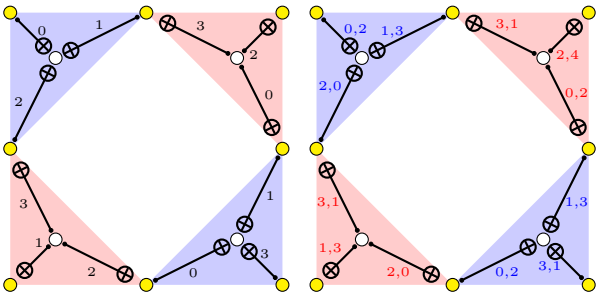



**Figure 4.1:** Parity check measurement schedule for the subsystem surface code using a homogeneous $(ZX)^r$ sequence (left) [36], a homogeneous $Z^r$ sequence (right, blue text) and a homogeneous $X^r$ sequence (right, red text). CNOT gates are labelled with the time step(s) they are applied in, which are given modulo 4, since all schedules have period 4.

standard $ZX$ schedule, we need only a single ancilla qubit for each triangle operator. For schedules which contain $ZZ$, we use two ancillas per $Z$ triangle operator to parallelise consecutive triangle operator measurements, and similarly we use two ancillas per $X$ triangle operator for parallelised schedules containing $XX$.

Each individual fault in the measurement circuit results in at most a single data qubit error, a property that is made possible by the weight-three gauge operators (see Ref. [36]). As a result of this bare-ancilla fault tolerance of the measurement circuits, we can correct up to the full code distance for all the codes we have constructed.

## 4.2 Matching graph using gauge-fixing

We now show how this additional gauge operator information can be used when decoding a CSS subsystem code using the MWPM decoder, which introduces the additional requirement that the code must have no more than two stabilisers of a given Pauli type acting non-trivially on each qubit. Subsystem codes which satisfy these properties include the subsystem surface code [36], the Bacon-Shor code [9, 6], and some 2D compass codes [138], including heavy-hexagon codes [48].

As an example, let us first consider the 2D matching graphs of the subsystem toric code, assuming perfect stabiliser measurements. Each node in the $X$-type (or $Z$-type) matching graph corresponds to an $X$ (or $Z$) stabiliser, and each edge corresponds to a $Z$ (or $X$) error on a qubit. For the stabiliser group of the subsystem toric code with no gauge operators fixed, both the $X$-type and $Z$-type matching

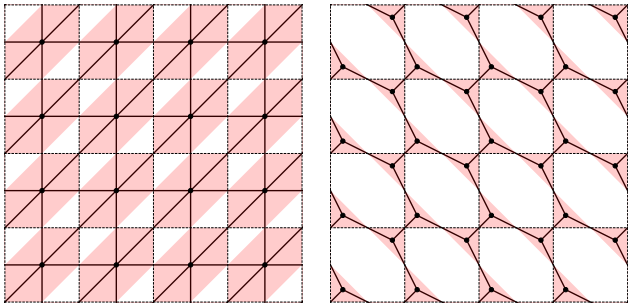


**Figure 4.2:** Matching graphs ($X$-type) for the subsystem toric code with no triangle operators fixed as stabilisers (left) and all triangle operators fixed as stabilisers (right).

graphs are triangular lattices, as shown in Figure 4.2 (left) for the $X$-type matching graph. This triangular lattice matching graph has a MWPM threshold of 6.5% with perfect measurements [86]. However, once we have measured all the $X$-type gauge operators, they become gauge-fixed as stabilisers (up to signs that can be accounted for in software), and the stabiliser group we obtain is that of the hexagonal toric code [86]. The new associated $X$-type matching graph instead has an improved MWPM threshold with perfect measurements of 15.6% [86], exceeding that of the toric code on a square lattice of 10.3% [64]. If we measure all the $Z$-type gauge operators, we instead obtain the dual of the hexagonal toric code, and now the $Z$-type matching graph is a hexagonal lattice.

When using the standard $ZX$ schedule for the subsystem toric code, the stabiliser group is indeed constantly switching (up to signs) between the hexagonal toric code and its dual, both abelian subgroups of the gauge group $\mathcal{G}$. However, each gauge operator is only ever fixed immediately after it is measured, and is randomised by the time the same gauge operator is next measured, since an anti-commuting gauge operator of the opposite Pauli-type is measured in between these consecutive measurements of the same gauge operator. However, by making more than one consecutive measurement of gauge operators of a given Pauli type, we will now show that we can gauge fix into the hexagonal toric code (and its dual) for longer durations, thereby making more valuable use of the individual gauge operator outcomes themselves.

The matching graph for these circuits are three-dimensional, as for the surface

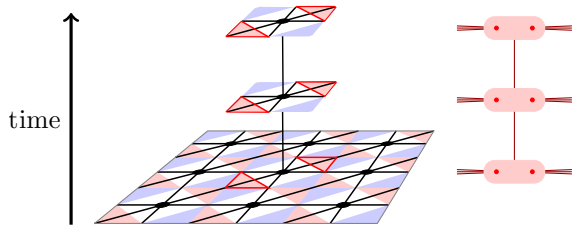



**Figure 4.3:** The 3D $X$-type matching graph for the subsystem toric code. Left: We show three time steps of the 3D matching graph for a single $X$ stabiliser (highlighted in red), with black lines denoting edges. Right: We also use this more simple, abstract notation to depict the same 3D matching graph in our work, restricted to a single face of the lattice. Here, each pale red rounded rectangle corresponds to an $X$ stabiliser in one of three consecutive time steps in the matching graph. Red dots denote $X$ triangle operator measurements (two of which within a face form a stabiliser), and red lines denote edges in the 3D matching graph.

code. Each node in the matching graph corresponds to a detector, and each edge $(u,v)$ corresponds to an error mechanism that can occur, creating a detection event at nodes $u$ and $v$. In order to handle measurement errors, each stabiliser measurement is repeated $T \geq L$ times [64], and in the bulk each detector is the parity of two consecutive measurements of a stabiliser. Measurement errors correspond to time-like edges, and memory (data qubit) errors correspond to space-like edges. There are also single circuit faults that can induce *diagonal* edges, which have nodes that differ in both space and time. We can label each detector (and its node in the matching graph) with a coordinate $(s,t)$, where $t$ is the time step and $s = g_0 \dots g_{m-1}$ denotes the stabiliser using its gauge factors $g_i \in \mathcal{G}^s$. If we denote by $M(g_i,t) \in \{0,1\}$ the measurement bit corresponding to the measurement of $g_i$ in time step $t$, then the detector $D(s,t)$ with coordinate $(s,t)$ is the parity of measurement bits

$$D(s,t) \coloneqq \bigoplus_{g_i \in \mathcal{G}^s,\ t' \in \{t, t_{prev}\}} M(g_i,t') \tag{4.2}$$

where here $t_{prev}$ is the time step of the most recent previous measurement of $s$ (here we assume for simplicity that all gauge factors of a stabiliser are measured together in the same time step). We depict the 3D matching graph for the subsystem toric code in Figure 4.3.

For the $ZX$ schedule used in the previous literature, gauge operators are never

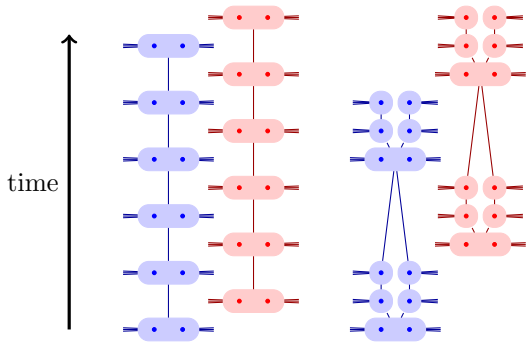



**Figure 4.4:** Matching graph for a single face of the subsystem toric code using a homogeneous $(ZX)^6$ schedule (left) and a homogeneous $(Z^3X^3)^2$ schedule (right). The vertical axis corresponds to time, with the direction of time being from bottom to top. Small blue and red filled circles correspond to $Z$ and $X$ gauge operator measurements respectively, with each vertical column of small filled circles corresponding to a single gauge operator. Large light blue and light red filled rounded rectangles (or rounded squares) correspond to stabilisers, being the product of the gauge operators they enclose. Diagonal edges (between stabilisers that differ in space and time) have been omitted for clarity. Blue and red lines correspond to edges in the $Z$ and $X$ matching graphs, respectively.

fixed and stabilisers are always the product of gauge operators, whereas for many of the schedules we use, we can fix a subset of the gauge operator measurements, and obtain (temporarily) stabilisers consisting of single gauge operators. A gauge operator $g$ is fixed as a stabiliser if no gauge operator $h$ which anti-commutes with $g$ has been measured since the last measurement of $g$. The matching graphs for the schedules $(ZX)^6$ and $(Z^3X^3)^2$ are shown in Figure 4.4. For the $(ZX)^6$ schedule, gauge operators can never be fixed as stabilisers, whereas for the $(Z^3X^3)^2$ schedule, two-thirds of the gauge operator measurements can be fixed as stabiliser measurements, and so the parity of two consecutive individual gauge operator measurements can be used to form a detector. Since each gauge operator has weight 3, by fixing some gauge operators as stabilisers, we are temporarily reducing the weight of some stabiliser measurements from 6 down to 3.

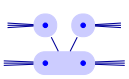
### 4.2.1 Vertex splitting and merging

Since the stabilisers can change between consecutive time steps when using gauge fixing, we must generalise our definitions of the detectors usually used for the subsystem surface code. We still form a detector as the parity of two consecutive

measurements of a stabiliser, but these stabilisers can change over time as gauge operators become gauge-fixed (temporarily belong to the stabiliser group). With our generalised definition of the detectors, a detector corresponding to stabiliser $s$ is flipped in time step $t$ if the eigenvalue of $s$ differs from that of the same product of gauge operators in time step $t-1$. There is a vertical edge in the matching graph between a detector measurement $s_t$ in time step $t$ and measurement $s_{t-1}$ in time step $t-1$ if $s_t$ and $s_{t-1}$ have at least one gauge factor in common. This vertical edge corresponds to a measurement error on one of the gauge factors that $s_t$ and $s_{t-1}$ have in common.

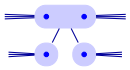
As an example we will now consider the case where a stabiliser has two gauge factors, as is the case for the subsystem toric code. Suppose we have two gauge operators $g_0$ and $g_1$ that factorise some stabiliser of the subsystem code. Furthermore, suppose we have some other gauge operator $g_2$ that anti-commutes with $g_0$ and $g_1$ individually, but by definition commutes with their product $g_0g_1$ (since $g_0g_1$ is a stabiliser of the subsystem code). For example, $g_0$ and $g_1$ could be $X$ triangle operators and $g_2$ a $Z$ triangle operator, all in the same face. Consider the sequence of measurements: $M(g_0,t-3)$, $M(g_1,t-3)$, $M(g_2,t-2)$, $M(g_0,t-1)$, $M(g_1,t-1)$, $M(g_0,t)$, $M(g_1,t)$. Just before the measurements of time step $t-1$, we know that $g_0g_1$ is in the ISG, but $g_0$ and $g_1$ are not in the ISG individually. Following Equation (4.2) we can therefore define a detector $D(g_0g_1,t-1) \coloneqq M(g_0,t-1) \oplus M(g_1,t-1) \oplus M(g_0,t-3) \oplus M(g_1,t-3)$ (as usually done for the subsystem toric code with a normal schedule). However, just before time step $t$, we know that $g_0$ and $g_1$ are individually in the ISG. We can therefore define detectors $D(g_0,t) \coloneqq M(g_0,t) \oplus M(g_0,t-1)$ and $D(g_1,t) \coloneqq M(g_1,t) \oplus M(g_1,t-1)$. In the absence of errors, all three detectors we have defined will be deterministic, as required.

We now consider the matching graph with these three detectors as nodes, and with spacelike and timelike edges corresponding to data qubit errors and measurement errors respectively. The stabiliser node is *split* into two nodes in time step $t$, with the matching graph locally looking like (with time propagating upwards):



and a measurement error in time step $t-1$ for $g_0$ will flip detectors $D(g_0g_1,t-1)$ and $D(g_0,t)$, which corresponds to flipping the vertical edge $((g_0g_1,t-1),(g_0,t))$ in this diagram. The same argument holds for a measurement error on $g_1$ in time step $t-1$ corresponding to flipping the other vertical edge $((g_0g_1,t-1),(g_1,t))$.

Similarly, we can instead have the sequence: $M(g_0,t-3)$, $M(g_1,t-3)$, $M(g_0,t-2)$, $M(g_1,t-2)$, $M(g_2,t-1)$, $M(g_0,t)$, $M(g_1,t)$. Here the gauge operators which were fixed at time $t-2$ are no longer fixed after $g_2$ is measured at time $t-1$. Defining detectors $D(g_0,t-2) \coloneqq M(g_0,t-3) \oplus M(g_0,t-2)$, $D(g_1,t-2) \coloneqq M(g_1,t-3) \oplus M(g_1,t-2)$ and $D(g_0g_1,t) \coloneqq M(g_0g_1,t) \oplus M(g_0g_1,t-2)$ leads to a matching graph which instead locally looks like:



and we find that a measurement error that occurs during the measurement of $M(g_0,t-2)$ flips detector $D(g_0,t-2)$ and detector $D(g_0g_1,t)$ corresponding to flipping the vertical edge $((g_0,t-2),(g_0g_1,t))$. While, in this example, we have considered stabilisers which have only two gauge factors (which is the case for subsystem toric codes), the definition of the detectors generalises straightforwardly to stabilisers with any number $m$ of gauge factor. For example, we have $m=4$ for the $\{8,4\}$ subsystem hyperbolic codes considered in Chapter 6, since these have four triangle operators (gauge factors) in each face of the lattice.

In a measurement round in which all gauge operators are fixed (matching graph nodes are split), there are two distinct advantages which gauge fixing can offer. Firstly, vertical time-like edges have a lower error probability, since they can only by flipped by a single gauge operator being measured incorrectly, rather than *any* of the gauge factors in a stabiliser being measured incorrectly, as would otherwise be the case. Secondly, the change in the structure of the matching graph leads to the average degree of the nodes being reduced.

The advantage that this can offer becomes clear when we again consider the (space-like) matching graph of the subsystem surface code when all gauge operators are fixed, compared to the matching graph when they are not fixed. We have found that the hexagonal lattice matching graph when gauge operators are fixed (Figure 4.2,

right) has a threshold of around 4.1% under a phenomenological noise model. On the other hand, for the triangular lattice matching graph when no gauge operators are fixed (Figure 4.2, left) we find a threshold of 2.0% with a phenomenological noise model (see Figure D.9). Furthermore, the outcomes of the weight-three checks are more reliable, since their measurement circuits are shorter. However, a potential disadvantage of gauge fixing is that by repeating $X$ checks, more errors accumulate for the next measurement of $Z$ checks, for which $Z$ gauge operators cannot be fixed. We will show in Section 4.3.1 how this trade-off leads to an optimal homogeneous schedule for the threshold under a circuit-level depolarising noise model.

## 4.3 Numerical simulations

For all of the numerical results in this section, we used a local variant of the minimum-weight perfect matching (MWPM) decoder, a pre-release of the first version of PyMatching [107] which used the Blossom V implementation of the Blossom algorithm [73, 131]. Note that this variant of the MWPM decoder has been superceded by Sparse Blossom, described in Chapter 2, which was released in PyMatching Version 2 [113]. We describe some of the methods we used to perform the simulations, such as the details of the noise model and the construction of the matching graph, in Appendix D.1.

### 4.3.1 Gauge-fixing for depolarising noise

We will now show how gauge-fixing can be used to improve the quantum error correcting performance of the subsystem toric code under a depolarising noise model. For this unbiased noise, we have used *balanced* schedules, which we define to be of the form $Z^aX^a$ for some $a \in \mathbb{Z}^+$. We find that schedules that allow gauge-fixing increase the threshold from $0.666(1)\%$ for the standard $(ZX)^r$ schedule used in Ref. [36] to $0.811(2)\%$ for the $(Z^4X^4)^r$ schedule, where gauge operators are fixed for three in every four rounds of measurements. In Figure 4.5, we show the thresholds for the $ZX$, $Z^2X^2$ and $Z^3X^3$ schedules. We see that both the $Z^2X^2$ and $Z^3X^3$ schedules are higher than the standard $ZX$ schedule, but the crossing is at a higher logical error rate. For these balanced schedules $(Z^aX^a)^r$ under depolarising

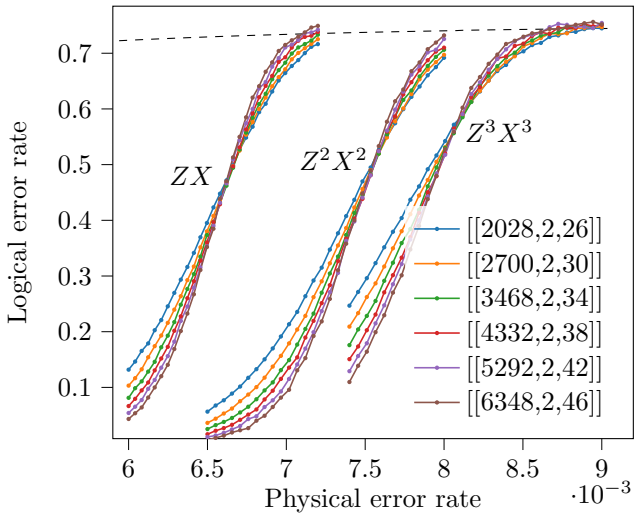



**Figure 4.5:** Threshold plots for subsystem toric codes using a $(ZX)^{92}$ schedule (left), $(Z^2X^2)^{46}$ schedule (middle) and $(Z^3X^3)^{31}$ schedule (right) using a depolarising noise model.

noise, we find that $a = 4$ is optimal (see Table D.4). Therefore, schedule-induced gauge fixing makes the threshold of the subsystem toric code under depolarising noise much more competitive with the rotated surface code, which we find has a threshold of around 0.97% under the same noise model and assumptions (state preparation and measurement each take half the time of a CNOT, and the logical error rate per time step is used). However, in Section 4.3.2 we show that schedule-induced gauge fixing with the subsystem toric code can be used to outperform the rotated surface code for small finite bias $\eta > 2.3$.

By using gauge fixing (setting $a > 1$) we reduce the average stabiliser weight in the 3D matching graph, since the stabilisers introduced from gauge fixing have weight 3. The mean stabiliser weight in the 3D ($X$ check) matching graph for a $\{2c, 4\}$ subsystem surface or hyperbolic code using a $(Z^qX^a)^r$ schedule (for any $q \geq 1$ or $r \geq 1$) is given by $3ca/(c(a-1)+1)$. So for the subsystem toric code ($c$=2), the mean stabiliser weights for the $(ZX)^r$, $(Z^2X^2)^r$ and $(Z^3X^3)^r$ schedules are 6, 4 and 3.6 respectively. We also reduce the average degree of nodes in the matching graph. For $a = 1$ the mean node degree is 14, whereas for $a > 1$, the mean node degree is $8ca/(c(a-1)+1)$, and so the $(ZX)^r$, $(Z^2X^2)^r$ and $(Z^3X^3)^r$ schedules have mean node degrees of 14, 32/3 and 9.6 respectively for the subsystem toric

| Schedule | $\lvert\bar{s}\rvert$ | $\lvert s\rvert_{\max}$ | $\lvert s\rvert_{\min}$ | $\bar{d}$ | $\Delta$ | $\delta$ |
|---|---|---|---|---|---|---|
| $Z^qX$ | 6 | 6 | 6 | 14 | 14 | 14 |
| $Z^qX^2$ | 4 | 6 | 3 | 10.67 | 16 | 8 |
| $Z^qX^3$ | 3.6 | 6 | 3 | 9.6 | 16 | 8 |
| $Z^qX^5$ | 3.33 | 6 | 3 | 8.89 | 16 | 8 |
| $Z^qX^{10}$ | 3.16 | 6 | 3 | 8.42 | 16 | 8 |

**Table 4.1:** The mean $\lvert\bar{s}\rvert$, maximum $\lvert s\rvert_{\max}$ and minimum $\lvert s\rvert_{\min}$ stabiliser weight and mean $\bar{d}$, maximum $\Delta$, and minimum $\delta$ degree of the $X$-check 3D matching graphs for various homogeneous schedules with the subsystem toric code.

code. More properties of matching graphs for some homogeneous schedules with the subsystem toric code are given in Table 4.1.

While we expect that reducing the average stabiliser weight and node degree in the matching graph should improve the threshold, increasing $a$ in balanced $Z^aX^a$ schedules also alters the edge fault probabilities. In time steps where gauge operators are fixed, $r_Z = 0$ in Table D.1, reducing the edge weights for some edges of type 0, 1 and 2. However, in the time steps where gauge operators are not fixed, $r_Z = a$, and so increasing $a$ also increases the edge-fault probability for these edges of type 0, 1 and 2. Therefore, increasing $a$ increases the proportion of time steps where a space-like slice of the matching graph is a degree-3 hexagonal lattice with small edge fault probabilites, but also *increases* the edge fault probabilities for the remaining time steps where the matching graph is *not* fixed, and is instead a degree-6 triangular lattice. There is therefore a trade-off between increasing the edge weights, and decreasing the stabiliser weights and node degrees, and the $a = 4$ schedule is the optimal compromise for schedules of the form $(Z^aX^a)^r$ for a circuit-level depolarising noise model.

Since changing the schedule alters both the matching graph via gauge fixing, as well as the edge fault probabilities, we can better understand how these two factors contribute to performance by studying them separately. In Figure 4.6 we plot the threshold as a function of $a$ for balanced schedules $Z^aX^a$ both with and without using gauge fixing. The thresholds that do not use gauge fixing are decoded by always merging gauge factors of a stabiliser into a single node in the matching graph, even in time steps where they could be split (gauge factors fixed) using the techniques

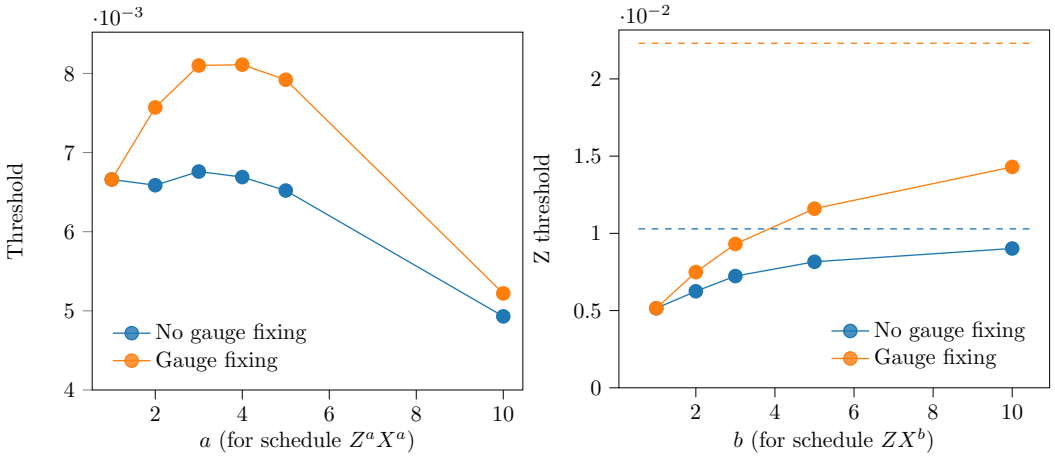



**Figure 4.6:** Left: Circuit level depolarising threshold as a function $a$ for schedules of the form $Z^aX^a$, with and without gauge fixing. Right: $Z$ thresholds as a function of $b$ for schedules of the form $ZX^b$, both with (orange) and without (blue) gauge fixing, using a circuit-level independent noise model. The orange and blue dashed lines are the threshold achievable under infinite bias (using an $X$ schedule) with and without gauge fixing respectively. Error bars are smaller than the marker size and have been omitted for clarity.

we have introduced. We see that for schedules that do not use gauge fixing, there is almost no improvement for $a > 1$, with performance degrading for $a > 4$. This demonstrates that almost all the improvement in threshold for depolarising noise is due to the use of gauge fixing, rather than the change in the noise model induced by the different schedule alone.

### 4.3.2 Tailoring the 3D matching graph to biased noise using gauge fixing

By using *unbalanced* schedules, where $X$ check operators are measured more frequently than $Z$ check operators (or vice versa), we can use gauge fixing to improve performance under biased noise models. Since we correct $X$ errors and $Z$ errors independently, we can define the $Z$ threshold $p_Z^{th}$ and $X$ threshold $p_X^{th}$ as the threshold for only $Z$-type or only $X$-type errors respectively. In Figure 4.7 we plot the $Z$ threshold for the unbalanced $ZX$, $ZX^2$, $ZX^{10}$ and $X$ schedules, under the independent circuit-level noise model. Increasing the ratio of $X$ checks to $Z$ checks significantly increases the $Z$ threshold from 0.52% for the $ZX$ schedule up to 2.22% for the $X$ schedule, which sets an upper bound.

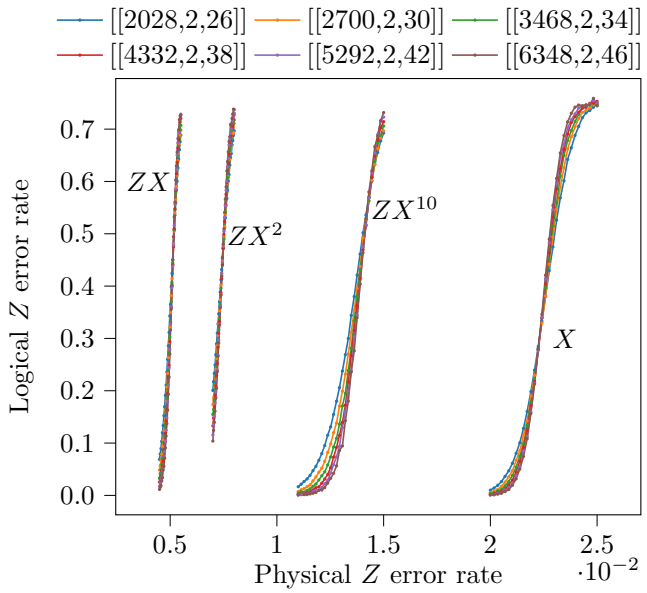



**Figure 4.7:** $Z$ thresholds for unbalanced schedules of the form $ZX^b$, as well as an $X$ schedule, which gives an upper bound on the $Z$ threshold achievable using unbalanced schedules.

By measuring $X$ checks more frequently, we also reduce the noise on data qubits caused by the CNOT gates used to measure $Z$ checks. To determine how much of the improvement in threshold comes from this reduced noise in the measurement schedule compared to the use of gauge fixing in the matching graph, we determine the thresholds both with and without using gauge fixing in Figure 4.6. We see that even without using gauge fixing, increasing the ratio of $X$ checks to $Z$ checks increases the $Z$ threshold, as expected. However, gauge fixing significantly boosts the $Z$ threshold further, and even a $ZX^5$ schedule using gauge fixing outperforms the best achievable $Z$ threshold without gauge fixing (using the $X$ schedule).

However, by increasing the ratio of $X$ to $Z$ checks, we also reduce the $X$ threshold of the code, which we must take into account when determining the total threshold under biased noise models. We now ask what the threshold is under the biased independent circuit-level noise model described in Section D.1.2, with bias parameter $\eta$. Specifically, for a given $\eta$, we wish to find the total physical error rate $p^{th}_{total}$ below which the total logical error probability $p^{log}_{total}$ of both logical $\bar{X}$ or $\bar{Z}$ errors vanishes as the distance $L$ of the code increases to infinity. A sufficient and necessary condition for a total error probability $p'_{total}$ to be below the accuracy

threshold for a decoder that decodes $Z$ and $X$ errors independently is that the probability of a $Z$-type error $p'_Z$ be below $p_Z^{th}$ and the probability of an $X$-type error $p'_X$ be below $p_X^{th}$.

The total error probability $p_{total}^{Z_{th}}$ when $p_Z = p_Z^{th}$ is

$$p_{total}^{Z_{th}} = p_Z^{th} + p_Z^{th}(1 - p_Z^{th})\frac{1}{\eta} \tag{4.3}$$

and the total error probability $p_{total}^{X_{th}}$ when $p_X = p_X^{th}$ is

$$p_{total}^{X_{th}} = p_X^{th} + p_X^{th}(1 - p_X^{th})\eta. \tag{4.4}$$

The total threshold $p_{total}^{th}$ is therefore given by

$$p_{total}^{th} = \min(p_{total}^{Z_{th}}, p_{total}^{X_{th}}). \tag{4.5}$$

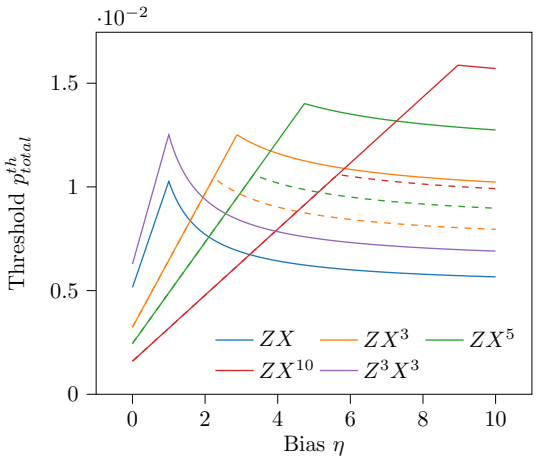



**Figure 4.8:** Threshold $p_{total}^{th}$ (see Eq. 4.5) as a function of bias for different homogeneous schedules and under a circuit-level independent noise model. Dashed lines use the same schedule as the corresponding solid line of the same colour, except gauge fixing is not used, for the purpose of comparison.

In Figure 4.8 we plot $p_{total}^{th}$ as a function of the bias parameter $\eta$ for the subsystem toric code, and for a few different choices of homogeneous schedule. For the $ZX$ schedule, used in Ref. [36], and the $Z^3X^3$ schedule with gauge fixing, the optimal bias is $\eta = p_Z/p_X = 1$. This is as expected, since the $X$ threshold is identical to the $Z$ threshold for these symmetric schedules. From Eqs. 4.3, 4.4 and

4.5 we see that at $\eta = 0$ and $\eta = \infty$ the total threshold is simply the $X$ threshold and $Z$ threshold, respectively.

For each of the schedules for which $p^{th}_{total}$ is plotted in Figure 4.8, there are two regimes: to the left and to the right of the peak. To the left of the peak, the threshold is limited by the $X$ threshold, and is therefore given by Equation (4.4), which is linear in $\eta$. To the right of the peak, the threshold is limited by the $Z$ threshold, and is therefore given by Equation (4.3), which is linear in $1/\eta$. The optimal $\eta$ for a given schedule can be found by setting $p^{Z_{th}}_{total} = p^{X_{th}}_{total}$.

Even for small finite bias, using unbalanced schedules and gauge fixing significantly improves the total threshold compared to the traditional $ZX$ schedule, with a $2.8\times$ increase in threshold at $\eta = 9$. With infinite bias the threshold rises to 2.22% which is $4.3\times$ higher than the threshold of 0.52% using standard $ZX$ schedule. Each dashed line in Figure 4.8 uses the same schedule as the corresponding solid line of the same colour, but without using gauge fixing to decode. For high bias, we see that approximately half of the improvement over the $ZX$ schedule can be attributed to the effect the new schedule has on the noise model, with the remainder attributed to the extra information used by gauge fixing when decoding.

For the rotated surface code, using the same schedule as in Ref. [46], we find a threshold under circuit-level independent noise of 0.741(2)%. Therefore, the subsystem toric code (with a $ZX^3$ schedule and using gauge fixing) outperforms the rotated surface code for biases $\eta > 2.3$.

Note that, for all the thresholds we have reported so far, we have used fully parallelised schedules. Whereas the $ZX$ schedule is fully parallelised with only $n_a = 1$ ancilla qubits per triangle operator, the unbalanced $ZX^b$ schedules require two ancilla qubits per $X$ check operator ($n_a = 1.5$), and the balanced $Z^aX^a$ schedules require two ancilla qubits per $X$ check operator *and* per $Z$ check operator ($n_a = 2$). Since there are $4n_a/3$ ancilla qubits per data qubit, this leads to a larger qubit overhead when using gauge fixing with parallelised schedules. We can choose not to parallelise the schedules, and instead simply omit gates in the $ZX$ schedule to construct our other schedules (e.g. an unparallelised $ZX^2$ schedule can be constructed

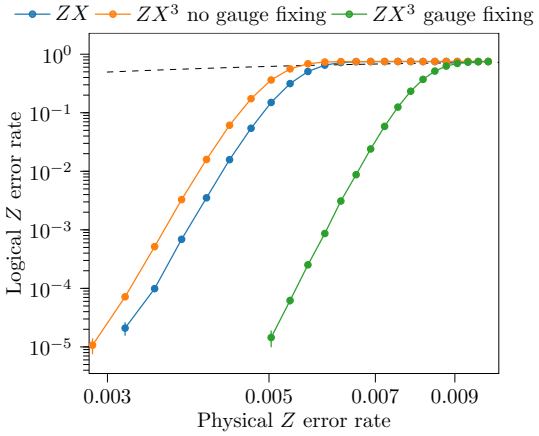



**Figure 4.9:** Logical $\bar{Z}$ error rate of the $[[2028, 2, 26]]$ subsystem toric code using a $(ZX)^{36}$ schedule, as well as a $(ZX^3)^{12}$ schedule (using only a single ancilla by introducing idle time steps) with and without using gauge fixing in the matching graph. All schedules use 144 time steps, and the independent circuit-level noise model was used. The dashed black line is the probability that either of two physical qubits will suffer a $Z$ error during 144 time steps without using error correction.

by omitting every other $Z$ measurement in the $ZX$ schedule). These schedules incur no qubit overhead, but instead introduce idle errors. The threshold with infinite bias using an unparallelised $X$ schedule is 1.25%, compared to 2.22% using a parallelised $X$ schedule, both an improvement over the 0.52% threshold using the $ZX$ schedule. Near the threshold, using additional ancillas is clearly worthwhile, whereas far below threshold it may be beneficial to use an unparallelised schedule, using the additional qubits to instead construct a code with a larger distance.

To analyse the performance below threshold, we compare a $ZX$ schedule to an unparallelised $ZX^3$ schedule ($n_a = 1$) using the $L = 26$ subsystem toric code, both with and without using gauge fixing to decode. When using gauge fixing, the logical $Z$ error rate is reduced by around four orders of magnitude compared to the $ZX$ schedule (see Figure 4.9). Without using gauge fixing, the logical error rate with the unparallelised $ZX^3$ schedule is slightly worse than with the $ZX$ schedule, since idle qubit errors are worse than qubit errors in the standard depolarising noise model [177].

## 4.4 Conclusion

In this chapter we introduced a technique, which we call schedule-induced gauge fixing, that improves the performance of a wide class of codes, especially under biased noise models. Schedule-induced gauge fixing changes the order in which check operators are measured in subsystem codes. While the check operators of subsystem codes do not all mutually commute, we find that grouping blocks of mutually commuting check operators together allows us to obtain more useful information without increasing the total number of measurements. By making consecutive measurements of the same gauge operators they can be treated temporarily as stabilisers, and we introduce a method for decoding, based on minimum-weight perfect matching (MWPM), that takes advantage of this additional information. When applied to the subsystem surface code with three-qubit check operators, we can switch repeatedly between the hexagonal surface code and its dual, both of which are abelian subgroups of the gauge group of the code. We find that the threshold under circuit-level depolarising noise can be increased from 0.67% to 0.81% by making four consecutive measurements of each gauge operator in the measurement schedule. The improvement is even more significant under biased noise models. With an independent $Z$-biased circuit-level noise model, $X$ check operators can be repeated (and fixed) more frequently, leading to an even higher threshold under small finite bias, up to 2.22% under infinite bias. Below threshold, gauge fixing reduces the logical error rate by several orders of magnitude for biased noise models.

Schedule-induced gauge fixing can be applied *in software*, with no changes to the underlying hardware interactions necessary. This allows both the code and the decoder to be tailored to the noise model even if it cannot be fully characterised prior to device fabrication. The same techniques can also be directly applied to a broad class of subsystem codes beyond the subsystem surface code, including the Bacon-Shor code [9], the heavy hexagon code [48], and some compass codes [138], and future work could investigate the performance improvements achievable using schedule-induced gauge fixing with these codes. It would also be interesting to generalise the decoding method to other subsystem codes where detection events

do not come in pairs, such as the gauge colour code [28], amongst others [23, 181]. We also note that schedule-induced gauge fixing has recently been applied to the problem of handling fabrication defects in surface codes [180, 173].

# Part II

# Practical High-Rate Quantum LDPC Codes

# Chapter 5

# Introduction

As we saw in Part I, the surface code has many advantages that make it particularly amenable to experimental implementation [64]. It has a high threshold approaching 1% for a circuit-based noise model, with operations that are geometrically local in 2D [85]. Furthermore, it is well known how to implement universal logic gates in surface codes fault-tolerantly [32, 85, 117, 140, 82], and surface code circuits can be decoded efficiently and accurately using the matching-based decoders described in Part I.

Unfortunately, the surface code requires a very high resource overhead to achieve the low logical error rates needed to perform useful fault-tolerant quantum computations. For example, to achieve a logical failure rate of one in a trillion (the "teraquop" regime [97]) requires using around 1000 physical qubits per logical qubit. This is a consequence of the fact that the surface code only encodes a single logical qubit per code block (it has a vanishing rate $k/n$), and its distance scales as the square root of the number of physical qubits, $d = \sqrt{n}$. In fact, it has been proven that, more generally, the parameters of stabiliser codes with stabiliser generators that are geometrically local in 2D are constrained by the trade-off $kd^2 = O(n)$ [33]. This suggests that more efficient quantum error correcting codes will require some long-range connections between qubits.

If we relax the constraint of geometric locality, we can consider the broader class of stabiliser codes called quantum low-density parity check (LDPC) codes [43]. A quantum LDPC code is a stabiliser code admitting a set of stabiliser generators

such that each stabiliser generator acts non-trivially on a constant number of qubits, and each qubit is in the support of a constant number of stabiliser generators. In other words, the Tanner graph of the code has bounded degree.

There have been recent breakthroughs in the development of quantum LDPC codes that have led to significant improvements in the asymptotic parameters of quantum codes [43, 103, 159, 42, 157, 135]. In fact, it has now been proven [157] that some families of lifted product [159] and balanced product [42] quantum LDPC codes have optimal asymptotic parameters $k = \theta(n)$ and $d = \theta(n)$.

However, demonstrating a significant advantage over the surface code for realistic noise models and reasonable system sizes is challenging. For example, Ref. [189] demonstrated a significant $14\times$ saving in resources relative to the surface code using hypergraph product codes for circuit-level noise, however this improvement was shown for very low physical error rates of around $0.01\%$ and for very large system sizes (around 13 million qubits). This is in large part due to the high-weight stabiliser generators common in many families of quantum LDPC codes, resulting in deep and complex syndrome extraction circuits, which can introduce many additional error mechanisms and lower their noise thresholds [53, 137, 189, 160]. The recent results of Refs. [35, 208] developed efficient syndrome extraction circuits for some families of quantum LDPC codes (hypergraph product, lifted product and quasi-cyclic codes) and showed good logical error rate performance under circuit-level noise. Some drawbacks of these approaches, however, are that they require high qubit connectivity (at least degree 6), and the BP-OSD decoder used has high computational complexity $O(m^3)$, where $m$ is the number of error mechanisms.

In this second part of the thesis, we show how a family of quantum LDPC codes called hyperbolic surface codes can be adapted such that they have good performance for circuit-level noise. As we will explain, we achieve this using generalisations of subsystem surface codes and Floquet codes to hyperbolic surfaces, thereby reducing the weight of check operators that must be measured, simplifying syndrome extraction circuits. We will show that our constructions are more efficient than the surface code even at high physical error rates, and the qubit connectivity in

our constructions is always 2, 3 or 4 (less than or equal to that of the surface code). Furthermore, all of our constructions can be decoded efficiently with matching-based decoders such as MWPM or Union-Find.

## 5.1 Chain complexes and $\mathbb{F}_2$-homology

As explained in Section 1.1.1, the stabiliser generators of CSS codes can be represented using a check matrix of the form

$$H = \begin{pmatrix} H_X & 0 \\ 0 & H_Z \end{pmatrix} \tag{5.1}$$

where $H_X \in \mathbb{F}_2^{r_x \times n}$ and $H_Z \in \mathbb{F}_2^{r_Z \times n}$ are binary matrices defining the $X$-type and $Z$-type stabiliser generators, respectively. Recall that the commutativity requirement is then given by the condition

$$H_X H_Z^T = 0. \tag{5.2}$$

This condition leads to a useful connection between CSS codes and chain complexes in $\mathbb{F}_2$-homology, which we will now review. For a more comprehensive introduction to chain complexes, and their use in constructions of quantum LDPC codes, we refer the reader to Refs. [104, 43].

In $\mathbb{F}_2$-homology, a chain complex $C$ with length $l+1$ is a collection of vector spaces $C_i := \mathbb{F}_2^{n_i}$

$$\{0\} \xrightarrow{\partial_{l+1}} C_l \xrightarrow{\partial_l} C_{l-1} \cdots \xrightarrow{\partial_1} C_0 \xrightarrow{\partial_0} \{0\}$$

and *boundary maps* $\partial_i : C_i \to C_{i-1}$, where the boundary maps satisfy the constraint

$$\partial_i \partial_{i+1} = 0 \tag{5.3}$$

for all $i \in \{0 \dots l\}$. We refer to each element of $C_i$ as an $i$-cell, and call elements of $\operatorname{im} \partial_{i+1}$ and $\ker \partial_i$ *boundaries* and *cycles*, respectively. From Equation (5.3) we know that $\operatorname{im} \partial_{i+1} \subseteq \ker \partial_i$ (every boundary is a cycle). However, each cycle is not

necessarily a boundary, and the $i$th *homology group* of $C$ is the quotient

$$\mathrm{H}_i = \ker \partial_i / \operatorname{im} \partial_{i+1}. \tag{5.4}$$

Associated with $C$ is another chain complex called a *cochain* complex with *coboundary* operators $\delta^i : C_i \to C_{i+1}$ defined as $\delta^i := \partial_{i+1}^T$:

$$\{0\} \xrightarrow{\delta^{-1}} C_0 \xrightarrow{\delta^0} C_1 \cdots \xrightarrow{\delta^{l-1}} C_l \xrightarrow{\delta^l} \{0\}.$$

Elements of $\ker \delta^i$ and $\operatorname{im} \delta^{i-1}$ are *cocycles* and *coboundaries* respectively, and the $i$th *cohomology* group is $\mathrm{H}^i = \ker \delta^i / \operatorname{im} \delta^{i-1}$.

We see that the commutativity condition for CSS codes, Equation (5.1), is equivalent to the defining property of chain complexes in $\mathbb{F}_2$-homology, given in Equation (5.3). A consequence of this is that we can use a chain complex with length at least two to define a CSS code. We associate each qubit with an $i$-cell, where $0 < i < l$. We then use the $i$th boundary operator as the $X$ check matrix $H_X = \partial_i$ and use the $i$th coboundary operator as the $Z$ check matrix $H_Z = \delta^i = \partial_{i+1}^T$. The commutativity condition $H_X H_Z^T = 0$ is then guaranteed to be satisfied by Equation (5.3). The $Z$ logicals are associated with elements of the $i$th homology group $\mathrm{H}_i$, and the $X$ logicals are associated with elements of the $i$th cohomology group $\mathrm{H}^i$. The number of logical qubits is therefore given by $\dim \mathrm{H}_i = \dim \mathrm{H}^i$.

Many different approaches have been taken to construct chain complexes defining families of quantum LDPC codes with improved parameters (see Ref. [43] for a review). In this thesis we focus on constructions derived from tilings of two-dimensional surfaces that generalise beyond the toric code.

As a concrete example, let us revisit the toric code and consider it from the perspective of homology. Recall that the toric code is defined on a square tiling of a torus. This square tiling defines a length-three chain complex, where each 0-cell corresponds to a vertex, each 1-cell corresponds to an edge and each 2-cell corresponds to a face. The boundary operator $\partial_1$ maps each edge to the vertices at its boundary (endpoints), and similarly $\partial_2$ maps each face to the edges on its boundary.

We associate a qubit with each edge of the lattice, i.e. a 1-cell. Each $X$ stabiliser generator is the set of edges adjacent to a vertex (the co-boundary of the vertex) and hence we have $H_X \coloneqq \delta^0 = \partial_1$. Each $Z$ stabiliser generator is the set of edges at the boundary of a face (the edges at its boundary), and hence $H_Z \coloneqq \partial_2$. Each $Z$ logical operator is an element of the 1st homology group $\mathrm{H}_1 = \ker \partial_1 / \operatorname{im} \partial_2 = \ker H_X / \operatorname{im} H_Z^T$. In other words, each $Z$ logical is a cycle of edges (and hence has no boundary), which is not a boundary of a set of faces (i.e. it is not a $Z$ stabiliser). These correspond to non-contractible loops, or homologically nontrivial loops, around the handle of the torus. Similarly, each $X$ logical operator is an element of the 1st co-homology group, and is hence a homologically nontrivial co-cycle around the handle of the torus.

More generally, for any tiling of an orientable surface with $w$ handles (genus $w$), it holds that $\dim H_1 = 2 - \chi = 2w$, where here $\dim H_1$ is the first homology group of the corresponding length-three chain complex, as explained above. Here $\chi \coloneqq |V| - |E| + |F|$ is the Euler characteristic of the tiling (with $|V|$ vertices, $|E|$ edges and $|F|$ faces). Hence, the number of logical qubits encoded by a surface code derived from a more general tiling of a surface is determined purely by its topology.

## 5.2 Tilings of hyperbolic surfaces

We will see in the next sections that quantum codes derived from tilings of surfaces with negative Gaussian curvature (hyperbolic surfaces) have favourable parameters relative to the surface code. This is a consequence of the the fact (due to the Gauss-Bonnet theorem) that closed surfaces with negative Gaussian curvature have an Euler characteristic that grows with their area, unlike Euclidean surfaces. In this section, we will review tilings of hyperbolic surfaces and their properties.

### 5.2.1 Wythoff's kaleidoscopic construction

We can obtain tilings of the hyperbolic plane using Wythoff's kaleidoscopic construction. We choose a *fundamental triangle* with internal angles $\pi/p$, $\pi/q$ and $\pi/t$. We can generate an infinite tiling through reflections in the sides of the triangle (we refer to each side of the triangle as a mirror). The group generated by these reflections (a

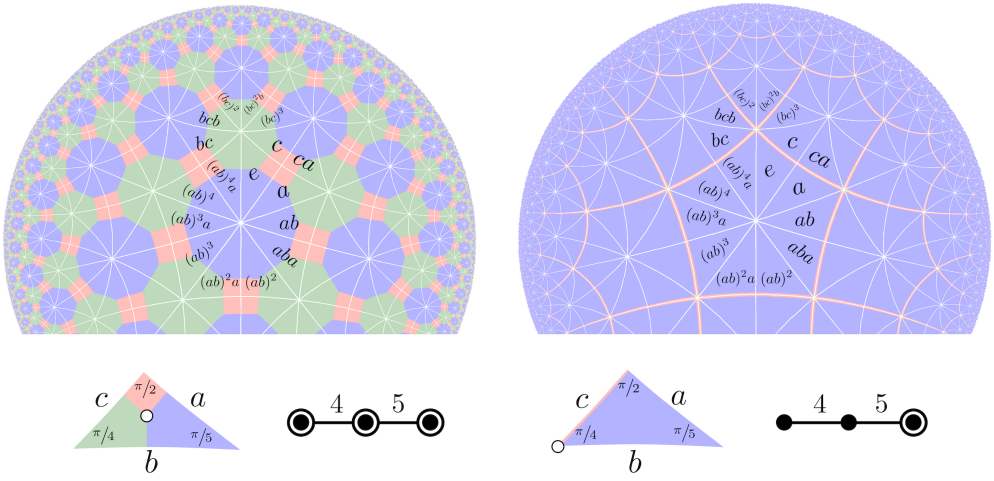



**Figure 5.1:** Two hyperbolic tilings generated by the hyperbolic triangle group $\Delta(2,4,5)$ using Wythoff's kaleidoscopic construction. Both tilings are generated through reflections across the sides of a fundamental triangle with internal angles $\pi/2$, $\pi/4$ and $\pi/5$. We show a white cut along each reflection line. Below each tiling we show the corresponding fundamental triangle and Coxeter diagram. A generator point is placed in the fundamental triangle and from each mirror a perpendicular line is drawn to the generator point. The only difference between the two tilings is the choice of generator point. Left: A uniform tiling with vertex configuration 4.8.10, which uses a generator point in the interior of the fundamental triangle. Right: A regular tiling with Schläfli symbol $\{5,4\}$ (vertex configuration $5.5.5.5 = 5^4$), which uses a generator point in the corner of the triangle with angle $\pi/4$ (the edges of the tiling are given in red).

symmetry group of the tiling) is the triangle group $\Delta(p,q,t)$, an example of a Coxeter group. We place a vertex, called a generator point, within the fundamental triangle, and from each mirror we draw a perpendicular line to the generator point. These lines are mapped by the reflections to edges in the tiling, and the generator point is similarly mapped to vertices of the tiling. For a given triangle group $\Delta(p,q,t)$ we can generate different tilings depending on the choice of generator point, and two such examples are shown in Figure 5.1 for $\Delta(2,4,5)$.

If we denote a reflection in each of the three sides by $a$, $b$ and $c$, where the angles between the pairs of mirrors $(a,b)$, $(b,c)$ and $(c,a)$ are $\pi/t$, $\pi/q$ and $\pi/p$ respectively, then the triangle group $\Delta(p,q,t)$ has presentation

$$\Delta(p,q,t) = \langle a,b,c \mid a^2 = b^2 = c^2 = (ab)^t = (bc)^q = (ca)^p = e \rangle \qquad (5.5)$$

where $e$ is the identity element. The relations for $(ab)^t$, $(bc)^q$ and $(ca)^p$ can be understood from the fact that $ab$, $bc$ and $ca$ are rotations by $2\pi/t$, $2\pi/q$ and $2\pi/p$, respectively.

#### 5.2.1.1 Regular hyperbolic tilings

A regular tiling, in which each face in the tiling is an $r$-gon and $s$ faces meet at each vertex, can be denoted by its Schläfli symbol $\{r,s\}$. Regular tilings of hyperbolic surfaces satisfy $1/r+1/s<1/2$. We will consider regular tilings in Chapter 6. We can construct such a regular $\{r,s\}$ tiling using Wythoff's construction with the triangle group $\Delta(2,s,r)$ by placing the generator point on the corner with internal angle $\pi/s$, as shown in Figure 5.1 (right) for a $\{5,4\}$ tiling. Fixing some triangle as the fundamental domain of $\Delta(2,s,r)$ (labelled as $e$ in Figure 5.1) we can label every triangle in the tiling uniquely with a group element. Each face of the $\{r,s\}$ tiling is then labelled by an element of $\Delta(2,s,r)$ up to reflections $a$ and $b$. More precisely, if we define the subgroup $\langle a,b\rangle$ of $\Delta(2,s,r)$ generated by $a$ and $b$ then each face is labelled by a left coset $g\langle a,b\rangle$ for some $g\in\Delta(2,s,r)$. Similarly, each vertex or edge of the tiling is labelled by a left coset $g\langle b,c\rangle$ or $g\langle c,a\rangle$, respectively. The boundary operators of faces and edges can therefore be constructed from the relevant intersection of these different cosets.

#### 5.2.1.2 Uniform degree-three hyperbolic tilings

In Chapter 7 we will consider degree-three uniform hyperbolic tilings constructed using a generator point in the interior of the fundamental triangle. Uniform tilings can be described by their *vertex configuration*, a sequence of numbers giving the number of sides of the faces around a vertex. For example, each vertex of a degree-three uniform tiling with vertex configuration $r.g.b$ has three faces around it, with $r$, $g$ and $b$ sides. When the tiling is constructed using the triangle group $\Delta(p,q,t)$, we have that $r=2p$, $g=2q$ and $b=2t$. Uniform tilings of hyperbolic surfaces satisfy $1/r+1/g+1/b<1/2$, whereas a Euclidean tiling (e.g. 6.6.6 or 4.8.8) satisfies $1/r+1/g+1/b=1/2$. Let us assign a colour to each face such that red, green and blue faces have $r$, $g$ and $b$ sides, respectively (see Figure 5.1). Again fixing some triangle as the fundamental domain, we see that each vertex of the $r.g.b$ tiling is

uniquely identified by an element of $\Delta(2r,2g,2b)$. The red, green and blue faces can each be identified using left cosets $g\langle a,c\rangle$, $g\langle b,c\rangle$ or $g\langle a,b\rangle$, respectively. Similarly, each edge is associated with a left coset $g\langle a\rangle$, $g\langle b\rangle$ or $g\langle c\rangle$. The boundary of each face can thus be determined from which edge cosets $g\langle a\rangle$, $g\langle b\rangle$ or $g\langle c\rangle$ are subgroups of the face's coset, and similarly the endpoints of an edge are simply the vertices corresponding to the elements contained in its coset.

### 5.2.2 Compactification

So far we have described infinite tilings, however we would like to construct quantum codes from finite tilings. We construct finite tilings using a compactification procedure [44, 40]. This can be achieved by finding a normal subgroup $\Gamma$ of $\Delta(p,q,t)$ that has no fixed points and gives a finite quotient group $\Delta(p,q,t)/\Gamma$, see [44, 40]. The quotient group $G^{+}_{p,q,t} \coloneqq \Delta(p,q,t)/\Gamma$ can then be used to define regular and uniform tilings using the Wythoff construction, and has presentation

$$G^{+}_{p,q,t} = \langle a,b,c \mid a^2 = b^2 = c^2 = (ab)^t = (bc)^q = (ca)^p = r_1 = \cdots = r_v = e\rangle \quad (5.6)$$

where here the additional relations $r_1,\ldots,r_v$ are the generators of $\Gamma$.

### 5.2.3 Properties of hyperbolic surface codes

For an $\{r,s\}$ regular tiling, we have that $r|F| = s|V| = 2|E|$ and hence the dimension of the first homology group is given by

$$\dim H_1 = 2-\chi = |E|(1-\frac{2}{r}-\frac{2}{s})+2. \quad (5.7)$$

Therefore, defining a surface code from such a tiling with $n = |E|$ physical qubits and $k = \dim H_1$ logical qubits we have a finite encoding rate $k/n > (1-2/r-2/s)$, provided the tiling is hyperbolic $(1/r+r/s<1/2)$.

Considering now a degree-three uniform $r.g.b$ tiling, let us denote the set of faces with $r$, $g$ and $b$ sides as $F_r$, $F_g$ and $F_b$, respectively. We also denote the edge set by $E$ and the vertex set by $V$. The tiling has $|E| = 3|V|/2$ edges, and since $r|F_r| = g|F_g| = b|F_b| = |V|$, we have that $|F| = |V|(1/r+1/g+1/b)$. Therefore, the

dimension of the first homology group is

$$\dim H_1 = 2 - |V| + |E| - |F| = |V| \left( \frac{1}{2} - \frac{1}{r} - \frac{1}{g} - \frac{1}{b} \right) + 2. \tag{5.8}$$

The distance of a hyperbolic surface code is determined by the minimum length of a non-contractible loop (combinatorial systole) of the tiling or its dual. For hyperbolic tilings it is known that the combinatorial systole grows at most logarithmically in the number of edges, and hence hyperbolic surface codes have distance scaling as $d \in O(\log n)$ [174, 78, 59, 44]. This leads to the parameters of hyperbolic surface codes satisfying $kd^2/n = O(\log^2(n))$, which is an asymptotic improvement over the parameters of Euclidean surface codes, for which $kd^2/n = O(1)$.

The larger stabiliser weight of hyperbolic surface codes leads to deeper syndrome extraction circuits, which can reduce the effective distance of the code and lower noise thresholds [53]. Furthermore, the high check weight increases the required qubit connectivity, which can lead to crosstalk in some architectures [48]. In the remainder of this thesis we describe constructions that are closely related (by a constant-depth unitary) to hyperbolic surface codes, but which have reduced check weight. In Chapter 6 we describe subsystem hyperbolic codes, which generalise the subsystem surface code to closed hyperbolic tilings [110]. In Chapter 7 we construct and analyse the performance of Floquet codes [102, 198] derived from colour code tilings of closed hyperbolic surfaces [111]. For a more comprehensive background on hyperbolic surface codes and their properties, as well as generalisations to higher dimensions, we refer the reader to Ref. [40].

# Chapter 6

# Subsystem hyperbolic and semi-hyperbolic codes

Most research demonstrating the potential advantages of quantum LDPC codes has assumed simplistic noise models that do not account for noise in the quantum circuit used to implement them. Once circuit-level noise is taken into account, the potential reduction in qubit overhead can be lost [53]. However, in this chapter, we show how we can reduce the qubit overhead for quantum error correction even with more realistic circuit level noise, using a construction for subsystem codes that encode a number of logical qubits $k$ proportional to the number of physical qubits $n$, while using only three-qubit check operators. These codes are derived from hyperbolic tilings, and we use the symmetry group of the tiling to derive quantum circuits for measuring the check operators that use only four time steps, which is optimal. From simulating their performance with circuit-level depolarising noise, we find that these finite-rate subsystem codes have a qubit overhead that is $4.3\times$ lower than the most efficient version of the surface code for error rates as high as 0.2%, which is a noise regime often considered for practical surface code quantum computing [94].

This chapter is organised as follows. In Section 6.1 we introduce our construction of subsystem hyperbolic codes, before discussing their properties in Section 6.2. We then show how efficient syndrome measurement circuits can be constructed in Section 6.3 and introduce subsystem semi-hyperbolic codes in Section 6.4. We then numerically assess their performance in Section 6.5 before concluding in Section 6.6.

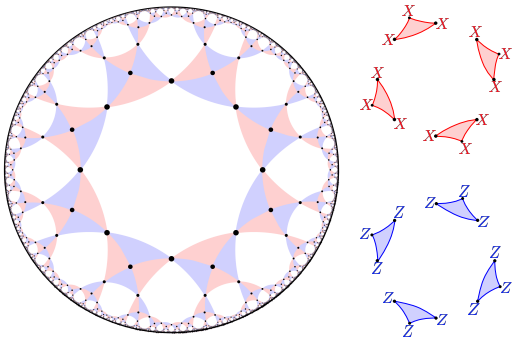


**Figure 6.1:** The $\{8,4\}$ subsystem hyperbolic code. A qubit (each represented by a black filled circle) is placed in the center of each edge and on each vertex of an $\{8,4\}$ tiling of a closed hyperbolic surface. A three qubit triangle operator is placed in each corner of each face. Each $X$ stabiliser is the product of the four $X$ triangle operators within a face (top right). Similarly, each $Z$ stabiliser is the product of the four $Z$ triangle operators within a face (bottom right).

The original research in this section has perviously been published in Ref. [112].

## 6.1 Subsystem hyperbolic codes

Our construction generalises the subsystem toric code of Ref. [36] (which we reviewed in Section 1.2.3) to more general tilings, including tilings of closed hyperbolic surfaces. One of the challenges of implementing circuits for hyperbolic surface codes is that their stabiliser weight is larger than for the toric code, making syndrome extraction more challenging. A key benefit of the subsystem hyperbolic code construction we now present is that syndrome extraction can be done with only weight-3 check operators. Hyperbolic codes are a promising candidate for experimental realisation in systems that allow variable length connectivity between qubits [130], such as modular architectures [153, 125], and reduced check-weight simplifies stabiliser readout, as well as reducing crosstalk [48].

For our construction we start with a tiling of a surface, where each face has an even number of sides, and four faces meet at each vertex. As was done for the subsystem toric code, we place a qubit in each corner of the lattice and in the middle of each edge. We then place a triangle operator in each corner of each face of the tiling, alternating between $X$ (red) and $Z$ (blue) triangle operators around each face and around each vertex. Each triangle operator is a gauge operator of the subsystem code. The $X$ triangle operators are three-qubit $XXX$ Pauli operators (acting non-

trivially on the qubits in their support), and similarly each $Z$ triangle operator is a three-qubit $ZZZ$ Pauli operator. An example using a regular $\{8,4\}$ tiling is shown in Figure 6.1. Each $Z$ stabiliser is the product of all $Z$ triangle operators within a face of the $\{8,4\}$ tiling, and similarly for $X$ stabilisers and $X$ triangle operators. Note that for a square lattice on a torus we recover the subsystem toric code, but our construction generalises to hyperbolic tilings.

By definition we have required that adjacent triangle operators related by a single rotation about a face or a vertex must be of opposite Pauli types. We will say that a tiling that allows such an assignment of triangle operators is *colourable*. This colourability requires that each face must have an even number of sides, and an even number of faces must meet at each vertex (so for regular $\{r,s\}$ tilings, both $r$ and $s$ must be even). We required exactly *four* faces to meet at each vertex to ensure that the stabilisers commute. In Appendix D.4 we show that a regular tiling of a *closed* surface is colourable if a particular function $f$ (which we define) extends to a homomorphism from the symmetry group of the tiling to the cyclic group $\mathbb{Z}_2$. As a result, the colourability property places a constraint on which compactifications of a hyperbolic tiling can be used to define subsystem codes (each relation in the quotient of the triangle group defining the tiling must contain an even number of reflections $a$ or $c$ across the sides of the fundamental triangle).

## 6.2 Properties of subsystem hyperbolic codes

We will now consider some more properties of subsystem hyperbolic codes, each derived from a $\{2c,4\}$ tiling with edges $E$, vertices $V$ and faces $F$. Since we place a qubit on each vertex, and in the centre of each edge of this tiling, our subsystem hyperbolic code will have $|E|+|V|$ data qubits. Each vertex in the tiling has degree 4, and so $2|V| = |E|$. Furthermore, we also place $n_a$ ancilla qubits within each triangle operator. While we can always use $n_a = 1$ ancillas per triangle operator by using schedules with some idle qubit locations (if necessary), we have parallelised many of our schedules which in some cases requires $n_a = 2$. Each vertex is adjacent to four triangle operators and each triangle operator is adjacent to a single vertex. Therefore,

in total there are $n = \frac{3}{2}|E|$ data qubits and $2n_a|E|$ ancilla qubits in our subsystem hyperbolic codes. For the subsystem toric code, where $|E| = 2L^2$, there are $3L^2$ data qubits and $4n_aL^2$ ancilla qubits.

The number of faces in the $\{r,s\}$ tiling satisfies $r|F| = 2|E|$. Since the product of all $X$-type (or $Z$-type) stabilisers is the identity, and since there are no other relations that the stabilisers satisfy, the number of independent stabilisers is $4|E|/r - 2$. Therefore, the total number of logical qubits (including gauge qubits) is $(3/2 - 4/r)|E| + 2$.

Aside from the triangle operators introduced within each face, the number of remaining bare logical operators (those in $C(\mathcal{G}) \setminus \mathcal{G}$) is determined from the topology of the tiling from which it is derived. Therefore, excluding gauge qubits, the number of logical qubits $k$ that a subsystem hyperbolic code derived from a $\{r,4\}$ tiling encodes is given by [44]

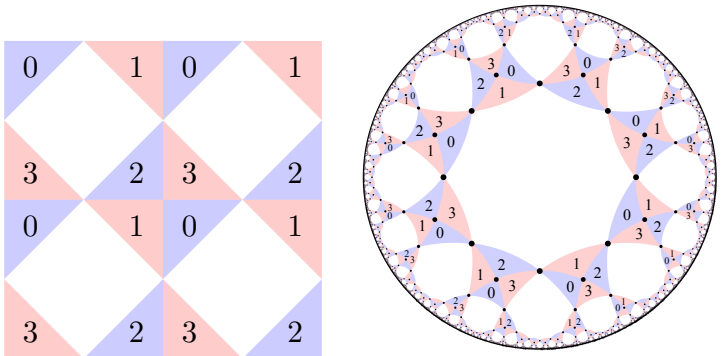
$$k = \frac{|E|}{2} - \frac{2|E|}{r} + 2. \tag{6.1}$$

This leaves $(1 - 2/r)|E|$ gauge qubits, or $r/2 - 1$ gauge qubits per face. The triangle operators act nontrivially on these gauge qubits. The encoding rate of the subsystem hyperbolic code is therefore

$$\frac{k}{n} = \frac{1}{3} - \frac{4}{3r} + \frac{2}{n}. \tag{6.2}$$

There are $4n_a/3$ ancilla qubits per data qubit, leading to $(4n_a/3 + 1)n$ qubits in total. Note that this expression does not depend on $r$: the number of ancilla qubits is proportional to the number of data qubits, and the constant of proportionality is the same regardless of which $\{2c,4\}$ tiling we use.

In Appendix D.7, we show that the distance $d$ of a subsystem hyperbolic or semi-hyperbolic code is bounded by $d_X/2 \leq d \leq d_X$, where $d_X$ is the $X$ distance of the *subspace* hyperbolic or semi-hyperbolic code derived from the same tiling. The $X$ distance of the subspace code is always less than or equal to its $Z$ distance for the codes we consider, and so the distance of the subsystem code is at least half, and at most the same as, the distance of the subspace code. We analyse the distances of the codes we construct in Appendix D.7, and find codes with distances that span this full




(a) (b)

**Figure 6.2:** (a) An $L = 2$ subsystem surface code. The four types of triangle operators are labelled as 0, 1, 2 and 3. (b) Labelling of the four types of triangle operators on an $\{8,4\}$-tiling of the hyperbolic plane. The neighbourhood of each triangle operator (the types and relative locations of triangle operators it overlaps with) is the same as in the toric code.

range.

## 6.3 Efficient circuits for syndrome measurement

Recall that in order to determine the syndrome used for decoding, we require a *stabiliser measurement schedule*, the sequence of gates applied to data and ancilla qubits in order to measure the eigenvalues of the stabilisers. We will now show that any valid stabiliser measurement schedule defined within a single face of the subsystem toric code and chosen to be periodic in space (i.e. identical for every vertex or face) can be generalised for a subset of $\{4c,4\}$ subsystem hyperbolic codes, for $c \in \mathbb{Z}^+$. The measurement schedule used by Bravyi *et al.* [36] is an example of such a periodic schedule.

We first assign an element of the cyclic group $\mathbb{Z}/4\mathbb{Z}$ to each of the four types of triangle operators within a face, and will call such an assignment a *labelling*. We choose to label the north-west, north-east, south-east and south-west triangle operators with the elements 0, 1, 2 and 3 of $\mathbb{Z}/4\mathbb{Z}$, respectively (see Figure 6.2(a)). Note that, for a translationally invariant schedule, each triangle operator with a given *label* in the subsystem toric code is assigned an identical schedule. Triangle operators with different labels have different measurement schedules. In order to apply this measurement schedule to the subsystem hyperbolic code, we label

every triangle operator as one of these four types in such a way that the schedule always looks locally the same as for the subsystem toric code to ensure that it remains correct. More precisely, for each triangle operator with a given label in the subsystem hyperbolic code, its neighbourhood of triangle operators it shares qubits with (and their labels) must be the same as for a triangle operator with the same label in the subsystem toric code. We will call a labelling that has this property a *valid labelling*, and a *schedulable* code is one that admits a valid labelling. In Appendix D.5, we show that a regular tiling of a closed hyperbolic surface admits a valid labelling if a particular function $h$ (which we define) extends to a homomorphism from the proper symmetry group of the tiling to the cyclic group $\mathbb{Z}/4\mathbb{Z}$. We show that a subset of $\{4c,4\}$ regular tilings of closed hyperbolic surfaces satisfy this property. An example of a valid scheduling of the $\{8,4\}$ tiling of the hyperbolic plane is shown in Figure 6.2(b).

## 6.4 Subsystem semi-hyperbolic codes

The $\{8,4\}$ subsystem hyperbolic code has stabilisers of weight 12, which is double that of the subsystem toric code. Despite the check operators still being weight 3, we find that the large stabiliser weight results in a lower threshold of $0.31(1)\%$ compared to $0.666(1)\%$ for the subsystem toric code. The intuition behind this is the following: if a stabiliser has higher weight, it provides less information about the location of an error and requires more gates to be used when measured, making it harder to measure precisely. Furthermore, the distance of these codes scales only logarithmically in $n$, which means that logical errors cannot be suppressed exponentially in system size.

To address these issues, we can construct subsystem codes derived from *semi-hyperbolic* tilings, introduced in Ref. [38]. The idea is to fine-grain the tiling leading to lower-weight stabilizers. A semi-hyperbolic tiling is derived from a $\{4,q\}$ regular tiling of a closed hyperbolic manifold for $q > 4, q \in \mathbb{Z}^+$. Each (square) face of the $\{4,q\}$ tiling is tiled with an $l \times l$ square lattice. By doing so, the curvature of the surface is weakened. The subspace quantum code derived from the semi-hyperbolic tiling (a semi-hyperbolic code) has larger distance and reduced check

weight compared to a code derived from the original $\{4,q\}$ tiling. This comes at the cost of requiring $l^2$ times more qubits and, since the number of logical operators is unchanged, the encoding rate is reduced by a factor of $l^2$. An important advantage of semi-hyperbolic codes is that, by increasing $l$, we obtain a family of codes with distance scaling like $\sqrt{n}$ (as for the toric code), while expecting to retain a reduced qubit overhead relative to the toric code [38]. The same advantages apply for the subsystem semi-hyperbolic codes we construct in this work.

Recall that the tilings that we derive subsystem hyperbolic codes from must have vertices of degree four, and each face must have $4c$ sides (where $c \in \mathbb{Z}^+$). On the other hand, a $\{4,q\}$ semi-hyperbolic tiling instead has faces with four sides, while vertices have degree four or $q$. We can therefore derive a subsystem code from the *dual lattice* of $\{4,4c\}$ semi-hyperbolic tiling. In Appendix D.5 we show that if an $\{8,4\}$ tiling is schedulable, then so is the semi-hyperbolic tiling derived from it. Therefore, each schedulable closed $\{8,4\}$ tiling defines a family of subsystem semi-hyperbolic codes (each code in the family having a different lattice parameter $l$), and where each code in the family is schedulable.

We say that an $l$, $\{4c,4\}$ subsystem semi-hyperbolic code is the code derived by placing a triangle operator in each corner of each face of the dual lattice of a semi-hyperbolic lattice, where that semi-hyperbolic lattice was constructed by tessellating each face of the $\{4,4c\}$ tiling with an $l \times l$ square lattice. The subsystem semi-hyperbolic codes we construct and analyse in this work are $l = 2$, $\{8,4\}$ subsystem semi-hyperbolic codes. The irregular tilings these codes are derived from therefore contain both square and octagonal faces, with four faces meeting at each vertex.

## 6.5 Numerical simulations

We have simulated the performance of $l = 2$ $\{8,4\}$ subsystem semi-hyperbolic codes under the circuit-level depolarising noise model. We are interested in finding the threshold value below which the logical error rate per logical qubit tends to zero as the code distance tends to infinity. Since the number of logical qubits $k$ increases with distance for this family of finite-rate codes, we fix the number of logical qubits

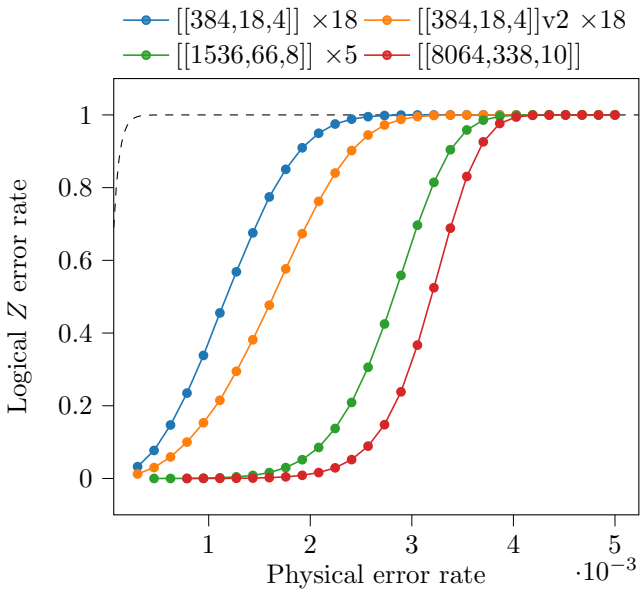



**Figure 6.3:** Performance of the extremal $l = 2$ $\{8,4\}$ subsystem semi-hyperbolic codes under a circuit-level depolarising noise model. Here, we fix the number of logical qubits to at least 338 for all codes, by using multiple copies of the smaller codes. A homogeneous $(ZX)^{20}$ schedule is used for all codes, and the $y$ axis is the probability that at least one logical $Z$ error occurs. The dashed black line is the probability of a $Z$ error occurring on at least one of 338 physical qubits without error correction under the same error model for the same duration (80 time steps). For each code that encodes $k < 338$ logical qubits, we use $m = \lfloor k/338 \rfloor$ copies and plot the failure rate as $p^*_{log} = 1-(1-p_{log})^m$.

by using multiple independent copies of the smaller codes. In Figure 6.3 we plot the probability that at least one of 338 logical qubits suffers a $Z$ failure as a function of the depolarising error rate $p$. The [[8064,338,10]] code has the lowest logical error rate per logical qubit for physical error rates below 0.42%, from which we conclude that the threshold is at least 0.42%. We have not been able to obtain an upper bound on the threshold, since all codes have an error rate (per 338 logical qubits) of one for physical error rates above 0.42%, within the precision provided by our numerical experiments.

We now analyse the performance of the [[8064,338,10]] $l = 2$ $\{8,4\}$ subsystem semi-hyperbolic code, which has the best ratio $n/(kd^2) = 0.24$ of the codes we have constructed. In Figure 6.4 we compare its performance with that of the $L = 4,6,8,9$ and 10 subsystem toric codes. We use 169 independent copies of the subsystem toric codes, in order to keep the number of logical qubits (338) constant,

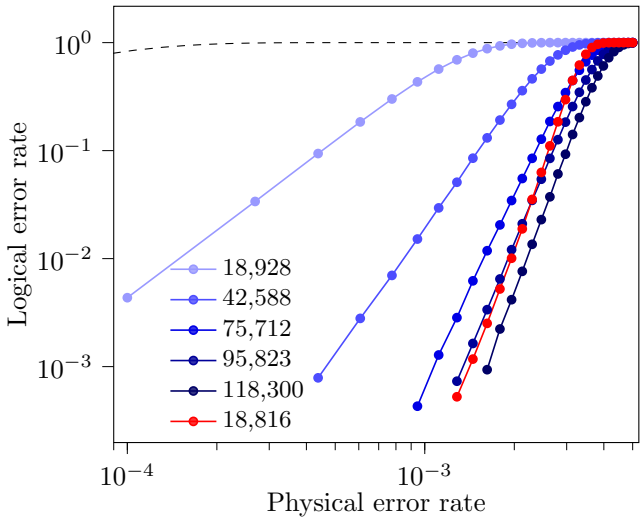



**Figure 6.4:** Comparison of the [[8064,338,10]] l=2 $\{8,4\}$ subsystem semi-hyperbolic code (red), which has 8,064 data qubits and 10,752 ancillas, with $L = 4,6,8,9$ and 10 subsystem toric codes (shades of blue), using a $(ZX)^{20}$ schedule (no gauge fixing) and a circuit-level depolarising noise model. We fix the number of logical qubits by plotting the probability that at least one of 169 independent copies of the subsystem toric codes suffers a logical $Z$ failure (i.e. we plot $1-(1-p_{log})^{169}$ for the subsystem toric codes where $p_{log}$ is the probability that a single copy of the code suffers a logical $Z$ error). The total number of physical qubits (including ancillas) is given in the legend. The black dashed line is the probability that at least one of 338 physical qubits would suffer a $Z$ failure without error correction over the same duration.

and the total number of physical qubits used (including ancillas) is given in the legend. We find that the [[8064,338,10]] subsystem semi-hyperbolic code (which uses 18,816 physical qubits), outperforms the $L = 4$ subsystem toric code (which uses 18,928 physical qubits to encode 338 logical qubits) by around three orders of magnitude at $p = 0.15\%$. At a physical error rate of 0.2% the performance of the [[8064,338,10]] subsystem semi-hyperbolic code is similar to the $L = 9$ subsystem toric code, which uses 95,823 physical qubits to achieve the same logical error rate. This demonstrates that the [[8064,338,10]] subsystem semi-hyperbolic code requires $5.1\times$ fewer resources to achieve the same level of protection that the subsystem toric code would provide at a physical error rate of 0.2%.

We also compare the [[8064,338,10]] subsystem semi-hyperbolic code with the rotated surface code, which is the leading candidate for realising fault-tolerant quantum computation, and has the optimal ratio $n/d^2 = 1$ for surface codes [24].

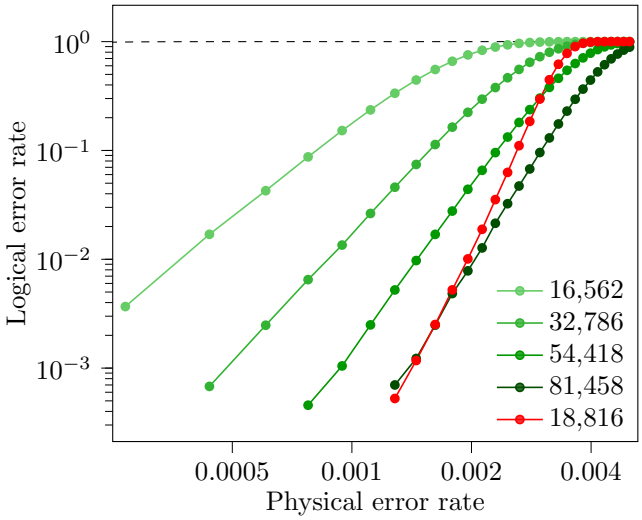



**Figure 6.5:** Comparison of the [[8064,338,10]] l=2 $\{8,4\}$ subsystem semi-hyperbolic code (red), with $L = 5,7,9$ and 11 rotated surface codes (shades of green), using a $(ZX)^{20}$ schedule (no gauge fixing) for the subsystem semi-hyperbolic code and a $(ZX)^{16}$ schedule for the rotated surface codes (both schedules require 80 time steps). We use a circuit-level depolarising noise model. We fix the number of logical qubits by plotting the probability that at least one of 338 independent copies of the rotated surface code suffers a logical $Z$ failure. The legend gives the total number of qubits (ancilla and data qubits) used. The black dashed line is the probability that at least one of 338 physical qubits would suffer a $Z$ failure without error correction over the same duration.

This comparison is shown in Figure 6.5, where we again keep the number of logical qubits fixed by using 338 independent copies of the rotated surface codes. At a circuit-level depolarising error rate of 0.15%, the subsystem semi-hyperbolic code, using 18,816 physical qubits, has a similar performance to $L = 11$ rotated surface codes using 81,458 physical qubits, a 4.3× reduction in qubit overhead. We also compare the performance of the [[8064,338,10]] subsystem semi-hyperbolic code with a distance 6 rotated surface code, which has a slightly lower encoding rate (including ancillas), and find that the subsystem semi-hyperbolic code has a lower logical error rate below 0.43%.

Furthermore, we can use schedule-induced gauge fixing for the subsystem hyperbolic and semi-hyperbolic codes just as we did for the subsystem toric code. In Figure 6.6 we plot the threshold of the $l = 2$, $\{8,4\}$ subsystem semi-hyperbolic codes under the independent circuit-level noise model using an $X$ schedule, and find a threshold of at least 2.4%, exceeding that of the subsystem toric code (2.22%).

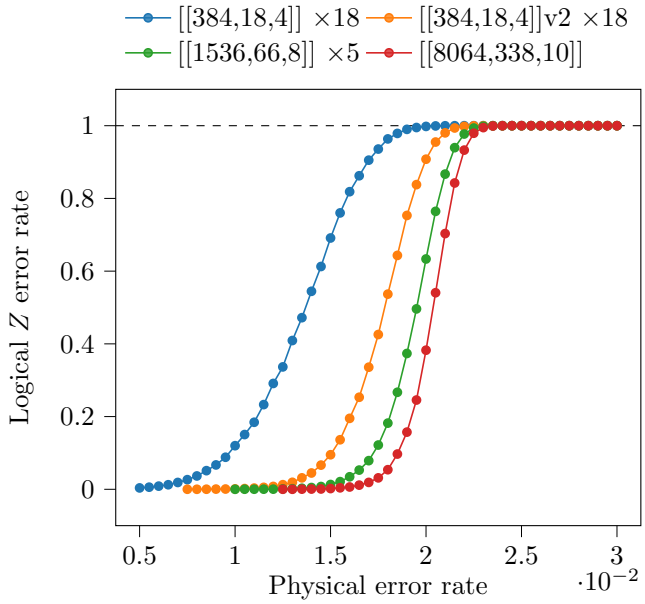



**Figure 6.6:** Performance of the extremal subsystem $\{8,4\}$ $l = 2$ semi-hyperbolic codes under a circuit-level independent noise model and using an $X$ schedule.

This threshold sets an upper bound on the thresholds that can be achieved using gauge fixing under biased noise models, and we expect that large gains can still be found even for small finite bias, as we found for the subsystem toric codes.

## 6.6 Conclusion

In this chapter, we introduced new constructions of quantum error correcting codes that improve upon the resource overhead of the widely-studied surface code. While the surface code requires four-qubit measurements and encodes a single logical qubit, we introduce families of quantum error correcting codes that use only three-qubit measurements and encode a number of logical qubits $k$ proportional to the number of physical qubits $n$.

A drawback of subsystem codes is that they typically have a smaller encoding rate $k/n$ compared to their subspace counterparts. To address this issue, we generalised the subsystem surface code to surfaces with negative curvature, constructing families of quantum LDPC subsystem codes with a finite encoding rate and only three-qubit check operators. We call these codes subsystem hyperbolic and subsystem semi-hyperbolic codes, and show how the symmetry group of the tiling can be used to construct check operator measurement circuits which require only four time

steps to implement. Thanks to the weight-three check operators, these measurement circuits allow us to correct up to the full code distance fault-tolerantly.

By simulating the performance of subsystem semi-hyperbolic codes under circuit-level depolarising noise, we find that they can require $4.3\times$ fewer physical qubits than the rotated surface code and $5.1\times$ fewer physical qubits than the subsystem toric code to achieve the same physical error rate at around 0.15% to 0.2%. Furthermore, these subsystem semi-hyperbolic codes belong to a family of codes that achieve distance scaling as $\sqrt{n}$, and that we expect to maintain a reduced qubit overhead relative to the surface code even at higher distances. These codes are also *locally* Euclidean, which is encouraging for the prospect of physical implementations in modular architectures [153, 125, 130].

We have also found a threshold of 0.42% for the subsystem semi-hyperbolic codes under a circuit-level depolarising noise. All of the techniques for schedule induced gauge-fixing that applied to the subsystem toric code can also be applied to subsystem semi-hyperbolic codes, and we find a threshold of 2.4% under infinite bias, exceeding that of the subsystem toric code.

Our work in this chapter has focussed on reducing the qubit overhead of quantum error correction, however reducing the time overhead of implementing logical gates is also an important problem. In Ref. [38] it was shown how lattice surgery and Dehn twists can be used to implement logical gates in hyperbolic codes. While these techniques should generalise straightforwardly to the subsystem hyperbolic codes we have introduced, in the future it would be interesting to compare the time overhead of these methods with those used for surface codes, as well as to investigate alternative methods for implementing fault-tolerant logical operations.

A key advantage of weight-three gauge operators is that they can be helpful for handling leakage errors [46], and direct three-qubit parity check measurements have been proposed in Ref. [66]. Since the average degree of the interaction graph is lower than the surface code, we also expect these codes to suffer from fewer frequency collisions and less crosstalk than the surface code in superconducting qubit architectures [48]. On the other hand, if high-weight stabiliser measurements

are available in hardware, then it may be possible to reduce the qubit overhead of our subsystem codes even further (likely at the cost of a lower threshold) by using a single ancilla qubit per stabiliser rather than per gauge operator, and measuring along the gauge operators to retain bare-ancilla fault-tolerance [136].

## Chapter 7

# Hyperbolic Floquet codes

In this chapter, we construct families of finite-rate codes derived from hyperbolic tilings that have weight-*two* check operators, a further reduction in check weight relative to the weight-three check operators of the subsystem hyperbolic codes introduced in the previous chapter. Our constructions are from the broader family of Floquet codes [102], which can be seen as a generalisation of subsystem codes, see e.g. Ref. [188] for a discussion. Similar to subsystem codes, Floquet codes involve the measurement of low-weight anti-commuting check operators. Unlike subsystem codes, however, Floquet codes do not require that the logical operators of the code remain static over time. Indeed, Floquet codes do not admit a static set of generators for all their logical Pauli operators, and the form of these logical operators instead evolves periodically during the syndrome extraction circuit. This additional flexibility leads to codes with weight-two check operators and good performance [97], especially in platforms that support direct two-qubit Pauli measurements, such as Majorana-based qubits [156]. Notably, the planar honeycomb code [100, 156, 95] is a specific Floquet code derived from a hexagonal lattice with open boundary conditions, and is therefore amenable to experimental realisation on a quantum computer chip (e.g. a solid state device such as a superconducting qubit architecture).

In Ref. [198], Vuillot showed how Floquet codes can be derived from any tiling suitable to define a colour code [25], and furthermore observed that Floquet codes derived from colour code tilings of closed hyperbolic surfaces, which we refer to as *hyperbolic Floquet codes*, would have a constant encoding rate and logarithmic

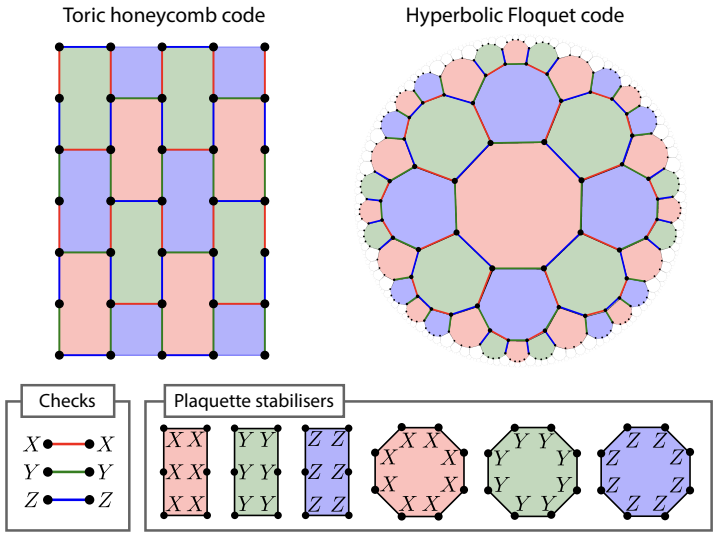



**Figure 7.1:** Two of the families of Floquet codes studied in this chapter. A data qubit is associated with each vertex of a lattice, a two-qubit check operator is associated with each edge and a plaquette stabiliser generator is associated with each face. Left: A toric honeycomb code, which is defined on a hexagonal lattice with periodic boundary conditions and three-colourable faces [102]. Here we consider a patch with 4 columns of data qubits and 6 rows (dimensions $4 \times 6$). Opposite sides of the lattice are identified. Right: A Floquet code derived from a tiling of a hyperbolic surface with three octogans meeting at each vertex (vertex configuration 8.8.8) and three-colourable faces. Here we draw a region of the tiled hyperbolic plane defining the code family, however each Floquet code in the family is derived from a tiling of a closed surface.

distance. However, there has not been prior work constructing explicit examples of hyperbolic Floquet codes or analysing their performance.

In this chapter, we construct explicit families of hyperbolic Floquet codes (see Figure 7.1) and analyse their performance numerically, comparing them to planar honeycomb codes and surface codes. We also construct Floquet codes derived from semi-hyperbolic lattices, which fine-grain hyperbolic tilings, leading to an improved distance scaling that enables exponential suppression of errors while retaining an advantage over honeycomb and surface codes. Furthermore, the thresholds of families of semi-hyperbolic Floquet codes are essentially the same as the threshold of the honeycomb code. All of our constructions have weight-two check operators and can be decoded efficiently using surface code decoders, such as minimum-weight perfect matching [64, 80, 113, 207].

For platforms that support direct two-qubit measurements, we find that semi-hyperbolic Floquet codes can require 48× fewer physical qubits than honeycomb codes even for high physical error rates of 0.3% to 1% (for the 'EM3' noise model) and for a system size of 21,504 physical qubits. These results imply that our constructions are over 100× more efficient than alternative compilations of the surface code to two-qubit measurements [49, 88], which have been shown to be less efficient than honeycomb codes for the same noise model [156, 88]. We also show that semi-hyperbolic Floquet codes can require as few as 32 physical qubits per logical qubit to achieve logical failure rates below $10^{-12}$ per logical qubit at a physical error rate of 0.1%, whereas honeycomb codes instead require 600 [97] to 2000 [156] physical qubits per logical qubit in the same regime.

We also consider a standard circuit-level depolarising noise model ("SD6"), for which two-qubit measurements are implemented using an ancilla, CNOT gates and single-qubit rotations. For this noise model, our semi-hyperbolic Floquet codes are 30× more efficient than planar honeycomb codes and over 5.6× more efficient than conventional surface codes for physical error rates of around 0.1% and below.

Finally, we construct small examples of hyperbolic Floquet codes that are amenable to near-term experiments. This includes codes derived from the Bolza surface, using as few as 16 physical qubits, and which we show are 3× to 6× more efficient than their Euclidean counterparts.

Although these (semi-)hyperbolic Floquet codes cannot be implemented using geometrically local connections in a planar Euclidean architecture, we show how they can instead be implemented using a *bilayer* or *modular* architecture. A bilayer architecture uses two layers of qubits, where connections within each layer do not cross, but may be long-range. A modular architecture consists of many small modules, where each module only requires local 2D planar Euclidean connectivity and connections between modules may be long-range. The long-range connections between modules could be mediated via photonic links in a trapped-ion architecture [150], for example.

This chapter is organised as follows. We start by reviewing Euclidean, hyper-

bolic and semi-hyperbolic colour code tilings in Section 7.1. In Section 7.2 we review Floquet codes, including how we construct Floquet code circuits from the colour code tilings of Section 7.1, as well as how these circuits can be decoded. Section 7.2 also describes the EM3 and SD6 noise models we use in our simulations. In Section 7.3 we present an analysis of the (semi-)hyperbolic Floquet codes we have constructed, including a study of their parameters (Section 7.3.1) and simulations comparing them to honeycomb and surface codes (Section 7.3.2). We then conclude in Section 7.4 with a summary of our results and a discussion of future work. See Ref. [111] for the preprint where most of the original research presented in this chapter has previously been published.

## 7.1 Colour code tilings

Floquet codes [102] can be defined on any tiling of a surface that is also suitable to define a 2D colour code [25]; namely, it is sufficient that the faces are 3-colourable and the vertices are 3-valent [198]. We will refer to such a tiling as a *colour code tiling*. In this section we will review colour code tilings and describe the tilings used to construct Floquet codes in this chapter.

We denote by $\mathcal{T} = (V, E, F)$ a colour code tiling with vertices $V$, edges $E \subset V^2$ and faces $F \subset 2^V$. Each face $f \in F$ is assigned a colour $C(f) \in \{R, G, B\}$ which can be red, green or blue ($R$, $G$ or $B$), such that two faces that share an edge have different colours:

$$f_1 \cap f_2 \in E \implies C(f_1) \neq C(f_2), \quad \forall (f_1, f_2) \in F^2. \tag{7.1}$$

We will also assign a colour $C(e) \in \{R, G, B\}$ to each edge $e$, given by the colour of the faces that it links. Equivalently, the colour of an edge is the complement of the colours of the two faces it borders. See Figure 7.1 for two examples of colour code tilings.

We now introduce some definitions which are applicable to any tiling $\mathcal{T} = (V, E, F)$ of a closed surface, including tilings that are not colour code tilings. The *dual* of $\mathcal{T}$, which we denote $\mathcal{T}^* = (V^*, E^*, F^*)$, has a vertex for each face of $\mathcal{T}$ and

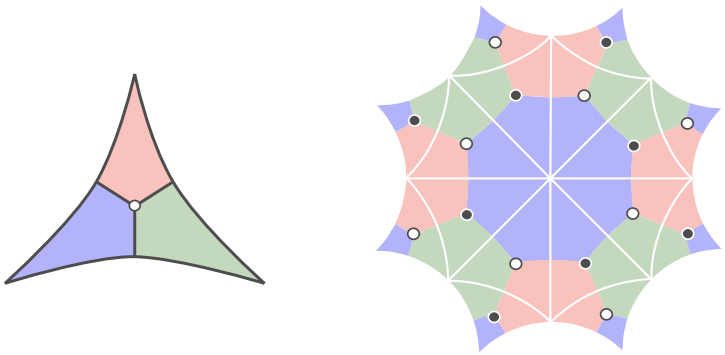


**Figure 7.2:** Construction of colour code tilings via the Wythoff construction. Left: Fundamental triangle with internal angles $\pi/p$, $\pi/q$ and $\pi/t$ belonging to the hyperbolic triangle group $\Delta(p,q,t)$. A single vertex is placed in the middle which is connected to the mid-point of the sides via half-edges. Right: The fundamental domain of the group $\Gamma_B$ defining the Bolza surface $\mathbb{H}^2/\Gamma_B$. In the figure opposite sides are identified, so that $\dim H_1(\mathbb{H}^2/\Gamma_B) = 4$. The Bolza surface supports the three-colourable 8.8.8 tiling. Neighbouring fundamental triangles of the tiling are related by a reflection along the triangle's side. The vertices in the middle of each fundamental triangle swap between black and white with each reflection.

two vertices in $\mathcal{T}^*$ are connected by an edge if the corresponding faces in $\mathcal{T}$ share an edge. A *cycle* in $\mathcal{T}$ is a set of edges (a subset of $E$) that forms a collection of closed paths in $\mathcal{T}$ (it has no boundary). A *co-cycle* in $\mathcal{T}$ is a set of edges (also a subset of $E$) that corresponds to a cycle in $\mathcal{T}^*$.

### 7.1.1 Hyperbolic colour code tilings

Suitable hyperbolic colour code tilings can be obtained directly using the Wythoff construction, as described in Section 5.2.1.2. This approach was used in introduced in Ref. [199] to define hyperbolic colour codes and pin codes. We obtain the desired colour code tiling from a triangle group $\Delta(p,q,t)$ by placing the generator point in the interior of the fundamental triangle, as described in Section 5.2.1.2. If $1/p+1/q+1/t < 1$ then we obtain a colour code tiling of the hyperbolic plane. Furthermore, we consider finite (compactified) tilings by finding an appropriate normal subgroup $\Gamma \subset \Delta(p,q,t)$ and constructing the quotient group $\Delta(p,q,t)/\Gamma$ (see Section 5.2.2). Note that when $\Gamma \subset \Delta(p,q,t)$ only contains elements consisting of an even number of reflections, the resulting surface is orientable and the graph of the tiling is bipartite. This is because every vertex is labelled by an element of the

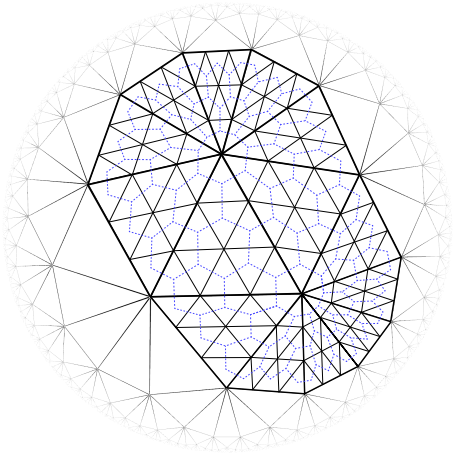


**Figure 7.3:** Obtaining a semi-hyperbolic colour code tiling by fine-graining a $3^8$ tiling and then taking its dual. We fine-grain by tiling each face of the $3^8$ tiling with a triangular lattice, such that each side of each face in the $3^8$ tiling becomes subdivided into $l$ edges (shown for a subset of the faces here, with $l = 3$). We take the dual of this lattice (blue dashed lines) to obtain a semi-hyperbolic colour code tiling of hexagons and octagons.

reflection group $h \in \Delta(p,q,t)/\Gamma$ and, assuming that $\Gamma$ contains only even parity elements, the assignment of a parity to each $h$ is well-defined. See Figure 7.2, where the parity of reflections of vertices are given by the colour black and white.

### 7.1.2 Properties of uniform tilings

Since we only consider colour code tilings, we can equivalently define $r$, $g$ and $b$ to be the number of sides that red, green and blue faces have in the uniform tiling, respectively. We construct Floquet codes from 6.6.6, 8.8.8, 4.8.10 and 4.10.10 hyperbolic tilings in this chapter; see Figure 7.1 for an illustration of 6.6.6 and 8.8.8 tilings.

The derived Floquet code will encode $k = \dim H_1$ logical qubits into $n = |V|$ physical qubits. Hence, Equation (5.8) implies that this family of codes has finite rate and that the proportionality depends on the plaquette stabiliser weights $r$, $g$, $b$.

### 7.1.3 Semi-hyperbolic colour code tilings

We also construct *semi-hyperbolic* colour code tilings, which interpolate between hyperbolic and Euclidean tilings, using a fine-graining procedure. Our semi-hyperbolic colour code tilings are inspired by the semi-hyperbolic tilings introduced in Ref. [39],

which used a slightly different method of fine-graining applied to 4.4.4.4.4 tilings (instead of colour code tilings) to improve the distance-scaling of standard hyperbolic surface codes. We used a similar method in Chapter 6 to construct subsystem semi-hyperbolic codes [112]. As we will show in Section 7.3, these tilings can be used to obtain Floquet codes with improved distance scaling relative to those derived from purely hyperbolic tilings, while retaining an advantage over honeycomb codes.

A semi-hyperbolic colour code tiling $\mathcal{T}_l = (V_l, E_l, F_l)$ is defined from a *seed* colour code tiling $\mathcal{T} = (V, E, F)$ as well as a parameter $l$, which determines the amount of fine-graining. To construct $\mathcal{T}_l$ we first take the dual of the seed tiling $\mathcal{T}$, which we recall we denote by $\mathcal{T}^* = (V^*, E^*, F^*)$. Since $\mathcal{T}$ is a colour code tiling and therefore 3-valent, all the faces of $\mathcal{T}^*$ are triangles, and we have that $|V^*| = |F|$, $|E^*| = |E|$ and $|F^*| = |V|$. If $\mathcal{T}$ is an 8.8.8 (also denoted $8^3$) tiling then $\mathcal{T}^*$ is a $3^8$ uniform tiling, i.e. with 8 triangles meeting at each vertex. We construct a new tiling $\mathcal{T}_l^*$ which fine-grains $\mathcal{T}^*$ by tiling each face of $\mathcal{T}^*$ with a triangular lattice, such that each edge in $\mathcal{T}^*$ is subdivided into $l$ edges in $\mathcal{T}_l^*$. Finally, we take the dual of $\mathcal{T}_l^*$ to obtain our semi-hyperbolic colour code tiling $\mathcal{T}_l$. In Figure 7.3 we give an example of this procedure starting from $\mathcal{T}^*$ (here a $3^8$ tiling) using $l = 3$ to obtain a colour code tiling of hexagons and octagons.

Given an *r.g.b* uniform colour code tiling $\mathcal{T}$, the semi-hyperbolic colour code tiling $\mathcal{T}_l$ derived from it using this procedure has a number of vertices, edges and faces given by:

$$|V_l| = l^2 |V|, \tag{7.2}$$

$$|E_l| = \frac{3l^2|V|}{2}, \tag{7.3}$$

$$|F_l| = \left(\frac{l^2}{2} - \frac{1}{2} + \frac{1}{r} + \frac{1}{g} + \frac{1}{b}\right)|V|. \tag{7.4}$$

Since we have not modified the topology of the surface, the dimension of the first homology group $\dim H_1 = 2 - |V_l| + |E_l| - |F_l|$ is independent of $l$ and still given by Equation (5.8).

## 7.2 Floquet codes

Floquet codes, introduced by Hastings and Haah [102], are quantum error correcting codes that are implemented by measuring two-qubit Pauli operators (they have weight-two checks). In Ref. [102] the authors introduced and studied the honeycomb code, which is a Floquet code defined from a hexagonal tiling of a torus, inspired by Kitaev's honeycomb model [128]. Shortly after, Vuillot showed that Floquet codes can be defined from any colour code tiling, and observed that the use of hyperbolic colour code tilings would lead to Floquet codes with a finite encoding rate and logarithmic distance [198].

We will focus our attention on Floquet codes derived from colour code tilings of *closed* surfaces. It is also possible to construct planar Floquet codes; however, introducing boundaries requires a modification to the measurement schedule and observables, see Refs. [100, 156, 95]. The definition of Floquet codes we use is consistent with Refs. [198, 97], which is related to the original definition of Ref. [102] by a local Clifford unitary.

### 7.2.1 Checks and stabilisers

We use the colour code tiling $\mathcal{T}$ to define a Floquet code, which we will denote by $\mathcal{F}(\mathcal{T})$. Each vertex in $\mathcal{T}$ represents a qubit in $\mathcal{F}(\mathcal{T})$ and each edge in $\mathcal{T}$ represents a two-qubit check operator. There are three types of check operators: $X$-checks (red edges), $Y$-checks (green edges) and $Z$-checks (blue edges). Specifically, the check operator for an edge $e = (v_1, v_2)$ is defined to be $P_e := P_{v_1}^{C(e)} P_{v_2}^{C(e)}$, where here $P_q^c$ denotes a Pauli operator acting on qubit $q$ labelled by the colour $c \in \{R, G, B\}$ which determines the Pauli type. i.e. $P_q^B$, $P_q^G$ and $P_q^R$ denote Pauli operators $X_q$, $Y_q$ and $Z_q$, respectively. For the Floquet codes we consider here, the Pauli type of an edge is fixed and is determined by its colour, and we will sometimes refer to checks of a given colour as $c$-checks, for $c \in \{R, G, B\}$. However, we note that alternative variants of Floquet codes have been introduced for which the Pauli type of an edge measurement is allowed to vary in the schedule (e.g. Floquet colour codes [126]).

Each stabiliser generator of the Floquet code consists of a cycle of checks surrounding a face. In other words we associate a stabiliser generator $P_f := \prod_{e \subset f} P_e$

with each face $f$. Stabiliser generators on red, green and blue faces are $X$-type, $Y$-type and $Z$-type respectively. We sometimes refer to these stabiliser generators as the *plaquette stabilisers* for clarity. Note that the plaquette stabilisers commute with all the checks. See Figure 7.1 for examples of the checks and stabilizers for Floquet codes defined on hexagonal (6.6.6) and hyperbolic (8.8.8) lattices.

### 7.2.2 The schedule and instantaneous stabiliser group

The Floquet code is implemented by repeating a *round* of measurements, where each round consists of measuring all red checks, then all green checks and finally all blue checks. We refer to the measurement of all checks of a given colour as a *sub-round* (so each round contains three sub-rounds).

Once the steady state is reached, after fault-tolerant initialisation of the Floquet code, the state is always in the joint $+1$-eigenspace of the plaquette stabilisers defined in Section 7.2.1, regardless of which sub-round was just measured. This property allows measurements of the plaquette stabilisers to be used to define detectors for decoding (see Section 7.2.7). However, the full stabiliser group of the code at a given instant includes additional stabilisers corresponding to the check operators measured in the most-recent sub-round. We refer to the full stabiliser group of the code after a given sub-round as the *instantaneous stabiliser group* (ISG) [102]. In the steady state of the code, just after measuring a sub-round of checks with colour $c \in R, G, B$, the ISG is generated by all the plaquette stabilisers, as well as the check operators of colour $c$. The phase of each check operator in the ISG is given by its measurement outcome in the most-recent sub-round. We will sometimes refer to the ISG immediately after measuring $c$-checks as $\mathrm{ISG}(c)$.

### 7.2.3 The embedded homological code

The state of a Floquet code can be mapped by a constant-depth unitary circuit to a 2D homological code, which we refer to as the embedded homological code. This mapping is shown in Ref. [102] for the honeycomb code and in Ref. [198] for general colour code tilings. The embedded homological code can be useful to understand and construct the logical operators of the code, as we will explain in Section 7.2.4. As an

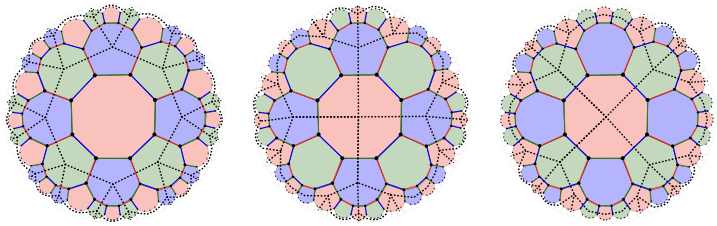


**Figure 7.4:** The dashed lines in the left, middle and right diagrams of the 8.8.8 tiling show the red ($\mathcal{T}_R^*$), green ($\mathcal{T}_G^*$) and blue ($\mathcal{T}_B^*$) restricted lattices, respectively. The restricted lattice $\mathcal{T}_c^*$ of colour $c \in \{R,G,B\}$ defines the embedded homological code after the $c$-checks are measured. Each edge, face and vertex in the restricted lattice corresponds to an effective qubit, plaquette stabiliser or site stabiliser in the embedded homological code, respectively. All three restricted lattices are regular tessellations, with four octagons meeting at each vertex.

example, consider the state of the Floquet code immediately after a red sub-round, during which any red edge $e = (u,v)$ has participated in an $X_uX_v$ measurement. This $X_uX_v$ measurement projects the qubits $u$ and $v$ into a two-dimensional subspace, which we can consider an effective qubit in an embedded 2D homological code. The effective $X$ and $Z$ operators associated with this effective qubit are defined to be $\hat{X}_e := Y_uY_v$ (or $Z_uZ_v$) and $\hat{Z}_e := X_uI_v$ (or $I_uX_v$), respectively.

More concretely, we denote by $\mathcal{T}^*$ the dual of the colour code tiling $\mathcal{T}$, where we define a vertex in $\mathcal{T}^*$ for each face of $\mathcal{T}$, and two nodes in $\mathcal{T}^*$ are connected by an edge iff their corresponding faces in $\mathcal{T}$ share an edge. Furthermore, the colour of a node in $\mathcal{T}^*$ is given by the colour of the corresponding face in $\mathcal{T}$. We then define the restricted lattice $\mathcal{T}_c^*$ for $c \in \{R,G,B\}$ to be the subgraph of $\mathcal{T}^*$ where all nodes of colour $c$ (and their adjacent edges) have been removed. We refer to $\mathcal{T}_R^*$ as the red restricted lattice (and similarly for green and blue). See Figure 7.4 for examples of the red, green and blue restricted lattices for an 8.8.8 hyperbolic tiling.

The embedded 2D homological code $\mathcal{C}(\mathcal{T}_c^*)$ associated with the Floquet code after measuring checks of colour $c$ is defined from $\mathcal{T}_c^*$ by associating qubits with the edges, $Z$-checks with the faces and $X$-checks with the vertices. An effective $\hat{X}$ operator on an effective qubit in $\mathcal{T}_c^*$ corresponds to a physical $P^c \otimes I$ operator on the Floquet code data qubits and similarly an effective $\hat{Z}$ operator corresponds to a physical $P^{\bar{c}} \otimes P^{\bar{c}}$ operator, where here $\bar{c} \in \{R,G,B\} \setminus \{c\}$. As an example of this

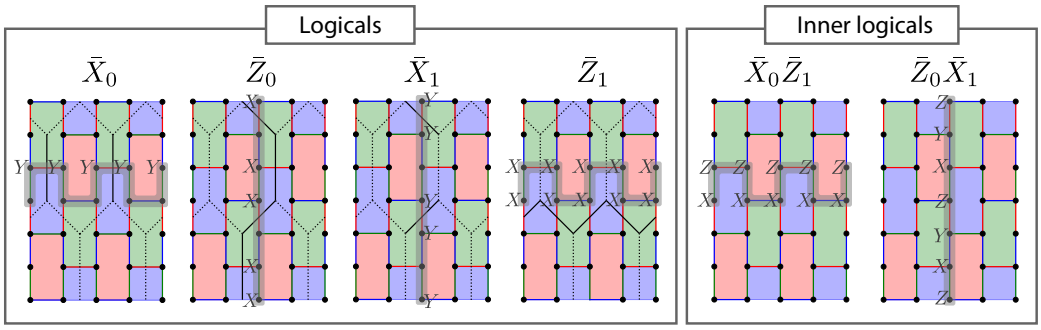



**Figure 7.5:** Left: Logical operators of the two logical qubits of the toric honeycomb code after measuring the red checks. These representatives of the logical operators commute with the round of green checks that follows. The overlaid black lattice (with dotted and solid lines) is the tiling of the corresponding embedded toric code, which has an effective qubit associated with each edge. The grey highlighted region is the logical path of the corresponding logical operator. The nontrivial support of the logical operator lies along its logical path in all sub-rounds, even though the Pauli operator itself changes. Right: The inner logical operators, which are products of the logical operators shown on the left. Each inner logical is a homologically non-trivial path of check operators which commutes with all check operators.

mapping, notice that a red $X^{\otimes 8}$ plaquette stabiliser in Figure 7.4 corresponds to a $\hat{Z}^{\otimes 8}$ plaquette operator in $\mathcal{C}(\mathcal{T}_R^*)$ and to an $\hat{X}^{\otimes 4}$ site operator in $\mathcal{C}(\mathcal{T}_G^*)$ or $\mathcal{C}(\mathcal{T}_B^*)$.

### 7.2.4 Logical operators

We can derive a set of logical operators of a Floquet code from the logical operators of the embedded 2D homological code [102, 198], as shown in Figure 7.5 for the toric honeycomb code. An unusual property of Floquet codes is that the logical operators generally do not commute with all of the checks. For example, none of the logical operators shown in Figure 7.5 (left) commute with the blue checks. Fortunately we can still preserve the logical operators throughout the schedule by updating the logical operators after each sub-round, multiplying an element of the ISG into each of them, such that they commute with the next sub-round of check operator measurements. In this section we will give a suitable basis for the logical operators and show how they are updated in each sub-round to commute with the checks. This was explained in Ref. [97] for the toric honeycomb code (see their Figure 1) whereas our description here is applicable to any Floquet code derived from a colour code tiling of a closed surface (Euclidean or hyperbolic).

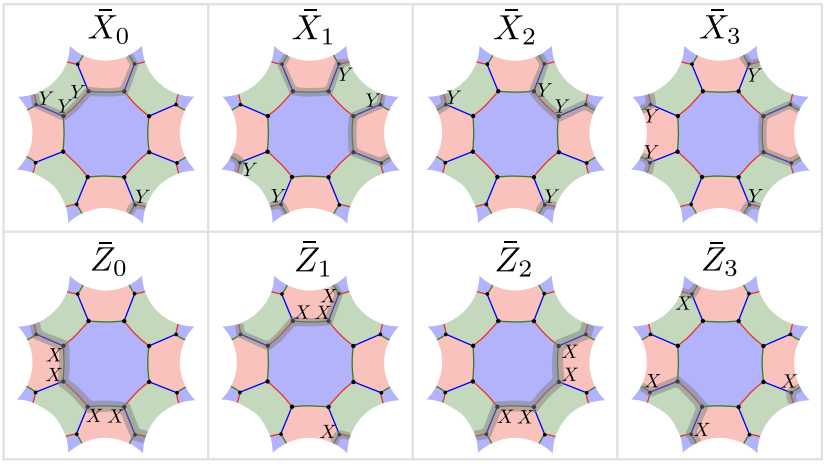



**Figure 7.6:** A symplectic basis for the logical operators of the hyperbolic Floquet code derived from the 8.8.8 tiling of the Bolza surface, which has genus 2 and encodes 4 logical qubits into 16 physical data qubits. Opposite sides of the tiling are identified. The logical $\bar{X}$ and $\bar{Z}$ operators of logical qubit $i$ are denoted by $\bar{X}_i$ and $\bar{Z}_i$, respectively. For each logical, the grey highlighted path is its associated homologically non-trivial logical path, which defines how the logical is updated in each sub-round (see Figure 7.7).

We will define a basis for the logical operators of a Floquet code $\mathcal{F}(\mathcal{T})$ from the logical operators of $\mathcal{C}(\mathcal{T}_R^*)$, the embedded homological code immediately after a red sub-round. After the first red sub-round of $\mathcal{F}(\mathcal{T})$, each $\bar{X}_i$ logical of $\mathcal{F}(\mathcal{T})$ is defined to be an $\bar{X}_i$ logical of $\mathcal{C}(\mathcal{T}_R^*)$ (a homologically non-trivial co-cycle), and each $\bar{Z}_i$ logical of $\mathcal{F}(\mathcal{T})$ is defined to be an $\bar{Z}_i$ logical of $\mathcal{C}(\mathcal{T}_R^*)$ (a homologically non-trivial cycle). See Figure 7.5 (left) for an example using the toric honeycomb code. With this choice of logical basis, none of the representatives of the $\bar{X}_i$ or $\bar{Z}_i$ operators commute with all the check operators, so they are what Ref. [102] refers to as *outer logical operators*.

We need representatives of logical $\bar{X}_i$ and $\bar{Z}_i$ operators that commute with the green sub-round that immediately follows, however the definition we have just given does not guarantee this property. This is because our choice of logicals so far is only defined up to an arbitrary element of $\mathrm{ISG}(R)$, which is generated by plaquette stabilisers as well as red checks, and the red checks do not generally commute with the green checks. We will now show how we can choose a representative for each $\bar{X}_i$ and $\bar{Z}_i$ operator that is guaranteed to commute with the subsequent green checks.

For each logical $\bar{X}_i$ or $\bar{Z}_i$ operator we associate a *logical path* $P(\bar{X}_i)$ or $P(\bar{Z}_i)$ (highlighted in grey in Figure 7.5 and Figure 7.6), a homologically non-trivial cycle on $\mathcal{T}$ that includes the nontrivial support of the logical operator. We find each logical path using a procedure that lifts the homologically non-trivial co-cycle (for $\bar{X}_i$) or cycle (for $\bar{Z}_i$) of the logical operator from the restricted lattice to a homologically non-trivial cycle on the colour code tiling $\mathcal{T}$. We will use the red restricted lattice $\mathcal{T}_R^*$ as an example. We can lift a *co-cycle* $Q$ in $\mathcal{T}_R^*$ to a logical path $P(X_i)$ in $\mathcal{T}$ by choosing a path that passes only through the red edges that $Q$ crosses and around the borders of red faces. We can lift a *cycle* $W$ in $\mathcal{T}_R^*$ to a logical path $P(Z_i)$ in $\mathcal{T}$ by finding a homologically non-trivial path in $\mathcal{T}$ that passes only through the borders of the blue and green faces in $\mathcal{T}$ associated with the nodes in $W$. Examples are shown in Figure 7.5. We choose the representative of each $\bar{X}_i$ to be the Pauli operator that acts as $Y$ on the endpoints of red edges that lie within its logical path $P(\bar{X}_i)$ and acts trivially on all other qubits. For each $\bar{Z}_i$, we choose its representative to be the Pauli operator that acts as $X$ on the endpoints of green edges that lie within the logical path $P(\bar{Z}_i)$ (and trivially elsewhere). It is straightforward to verify that these are valid representatives of the logical operators and that they always commute with the green check operator measurements in the sub-round that immediately follows. Note that for each of the logical paths $P(\bar{X}_i)$ and $P(\bar{Z}_i)$ we could have picked *any* homologically non-trivial cycle belonging to the same homology class (the product of the Pauli operators obtained from two such choices belonging to the same homology class is in the ISG). The use of the restricted lattice is helpful to understand the logical operators and paths but is not especially helpeful for constructing them; we can instead find homologically non-trivial cycles on the colour code lattice directly. Logical operators and paths for the toric honeycomb code are shown on the left of Figure 7.5. Similarly, logical operators and paths of the hyperbolic Floquet code derived from the Bolza surface (the Bolza Floquet code) are shown in Figure 7.6.

We have so far only described the initial representatives of the logical operators after the first red sub-round such that they commute with the subsequent green sub-round, however we will now explain how the logical operators are updated after each

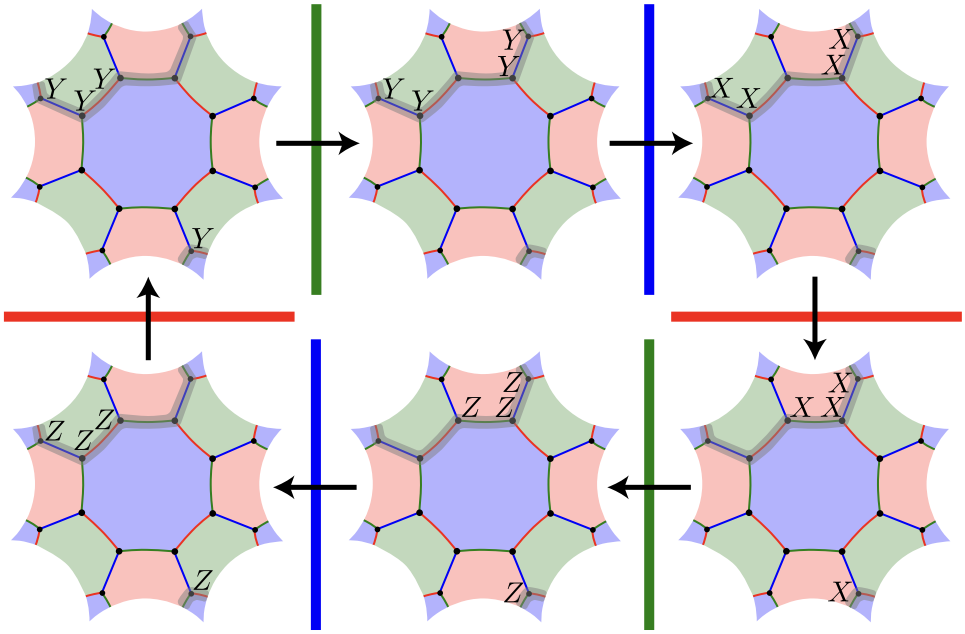



**Figure 7.7:** Each logical operator changes over a periodic sequence of six sub-rounds (two full rounds), as shown here for a logical operator ($\bar{X}_0$ in Figure 7.6) of the Bolza Floquet code. Opposite sides of the tiling are identified. Crossing a bar of colour $c \in \{R, G, B\}$ means measuring the $c$-checks and then multiplying into each logical operator the $c$-checks that lie within its logical path. Note that the $\bar{X}_0$ logical operator (top left) is mapped to the $\bar{Z}_1$ logical operator (bottom right) every three sub-rounds and vice versa. This transition is an automorphism of $\mathrm{ISG}(R)$ which acts on each logical operator by multiplying by the inner logical operator that lies along the same logical path.

sub-round. After measuring the checks of colour $c \in \{R, G, B\}$, we multiply into each logical operator the $c$-coloured checks that lie within its logical path. Updating the choice of representative of the logical operator this way in each sub-round, we guarantee that it will commute with the sub-round of checks that immediately follows. This can be understood by noticing how the form of the logical operators changes over a period of six sub-rounds (2 full rounds):

| $r \pmod 6$ | ISG | Form of $\bar{X}_i$ on $P(\bar{X}_i)$ | $\bar{X}_i$ in $\mathcal{T}_c^*$ | Form of $\bar{Z}_i$ on $P(\bar{Z}_i)$ | $\bar{Z}_i$ in $\mathcal{T}_c^*$ |
|---|---|---|---|---|---|
| 0 | $R$ | $Y$ on red edges | co-cycle in $\mathcal{T}_R^*$ | $X$ on green edges | cycle in $\mathcal{T}_R^*$ |
| 1 | $G$ | $Y$ on blue edges | cycle in $\mathcal{T}_G^*$ | $Z$ on green edges | co-cycle in $\mathcal{T}_G^*$ |
| 2 | $B$ | $X$ on blue edges | co-cycle in $\mathcal{T}_B^*$ | $Z$ on red edges | cycle in $\mathcal{T}_B^*$ |
| 3 | $R$ | $X$ on green edges | cycle in $\mathcal{T}_R^*$ | $Y$ on red edges | co-cycle in $\mathcal{T}_R^*$ |
| 4 | $G$ | $Z$ on green edges | co-cycle in $\mathcal{T}_G^*$ | $Y$ on blue edges | cycle in $\mathcal{T}_G^*$ |
| 5 | $B$ | $Z$ on red edges | cycle in $\mathcal{T}_B^*$ | $X$ on blue edges | co-cycle in $\mathcal{T}_B^*$ |

where here each row describes the form of the logical operators on $\mathcal{T}$ and $\mathcal{T}_c^*$ after a given sub-round $r$, which depends on $r \pmod 6$. See Figure 7.7 for the cycle of a logical operator in the Bolza Floquet code, as well as Figure 1 of Ref. [97] for a similar figure using the honeycomb code. An $\bar{X}$-type logical operator in $\mathrm{ISG}(R)$ is mapped to a $\bar{Z}$-type logical operator in $\mathrm{ISG}(R)$ (and vice versa) every three sub-rounds by an automorphism of $\mathrm{ISG}(R)$. The same applies to $\mathrm{ISG}(G)$ and $\mathrm{ISG}(B)$. Since we must multiple checks into the logicals in each sub-round, the measurement of a logical operator in a Floquet code includes the product of measurement outcomes spanning a sheet in space-time, rather than just a string of measurements in the final round as done for transversal measurement in the surface code.

So far we have only consider outer logical operators, which move after each sub-round. However there are some logical operators, called *inner logical operators* in Ref. [102], which commute with all the checks. These inner logical operators act non-trivially on the encoded logical qubits of the Floquet code and are products of checks lying along homologically nontrivial cycles of the lattice. Each inner logical operator is formed from the product of an $\bar{X}$ and a $\bar{Z}$ logical operator associated with the same homologically nontrivial cycle of the lattice. For example, Figure 7.5 (right) shows the two inner logical operators, $\bar{X}_0\bar{Z}_1$ and $\bar{Z}_0\bar{X}_1$, in the toric honeycomb code. One of the four inner logical operator of the Bolza Floquet code can be formed from the product of the $\bar{X}_0$ and $\bar{Z}_1$ logical operators in Figure 7.6. However, even though the inner logical operators are formed from a product of check operators, the measurement schedule of the Floquet code is chosen such that it never measures the inner logical operators, only the plaquette stabilisers [102, 198].

### 7.2.5 Circuits and noise models

In this chapter we consider two different noise models, corresponding to the EM3 and SD6 noise models used in Ref. [95]. Both noise models are characterised by a noise strength parameter $p \in [0,1]$ (the physical error rate).

#### 7.2.5.1 EM3 noise model

The EM3 noise model (entangling measurement 3-step cycle) assumes that two-qubit Pauli measurements are available natively in the platform (and hence no ancilla qubits are needed), as is the case for Majorana-based architectures [49, 156]. When implementing the measurement of a two-qubit Pauli operator $P^c \otimes P^c$, with probability $p$ we insert an error chosen uniformly at random from the set $\{I,X,Y,Z\}^{\otimes 2} \times \{\text{flip}, \text{no flip}\}$. Here the two qubit Pauli error in $\{I,X,Y,Z\}^{\otimes 2}$ is applied immediately before the measurement, and the "flip" operation flips the outcome of the measurement. We assume each measurement takes a single time step, and hence each cycle of three sub-rounds takes three time steps. Initialisation of a qubit in the $Z$ basis is followed by an $X$ error, inserted with probability $p/2$. Similarly, the measurement of a data qubit in the $Z$ basis is preceded by an $X$ error, inserted with probability $p/2$. Single-qubit initialisation and measurement in the $X$ or $Y$ basis is achieved using noisy $Z$-basis initialisation or measurement and noiseless single-qubit Clifford gates. We do not need to define idling errors because data qubits are never idle in this noise model.

#### 7.2.5.2 SD6 noise model

The SD6 noise model (standard depolarizing 6-step cycle) facilitates the check measurements via an ancilla qubit for each edge of the colour code tiling. Therefore, this noise model introduces $|E| = 3|V|/2$ ancilla qubits and requires $2.5\times$ more qubits than the $EM3$ noise model for a given colour code tiling $\mathcal{T}$. Note that the distance in this noise model can be higher than $EM3$ for a given tiling though, so we do not necessarily require $2.5\times$ more qubits for a given distance.

Each measurement of a two-qubit check operator $P^c \otimes P^c$ is implemented using CNOT gates and single-qubit Clifford gates, as well as measurement and reset of the

ancilla qubit associated with the edge. We use the same circuit as in Ref. [97]. Each CNOT gate is followed by a two-qubit depolarising channel, which with probability $p$ inserts an error chosen uniformly at random from the set $\{I,X,Y,Z\}^{\otimes 2} \setminus \{I \otimes I\}$. Each single-qubit Clifford gates is followed by a single-qubit depolarising channel, which with probability $p$ inserts an error chosen uniformly at random from the set $\{X,Y,Z\}$. Single-qubit initialisation is followed by an $X$ error with probability $p$ and single-qubit measurement is preceded by an $X$ error with probability $p$. Each cycle of three sub-rounds takes six time steps, with CNOT gates, single-qubit gates, initialisation and measurement all taking one time step each. In each time step in the bulk, each data qubit is either involved in a CNOT gate or a $C_{ZYX} := HS$ gate (which maps $Z \to Y \to X \to Z$ under conjugation) and therefore never idles.

### 7.2.6 Implementing hyperbolic and semi-hyperbolic connectivity

Although hyperbolic and semi-hyperbolic Floquet codes cannot be implemented using geometrically local connections in a planar Euclidean layout of qubits, they can instead be implemented using *biplanar* or *modular* architectures.

In a biplanar architecture, the connections (couplers) between qubits can be partitioned into two layers, such that the couplers within each layer do not cross. More concretely, consider the qubit connectivity graph $G_c = (V_c, E_c)$, which by definition contains an edge $(q_i, q_j) \in E_c$ if and only if qubits $q_i$ and $q_j$ directly interacted in the quantum circuit implementing the code. Recall that the *thickness* of a graph is the minimum number of planar graphs that the graph's edge set can be partitioned into. A graph is biplanar if it has thickness 2; i.e. $G_c$ is biplanar if we can partition the edge set as $E_c = E_c^1 \cup E_c^2$, where the edge sets $E_c^1$ and $E_c^2$ each define a planar graph. It is known that the thickness of a graph with maximum degree $d$ has thickness at most $t = \lceil \frac{d}{2} \rceil$ (see Corollary 5 of Ref. [101], as well as Proposition 1 of Ref. [189] for this application to QEC). Therefore, since $G_c$ has maximum degree 3 for any (semi-)hyperbolic Floquet code, it must also be biplanar. Note that the subsystem hyperbolic codes of Ref. [112] are also biplanar, since their qubit connectivity graphs always have degree 4. Biplanarity of the connectivity graph implies that we can fix the position of the qubits and have two layers of connections

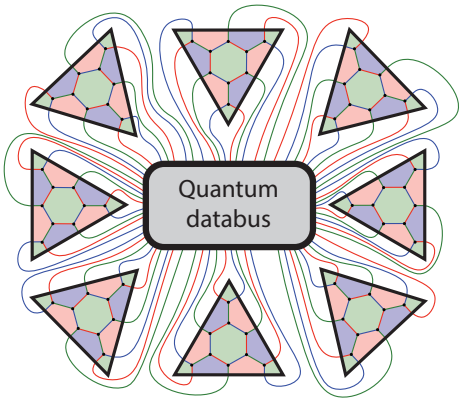



**Figure 7.8:** A modular architecture for a semi-hyperbolic Floquet code. Each module has the connectivity of a planar Euclidean chip, and arbitrary connectivity is permitted between modules. In this example, each module is the result of fine-graining the neighbourhood of one vertex of the seed tiling $\mathcal{T}$ used to construct an $l = 3$ semi-hyperbolic Floquet code $\mathcal{F}(\mathcal{T}_3)$. The 'quantum databus' here facilitates long-range two-qubit measurements between separate modules.

(e.g. above and below), each of which is planar. That both $E_c^1$ and $E_c^2$ can be planar with fixed qubit positions can be understood from Theorem 8 of Ref. [101], which shows that a planar graph always has a planar representation with nodes placed in arbitrary positions.

We can also implement (semi-)hyperbolic Floquet codes using a modular architecture [150], in which long-range links connect small modules of qubits. Examples of qubit platforms that could support modular architectures include ions [150, 178, 162], atoms [168, 20] or superconducting qubits [170, 190, 58, 211, 121]. We break up the Floquet code into small modules, where each module supports planar Euclidean connectivity and long-range two-qubit Pauli measurements are used to connect different modules. For a hyperbolic Floquet code, variable length connections can be used to embed a small region of the lattice into each module [130]. For a semi-hyperbolic Floquet code, local regions of the code have identical connectivity to a honeycomb code; hence, each module can consist of a small hexagonal lattice of qubits obtained from fine-graining the neighbourhood of one vertex of the seed tiling $\mathcal{T}$ that the semi-hyperbolic Floquet code $\mathcal{F}(\mathcal{T}_l)$ was derived from (see Figure 7.8). By connecting these modules using long-range two-qubit measurements, we can implement the connectivity required for the tiling of the closed hyperbolic surface.

### 7.2.7 Detectors and decoding

In order to decode Floquet codes, we must make an appropriate choice of *detectors*. Following the definition in the Stim [92] documentation, a detector is a parity of measurement outcomes in the circuit that is deterministic in the absence of errors. In the literature, detectors have also been referred to as error-sensitive events [183] or checks [27, 61]. In a Floquet code, each detector in the bulk of the schedule is formed by taking the parity of two consecutive measurements of a plaquette stabiliser. For example, in the honeycomb code, each plaquette stabiliser is the product of the six edge check measurements around a face, and so a detector is the parity of 12 measurement outcomes. For a Floquet code derived from an $r.g.b$ tiling, detectors are the parity of $2r$, $2g$ or $2b$ measurement outcomes. With this choice of detectors, it is known that any Floquet code can be decoded with minimum-weight perfect matching (MWPM) [64, 80, 113, 207] or Union-Find (UF) [60, 119] decoders [102, 198, 97] by defining a *detector graph* for a given noise model. Each node in the detector graph represents a detector and each edge represents an error mechanism that flips the detectors at its endpoints with probability $p$, and has an edge weight given by $\log((1-p)/p)$. Given this detector graph and an observed syndrome, MWPM or UF can be used to find a minimum-weight or low-weight set of edges consistent with the syndrome, respectively. Some additional accuracy can be achieved by also exploiting knowledge of hyperedge error mechanisms using a correlated matching [81] or belief-matching [115] decoder, however in this chapter we use the PyMatching implementation of MWPM [113] to decode and use Stim to construct the detector graphs and simulate the circuits [92].

## 7.3 Constructions

In this section we present a performance analysis of our constructions of hyperbolic and semi-hyperbolic codes, including a comparison with honeycomb codes and surface codes. Section 7.3.1 contains our analysis of the code parameters, whereas are simulations using EM3 and SD6 noise models are presented in Section 7.3.2.

### 7.3.1 Code parameters

The number of data qubits $n = |V|$ in a Floquet code is determined by the number of vertices in the colour code tiling $\mathcal{T} = (V, E, F)$ and the number of logical qubits is given by the dimension of its first homology group $k = \dim H_1$. From Equation (5.8) and Equation (7.2) we see that the encoding rate $k/n$ is determined by the plaquette stabiliser weights and the fine-graining parameter $l$. No ancilla qubits are required for the EM3 noise model ($n_{anc} = 0$), whereas for the SD6 noise model we have $n_{anc} = |E| = 3n/2$ ancillas. We denote the total number of physical qubits by $n_{tot} \coloneqq n + n_{anc}$.

The distance of a circuit implementing a Floquet code is the minimum number of error mechanisms required to flip at least one logical observable measurement outcome without flipping the outcome of any detectors. Therefore the distance depends not only on the colour code tiling used to define the Floquet code, but also the choice of circuit and noise model. We consider three different types of distance of a Floquet code. One of these is a "circuit agnostic" distance which we will refer to as the Floquet code's *embedded distance*. The embedded distance $d$ of a Floquet code is defined to be the smallest distance of any of its three embedded 2D homological codes (the smallest homologically non-trivial cycle or co-cycle in $\mathcal{T}_R^*$, $\mathcal{T}_G^*$ or $\mathcal{T}_B^*$). The embedded distance is more efficient to compute than the distance of a Floquet code circuit, since it involves shortest path searches on the 2D restricted lattices (using the method described in Appendix B of [39]), rather than on the much larger 3D detector graph associated with a specific circuit over many rounds (which we compute using `stim.Circuit.shortest_graphlike_error` in Stim [92]). We also consider the EM3 distance $d_e$ of a Floquet code, which is the graphlike distance of its circuit for an EM3 noise model. Empirically, we find that the EM3 distance of a Floquet code almost always matches its embedded distance (see Table 7.1), even though the embedded distance does not consider measurement errors or the details of a circuit-level noise model. Finally, we define the SD6 distance $d_s$ to be the graphlike distance of the Floquet codes circuit for an SD6 noise model.

To achieve an EM3 distance $d_e$ for either the toric or planar honeycomb code,

**Table 7.1:** Parameters of some of the hyperbolic Floquet codes we have constructed. Here $n$, $k$ and $d$ are the number of physical data qubits, number of logical qubits and embedded distance, respectively. The EM3 distance $d_e$ and SD6 distance $d_s$ are the graphlike distances of 16-round circuits using EM3 and SD6 noise models respectively, computed using Stim [92]. We omit $d_s$ where the calculation is too computationally expensive. We only include a code if $d$, $d_e$ or $d_s$ is at least as large as the corresponding distance for all smaller codes in the family.

8.8.8

| $n$ | $k$ | $d$ | $kd^2/n$ | $d_e$ | $d_s$ |
|---|---|---|---|---|---|
| 16 | 4 | 2 | 1.00 | 2 | 2 |
| 32 | 6 | 2 | 0.75 | 2 | 3 |
| 64 | 10 | 2 | 0.62 | 2 | 4 |
| 256 | 34 | 4 | 2.12 | 4 | 4 |
| 336 | 44 | 3 | 1.18 | 3 | 6 |
| 336 | 44 | 4 | 2.10 | 4 | 6 |
| 512 | 66 | 4 | 2.06 | 4 | 6 |
| 720 | 92 | 4 | 2.04 | 4 | 6 |
| 1,024 | 130 | 4 | 2.03 | 4 | 6 |
| 1,296 | 164 | 4 | 2.02 | 4 | 6 |
| 1,344 | 170 | 6 | 4.55 | - | 6 |
| 2,688 | 338 | 6 | 4.53 | - | - |
| 4,896 | 614 | 6 | 4.51 | - | - |
| 5,376 | 674 | 6 | 4.51 | - | - |
| 5,760 | 722 | 6 | 4.51 | - | - |

4.10.10

| $n$ | $k$ | $d$ | $kd^2/n$ | $d_e$ | $d_s$ |
|---|---|---|---|---|---|
| 120 | 8 | 3 | 0.60 | 3 | 4 |
| 160 | 10 | 4 | 1.00 | 4 | 4 |
| 320 | 18 | 5 | 1.41 | 4 | 6 |
| 600 | 32 | 6 | 1.92 | 6 | 6 |
| 3,600 | 182 | 8 | 3.24 | - | - |
| 10,240 | 514 | 9 | 4.07 | - | - |
| 19,200 | 962 | 10 | 5.01 | - | - |

4.8.10

| $n$ | $k$ | $d$ | $kd^2/n$ | $d_e$ | $d_s$ |
|---|---|---|---|---|---|
| 240 | 8 | 4 | 0.53 | 4 | 6 |
| 640 | 18 | 6 | 1.01 | 6 | 8 |
| 1,440 | 38 | 8 | 1.69 | 8 | 8 |
| 7,200 | 182 | 10 | 2.53 | - | - |

we need at least $2d_e$ columns of qubits and $3d_e$ rows, and hence use a patch with dimensions $2d_e \times 3d_e$ using $n_{tot} = n = 6d_e^2$ physical qubits [97, 100, 95, 156]. See Figure 7.1 for an example of a $4 \times 6$ toric honeycomb code ($d_e = 2$).

#### 7.3.1.1 Hyperbolic Floquet codes

Although planar and toric honeycomb codes can only encode one or two logical qubits, respectively, hyperbolic Floquet codes can encode a number of logical qubits $k$ proportional to the number of physical qubits $n$. From Equation (5.8) we see that a hyperbolic Floquet code has a finite encoding rate $k/n = \dim H_1/|V| > R$, where $R$ is $1/8$, $1/40$ and $1/20$ for the 8.8.8, 4.8.10 and 4.10.10 colour code tilings, respectively.

It follows from the $d \in O(\log n)$ distance scaling of hyperbolic surface codes that the embedded distance of a hyperbolic Floquet code therefore also scales as $d = O(\log n)$ [198]. This leads to the parameters of hyperbolic Floquet codes satisfying

$kd^2/n = O(\log^2(n))$, which is an asymptotic improvement over the parameters of Euclidean surface codes and Floquet codes, for which $kd^2/n = O(1)$.

We give the parameters of some of the hyperbolic Floquet codes we constructed in Table 7.1. These belong to three families of hyperbolic Floquet codes, derived from 8.8.8, 4.10.10 and 4.8.10 tessellations. We have picked these tilings as they have the lowest stabiliser weight and give the highest relative distance, although they give the lowest encoding rate, see discussion in Ref. [44]. All of the codes constructed have $kd^2/n$ exceeding that of planar and toric honeycomb codes for which $kd^2/n$ is $1/6$ and $1/3$, respectively. The largest improvement in parameters is given by the [[19200,962,10]] 4.10.10 hyperbolic Floquet code which has $kd^2/n = 5.01$, a $30.1\times$ larger ratio than is obtained using the planar honeycomb code.

#### 7.3.1.2 Semi-hyperbolic Floquet codes

One potential concern is that hyperbolic Floquet codes can only achieve error suppression that is polynomial in the system size, owing to their $O(\log n)$ distance scaling. However, in this section we will show that we can achieve exponential error suppression using the semi-hyperbolic colour code tilings we introduced in Section 7.3.1.2, which fine-grain the hyperbolic lattice and have $O(\sqrt{n})$ distance scaling, while still retaining an advantage over Euclidean Floquet codes.

We define a family of semi-hyperbolic Floquet codes from a hyperbolic colour code tiling $\mathcal{T}$ and the fine-graining parameter $l$ to obtain a semi-hyperbolic tiling $\mathcal{T}_l$ (see Section 7.1.3), which we use to define the semi-hyperbolic Floquet code $\mathcal{F}(\mathcal{T}_l)$. Let us denote the parameters of the Floquet code $\mathcal{F}(\mathcal{T})$ derived from $\mathcal{T}$ by $[[n,k,d]]$ and we denote the parameters of $\mathcal{F}(\mathcal{T}_l)$ by $[[n_l,k_l,d_l]]$. The topology of the surface is unchanged by the fine-graining procedure so we have $k_l = k$ and from Equation (7.2) we have that $n_l = l^2 n$. The minimum length of a homologically non-trivial cycle or co-cycle in the restricted lattices of $\mathcal{T}_l$ will increase by a factor proportional to $l$, and so we have $d_l \geq C_1 l d$ for some constant $C_1$. This leads to parameters $[[n_l,k_l,d_l]]$ of the semi-hyperbolic Floquet code $\mathcal{F}(\mathcal{T}_l)$ satisfying $k_l d_l^2/n_l \geq C_1^2 kd^2/n$. If we fix the tiling $\mathcal{T}$ from which we define a family of semi-hyperbolic Floquet codes $\mathcal{F}(\mathcal{T}_l)$ then we have $k_l d_l^2/n_l \geq C_2$ where $C_2 = C_1^2 kd^2/n$ is a constant determined by

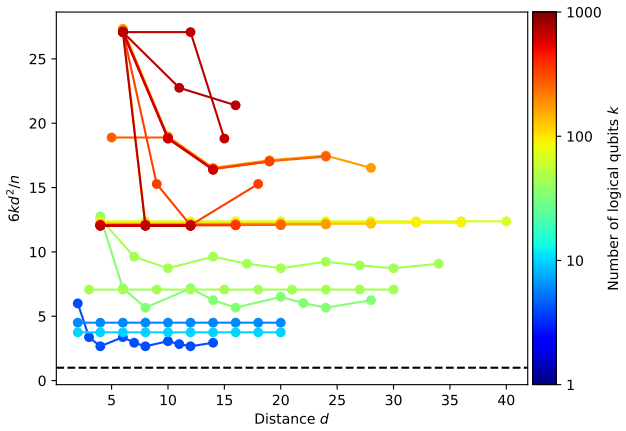



**Figure 7.9:** Parameters of semi-hyperbolic Floquet codes. Each line corresponds to a family of semi-hyperbolic codes constructed by fine-graining an 8.8.8 colour code tiling, with $l$ increasing by one from left to right (starting at $l = 1$). We plot $6kd^2/n$ on the $y$-axis against $d$ on the $x$-axis (where $d$ is the embedded distance of the Floquet code). Note that $6kd^2/n = 1$ for the planar Floquet code (black dashed line) and hence this ratio corresponds to the multiplicative saving in the number of physical qubits relative to the planar Floquet code to obtain a fixed target $k$ and $d$.

$\mathcal{T}$. Although a family of semi-hyperbolic Floquet codes defined this way does not have an *asymptotic* improvement over the parameters of the honeycomb code, it is still possible to obtain a constant factor improvement if $C_2 > 1/6$.

From our analysis of the parameters of the semi-hyperbolic Floquet codes we have constructed, we find that this constant factor improvement over the planar honeycomb code can be substantial. In Figure 7.9 we show the parameters of families of semi-hyperbolic Floquet codes derived from 8.8.8 colour code tilings. For some families of semi-hyperbolic codes we have $C_1 = 1$, i.e. the line is horizontal on the plot, at least for the system sizes we consider. For these semi-hyperbolic Floquet code families, we can increase the distance by a factor of $l$ using exactly $l^2\times$ more physical qubits. Many of these families of semi-hyperbolic codes retain an order-of-magnitude reduction in qubit overhead relative to the planar honeycomb code even for very large distances.

In Figure 7.10 we plot the number of logical qubits that can be encoded (with embedded distance at least 12) as a function of the number of physical qubits available, using multiple copies of semi-hyperbolic Floquet codes and honeycomb codes.

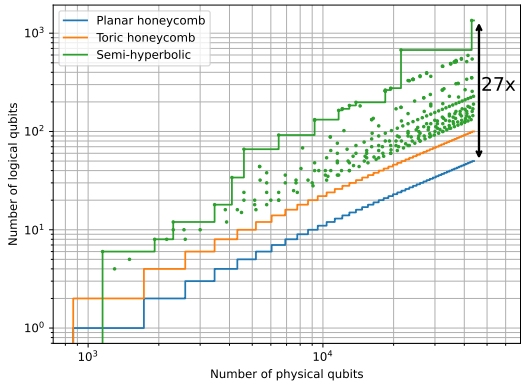



**Figure 7.10:** The number of logical qubits that can be encoded with embedded distance $d$ at least 12 using multiple copies of a semi-hyperbolic, toric honeycomb or planar honeycomb floquet code. The EM3 distance is the distance of the code for the EM3 noise model, for which two-qubit Pauli operators can be measured directly without the need for ancillas. Each green circle corresponds to multiple copies of a semi-hyperbolic floquet code, and the green line is the Pareto frontier for the copies of semi-hyperbolic codes considered.

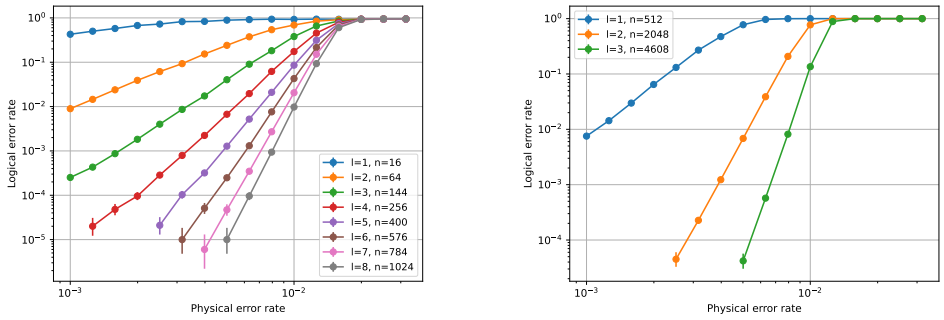
We can encode up to $27\times$ more logical qubits using semi-hyperbolic Floquet codes relative to planar honeycomb codes, including a $> 10\times$ increase in the encoding rate for smaller system sizes using fewer than $10,000$ physical qubits.

### 7.3.2 Simulations

So far we have focused our analysis on code parameters, however logical error rate performance for realistic noise models is more relevant to understand the practical utility of the constructions. We have carried out numerical simulations to compare the performance of semi-hyperbolic Floquet codes with planar honeycomb codes (for the EM3 noise model) as well as surface codes (for the EM3 and SD6 noise models). All simulations were carried out using Stim [92] to simulate the circuits and construct matching graphs and PyMatching [113] to decode. For the simulations of planar honeycomb codes and surface codes, we used the open-source simulation software [96] written for Ref. [95]. Note that we have made some Stim circuits of hyperbolic and semi-hyperbolic Floquet codes available on GitHub [108].

#### 7.3.2.1 Thresholds

We expect families of semi-hyperbolic Floquet codes to have the same threshold as planar and toric honeycomb codes, since the bulk of a semi-hyperbolic Floquet code




**(a)** **(b)**

**Figure 7.11:** Threshold plots of two families of semi-hyperbolic Floquet codes, using an EM3 noise model and 64 rounds. (a) A family of semi-hyperbolic Floquet codes derived from the Bolza surface (see Table 7.2 for code parameters). (b) A family of semi-hyperbolic Floquet codes derived from a genus-33 closed hyperbolic surface (with code parameters given in Table 7.3). For both plots, the legend gives the fine-graining parameter $l$ and number of physical qubits $n$ and we observe a threshold of around 1.5%-2%, consistent with the threshold of the honeycomb code [97, 95, 156]. Note that here (and throughout this paper) the logical error rate is the probability that *any* of the encoded logical qubits fail.

looks identical to a honeycomb code as we increase the fine-graining parameter $l$. We demonstrate this numerically for two families of semi-hyperbolic Floquet codes in Figure 7.11, which each have a threshold of at least 1.5% to 2%, consistent with the threshold of the honeycomb code [97, 95, 156].

#### 7.3.2.2 Overhead reduction

In Figure 7.12a we consider the Majorana-inspired EM3 noise model, which assumes noisy direct measurement of two-qubit Pauli operators. We simulate the performance of an $l = 2$ [[21504,674,12]] semi-hyperbolic Floquet code, which belongs to the family of semi-hyperbolic Floquet codes described in Table 7.4. We compare the logical error rate performance of the [[21504,674,12]] code with honeycomb codes for encoding 674 logical qubits over 192 time steps. We find that the semi-hyperbolic Floquet code matches the performance of $d = 16$ honeycomb codes using $1,035,264$ physical qubits, corresponding to a $48\times$ reduction in qubit overhead. Honeycomb codes are already known to be $2\times$ to $6\times$ more efficient than surface codes for comparable noise models that assume compilation into direct two-qubit

**Table 7.2:** Parameters of a family of semi-hyperbolic floquet codes derived from the Bolza surface. Here, $n_{tot}$ is the total number of physical qubits (including ancillas) for the SD6 noise model. We verified that the EM3 distance $d_e$ equals the embedded distance $d$ reported here for all codes in the table.

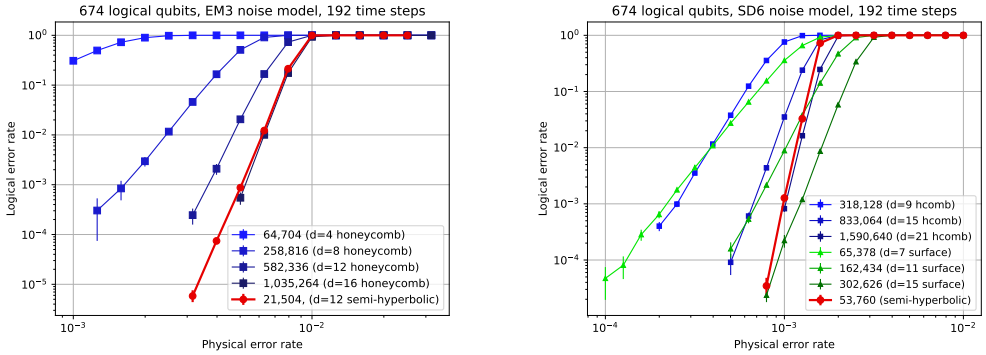
| $l$ | $k$ | $n$ | $d$ | $kd^2/n$ | $n_{tot}$ | $d_s$ | $kd_s^2/n_{tot}$ |
|---|---|---|---|---|---|---|---|
| 1 | 4 | 16 | 2 | 1.00 | 40 | 2 | 0.40 |
| 2 | 4 | 64 | 3 | 0.56 | 160 | 4 | 0.40 |
| 3 | 4 | 144 | 4 | 0.44 | 360 | 6 | 0.40 |
| 4 | 4 | 256 | 6 | 0.56 | 640 | 8 | 0.40 |
| 5 | 4 | 400 | 7 | 0.49 | 1,000 | 10 | 0.40 |
| 6 | 4 | 576 | 8 | 0.44 | 1,440 | 12 | 0.40 |
| 7 | 4 | 784 | 10 | 0.51 | 1,960 | 14 | 0.40 |
| 8 | 4 | 1,024 | 11 | 0.47 | 2,560 | 16 | 0.40 |
| 9 | 4 | 1,296 | 12 | 0.44 | 3,240 | 18 | 0.40 |
| 10 | 4 | 1,600 | 14 | 0.49 | 4,000 | 20 | 0.40 |

**Table 7.3:** Parameters of a family of semi-hyperbolic floquet codes derived from a genus-33 closed hyperbolic surface.

| $l$ | $k$ | $n$ | $d$ | $kd^2/n$ |
|---|---|---|---|---|
| 1 | 66 | 512 | 4 | 2.06 |
| 2 | 66 | 2,048 | 8 | 2.06 |
| 3 | 66 | 4,608 | 12 | 2.06 |
| 4 | 66 | 8,192 | 16 | 2.06 |
| 5 | 66 | 12,800 | 20 | 2.06 |
| 6 | 66 | 18,432 | 24 | 2.06 |
| 7 | 66 | 25,088 | 28 | 2.06 |
| 8 | 66 | 32,768 | 32 | 2.06 |
| 9 | 66 | 41,472 | 36 | 2.06 |
| 10 | 66 | 51,200 | 40 | 2.06 |

Pauli measurements [49, 156, 88]. Therefore, given a platform permitting direct two-qubit Pauli measurements and semi-hyperbolic qubit connectivity (e.g. using a modular architecture), our results suggest that semi-hyperbolic Floquet codes can require over $100\times$ fewer physical qubits than the surface code circuits from Refs. [49, 156, 88] for an EM3 noise model at around $p = 0.1\%$.

Using a least-squares fit of the four left-most data points to extrapolate the red curve in Figure 7.12a to lower physical error rates, we estimate that the [[21504,674,12]] semi-hyperbolic Floquet code has a logical error rate of $\approx 1.8 \times 10^{-11}$ at $p = 0.1\%$ for the EM3 noise model. This implies a logical er-




**(a)** **(b)**

**Figure 7.12:** Logical error rate vs. physical error rate for protecting 674 logical qubits using 674 copies of planar honeycomb codes (shades of blue) and a [[21504,674,12]] semi-hyperbolic Floquet code derived from a 8.8.8 tessellation. In (a) an EM3 noise model is used (direct pair measurements without the use of an ancilla) whereas in (b) an SD6 noise model (ancilla-assisted) is used and we also compare with standard surface code circuits (shades of green). For all codes, we simulate 192 time steps and give the average of the logical $X$ and $Z$ error rates on the $y$-axis. For the honeycomb and surface codes we plot $1-(1-p_{log})^{674}$ where $p_{log}$ is the logical error rate for a single patch (one logical qubit). The legend gives the total number of physical qubits (including ancillas for the SD6 noise model).

**Table 7.4:** Parameters of a family of semi-hyperbolic floquet codes encoding 674 logical qubits, constructed by fine-graining an 8.8.8 tiling of a genus-337 hyperbolic surface. Note that $d$ here denotes the embedded distance.

| $l$ | $k$ | $n$ | $d$ | $kd^2/n$ |
|---|---|---|---|---|
| 1 | 674 | 5,376 | 6 | 4.51 |
| 2 | 674 | 21,504 | 12 | 4.51 |
| 3 | 674 | 48,384 | 15 | 3.13 |

ror rate per logical qubit of $\approx 2.7\times10^{-14}$, or $\approx 4.2\times10^{-16}$ per logical qubit per round. We therefore project that we can reach well below the "teraquop regime" [97] of $10^{-12}$ logical failure rates using only 32 physical qubits per logical qubit. In contrast, planar honeycomb codes have been shown to require from 600 [97] to 2000 [156] physical qubits per logical qubit to achieve the teraquop regime using the same noise model in prior work.

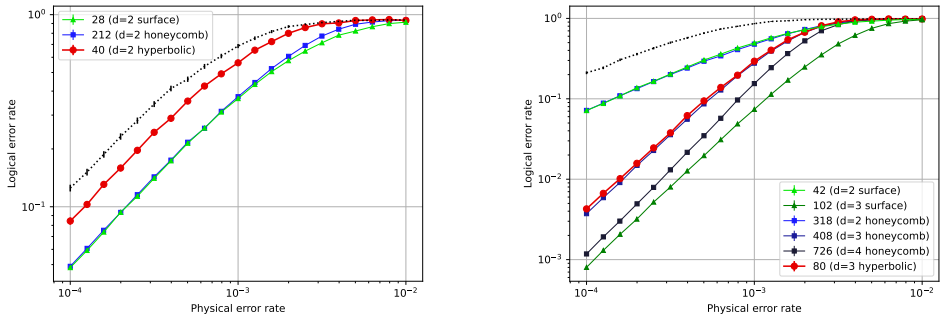
In Figure 7.12b, we instead consider the SD6 noise model, which uses an ancilla qubit to measure each check operator in the presence of standard circuit-level depolarising noise. We compare a semi-hyperbolic Floquet code (the same as used

in Figure 7.12a) with honeycomb and surface codes for encoding 674 logical qubits over 192 time steps. In this noise model, we find that the semi-hyperbolic Floquet code matches the performance of planar honeycomb codes with SD6 distance $d_s = 21$ which require $n_{tot} = 1,590,640$ qubits, whereas the semi-hyperbolic Floquet code only needs $53,760$ qubits (including ancillas), a $29.6\times$ saving in resources. At a physical error rate slightly below $0.1\%$, our semi-hyperbolic Floquet code achieves a $5.6\times$ reduction in qubit overhead relative to surface codes, matching the logical error rate of $d = 15$ surface codes which use $302,626$ physical qubits. Note that the curve for the $d = 15$ surface code has a shallower gradient than that of the semi-hyperbolic Floquet code, indicating that the semi-hyperbolic Floquet code has a higher SD6 distance. Therefore, we would expect that the semi-hyperbolic Floquet code will offer a bigger advantage over surface codes at lower physical error rates. If the semi-hyperbolic Floquet code has SD6 distance $d_s = 21$, then it would have similar performance to $d = 21$ surface codes at very low physical error rates, which would use $593,794$ physical qubits (881 per patch), $11\times$ more than the 53,760 needed for our semi-hyperbolic Floquet code.

While the check weight is still two, compared to the honeycomb code, hyperbolic Floquet codes derived from a $r.g.b$ tiling have a higher plaquette stabiliser weights $2r$, $2g$ and $2b$, and each detector is formed from the parity of $2r$, $2g$ or $2b$ check operator measurements. Despite this, we show in this section that, even at high physical error rates, hyperbolic Floquet codes have a logical error rate performance that is comparable to that of the honeycomb code, but with significantly reduced resource overheads.

### 7.3.3 Small examples

We also study the performance of small hyperbolic and semi-hyperbolic Floquet codes that might be amenable to experimental realisation in the near future. In Figure 7.13 we show the performance of a small hyperbolic Floquet codes (with SD6 distance 2 and 3) for an SD6 noise model, compared to small planar honeycomb and surface codes. To achieve a given logical performance, the hyperbolic Floquet codes in both Figure 7.13a and Figure 7.13b use around $5\times$ fewer physical qubits




**(a)** **(b)**

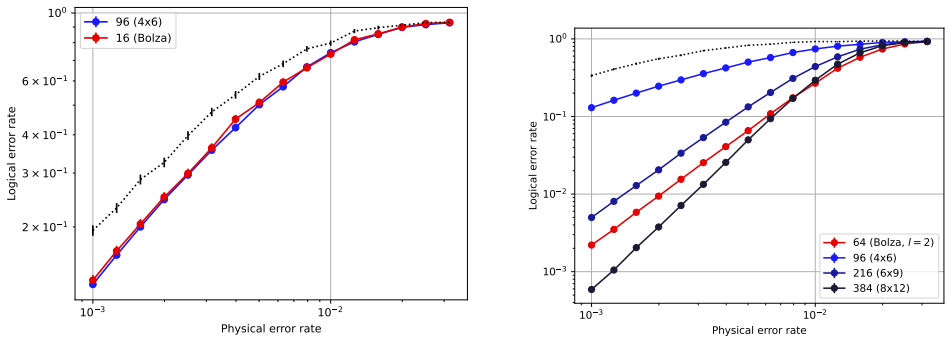
**Figure 7.13:** Logical error rates vs. physical error rates for encoding $k$ logical qubits using $k$ copies of small planar honeycomb or surface codes, compared to a hyperbolic floquet code encoding $k$ logical qubits, for 96 time steps using a standard depolarising (SD6) noise model. In (a) we have $k = 4$ for the Bolza Floquet code, which has SD6 distance $d_s = 2$, whereas in (b) we have $k = 6$ for a hyperbolic Floquet code with $d_s = 3$. The hyperbolic Floquet codes in (a) and (b) are both derived from 8.8.8 tilings. The legend gives the total number of qubits, with the SD6 distance given in brackets.

than planar honeycomb codes but use a similar number of physical qubits to surface codes.

For the EM3 noise model, we compare to planar honeycomb codes in Figure 7.14a and find that the $d = 2$ hyperbolic Floquet code derived from the Bolza surface (using 16 qubits) has identical performance to four copies of a $d = 2$ planar honeycomb code using $6\times$ more physical qubits. We also study the performance (again for the EM3 noise model) of an $l = 2$ semi-hyperbolic Floquet code derived from the Bolza surface in Figure 7.14b and find it to be $3.4\times$ more efficient than four copies of $d = 3$ planar honeycomb codes achieving a similar logical error rate.

## 7.4 Conclusion

In this chapter, we have constructed Floquet codes derived from colour code tilings of closed hyperbolic surfaces. These constructions include hyperbolic Floquet codes, obtained from hyperbolic tilings, as well as semi-hyperbolic Floquet codes, which are derived from hyperbolic tilings via a fine-graining procedure. We have given explicit examples of hyperbolic Floquet codes with improved encoding rates relative to honeycomb codes and have shown how semi-hyperbolic Floquet codes can retain




**(a)** **(b)**

**Figure 7.14:** Logical error rates vs. physical error rates for encoding 4 logical qubits using 4 copies of small planar honeycomb codes (blue) or a (semi-)hyperbolic floquet code encoding 4 logical qubits (red), for the EM3 noise model (check operators are measured directly without ancillas). In (a) we use a hyperbolic code derived from the Bolza surface that has distance 2 in this noise model, whereas in (b) we use an $l = 2$ semi-hyperbolic code derived from the Bolza surface that has distance 3. The legend gives the total number of qubits, with the dimension of the lattice given in parentheses for the planar honeycomb codes. All circuits use 16 rounds of measurements, and the $y$ axis gives the average of the logical $X$ and $Z$ error rates. The black dotted line is the logical failure rate for the (semi-)hyperbolic code for a decoder which always predicts that no logical observable has been flipped. Note that the $4 \times 6$, $6 \times 9$ and $8 \times 12$ planar honeycomb codes have distances 2, 3 and 4 respectively for this EM3 noise model.

this advantage while enabling improved $\sqrt{n}$ distance scaling.

We have used numerical simulations to analyse the performance of our constructions for two noise models: a Majorana-inspired 'EM3' noise model, which assumes direct noisy two-qubit measurements, and a standard circuit-level depolarising 'SD6' noise model, which uses ancilla qubits to assist the measurement of check operators. For the EM3 noise model, we compare a semi-hyperbolic Floquet code that encodes 674 logical qubits into 21,504 physical qubits with 674 copies of planar honeycomb codes, and find that the semi-hyperbolic Floquet code uses 48$\times$ fewer physical qubits at physical error rates as high as 0.3% to 1%. We demonstrate that this semi-hyperbolic Floquet code can achieve logical error rates below $10^{-12}$ at 0.1% EM3 noise using as few as 32 physical qubits per logical qubit. This is a significant improvement over planar honeycomb codes, that require 600 to 2000 physical qubits per logical qubit in the same regime [97, 156]. For the SD6 noise model at a noise

strength of 0.1%, we show that the same semi-hyperbolic Floquet code uses around 30× fewer physical qubits than planar honeycomb codes, and around 5.6× fewer qubits than surface codes. We also construct several small examples of hyperbolic and semi-hyperbolic Floquet codes amenable to near-term experiments, including a 16-qubit hyperbolic Floquet code derived from the Bolza surface. These small instances are around 3× to 6× more efficient than planar honeycomb codes and are comparable to standard surface codes in the SD6 noise model.

An interesting avenue of future research might be to study the performance of other recent variants and generalisations of Floquet codes [56, 126, 2, 210, 14, 182, 57, 188, 68]. For example, the CSS Floquet code introduced in Refs. [56, 126] is also defined from any colour code tiling, but with a modified choice of two-qubit check operators. CSS Floquet codes can therefore also be constructed from the hyperbolic and semi-hyperbolic colour code tilings in Section 7.1 and, since they have the same embedded homological codes as the Floquet codes we study here, we would expect them to offer similar advantages over CSS honeycomb codes.

Another possible research direction would be to study and design more concrete realisations of hyperbolic and semi-hyperbolic Floquet codes in modular architectures. This could include studying protocols for realising these modular architectures where links between modules are more noisy or slow [84, 152, 153, 167], motivated by physical systems that can be used to realise them, such as ions [150, 178, 162], atoms [168, 20] or superconducting qubits [170, 190, 58, 211, 121].

Finally, we have focused here on error corrected quantum *memories*, but more work is needed to demonstrate how our constructions can be used to save resources within the context of a quantum computation. We note that the Floquet code schedule applies a logical Hadamard to all logical qubits in the code every three sub-rounds; however, to be useful in a computation we must either be able to read and write logical qubits to the memory or address logical qubits individually with a universal gate set. We expect that reading and writing of logical qubits to and from honeycomb codes can be achieved using Dehn twists and lattice surgery by adapting the methods developed in Ref. [39] for hyperbolic surface codes to (semi-)hyperbolic Floquet

codes, however a more detailed analysis is required. Further work could also investigate (semi-)hyperbolic Floquet codes that admit additional transversal logical operations [166, 203] such as fold-transversal logical gates [41], examples of which have been demonstrated for hyperbolic surface codes.

# General Conclusions

In this thesis we have described techniques for making quantum error correction more efficient and practical. Part I focussed on decoders for 2D topological codes such as the surface code. In Chapter 1 we introduced the relevant background material, as well as presenting our optimal local unitary encoding circuits for the surface code. In Chapter 2 we presented sparse blossom, our implementation of the minimum-weight perfect matching (MWPM) decoder, which we released in the PyMatching software package. Sparse blossom avoids explicitly constructing a large auxiliary graph, commonly used in MWPM decoder implementations, leading to a running time that scales linearly in the number of detection events below threshold. Our implementation can process surface code syndrome data in less than one microsecond per round up to distance-17, which matches the rate at which it would be generated in superconducting quantum computers [5]. In Chapter 3 we described our belief-matching decoder, which improves on the accuracy of the MWPM decoder by exploiting error mechanisms that flip more more than two detection events (non-graphlike error mechanisms). Belief-matching increases the threshold of the surface code with circuit-level noise, and reduces the required resource overhead below threshold. We concluded Part I in Chapter 4 with schedule-induced gauge fixing (SIGF), which changes the order of check operator measurements in subsystem codes such that gauge operators are temporarily fixed as stabilisers. We showed that SIGF increases the noise threshold of the subsystem surface code, with the biggest improvements shown for biased noise.

In Part II we presented constructions of quantum LDPC codes that require fewer resources than the surface code for storage, even for a noise model in which gates

in the syndrome measurement circuit are noisy. Our constructions are derived from tilings of closed hyperbolic surfaces and are related to hyperbolic surface codes by constant-depth unitaries. Our subsystem hyperbolic codes are generalisations of the subsystem surface code, which have three-qubit check operators. We exploited symmetries in the tiling to construct optimal-depth parity check measurement schedules, and showed that our subsystem hyperbolic codes could require $4.3\times$ fewer physical qubits than the surface code even at physical error rates as high as 0.2%. In Chapter 7 we constructed Floquet codes derived from tilings of closed hyperbolic surfaces. Some of our constructions have a qubit overhead that is $48\times$ lower than that of surface codes for a noise model where two-qubit Pauli measurements can be implemented directly. We also presented small examples, including a code derived from the Bolza surface that encodes 4 logical qubits into 16 physical qubits.

Many avenues of research follow naturally from the work presented in this thesis. Firstly, while sparse blossom is designed to decode a batch of syndrome data, a real-time decoder must be capable of decoding a *stream* of measurement data. Furthermore, in order to reduce power consumption and achieve deterministic running times, FPGA or ASIC implementations may be necessary. While there has been recent progress in this direction, current approaches sacrifice accuracy relative to MWPM or belief-matching decoders [141, 13]. Additionally, the belief-matching decoder we presented is asymptotically efficient but significantly slower than sparse blossom in practice, and further optimisations may provide a more favourable trade-off of speed and accuracy. Finally, it will be important to develop more efficient and accurate decoders for more quantum codes not amenable to MWPM or UF, such as colour codes and more general quantum LDPC codes. While the BP-OSD decoder has good accuracy for a wide range of quantum LDPC codes [158, 169, 35, 208], it has an expected running that is cubic in the number of error mechanisms, in contrast to MWPM and UF which have a linear expected running time below threshold. Therefore, an important research direction is the development of more efficient decoders for more general quantum LDPC codes.

Important questions also remain for the development of practical quantum

LDPC codes. In this thesis we have focused on the performance of our constructions for storing quantum information, however ultimately we would like to use these codes to reduce resources within the context of a quantum computation. To be useful as a quantum memory within this context, it is important that quantum information into and out of storage efficiently. In Ref. [39] a proposal was made for moving logical qubits within hyperbolic surface codes using Dehn twists, and for teleporting qubits into and out of storage using lattice surgery. Certain symmetries of codes, such as automorphisms and ZX-dualities [41], can also be used to implement some logical gates in LDPC codes. All of these techniques could be studied in more detail for the subsystem hyperbolic and hyperbolic Floquet codes explored in this thesis. An ultimate goal would be to demonstrate that these techniques can be used to reduce the overall space-time cost of a quantum algorithm for realistic physical systems capable of implementing the required long-range connectivity.

# Appendix A

# Optimal Unitary Encoding Circuits

In this section of the Appendix, we present some additional unitary encoding circuits for surface does, which we originally published in Ref. [116].

## A.1 Planar base cases and rectangular code

In Figure A.1 we provide encoding circuits for the $L = 2$, $L = 3$ and $L = 4$ planar codes, requiring 4, 6 and 8 time steps respectively. These encoding circuits are used as base cases for the planar encoding circuits described in Section 1.3.1. In Figure A.2 we provide encoding circuits that either increase the width or height of a planar code by two, using three time steps.

## A.2 Rotated Surface Code

In Figure A.3 we demonstrate a circuit that encodes an $L = 7$ rotated surface code from a distance $L = 5$ rotated code. For a given distance $L$, the rotated surface code uses fewer physical qubits than the standard surface code to encode a logical qubit [24]. Considering a standard square lattice with qubits along the edges, a rotated code can be produced by removing qubits along the corners of the lattice boundary, leaving a diamond of qubits from the centre of the original lattice. The diagram in Figure A.3 shows the resultant code, rotated 45° compared to the original planar code, and with each qubit now denoted by a vertex rather than an edge. For a distance $L$ code the rotated surface code requires $L^2$ qubits compared to $L^2 + (L-1)^2$ for the planar code.

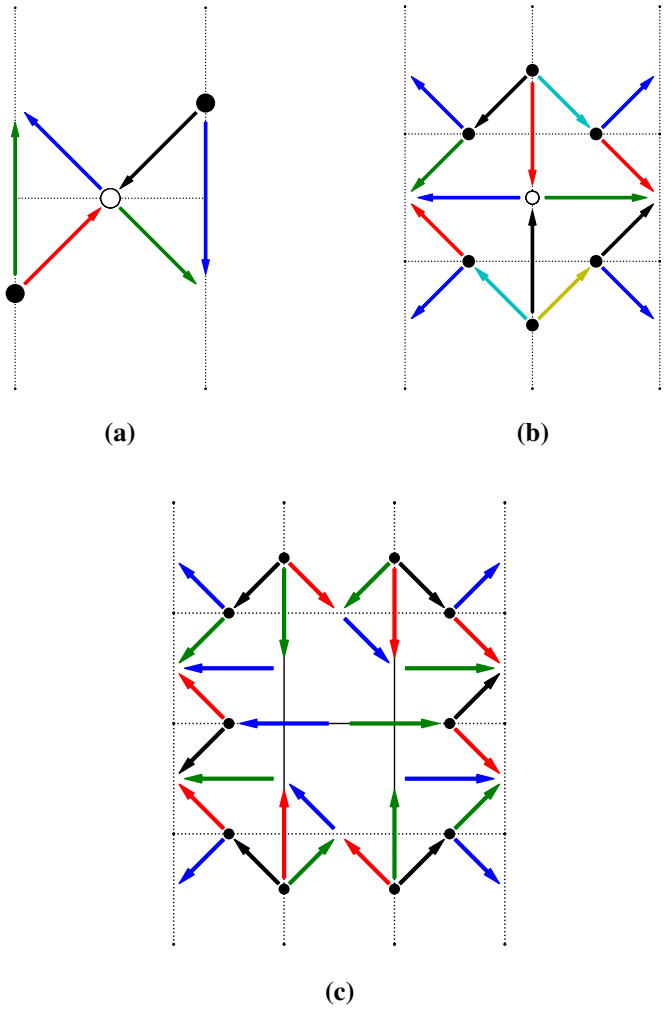
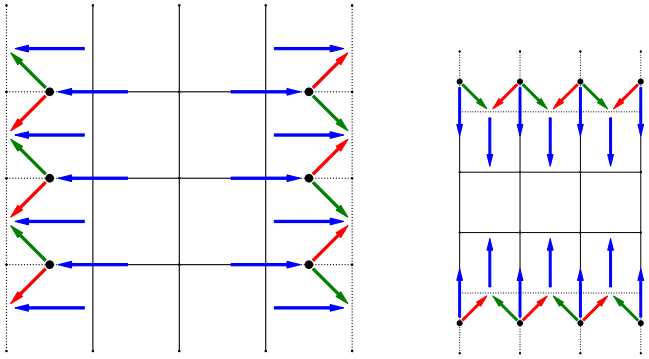



**Figure A.1:** Encoding circuits for the $L = 2$, $L = 3$ and $L = 4$ planar codes. Each edge corresponds to a qubit, each arrow denotes a CNOT gate pointing from control to target, and each filled black circle denotes a Hadamard gate applied at the beginning of the circuit. The colour of each CNOT gate corresponds to the time step it is implemented in, with blue, green, red, black, cyan and yellow CNOT gates corresponding to the first, second, third, fourth, fifth and sixth time steps respectively. The hollow circle in each of (a) and (b) denotes the initial unencoded qubit. The circuit in (c) encodes an L=4 planar code from an L=2 planar code, with solid edges denoting qubits initially encoded in the L=2 code.



(a) (b)

**Figure A.2:** (a) Circuit to increase the width of a planar code by two. (b) Circuit to increase the height of a planar code by two. Notation is the same as in Figure A.1.

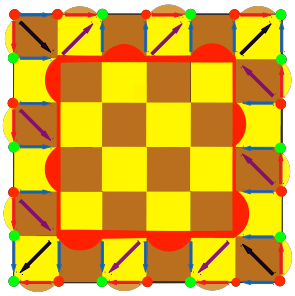


**Figure A.3:** Encoding circuit for the $L = 7$ rotated code from an $L = 5$ rotated surface code (shown as a red outline). The colour of each arrow denotes the time step the gate is applied in. The gates are applied in the order: blue, red, black, purple. The additional qubits are initialised in the $|+\rangle$ (red) or $|0\rangle$ (green) state. The yellow squares denote a $Z$ stabiliser on the four corner qubits, and the brown squares represent an $X$ operator on the four corner qubits. The rotated code has additional stabilizers between states on along the edges. In the $L = 5$ code these are shown as a red arch (with $Z$ and $X$ stabilisers on the vertical and horizontal edges respectively), and the yellow and brown arches in the $L = 7$ code edge are Z and X stabilizers between the two edge qubits.

The encoding circuit in Figure A.3 takes 4 steps to grow a rotated code from a distance $L = 5$ to $L = 7$. This is a fixed cost for any distance $L$ to $L+2$. To produce a distance $L = 2m$ code this circuit would be applied repeatedly $m + O(1)$ times to an $L = 2$ or $L = 3$ base case, requiring a circuit of total depth $2L + O(1)$. The circuit in Figure A.3 can be verified by seeing that a set of generators for the $L = 5$ rotated code (along with the single qubit $Z$ and $X$ stabilisers of the ancillas) is mapped to a set of generators of the $L = 7$ rotated code, as well as seeing that the $X$ and $Z$ logicals of the $L = 5$ code map to the $X$ and $Z$ logicals of the $L = 7$ rotated code.

# Appendix B

# Sparse Blossom

## B.1 The blossom algorithm

The blossom algorithm, introduced by Jack Edmonds [73, 72], is a polynomial-time algorithm for finding a minimum-weight perfect matching in a graph. In this section we will outline some of the key concepts in the original blossom algorithm. We do not explain the original blossom algorithm in full, since there is significant overlap with our sparse blossom algorithm, which we described in Section 2.2. We refer the reader to references [73, 72, 87, 131] for a more complete overview of the blossom algorithm.

We will first introduce some terminology. Given some matching $M \subseteq E$ in a graph $G = (V, E)$, we say that an edge in $E$ is *matched* if it is also in $M$, and unmatched otherwise, and a node is matched if it is incident to a matched edge, and unmatched otherwise. A maximum cardinality matching is a matching that contains as many edges as possible. An *augmenting path* is a path $P \subseteq E$ which alternates between matched and unmatched edges, and begins and terminates at two distinct unmatched nodes.

Given an augmenting path $P$ in $G$, we can always increase the cardinality of the matching $M$ by one by replacing $M$ with the new matching $M' = M \oplus P$. We refer to this process, of adding each unmatched edge in $P$ to $M$ and removing each matched edge in $P$ from $M$, as *augmenting* the augmenting path $P$ (see Figure B.1(a)). Berge's theorem states that a matching has maximum cardinality if and only if there is no

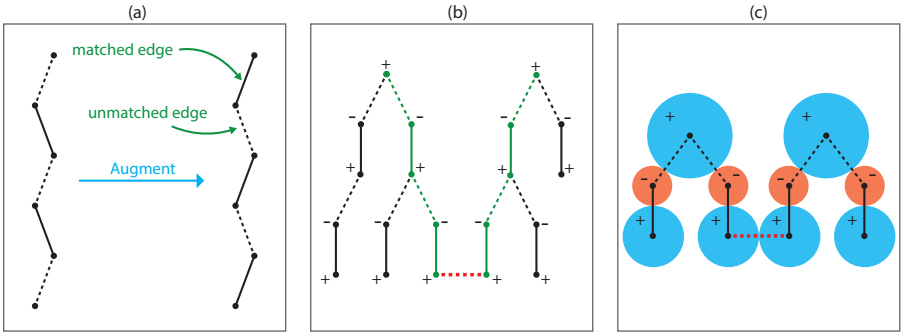



**Figure B.1:** (a) Augmenting an augmenting path. Matched edges become unmatched, and unmatched edges become matched. (b) Examples of two alternating trees in the blossom algorithm for finding a maximum matching. Each tree has one unmatched node. The two trees have become connected via the red dashed edge. The path between the roots of the two trees, through the green edges and red edge, is an augmenting path. (c) An example of two alternating trees in the blossom algorithm for finding a minimum-weight perfect matching. Each node $v$ now has a dual variable $y_v$ which, when $y_v$ is positive, we can interpret as the radius of a region centred on the node. A new edge $(u,v)$ with weight $w_{u,v}$ can only be explored by the alternating tree if it is *tight*, meaning that the dual variables $y_u$ and $y_v$ satisfy $y_u + y_v = w_{u,v}$.

augmenting path [17].

### B.1.1 Solving the maximum cardinality matching problem

We will now give an overview of the original *unweighted* version the blossom algorithm, which finds a maximum cardinality matching (as introduced by Edmonds in [73]). The unweighted blossom algorithm is used as a subroutine by the more general blossom algorithm for finding a minimum-weight perfect matching (discovered, also by Edmonds, in [72]), which we will outline in Appendix B.1.2. The algorithm is motivated by Berge's theorem. Starting with a trivial matching, it proceeds by finding an augmenting path, augmenting the path, and then repeating this process until no augmenting path can be found, at which point we know that the matching is maximum. Augmenting paths are found by constructing *alternating trees* within the graph. An alternating tree $T$ in the graph $G$ is a tree subgraph of $G$ with an unmatched node as its root, and for which every path from root to leaf alternates between unmatched and matched edges, see Figure B.1(b). There are two types of nodes in $T$: "outer" nodes (labelled "+") and "inner" nodes (labelled "−"). Each

inner node is separated from the root node by an odd-length path, whereas each outer node is separated by an even-length path. Each inner node has a single child (an outer node). Each outer node can have any number of children (all inner nodes). All leaf nodes are outer nodes.

Initially, every unmatched node is a trivial alternating tree (a root node). To find an augmenting path, the algorithm searches the neighboring nodes in $G$ of the outer nodes in each tree $T$. If, during this search, an edge $(u,v)$ is found such that $u$ is an outer node of $T$ and $v$ is an outer node of some other tree $T' \neq T$ then an augmenting path has been found, which connects the roots of $T$ and $T'$, see Figure B.1(b). This path is augmented, the two trees are removed, and the search continues. If an edge $(u,v)$ is found between an outer node $u$ in $T$ and a matched node $v$ not in any tree (i.e. $v$ is matched), then $v$ and its match are added to $T$. Finally, if an edge $(u,v)$ is found between two outer nodes of the same tree then an odd-length cycle has been found, and forms a *blossom*. A key insight of Edmonds was that a blossom can be treated as a virtual node, which can be matched or belong to an alternating tree like any other node. However, we will explain how blossoms are handled in more detail in the context of our sparse blossom algorithm in Section 2.2.

### B.1.2 Solving the minimum-weight perfect matching problem

The extension from finding a maximum cardinality matching to finding a minimum-weight perfect matching is motivated by formulating the problem as a linear program [72]. Constraints are added on how the alternating trees are allowed to grow, and these constraints ensure that the weight of the perfect matching is minimal once it has been found. The formulation of the problem as a linear program is not required either to understand the algorithm, or for the proof of correctness. However, it does provide useful motivation, and the constraints and definitions used in the linear program are also used in the blossom algorithm itself. We will therefore describe the linear program here for completeness.

We will denote the boundary edges of some subset of the nodes $S \subseteq V$ by $\delta(S) := \{(u,v) \in E \mid u \in S, v \in V \setminus S\}$, and will let $\mathcal{O}$ be the set of all subsets of $V$ of odd cardinality at least three, i.e. $\mathcal{O} := \{o \subseteq V : |o| > 1, |o| \mod 2 = 1\}$. We

denote the edges incident to a single node $v$ by $\delta(v)$ (e.g. $\delta(v) = \delta(\{v\})$). We will use an incidence vector $\mathbf{x} \in \{0,1\}^{|E|}$ to represent a matching $M \subseteq E$ where $x_e = 1$ if $e \in M$ and $x_e = 0$ if $e \notin M$. We denote the weight of an edge $e \in E$ by $w_e$. The minimum-weight perfect matching problem can then be formulated as the following *integer* program:

$$\text{Minimise} \quad \sum_{e \in E} w_e x_e \tag{B.1a}$$

$$\text{subject to} \quad \sum_{e \in \delta(v)} x_e = 1 \quad \forall v \in V \tag{B.1b}$$

$$x_e \in \{0,1\} \quad \forall e \in E \tag{B.1c}$$

Edmonds introduced the following linear programming relaxation of the above integer program:

$$\text{Minimise} \quad \sum_{e \in E} w_e x_e \tag{B.2a}$$

$$\text{subject to} \quad \sum_{e \in \delta(v)} x_e = 1 \quad \forall v \in V \tag{B.2b}$$

$$\sum_{e \in \delta(S)} x_e \geq 1 \quad \forall S \in \mathcal{O} \tag{B.2c}$$

$$x_e \geq 0 \quad \forall e \in E. \tag{B.2d}$$

Note that the constraints in Equation (B.2c) are satisfied by *any* perfect matching, but Edmonds showed that adding them ensures that the linear program has an integral optimal solution. In other words, the integrality constraint (Equation (B.1c)) can be replaced by the inequalities in Equation (B.2c) and Equation (B.2d).

Every linear program (referred to as the *primal* linear program, or primal problem) has a *dual* linear program (or dual problem). The dual of the above primal

problem is:

$$\text{Maximise} \quad \sum_{v \in V} y_v + \sum_{S \in \mathcal{O}} y_S \tag{B.3a}$$

$$\text{subject to} \quad slack(e) \geq 0 \quad \forall e \in E \tag{B.3b}$$

$$y_S \geq 0 \quad \forall S \in \mathcal{O} \tag{B.3c}$$

where the *slack* of an edge is defined as

$$slack(e) := w_e - \sum_{u \in e} y_u - \sum_{S \in \mathcal{O}: e \in \delta(S)} y_S. \tag{B.4}$$

We say that an edge is *tight* if it has zero slack. Here we have defined a dual variable $y_v \in \mathbb{R}$ for each node $v \in V$, as well as a dual variable $y_S \in \mathbb{R}$ for each set $S \in \mathcal{O}$. While each variable $y_S$ is constrained to be non-negative (Equation (B.3c)), each $y_v$ is permitted to take any value. Although we have an exponential number of $y_S$ variables, this turns out not to be an issue since only $O(|V|)$ are non-zero at any given stage of the blossom algorithm.

We now recall some terminology and general properties of linear programs (see [146, 132] for more details). A solution of a linear program is *feasible* if it satisfies the constraints of the linear program. Without loss of generality, we assume that the primal linear program is a minimisation problem (in which case its dual is a maximisation problem). By the *strong duality theorem*, if both the primal and the dual linear program have a feasible solution, then they both also have an *optimal* solution. Furthermore, the minimum of the primal problem is equal to the maximum of its dual, providing a "numerical" proof of optimality.

We can obtain a "combinatorial" proof of optimality for any linear program using the *complementary slackness* conditions. Each constraint in the primal problem is associated with a variable of the dual problem (and vice versa). Let us associate the $i$th primal constraint with the $i$th dual variable (and vice versa). The complementary slackness conditions state that, if and only if we have a pair of optimal solutions, then if the $i$th dual variable is greater than zero then the $i$th primal constraint is satisfied

with equality. Similarly, if the $i$th primal variable is greater than zero then the $i$th dual constraint is satisfied with equality. More concretely, for the specific primal-dual pair of linear programs we are considering, the complementary slackness conditions are:

$$slack(e) > 0 \implies x_e = 0 \tag{B.5}$$

$$y_S > 0 \implies \sum_{e \in \delta(S)} x_e = 1 \qquad (S \in \mathcal{O}) \tag{B.6}$$

These conditions are used as a stopping rule in the blossom algorithm (with Equation (B.5) satisfied throughout) and provide a proof of optimality.

While it is convenient to use the strong duality theorem, since it applies to *any* linear program, its correctness is not immediately intuitive and its proof is quite involved (see [146, 132]). Fortunately, we can obtain a simple proof of optimality of the minimum-weight perfect matching problem directly, without the need for duality theory [87]. First, we note that for any feasible dual solution, we have that *any* perfect matching $N$ satisfies

$$\sum_{e \in N} w_e = \sum_{e \in N} \left( slack(e) + \sum_{v \in e} y_v + \sum_{S \in \mathcal{O}: e \in \delta(S)} y_S \right) \geq \sum_{v \in V} y_v + \sum_{S \in \mathcal{O}} y_S, \tag{B.7}$$

where here the equality is from the definition of $slack(e)$ and the inequality uses Equation (B.3b) and Equation (B.3c) and the fact that $N$ is a perfect matching. However, if we have a perfect matching $M$ which additionally satisfies Equation (B.5) and Equation (B.6) we instead have

$$\sum_{e \in M} w_e = \sum_{v \in V} y_v + \sum_{S \in \mathcal{O}} y_S, \tag{B.8}$$

and thus the perfect matching $M$ has minimal weight.

So far in this section, we have only considered the case that each edge is a *pair* of nodes (a set of cardinality two). Let us now consider the more general case (required for decoding) where we can also have half-edges. More specifically, we now have the edge set $E = E_1 \cup E_2$ where each $(u,v) \in E_2$ is a regular edge and

each $(u,) \in E_1$ is a half-edge (a node set of cardinality one). We note that a perfect matching is now defined as a subset of this more general edge set, but its definition is otherwise unchanged (a perfect matching is an edge set $M \subseteq E := E_1 \cup E_2$ such that each node is incident to exactly one edge in $M$). We extend our definition of $\delta(S)$ to be

$$\delta(S) := \{(u,v) \in E_2 \mid u \in S, v \in V \setminus S\} \cup \{(u,) \in E_1 \mid u \in S\}. \tag{B.9}$$

With this modification, the simple proof of correctness above still holds and the $slack(e)$ of a half-edge $e \in E_1$ is well defined by Equation (B.4).

The blossom algorithm for finding a minimum-weight perfect matching starts with an empty matching and a feasible dual solution, and iteratively increases the cardinality of the matching and the value of the dual objective while ensuring the dual problem constraints remain satisfied. Eventually, we will have a pair of feasible solutions to the primal and dual problem satisfying the complementary slackness conditions (Equation (B.5) and Equation (B.6)) at which point we know we have a perfect matching of minimal weight. The algorithm proceeds in stages, where each stage consists of a "primal update" and a "dual update". We repeat these primal and dual updates until no more progress can be made at which point, provided the graph admits a perfect matching, the complementary slackness conditions will be satisfied and so the minimum-weight perfect matching has been found. We will now outline the primal and dual update in more detail.

In the *primal update*, we consider only the subgraph $H$ of $G$ consisting of *tight* edges and try to find a matching of higher cardinality, essentially by running a slight modification to the unweighted blossom algorithm on this subgraph. In [131], the four allowed operations in the primal update are referred to as "GROW", "AUGMENT", "SHRINK" and "EXPAND". The first three of these already occur in the unweighted variant of blossom discussed in Appendix B.1.1. The GROW operation consists of adding a matched pair of nodes to an alternating tree. AUGMENT is the process of augmenting the path between the roots of two trees when they become connected. SHRINK is the name for the process of forming a blossom when an odd length cycle

is found. The operation that differs slightly in the weighted variant is EXPAND. This EXPAND operation can occur whenever the dual variable $y_S$ for a blossom $S$ becomes zero; when this happens the blossom is removed, the odd-length path through the blossom is added into the alternating tree, and nodes in the even-length path become matched to their neighbours. This differs slightly from the unweighted variant as we described it, where blossoms are only expanded when a path they belong to becomes augmented (at which point *all* the nodes in a blossom cycle become matched). We refer the reader to [131] for a more complete description of these operations in the primal update (and associated diagrams), although we reiterate that very similar concepts will be covered in more detail when we describe sparse blossom in Section 2.2.

In the *dual update*, we try to increase the dual objective by updating the value of the dual variables, ensuring that edges in alternating trees and blossoms remain tight, and also ensuring that the dual variables remain a feasible solution to the dual problem (the inequalities Equation (B.3b) and Equation (B.3c) must remain satisfied). Loosely speaking, the goal of the dual update is to increase the dual objective in such a way that more edges become tight, while ensuring existing alternating trees, blossoms and matched edges remain intact. The only dual variables we update are those belonging to nodes in an alternating tree. For each alternating tree $T$ we choose a dual change $\delta_T \geq 0$ and we *increase* the dual variable of every outer node $u$ with $y_u := y_u + \delta_T$ but *decrease* the dual variable of every inner node $u$ with $y_u := y_u - \delta_T$. Recall that each node in $T$ is either a regular node or a blossom, and if the node is a blossom then are changing the *blossom*'s dual variable (while leaving the dual variables of the nodes it contains unchanged). Note that this change ensures that all tight edges within a given alternating tree remain tight, but since outer node dual variables are increasing, it is possible that some of their neighbouring (non-tight) edges may *become* tight (hopefully allowing us to find an augmenting path between alternating trees in the next primal update). The constraints of the dual problem (the inequalities Equation (B.3b) and Equation (B.3c)) impose constraints on the choice of $\delta_T$; in particular, the slacks of all edges must remain non-negative, and *blossom*

dual variables must also remain non-negative.

There are many different valid strategies that can be taken for the dual update. In a *single tree* approach, we pick a single tree $T$ and update the dual variables only of the nodes in $T$ by the maximum amount $\delta_T$ such that the constraints of the dual problem remain satisfied (e.g. we change the dual variables until an edge becomes tight or a blossom dual variable becomes zero). In a *multiple tree fixed* $\delta$ approach, we update the dual variables of *all* alternating trees by the same amount $\delta_T$ (again by the maximum amount that ensures the dual constraints remain satisfied). In a *multiple tree variable* $\delta$ approach, we choose a different $\delta_T$ for each tree $T$. Our variant of the blossom algorithm (sparse blossom) uses a multiple tree fixed $\delta$ approach. See [131] for a more detailed discussion and comparison of these different strategies.

In Figure B.1(c) we give an example with two alternating trees, and visualise a dual variable as the radius of a circular region centred on its node. Visualising dual variables this way, an edge between two trivial nodes is tight if the regions at its endpoints touch. In this example, we update the dual variables (radiuses) until the two alternating trees touch, at which point the edge joining the two trees becomes tight, and we can augment the path between the roots of the two trees. Note that we can only visualise dual variables as region radiuses like this when they are non-negative. While dual variables of blossoms are always non-negative (as imposed by Equation (B.3c)), dual variables of regular nodes *can* become negative in general. However, when running the blossom algorithm on a *path graph*, the dual variable of every regular node is also always non-negative, owing to the structure of the graph. This can be understood as follows. Consider any regular inner node $v$ that is not a blossom, which by definition must have exactly one child outer node $w$ in its alternating tree (its match), as well as its one parent outer node $u$. Recall that the path graph is a complete graph where the weight $w(x,y)$ of each edge $(x,y)$ is the length of shortest path between nodes $x$ and $y$ in some other graph (e.g. in our case always the matching graph). Therefore there is also an edge $(u,w)$ in the path graph with weight $w(u,w) \leq w(u,v) + w(v,w)$, since we know that there is at least

one path from $u$ to $w$ of length $w(u,v)+w(v,w)$, corresponding to the union of the shortest path from $u$ to $v$ and the shortest path from $v$ to $w$. Therefore, we cannot have $y_v < 0$ without having $slack((u,w)) < 0$ which would violate Equation (B.3b). More specifically, if $y_v = 0$ then we know that the edge $(u,w)$ *must* be tight, which means we can form a new blossom from the blossom cycle $(u,v,w)$ and this blossom can become an *outer* node in the (possibly now trivial) alternating tree.

## B.2 Comparison between blossom and sparse blossom

In Table B.1 we summarise how some of the concepts in the traditional blossom algorithm translate into concepts in sparse blossom. If the traditional blossom algorithm is run on the path graph $\bar{\mathcal{G}}[\mathcal{D}]$ using a multiple tree approach (and with all dual variables initialised to zero at the start), then a valid state of blossom at a particular stage corresponds to a valid state of sparse blossom for the same problem at the appropriate point in the timeline. The dual variables in blossom define the region radiuses in sparse blossom (and these radiuses can be used to construct the corresponding exploratory regions). Likewise the edges in the alternating trees, blossom cycles and matches in traditional blossom can all be translated into compressed edges in the corresponding entities in sparse blossom. We note, however, that when multiple alternating tree events happen at the same time in sparse blossom, *any* ordering for the processing of these events is a valid choice. So just because we can translate a valid state of one algorithm to that of the other, does not imply that two implementations of the algorithms (or the same algorithm) will have the same sequence of alternating tree manipuations (indeed it is unlikely that they will). This correspondence between sparse blossom run on the matching graph and traditional blossom run on a path graph, and the correctness of blossom itself for finding a MWPM, is one way understanding why sparse blossom correctly finds a MWEM in the matching graph.

## B.3 Compressed tracking in Union-Find

Compressed tracking can be naturally adapted to the Union-Find decoder [60, 119], as shown in Figure B.2. Each detection event is initialised with a region, which we refer to as a cluster, to be consistent with Ref. [60]. Using the same approach as for the compressed tracking in sparse blossom, we can track the detection event that each detector node has been reached from, as well as the observables that have been crossed to reach it. More explicitly, let $\mathcal{S}(u)$ again denote the *source detection event* of a detector node $u$, and we initialise $\mathcal{S}(x) = x$ for each detection event $x$ at the start of the algorithm. We denote the set of logical observables crossed an odd number of times from a node's source detection event by $l(u)$, and we initialise $l(x)$ as the empty set for each detection event $x$. We grow a cluster $C$ by adding a node $v$ from an edge $e := (u, v)$ on the boundary of $C$ (i.e. an edge such that $u \in C$ and $v \notin C$). As we add $v$ to $C$, we set $\mathcal{S}(v) = \mathcal{S}(u)$ and $l(v) = l(u) \oplus l(e)$. Recall that $l(e)$ is the set of logical observables flipped by edge $e$ and $\oplus$ denotes the symmetric difference of sets. Since we store $l(e_i)$ as a bitmask for each edge $e_i \in \mathcal{E}$, the symmetric difference of two edges $l(e_i) \oplus l(e_j)$ can be implemented particularly efficiently using a bitwise XOR.

We represent each cluster as a *compressed cluster tree*. Each node in the compressed cluster tree corresponds to a detection event, in contrast to the cluster tree introduced in [60], where each node is a detector node. Each edge $c := (x, y)$ in the cluster tree is a compressed edge, representing a path $P$ through the detector graph between detection events $x$ and $y$. In contrast to compressed edges in sparse blossom, this path is in general not the minimum-length path. The compressed edge $c$ is assigned the logical observables crossed an odd number of times by $P$, denoted $l(c)$ or $l(x, y)$. We check which cluster a detector node $u$ belongs to by calling Find$(\mathcal{S}(u))$.

We modify the path compression step of the Find operation such that whenever a path $B$ (consisting of compressed edges) through the cluster tree between two detection events $f$ and $g$ is replaced by a compressed edge $c := (f, g)$, the set of logical observables $l(c)$ of the new compressed edge is calculated $l(c) := \bigoplus_{c_i \in B} l(c_i)$.

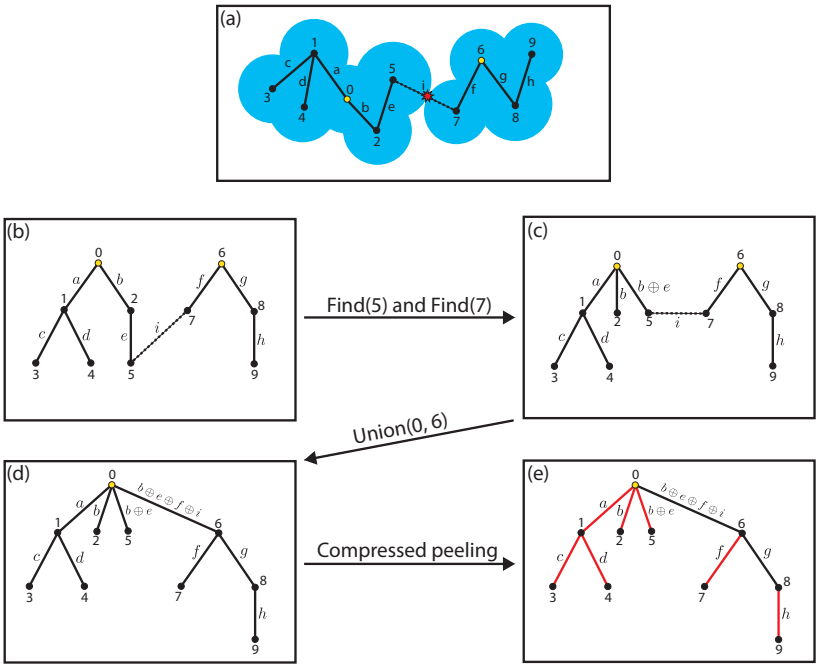



**Figure B.2:** Compressed tracking in Union-Find. (a) Two clusters collide. Each node in the cluster tree denotes a detection event, and each edge is a compressed edge representing a path between the two detection events in the detector graph (not necessarily a shortest path). Each edge is labelled with a letter denoting the bitmask of the observables crossed along the path it represents. This differs from traditional Union-Find implementations, where every detector node in a cluster is a node in the cluster tree. When two clusters collide, we store a compressed edge for the path between detection events along which the collision occurred (the collision edge). (b) The two cluster trees, along with the collision edge (dashed line). (c) After finding the source detection events (5 and 7) involved in the collision, we call Find(5) and Find(7). When using path compression (which here connects node 5 to the root node), we ensure the observable bitmask is kept up-to-date by taking the sum (modulo 2) of bitmasks along the path to the root node. (d) Union(0, 6) adds the smaller cluster as a subtree of the root node of the larger cluster (node 6 becomes a child of node 0). We store the observable bitmask along edge (0, 6), by taking the sum (modulo 2) of the observable bitmasks on edges (0, 5), (6, 7) and the collision edge (5, 7). (e) The combined cluster now has even parity. We use compressed peeling to highlight a set of edges (shown in red) such that each node is incident to an odd number of edges. We take the sum (modulo 2) of the observable bitmasks on these highlighted edges to find the predicted logical observable.

A similar modification can be made if path splitting or path halving is used instead of path compression for the Find operation.

The Union operation is adapted in a similar manner. Suppose a cluster $C_i$ collides with a cluster $C_j$ along an edge $(u,v)$. In other words, $(u,v)$ is an edge in the detector graph such that $u \in C_i$ and $v \in C_j$. We construct the collision edge $(\mathcal{S}(u),\mathcal{S}(v))$ from local information at the point of collision, and assign it the set of logical observables $l(\mathcal{S}(u),\mathcal{S}(v)) \coloneqq l(u) \oplus l(v) \oplus l(u,v)$. When we merge cluster $C_i$ with cluster $C_j$ using the Union operation, we add the root node $\mathcal{R}(C_i)$ of the smaller cluster tree (say $C_i$) as a child of the root node $\mathcal{R}(C_j)$ of the larger cluster tree $C_j$ by adding a compressed edge $c_{ij} \coloneqq (\mathcal{R}(C_i),\mathcal{R}(C_j))$ to the tree. We assign its set of logical observables $l(c_{ij})$ to be $l(c_{ij}) \coloneqq \bigoplus_{c_k \in P_{ij}} l(c_k)$, where $P_{ij}$ is the path through the tree between $\mathcal{R}(C_i)$ and $\mathcal{R}(C_j)$.

Finally, once all clusters have even parity (an even number of detection events, or connected to the boundary), we can apply the peeling algorithm [63] to the compressed cluster trees, which returns a set of compressed edges $\mathcal{P}$. We say the compressed edges in $\mathcal{P}$ are *highlighted* edges in the cluster tree. The set of logical observables that the decoder predicts to have been flippsed is then $\bigoplus_{c_i \in \mathcal{P}} l(c_i)$. This stage is much more efficient than the traditional peeling step of UF, as we do not need to construct a spanning tree in each cluster. Instead, we only run peeling on our compressed representation of the cluster trees. Compressed peeling is linear in the number of compressed edges in the cluster tree. For completeness, we give pseudocode for compressed peeling in Algorithm 2, however this is simply a recursive definition of the peeling algorithm of [63] applied to the case of a compressed cluster tree comprised of a graph of compressed edges joining detection events, rather than to a spanning tree subgraph of a conventional union-find cluster in the detector graph (comprised of detector nodes and detector graph edges). The procedure is recursive and takes as input a node $x$ in the compressed cluster tree, returning $p_x$ and $l_x$. Here, $l_x$ is the set of logical observables flipped by flipping the highlighted compressed edges that are descendants of $x$ in the cluster tree (a descendant edge of $x$ is an edge in the subtree rooted at $x$). The auxiliary variable $p_x$ (used in the recursion) is the

parity of the number of highlighted child edges of $x$ in the cluster (where a child edge is an edge connecting $x$ to one of its children in the cluster tree). Initially, we assume no node in the cluster tree is connected to the boundary, in which case we initialise $p_x^{init} = even$ for each node $x$. We run compressed peeling on the root node $r$ to find the set of logical observables $l_r$ flipped by all highlighted edges in the cluster tree. Note that $p_r$ should always be odd for the root node if there is no boundary.

Recall that peeling should find a set of highlighted edges (edges in $\mathcal{P}$) such that each node in the tree is incident to an odd number of highlighted edges. We will first show that $l_x$ (returned by compressed peeling) is indeed the set of logical observables flipped by highlighted edges that are descendants of $x$, and that $p_x$ is the parity of the number of highlighted child edges of $x$. Consider the base case that $x$ is a leaf node, in which case it has no child edges or descendants. In this case, compressed peeling correctly sets $p_x$ to $even$ (since we initialise $p_x^{init} = even$) and $l_x$ to the empty set. Now consider the inductive step, where $x$ has children $\mathcal{C}(x)$ in the cluster tree. For each $y \in \mathcal{C}(x)$ we highlight the edge $(x,y)$ if $p_y$ is even, and $p_x$ is set to the parity of the number of these highlighted child edges of $x$, as required. The main loop of the algorithm sets $l_x$ to

$$l_x = \left( \bigoplus_{y \in \mathcal{C}(x)} l_y \right) \oplus \left( \bigoplus_{(x,w):w \in \mathcal{C}(x),(x,w) \in \mathcal{P}} l(x,w) \right)$$

and if we assume $l_y$ is the set of logical observables flipped by highlighted descendant edges of $y$ then clearly $l_x$ is the set of logical observables flipped by highlighted descendant edges of $x$. Finally, note that since we apply compressed peeling to a tree, the function is called on each node $x$ exactly once, and we highlight an edge $(x,y)$ to a child $y$ of $x$ if and only if $p_y$ is even. Therefore, each node becomes incident to an odd number of highlighted edges, as required.

We haven't yet considered the boundary. If there is a compressed edge $(x,b)$ in a cluster tree $C$ connecting a detection event $x$ to the boundary $b$ (there can be at most one such edge since a cluster becomes neutral once it hits the boundary), we first add $l(x,b)$ to the solution and remove $(x,b)$ from $C$, then we set $p_x^{init} = odd$

before applying compressed peeling to the root node $r$ of the remaining cluster tree, adding the resulting $l_r$ to the solution.

**Algorithm 2** Compressed Peeling

```
procedure CompressedPeeling(x)
    p_x ← p_x^init                ▷ Parity of the number of highlighted child edges of x
    l_x ← {}                      ▷ Observables flipped by highlighted descendant edges of x
    for each child y of x do
        p_y, l_y ← CompressedPeeling(y)
        l_x ← l_x ⊕ l_y
        if p_y is even then
            l_x ← l_x ⊕ l(x,y)    ▷ Compressed edge (x,y) becomes highlighted
            flip p_x
    return p_x, l_x
```

## B.4 Worst-case running time

In this section, we will find a worst-case upper bound of the running time of sparse blossom. Note that this upper bound is likely loose, and furthermore differs greatly from the expected running time of sparse blossom for typical QEC problems, which we believe to be linear in the size of the graph. Let us denote the number of detector nodes by $n$, the number of edges in the detector graph by $m$ and the number of detection events by $q$. First, note that each alternating tree always contains exactly one unmatched detection event, in the sense that only a single growing region needs to become matched for the whole tree to shatter into matched regions. Therefore, the alternating tree events that grow the alternating tree, form blossoms or shatter blossoms (events of type a, c, d or e in Figure 2.2) do not change the number of detection events that remain to be matched. On the other hand, when a tree hits another tree (type b), the number of unmatched detection events reduces by two, and when a tree hits the boundary (type f), or a boundary match (type g), then the number of unmatched detection events reduces by one. We refer to an alternating tree event that reduces the number of unmatched detection events as an *augmentation*, and refer to a portion of the algorithm between consecutive augmentations as a *stage*. Clearly, there can be at most $q$ augmentations and at most $q$ stages.

We now bound the complexity of each augmentation, and of each stage. In

each stage, there are at most $O(q)$ blossoms formed or shattered, and at most $O(q)$ matches added to trees. When a blossom forms, each node it owns updates its cached topmost blossom ancestor and wrapped radius, with cost proportional to the depth of the blossom tree, and this step has complexity $O(nq)$. Additionally, every node owned by the blossom is rescheduled, with $O(m)$ cost. Updating the blossom structure and alternating tree structure (e.g. the compressed edges) is at most $O(q)$. In total, forming a blossom has a running time of at most $O(nq+m)$, and the same upper bound applies for shattering a blossom. When a match is added to an alternating tree, the complexity is $O(m)$ from rescheduling the nodes, which exceeds the $O(q)$ cost of updating the alternating tree structure. Finally, there is the cost of growing and shrinking regions. In each stage, a node can only be involved in $O(1)$ ARRIVE or LEAVE flooder events: once a region is growing, it (or its topmost blossom ancestor) continues to grow until the next augmentation. Therefore, in each stage the worst-case running time is dominated by the $O(nq^2+mq)$ cost associated with up to $q$ blossoms forming or shattering. There are $q$ stages, leading to a $O(nq^3+mq^2)$ worst-case complexity.

## B.5 Handling negative edge weights

Recall that an edge weight $\mathbf{w}[i] = \log((1-\mathbf{p}[i])/\mathbf{p}[i])$ can become negative since we can have $\mathbf{p}[i] > 0.5$. It is therefore necessary to to handle negative edge weights to decode correctly for these error models. For example, consider the distance three repetition code check matrix

$$H = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \tag{B.10}$$

with prior distribution $\mathbf{p} = (0.9, 0.9, 0.9)$ and an error $\mathbf{e} = (0,1,1)$ with syndrome $\mathbf{s} = H\mathbf{e} = (1,0,1)$. The two errors consistent with the syndrome are $(0,1,1)$, which has prior probability $0.1 \times 0.9^2 = 0.081$, and $(1,0,0)$, which has prior probability $0.9 \times 0.1^2 = 0.009$. Recall that MWPM decoding uses a weights vector $\mathbf{w} \in \mathbb{R}^3$ and

finds a correction $\mathbf{c} \in \mathbb{F}_2^3$ satisfying $H\mathbf{c} = \mathbf{s}$ of minimal weight $\sum_i \mathbf{w}[i]\mathbf{c}[i]$. Therefore, it is important that the edge weights $\mathbf{w}[i] = \log((1-\mathbf{p}[i])/\mathbf{p}[i])$ are indeed negative here, as this leads the decoder to predict the more probable (albeit higher *hamming* weight) $\mathbf{c} = (0,1,1)$ instead of the incorrect $\mathbf{c} = (1,0,0)$.

We now show how negative edge weights can be handled for the MWPM decoding problem for some check matrix $H \in \mathbb{F}_2^{n\times m}$ with weights vector $\mathbf{w} \in \mathbb{R}^m$ and an error $\mathbf{e} \in \mathbb{F}_2^m$ with syndrome $\mathbf{s} = H\mathbf{e} \in \mathbb{F}_2^n$. Even though sparse blossom only handles non-negative edge weights, we can still perform MWPM decoding when there are negative edge weights using the following procedure, which uses sparse blossom as a subroutine:

1. Define $\mathbf{b} \in \mathbb{F}_2^m$ such that $\mathbf{b}[i] = 1$ if $\mathbf{w}[i] < 0$ and $\mathbf{b}[i] = 0$ otherwise.

2. From $\mathbf{b}$, define adjusted edge weights $\mathbf{v} \in \mathbb{R}^m$ where $\mathbf{v}[i] \coloneqq (1-2\mathbf{b}[i])\mathbf{w}[i]$, as well as the adjusted syndrome $\mathbf{s}' = \mathbf{s} + H\mathbf{b}$.

3. Use sparse blossom to find a correction $\mathbf{c}'$ satisfying $H\mathbf{c}' = \mathbf{s}'$ of minimal adjusted weight $\sum_i \mathbf{v}[i]\mathbf{c}'[i]$. Note that the definition of $\mathbf{v}$ guarantees that every element $\mathbf{v}[i]$ is non-negative.

4. Return the correction $\mathbf{c} \coloneqq \mathbf{c}' + \mathbf{b}$, which is guaranteed to satisfy $H\mathbf{c} = \mathbf{s}$ with minimal total weight $\sum_i \mathbf{w}[i]\mathbf{c}[i]$.

We can verify the correctness of this procedure as follows. Firstly, note that $\mathbf{c}'$ satisfies $H\mathbf{c}' = \mathbf{s}'$ if and only if $\mathbf{c}$ satisfies $H\mathbf{c} = \mathbf{s}$. Secondly, note that

$$\sum_i \mathbf{w}[i]\mathbf{c}[i] = \sum_i (\mathbf{w}[i]\mathbf{c}'[i] + \mathbf{w}[i]\mathbf{b}[i] - 2\mathbf{w}[i]\mathbf{b}[i]\mathbf{c}'[i]) = \sum_i \mathbf{v}[i]\mathbf{c}'[i] + \sum_i \mathbf{w}[i]\mathbf{b}[i]. \tag{B.11}$$

Therefore, if we find a $\mathbf{c}'$ of minimal adjusted weight $\sum_i \mathbf{v}[i]\mathbf{c}'[i]$ satisfying $H\mathbf{c}' = \mathbf{s}'$, it is guaranteed that the correction $\mathbf{c} \coloneqq \mathbf{c}' + \mathbf{b}$ has minimal weight $\sum_i \mathbf{w}[i]\mathbf{c}[i]$ satisfying $H\mathbf{c} = \mathbf{s}$. Intuitively, wherever we have an error mechanism with high error probability ($> 50\%$), we are re-framing the decoding problem to instead predict if the error mechanism *didn't* occur. The handling of negative edge weights was also discussed

in [107], as well as in Corollary 12.12 of [132], where what Korte et al. refer to as a minimum-weight $T$-join is equivalent to what we call a MWEM.

| Blossom concepts (multiple tree approach, applied to $\bar{\mathcal{G}}[\mathcal{D}]$) | Sparse blossom concepts |
|---|---|
| Dual variable $y_u$ (for a node $u \in \mathcal{D}$) or $y_S$ (for a set of nodes $S \subseteq \mathcal{D}$ of odd cardinality at least three) | Radius $y_R$ of a graph fill region $R$, an exploratory region containing nodes and edges in the matching graph including an odd number of detection events. |
| A tight edge $(u,v)$ between two nodes $u$ and $v$ in $\mathcal{D}$ that is in the matching or belongs to an alternating tree or a blossom cycle. Since $(u,v)$ is an edge in the path graph, its weight is the length of a shortest path between the two detection events $u$ and $v$ in the matching graph $\mathcal{G}$ (found using a Dijkstra search when the path graph was constructed). It is *tight* since the dual variables associated with $u$ and $v$ result in zero slack as defined in Equation (B.4). | A compressed edge $(u,v)$ associated with a region edge, belonging to a match, alternating tree or blossom cycle. The compressed edge $(u,v)$ represents a shortest path between two detection events $u$ and $v$ in the matching graph $\mathcal{G}$. Associated with $(u,v)$ is the set of logical observables $l(u,v)$ flipped by flipping edges in $\mathcal{G}$ along the corresponding path between $u$ and $v$. The two regions in the corresponding region edge are touching (the edge is tight). |
| An edge $(u,v)$ between two nodes $u$ and $v$ in $\mathcal{D}$ that is *not* tight. Its weight is the length of the shortest path between $u$ and $v$ in the matching graph $\mathcal{G}$, found using a Dijkstra search when constructing the path graph. For typical QEC problems, the vast majority of edges in $\bar{\mathcal{G}}[\mathcal{D}]$ never become tight, but for standard blossom they still must be explicitly constructed using a Dijkstra search. | There is no analogous data structure for this edge in sparse blossom. The shortest path between $u$ and $v$ is not currently fully explored by graph fill regions. This means that either the path has not yet been discovered by sparse blossom (and may never be), or perhaps it had previously been discovered (belonging to a region edge) but at least one of the regions owning $u$ or $v$ since shrunk (e.g. it was matched and then became a shrinking region in an alternating tree). |
| In the dual update stage, update each dual variable $y_u$ of an outer node with $y_u \coloneqq y_u + \delta$ and update each dual variable $y_u$ of an inner node with $y_u \coloneqq y_u - \delta$. The variable $\delta \in \mathbb{R}_{\geq 0}$ is set to the maximum value such that the dual variables remain a feasible solution to the dual problem. | As time $\Delta t$ passes, each growing region $R$ explores the graph and its radius $y_R$ increases by $\Delta t$ and each shrinking region $R$ shrinks in the graph and its radius $y_R$ decreases by $\Delta t$. Eventually at $\Delta t = \delta$ a collision or implosion occurs (one of the matcher events in Figure 2.2), which must be handled by the matcher. |
| An edge $(u,v)$ between a node $u$ in an alternating tree $T$ and a node $v$ in another alternating tree $T'$ becomes tight after a dual update. The path between the root of $T$ and the root of $T'$ is augmented and all nodes in the two trees become matches. | A growing region $R$ of an alternating tree $T$ collides with a growing region $R'$ of another alternating tree $T'$. The collision edge $(R,R')$ is constructed from local information at the point of collision. $R$ is matched to $R'$ with the collision edge $(R,R')$ as a match edge. All other regions in the trees also become matched. All regions in the two trees become frozen. |
| An edge $(u,v)$ between two outer nodes $u$ and $v$ in the same tree $T$ becomes tight after a dual update. This forms an odd-length cycle in $T$ which becomes a *blossom*, which itself becomes an outer node in $T$. | Two growing regions $R$ and $R'$ from the same alternating tree $T$ collide. The discovered collision edge $(R,R')$ as well as the regions and tree-edges along the path between $R$ and $R'$ in $T$ become a blossom cycle in a newly formed blossom. The blossom starts growing, and is now a growing region in $T$. |

**Table B.1:** The correspondence between concepts in the standard blossom algorithm and concepts in sparse blossom. For the standard blossom algorithm we assume a "multiple tree with fixed $\delta$" approach is used, and further we assume that the algorithm is applied to the path graph $\bar{\mathcal{G}}[\mathcal{D}] = (\mathcal{D}, \bar{\mathcal{E}})$.

# Appendix C

# Belief propagation review

## C.1 Belief propagation algorithm

The belief propagation (BP) algorithm (also known as *sum-product algorithm*) has been shown to be effective at decoding classical LDPC codes [144, 143]. It has a running time that is linear in the block length of the code. In this section, we will review BP for syndrome decoding of a binary $r \times n$ parity check matrix $H$. We assume we have measured a syndrome $\mathbf{s} \in \mathbb{F}_2^r$, and the role of the decoder is to infer a likely error $\mathbf{e}$, which must satisfy $H\mathbf{e} = \mathbf{s}$. Note that here $H$ could correspond to the parity check matrix of a binary linear code, or to a detector check matrix (Section 1.4.2).

BP is a message passing algorithm, in which messages are passed along the edges of a Tanner graph. The Tanner graph is a graphical model describing a factorisation of the joint probability distribution of the error $\mathbf{e}$ and syndrome vector $\mathbf{s}$. The factorised joint probability distribution $P(\mathbf{e},\mathbf{s})$ is

$$\begin{aligned} P(\mathbf{e},\mathbf{s}) &= P(\mathbf{e})\mathbb{1}[\mathbf{s} = H\mathbf{e}] \\ &= \prod_i P(e_i) \prod_j \mathbb{1}[s_j = \sum_{a \in \partial j} e_a] \end{aligned} \tag{C.1}$$

where $P(\mathbf{e})$ is the prior probability distribution over the noise vector (i.e. $P(e_i) = p$ for a binary symmetric channel with a probability $p$ of error) and $\partial j \subseteq \{1,2,\ldots,n\}$ denotes the indices of the variables involved in the $j^{\text{th}}$ parity check in $H$. The Tanner graph $\mathcal{T}(H)$ of Equation (C.1) has a factor node $f_j$ for each parity check and a

variable node $v_i$ corresponding to the random variable (error mechanism) associated with each element $e_i$ of $\mathbf{e}$. There is an edge between $f_j$ and $v_i$ in the factor graph if variable $i$ is involved in parity check $j$.

The problem BP approximately solves is the bitwise decoding problem of finding the marginal posterior probabilities of each bit, $P(e_i = 1|\mathbf{s})$. BP solves the bitwise decoding problem exactly on tree graphs, but is not exact on graphs that contain loops. While Tanner graphs of classical and quantum codes generally do contain loops, BP can often still perform well in practice provided the girth of the graph is sufficiently large.

When implementing BP, we can represent each binary random variable $U$ using a log-likelihood ratio (LLR) defined as $L(U) = \log(P(U = 0)/P(U = 1))$. BP involves repeating multiple iterations, where each iteration consists of a "check-to-variable" step and a "variable-to-check" step. In the check-to-variable step, which iterates over the rows of $H$, each parity check factor $f_j$ sends a message $Q_{f_j \to v_i}$ to its adjacent variable nodes $\{v_i : i \in \partial_j\}$:

$$Q_{f_j \to v_i} \coloneqq (-1)^{s_j} 2 \tanh^{-1} \left( \prod_{i' \in \partial j \setminus i} \tanh \left( Q_{v_{i'} \to f_j}/2 \right) \right) \tag{C.2}$$

where each $Q_{v_i \to f_j}$ is initialised before the first iteration to the LLR of the prior of variable $v_i$, which is $L(v_i) = \log((1-p)/p)$ for the binary symmetric channel with bit-flip probability $p$.

In the vertical step, which iterates over the columns of $H$, each variable node $v_i$ sends a message $Q_{v_i \to f_j}$ to its adjacent factor nodes $\{f_j : j \in \partial_i\}$:

$$Q_{v_i \to f_j} \coloneqq L(v_i) + \sum_{j' \in \partial i \setminus j} Q_{f_{j'} \to v_i}. \tag{C.3}$$

We can also obtain an estimate $Q_{v_i}$ of the LLR of the posterior of each variable $v_i$ at each step:

$$Q_{v_i} \coloneqq L(v_i) + \sum_{j \in \partial i} Q_{f_j \to v_i}. \tag{C.4}$$

These estimates of the posteriors of each variable are also called the *soft decisions*.

From the soft decisions we then obtain *hard decisions* $\mathbf{c} \in \mathbb{F}_2^n$, where $c_i := 0$ if $Q_{v_i} > 0$ and $c_i := 1$ otherwise. If $\mathbf{c}$ is a *valid correction* satisfying $H\mathbf{c} = \mathbf{s}$ then BP stops and returns $\mathbf{c}$ as a correction. If $H\mathbf{c} \neq \mathbf{s}$ then BP instead continues and runs the next iteration of the check-to-variable and variable-to-check steps. If BP reaches a maximum number of iterations without finding a valid correction then it instead returns a heralded failure. There are therefore two possible failure mechanisms from BP: it either returns a valid correction that removes the syndrome but leaves a residual error $\mathbf{e}+\mathbf{c}$ corresponding to a logical failure, or it fails to find a valid solution, resulting in a heralded failure.

## C.2 The tanh update

The update rules in Equation (C.2) and Equation (C.3) are called the "tanh rule". The tanh update rule involves expensive hyperbolic tangent computations and has numerical instability issues, which we will review in this section. We will then provide alternative update rules in the sections that follow which remedy these issues. We refer the reader to Ref. [51] for more details.

The log-likelihood ratio $L(U)$ of a binary random variable $U$ is defined as

$$L(U) = \log\left(\frac{P(U=0)}{P(U=1)}\right) \tag{C.5}$$

where here the natural logarithm is used. It follows that $P(U=0) = \frac{e^{L(U)}}{1+e^{L(U)}}$ and $P(U=1) = \frac{1}{1+e^{L(U)}}$. In this representation, the sum (modulo 2) of two independent binary random variables $U$ and $V$ is given by

$$\begin{aligned} L(U\oplus V) &:= \log\left(\frac{P((U\oplus V)=0)}{P((U\oplus V)=1)}\right) \\ &= \log\left(\frac{P(U=0)P(V=0)+P(U=1)P(V=1)}{P(U=0)P(V=1)+P(U=1)P(V=0)}\right) \\ &= \log\left(\frac{e^{L(U)+L(V)}+1}{e^{L(U)}+e^{L(V)}}\right) \\ &= 2\tanh^{-1}\left(\tanh\left(\frac{L(U)}{2}\right)\tanh\left(\frac{L(V)}{2}\right)\right). \end{aligned} \tag{C.6}$$

The identity in Equation (C.6) is the origin of the "tanh rule" check-to-variable update rule for implementing BP using LLRs (see Equation (C.2)). However, using the tanh rule directly in an implementation of BP can lead to at least two problems. Firstly, once the LLR messages become sufficiently large, then a direct implementation of the check-to-variable tanh rule updates as given in Equation (C.2) using floating point arithmetic is numerically unstable. Secondly, the hyperbolic tangent computations can be too expensive for some hardware implementations.

The numerical instability of the tanh rule can be understood by considering the simpler problem of evaluating

$$f(x) \coloneqq \tanh^{-1}(\tanh(x)) = x \tag{C.7}$$

using floating point arithmetic. Consider the regime $x \gg 0$ where $\tanh(x) \approx 1$. In this regime, the round-off error when evaluating $\tanh(x)$ using IEEE double precision is approximately constant and equal to $\partial_{\tanh(x)} \approx 2^{-54}$. This results in an error in $f(x)$ of

$$\partial_{f(x)} \approx \frac{df(x)}{d\tanh(x)} \partial_{\tanh(x)} \tag{C.8}$$

$$\approx \frac{e^{2x}}{4} \partial_{\tanh(x)} \approx \frac{e^{2x}}{2^{56}}. \tag{C.9}$$

Since tanh and $\tanh^{-1}$ are both antisymmetric, the same holds for $x \ll 0$, i.e. the numerical error scales exponentially in $|x|$, as $\partial_{f(x)} \approx e^{2|x|}/2^{56}$, as also shown empirically in Fig. C.1. One approach to handle these underflows is to cap the magnitude of the log-likelihood ratios when evaluating the check-to-variable messages. Alternatively, Equation (C.2) can be implemented exactly using the Jacobian approach, outlined in Appendix C.2.1, which does not suffer from these numerical stability issues.

However, the Jacobian approach requires evaluating logarithms and exponentials and, as with the tanh rule, is not so amenable to fast implementations in hardware. The min-sum update rule is an *approximation* of the tanh rule which avoids its

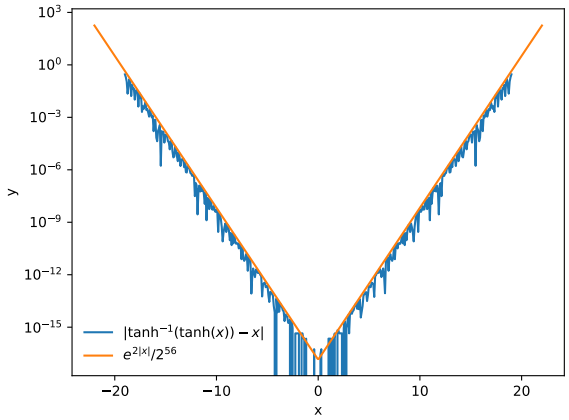



**Figure C.1:** The numerical error caused by underflow when evaluating $\tanh^{-1}(\tanh(x))$.

numerical instability issues as well as requiring only real additions, simplifying its implementation. We review the min-sum update rule in Appendix C.2.2

### C.2.1 The Jacobian approach

The check-to-variable tanh update rule of Equation (C.2) can be rewritten in a form that does not suffer from numerical underflow issues. To derive this alternative update rule [51, 76], first notice that

$$\begin{aligned} e^a + e^b &= e^a(1+e^{b-a}) \\ &= e^{a+\log\left(1+e^{b-a}\right)} \\ &= e^{\max(a,b)+\log\left(1+e^{-|a-b|}\right)} \end{aligned} \tag{C.10}$$

from which we see that we can rewrite Equation (C.6) as

$$\begin{aligned} L(U \oplus V) &= \log\left(\frac{e^{L(U)+L(V)}+1}{e^{L(U)}+e^{L(V)}}\right) \\ &= L(V) + \max(L(U), -L(V)) - \max(L(U), L(V)) \\ &\quad + \log\left(1+e^{-|a+b|}\right) - \log\left(1+e^{-|a-b|}\right) \\ &= \operatorname{sign}(L(U))\operatorname{sign}(L(V))\min(|L(U)|, |L(V)|) \\ &\quad + \log\left(1+e^{-|a+b|}\right) - \log\left(1+e^{-|a-b|}\right). \end{aligned} \tag{C.11}$$

In order to implement the log-likelihood check-to-variable update rule using Equation (C.11) [51], we first assume that the factor $f_j$ is connected to $k$ variable nodes on the factor graph labelled $v_1, v_2, \ldots, v_k$, (i.e. we assume $\partial j = \{1, 2, \ldots, k\}$). We now define $f_1 = v_1, f_2 = f_1 \oplus v_2, \ldots, f_k = f_{k-1} \oplus v_k$ and $g_k = v_k, g_{k-1} = g_k \oplus v_{k-1}, \ldots, g_1 = g_2 \oplus v_1$, and then calculate all $L(f_i)$ and $L(g_i)$ using Equation (C.11) recursively (e.g. $L(f_i) = L(f_{i-1} \oplus v_i)$ and $L(g_i) = L(g_{i+1} \oplus v_i)$). Now since $v_1 \oplus v_2 \oplus \ldots \oplus v_k = z_j$, we can write $v_i = z_j \oplus f_{i-1} \oplus g_{i+1}$, from which we see that $L(v_i) = (-1)^{z_j} L(f_{i-1} \oplus g_{i+1})$. Therefore, the horizontal messages are computed using this 'Jacobian rule' for $1 < i < k$ as

$$Q_{f_j \to v_i} = (-1)^{z_j} L(f_{i-1} \oplus g_{i+1}), \tag{C.12}$$

and the remaining messages are calculated as $Q_{f_j \to v_0} = (-1)^{z_j} g_1$ and $Q_{f_j \to v_k} = (-1)^{z_j} f_{k-1}$.

### C.2.2 The min-sum update

While the tanh update rule and Jacobian update rule can both be implemented efficiently in a time linear in the blocklength, for some practical applications it becomes necessary to use an even simpler decoder, which avoids the costly evaluation of the hyperbolic tangent. A widely used update rule is the min-sum rule, which approximates the tanh rule while being significantly simpler to implement in hardware [79, 51]. From Equation (C.11) we see that $|L(U \oplus V)| \leq \min(|L(U)|, |L(V)|)$, which leads us to the min-sum check-to-variable update rule

$$Q_{f_j \to v_i} = (-1)^{z_j} \alpha \min_{i' \in \partial j \setminus i} (Q_{v_{i'} \to f_j}) \prod_{i' \in \partial j \setminus i} \operatorname{sign}(Q_{v_{i'} \to f_j}) \tag{C.13}$$

where here $0 \leq \alpha \leq 1$ is a constant chosen to better approximate the tanh rule. By setting $\alpha = 2^{-a} + 2^{-b}$ for some $a, b \in \{1, 2, 3\}$, multiplications can be replaced with bit shifts and additions, which simplifies FPGA implementations [196].

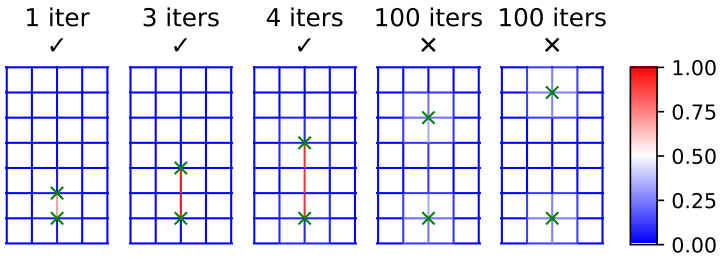



**Figure C.2:** The marginals output by BP for an $L = 15$ toric code for five different syndromes (and with a prior of 0.05 for each variable node in the Tanner graph). We consider the problem of decoding $Z$ errors using $X$ stabiliser measurements, and a qubit is associated with each edge and an X stabiliser with each vertex of the lattice. For each syndrome we only show a $4 \times 7$ region of the lattice. A -1 $X$ stabiliser measurement is displayed as a green cross, and the colour of each edge shows the marginal probability output by BP (according to the colour bar on the right). The title for each example syndrome shows the number of iterations used by BP, as well as a tick if BP converged or a cross if it did not converge. We use a maximum number of iterations of 100.

## C.3 Locality of BP

In Figure C.2 we consider the simple problem of decoding a single vertical string-like error in the 2D toric code using BP. In other words, the syndrome is a pair of -1 $X$ stabiliser measurements (detection events), separated by some number $l$ of vertical edges in a vertical column of the lattice. We have deliberately set up the problem in such a way that there is only a single minimum-weight solution, to reduce the impact of "split-beliefs" due to degenerate solutions in quantum codes.

We find that, for $l \leq 3$, BP converges quickly, as shown by the marginals in the left three examples in Figure C.2. However, when the defects are separated by 4 or more edges (e.g. $l = 4$ and $l = 5$ for the right two examples in Figure C.2), BP instead fails to converge. This suggests that information is only propagated effectively by BP within some local region of the lattice. Since BP is exact on tree graphs, and split beliefs are not a problem for this example, we expect that this issue arises due to loops in the Tanner graph. For the 2D toric code, the shortest loop in the Tanner graph for $Z$ errors (e.g. considering only the $X$ check matrix) has length 8; for each length-4 loop around a face of the lattice, there is a corresponding length-8 loop in the Tanner graph. Similarly, the defects separated by $l = 4$ edges in the lattice are in fact also separated by 8 edges in the Tanner graph. This suggests that the ability of BP to effectively

propagate information significantly degrades beyond a radius in the Tanner graph that corresponds to its girth (some information propagates, but the strength of the signal is reduced). We refer to this as the problem of *bounded information spread* in BP. Note that we are not claiming that *no* information propagates beyond the radius equal to the girth. Indeed, we verified that there is still enough information in the BP marginals for $l = 4$ and $l = 5$ for OSD to find the minimum-weight correction in Figure C.2, since the marginal probabilities are slightly larger along the minimum-weight path (albeit much less than 0.5). However, in a more realistic setting in which more defects are present, we might expect the messages propagated from nearby defects (within the radius equal to the girth) to overwhelm or 'hide' the much weaker messages passed from further away in the Tanner graph. Therefore, there may be limited benefit in increasing the maximum number of iterations of BP with system size for codes with Tanner graphs that have constant girth, and so it is reasonable to leave this parameter as a constant (30 in our case). Note that the full Tanner graph of a quantum code or stabiliser circuit in general has girth 4, in contrast to the Tanner graph of either the $X$ check matrix or $Z$ check matrix alone when decoding $X$ and $Z$ errors independently, as we have done here, which potentially worsens the problem of bounded information spread in BP. This is due to the commutativity condition of quantum codes. For example, for the full Tanner graph of a CSS quantum code, each $X$ stabiliser must overlap on an even number of qubits with each $Z$ stabiliser. Consider an $X$ stabiliser $S_X$ and $Z$ stabiliser $S_Z$ which overlap on two qubits $i$ and $j$. In the full Tanner graph there will be variable nodes corresponding to $Y$ errors $Y_i$ and $Y_j$. As a result there will be a loop (containing four edges) in the full Tanner graph that visits nodes in the order $(S_X, Y_i, S_Z, Y_j, S_X)$.

# Appendix D

# Subsystem codes

## D.1 Methods for numerical simulations

### D.1.1 Matching graph edge weights

In order to decode the subsystem surface codes using minimum-weight perfect matching, we construct a matching graph, where each individual fault that can occur flips an edge in the matching graph [64, 36]. We assign each edge a weight $w = \log((1-p)/p)$, where $p$ is the total probability that any individual fault will result in the edge being flipped [64, 202, 119].

In this section we will explain how we constructed the matching graph and calculated the edge weights, however we note that this work was conducted before the introduction of the Stim software package [91], which automates the construction

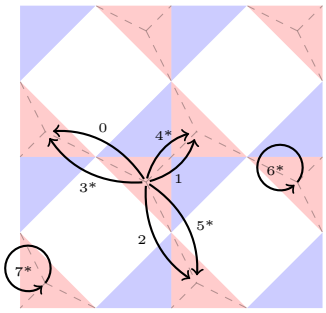



**Figure D.1:** The different types of edges in the 3D matching graph of the subsystem surface code for $X$-type checks only, when all $X$-type gauge operators are fixed. Each unique edge type is labelled with a number. If an asterisk is present in the label, the edge is from time step $t$ to $t+1$, otherwise the edge is purely space-like. The whole $X$ matching graph for a single time step is drawn with grey dashed lines.

of the matching graph given a description of the circuit, detectors, logical observables and noise model.

We will first consider the matching graph obtained by only measuring $X$-type check operators and fixing *all* $X$-type gauge operators as stabilisers. We will see later that all other matching graphs for arbitrary homogeneous schedules can be obtained by merging edges and/or nodes in this matching graph. There are two types of $X$-type gauge operators in the subsystem surface code, as shown in Figure 6.2, labelled by 1 and 3, which we will refer to as $T_1$ and $T_3$, respectively. Every space-like or diagonal edge is from a $T_1$ to a $T_3$ (or vice versa), and the neighbourhood of every triangle operator with the same label is identical. All seven types of edges in the matching graph for $X$-type checks are shown in Figure D.1. All edges are undirected, but are denoted by directed arrows in the diagram to remove any ambiguity in the definition of the diagonal edges. The purely space-like edges are labelled 0, 1 and 2, purely time-like errors are labelled 6 and 7 and diagonal edges are labelled 3, 4 and 5. Diagonal and time-like errors are drawn from time step $t$ to time step $t+1$, whereas space-like edges connect nodes within a single time step. Therefore, each node in this matching graph has degree 8 (since each node is both the source and target of a time-like edge).

If an $X$-type check operators is measured directly after a $Z$-type check operator that anti-commutes with it, then this $X$-type check operator cannot be fixed, and the matching graph shown in Figure D.1 is not quite valid. However, we can use the node merging procedure detailed in Section 4.2.1 to give the matching graph the correct structure. When the $X$-type check operators within a face of the lattice cannot be fixed, then the corresponding $X$-type matching graph nodes from that face (each node $v_{g_i}$ corresponding to a gauge factor $g_i$) are *merged* into a single node $v_s$. The edges incident to $v_s$ each correspond to an edge incident to a gauge factor vertex $v_{g_i}$. This process can result in more than one edge (a multi-edge) between the same pair of nodes (such as for time-like edges in homogeneous $(ZX)^r$ schedules). When this happens, we replace the multi-edge with a single edge, and assign it a flip probability equal to the probability that an odd number of edges in the multi-edge would flip.

| Edge type | $G_1^X$ | $G_1^Z$ | $G_2^Z$ | $P_X$ | $M_X$ |
|---|---|---|---|---|---|
| 0 | 2 | $2r_Z$ | 0 | 0 | 0 |
| 1 | 2 | $2r_Z$ | $2r_Z$ | 0 | 0 |
| 2 | 2 | $2r_Z$ | $2r_Z$ | 0 | 0 |
| 3 | 2 | 0 | 0 | 0 | 0 |
| 4 | 2 | 0 | 0 | 0 | 0 |
| 5 | 2 | 0 | 0 | 0 | 0 |
| 6 | 3 | 0 | 0 | 1 | 1 |
| 7 | 3 | 0 | 0 | 1 | 1 |

**Table D.1:** Number of single faults that can cause each type of edge to flip in the 3D matching graph for $X$-type check operators. Each $G_1^X$ or $G_1^Z$ fault is a single Pauli error arising from a CNOT gate in the measurement circuit for an $X$-type or $Z$-type gauge operator respectively. Each $G_2^Z$ fault is a *pair* of Pauli errors arising from a single CNOT gate in the measurement circuit for a $Z$-type gauge operator. $P_X$ and $M_X$ are state preparation and measurement errors in the $X$-type check operator measurement schedule, respectively. $r_Z$ is the number of rounds of $Z$-type check operator measurements that have occurred since the last $X$-type check operator measurement. For example, $r_Z = 1$ always for $(ZX)^r$ schedules, and $r_Z = 2$ always for $(ZZX)^r$ schedules. The edge types are shown in Figure D.1. Faults for the $Z$ matching graph can be found by exchanging $Z$ and $X$ in the table.

In order to calculate the probability $p$ that each edge flips (both for edge weights and for simulations), we count the number of single faults (of each type) that can lead to each type of edge flipping. These counts are given in Table D.1 for the $X$ matching graph (for $X$-type check operators). The operators $G_1^X$ and $G_1^Z$ are Pauli errors from CNOT gates in the $X$ or $Z$ measurement schedule respectively, corresponding to either a $XI$, $IX$ or $XX$ error acting after the gate. In the standard depolarising model, $G_1^X$ or $G_1^Z$ errors occur with probability $4p/15$. See Table D.2 for the gate error probabilities under the independent noise model we use. $G_2^Z$ errors correspond to a pair of $G_1^Z$ errors from the same CNOT gate in the $Z$ measurement circuit that both cause the same edge to flip. For example, both $XI$ and $XX$ errors on a CNOT gate may cause the same edge to flip, and since these errors are mutually exclusive on the same gate, the chance of either of these errors occurring is exactly twice the probability that one of them occurs. The number of $G_1^Z$ or $G_2^Z$ errors that can cause an edge to flip depends on $r_Z$, the number of $Z$ check operator measurements that have occurred since the most recent prior $X$ check operator measurement. We can recover the matching graph for the standard $(ZX)^r$ schedule used in Ref. [36] by

setting $r_Z = 1$ and merging all nodes within each face (up to small differences in the error model, shown in Table D.2).

### D.1.2 Noise models

We consider two different types of noise models: a circuit-level depolarising noise model, and a circuit-level independent noise model. The depolarising noise model is widely used in the literature, and is useful for comparing to previous work. Later we will consider biased noise, for which we use the independent noise model.

The circuit-level depolarising noise model is the same as that used in Refs. [48, 119], and is parameterised by a single variable $p$. Ancilla state preparation and measurement errors each occur with probability $2p/3$. With probability $p$, each CNOT gate is followed by a two-qubit Pauli error drawn uniformly from $\{I,X,Y,Z\}^{\otimes 2} \setminus I \otimes I$. A single qubit Pauli error drawn uniformly from $\{X,Y,Z\}$ occurs with probability $p$ after each idle single qubit gate location. Note that many of our syndrome extraction circuits are fully parallelised, and do not contain single qubit gates or idle locations.

In our circuit-level independent noise model, $Z$-type errors and $X$-type errors are independent. For a given error probability parameterised by $p_0$, we choose a high-rate error probability for $Z$-type errors $p_Z = p_0\eta/(\eta+1)$ and the low-rate error probability $p_X = p_0/(\eta+1)$ for $X$-type errors. The bias $\eta = p_Z/p_X$ parameterises the relative strengths of $Z$-type and $X$-type errors. The total probability of any error is:

$$\begin{aligned} p_{tot} &= 1-(1-p_X)(1-p_Z) \\ &= p_0 - \frac{p_0^2\eta}{(\eta+1)^2}. \end{aligned} \tag{D.1}$$

Each with probability $p_Z$, a CNOT gate is followed by an error in $\{IZ, ZI, ZZ\}$, chosen uniformly at random, an $X$-type ancilla is prepared or measured in an orthogonal state, and a single qubit idle for one time step undergoes a $Z$ error. Similarly, each with probability $p_X$, a CNOT gate is followed by an error randomly chosen from $\{IX, XI, XX\}$, a $Z$-type ancilla is prepared or measured in an orthogonal state, and a single qubit idle for one time step undergoes an $X$ error. Biased noise models are

| Error type | $G_1^X$ | $G_2^X$ | $G_1^Z$ | $G_2^Z$ | $P_X$ | $M_X$ | $P_Z$ | $M_Z$ |
|---|---|---|---|---|---|---|---|---|
| Depolarising | $\frac{4}{15}p$ | $\frac{8}{15}p$ | $\frac{4}{15}p$ | $\frac{8}{15}p$ | $\frac{2}{3}p$ | $\frac{2}{3}p$ | $\frac{2}{3}p$ | $\frac{2}{3}p$ |
| Independent | $\frac{1}{3}p_X$ | $\frac{2}{3}p_X$ | $\frac{1}{3}p_Z$ | $\frac{2}{3}p_Z$ | $p_X$ | $p_X$ | $p_Z$ | $p_Z$ |
| Ref. [36] | $\frac{1}{4}p$ | $\frac{1}{2}p$ | $\frac{1}{4}p$ | $\frac{1}{2}p$ | $p$ | $p$ | $p$ | $p$ |

**Table D.2:** The probability of a fault occurring for each type of circuit element under the two error models considered in this work, as well as for the depolarising error model used in Ref. [36] for reference.

common in many physical realisations of quantum computers, and bias-preserving CNOT gates can be realised using stabilized cat qubits [165]. We note that our techniques significantly improve performance even for small finite bias ($\eta \leq 10$), which may be achievable even with CNOT gates that do not fully preserve bias, as is the case in many architectures [7, 99].

The probability of each different type of circuit element undergoing a fault for our two error models (as well as the error model in Ref. [36] for comparison) is given in Table D.2.

## D.2 Broader applications of our techniques

### D.2.1 Inhomogeneous schedules

We have so far only considered homogeneous schedules, however sometimes it may be advantageous to use schedules that are *inhomogeneous*, where check operators in different faces of the lattice are given different schedules.

As an example, consider two different unparallelised $ZX^4$ schedules, which we call $L_0$ and $L_1$, obtained by omitting three quarters of the $Z$ check operator measurements in the $ZX$ schedule, and such that $L_1$ is identical to $L_0$ other than a lag of 4 check operator measurements. A section of 8 rounds of $X$ check operator measurements for these schedules looks like

| $(ZX)^8$ | $Z$ | $X$ | $Z$ | $X$ | $Z$ | $X$ | $Z$ | $X$ | $Z$ | $X$ | $Z$ | $X$ | $Z$ | $X$ | $Z$ | $X$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $L_0$ | | $X$ | | $X$ | | $X$ | $Z$ | $X$ | | $X$ | | $X$ | | $X$ | $Z$ | $X$ |
| $L_1$ | | $X$ | $Z$ | $X$ | | $X$ | | $X$ | | $X$ | $Z$ | $X$ | | $X$ | | $X$ |

where each column corresponds to a measurement round of either $X$-type or $Z$-type check operators. We can assign either the $L_0$ or $L_1$ schedule to each face of the

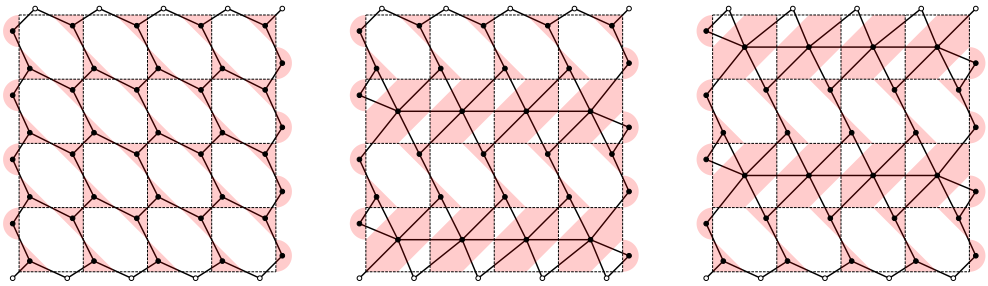


**Figure D.2:** Matching graphs ($X$-type) for the $L = 5$ subsystem surface code with triangle operators fixed in all rows (left), odd rows (middle) and even rows (right). Filled and hollow circles correspond to stabilisers and boundary nodes respectively.

planar subsystem surface code independently, since each schedule is a subset of the $ZX$ schedule, for which we have a consistent measurement circuit for every face. Let $G_0^X$ be the set of $X$ triangle operators in faces assigned the $L_0$ schedule, and let $G_1^X$ be the set of $X$ triangle operators in faces assigned the $L_1$ schedule. Note that in each round of $X$ check operator measurements, either $G_0^X$, $G_1^X$ or $G_0^X \cup G_1^X$ may be fixed.

Can an inhomogeneous schedule be used to increase the $Z$ distance of a subsystem code? For the planar subsystem surface code, the only $Z$ logical is a Pauli $Z$ operator applied to each qubit in a column of the lattice, corresponding to a path in the matching graph joining the north and south boundaries. Consider the inhomogeneous schedule where we alternate between using the $L_0$ and $L_1$ schedule in each row of the lattice: we assign the schedule $L_{(i \mod 2)}$ to faces in the $i^{\text{th}}$ row of the lattice. For a planar subsystem surface code with an odd distance, in each round of $X$ check operator measurements at least half of the gauge operators can be fixed: we can fix gauge operators in all rows, then in even rows, then all rows again, then odd rows, and so on in a cycle. In Figure D.2 we plot space-like slices (single time steps) of the 3D matching graph for when all rows, odd rows and even rows of gauge operators are fixed. Within each of these slices of the 3D matching graph, the shortest path between the north and south boundary is *larger* than the $Z$ distance of the subsystem surface code itself. We expect that the shortest path between the north and south boundaries of the overall 3D lattice is also larger, leading to an increased $Z$ distance of $d_Z = \lfloor 3(L-1)/2 \rfloor + 1$, but do not prove this here. The $X$ distance cannot increase

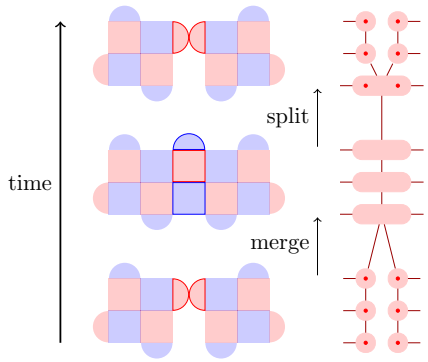



**Figure D.3:** A slice of the matching graph for lattice surgery, which can be interpreted as switching between different gauge fixes of a subsystem code. Left: the three stages of lattice surgery are shown for a distance 3 rotated surface code. Red (blue) squares and semi-circles denote $X$ ($Z$) stabilisers, with data qubits at their corners. Right: a slice of the matching graph for the $X$ stabilisers at the boundaries of the two codes where the merge takes place (denoted with red borders in the left diagram). Stabiliser measurements are repeated three times for each stage of lattice surgery, with the generalised difference syndrome used to connect the stabiliser with its gauge factors.

in this schedule, since none of the $Z$ gauge operators can be fixed.

Note that homogeneous schedules cannot increase the $Z$ or $X$ distance of the code, since there are always time steps where all $X$ gauge operators are measured simultaneously, as well as time steps where all $Z$ gauge operators are measured simultaneously. Measuring all $X$ gauge operators removes all $Z$ gauge operators from the stabiliser group, leaving time steps where none of the $Z$ gauge operators can be fixed (and therefore not increasing the $X$ distance), and similarly there are also time steps where no $X$ gauge operators can be fixed.

### D.2.2 Lattice surgery and code deformation

It was shown in Ref. [200] that the techniques of lattice surgery [117] and code deformation [22] can be interpreted as switching between different gauge fixes of a subsystem code. We can use this perspective to apply some of the techniques in this work to lattice surgery and code deformation. As an example, consider performing lattice surgery on two rotated surface code patches. During the merging step of lattice surgery, the weight two $X$ stabilisers on the opposing boundaries of the two patches are merged into weight 4 square stabilisers. These weight four stabilisers can be interpreted as stabilisers of a subsystem code, with the weight two checks that

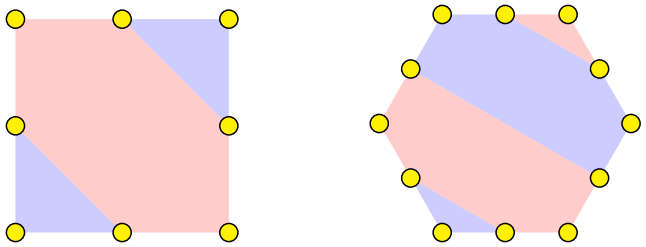


**Figure D.4:** Gauge fixings of a square (left) and hexagonal (right) face of a subsystem toric and $\{6,4\}$ subsystem hyperbolic code, respectively. Yellow filled circles are data qubits, and $X$ and $Z$ stabilisers are denoted by red and blue filled polygons, respectively.

they are merged from being gauge operators of the subsystem code. This procedure is shown for distance 3 codes in the left side of Figure D.3, for which a single pair of weight two $X$ checks (with red borders) is merged into a single square stabiliser. Since each pair of these weight two boundary $X$ checks is a pair of gauge factors of the corresponding weight 4 stabiliser, we can use the merging and splitting technique given in Section 4.2.1 to construct the matching graph and decode them. This is shown on the right side of Figure D.3, where three repetitions are used for each of the three stages of lattice surgery. With this technique, each of the consecutive stages of lattice surgery can be connected using the generalised difference syndrome, leading to a single matching graph that can be used for error correction with the overlapping recovery method of Ref. [64], and with information from the weight two boundary $X$ checks used directly where possible. The same ideas can also be readily applied to code deformation, which can also be viewed as gauge fixing of a subsystem code [200], and involves merging surface code patches in a similar manner [22].

### D.2.3 Subspace codes from gauge fixing

Another use of gauge fixing is to derive families of subspace codes from subsystem surface, toric and hyperbolic codes, by choosing different abelian subgroups of the gauge group $\mathcal{G}$ to be the stabiliser group $\mathcal{S}$, permanently fixing some gauge operators as stabilisers. For example, by fixing all the $X$-type triangle operators in the subsystem toric code as stabilisers we obtain the hexagonal toric code, and by fixing $X$-type triangle operators in the $\{8,4\}$ subsystem hyperbolic code as stabilisers

we obtain the $\{12,3\}$ hyperbolic code.

By fixing different subsets of the triangle operators in the subsystem toric code, we can interpolate between the hexagonal toric code and its dual. To achieve this we define *hexagonal surface density* codes, inspired by the surface-density and Shor-density codes of Ref. [138]. To construct a (subspace) hexagonal surface density code with parameter $q_f$ from a subsystem toric code, we fix the $X$-type gauge operators in each face with probability $q_f$, else we fix the $Z$-type gauge operators. When $q_f = 1$ we obtain the hexagonal surface code, and at $q_f = 0$ we construct its dual, but setting $0 < q_f < 1$ allows us to interpolate between these two extremes. With $q_f = 0.5$, there are both weight 6 and weight 3 $X$-type and $Z$-type stabilisers, and both $X$-type and $Z$-type stabilisers have average weight 4. The same idea can be directly applied to subsystem hyperbolic codes: applied to the $\{8,4\}$ subsystem hyperbolic code, we can interpolate between the $\{12,3\}$ hyperbolic code and its dual, for example.

For the subsystem hyperbolic codes, we can choose to fix only a subset of the triangle operators within each face. Consider the code obtained by fixing a single $Z$ triangle operator (chosen at random) within each face of the $\{6,4\}$ subsystem hyperbolic code, as well as the single $X$ triangle operator that commutes with it (an example of this for a single face is shown in Figure D.4). For both the $X$ and $Z$ stabilisers, half have weight 6, and the other half have weight 3. This hyperbolic code, derived from an *irregular* lattice, has average stabiliser weight 4.5 for both $X$ and $Z$ stabilisers, an improvement on the weight 5 stabilisers in the $\{5,5\}$ hyperbolic code, which has the smallest stabiliser weight of hyperbolic codes derived from self-dual regular lattices.

We can also use our choice of abelian subgroup of the gauge group to tailor codes to spatially inhomogeneous noise models, where the noise is biased towards $Z$-type errors in some regions of the lattice, and biased towards $X$-type errors in other regions. We can fix $X$-type gauge operators in regions where there is a $Z$ bias, locally reducing the vertex degree and stabiliser weight in the $X$-type matching graph, and likewise we can fix $Z$-type gauge operators where there is $X$ bias. This method of

tailoring a code to spatially inhomogeneous noise models has been demonstrated in Ref. [138] using gauge fixes of the Bacon-Shor code, and the same ideas can be readily applied here to gauge fixes of subsystem surface, toric and hyperbolic codes.

## D.3 Tessellations of closed surfaces

We will now give some additional background on tessellations of closed Euclidean and hyperbolic surfaces, since these tessellations are used to construct the subsystem hyperbolic and semi-hyperbolic codes in this work. An $\{r,s\}$ tessellation subdivides a surface into disjoint faces, where each face is an $r$-gon, and $s$ faces meet at each vertex. Using *Wythoff's kaleidoscopic construction*, an $\{r,s\}$-tessellation can be related to a symmetry group $G_{r,s}$ of distance-preserving maps (isometries). $G_{r,s}$ is generated by reflections on the edges of one of the $2r$ right triangles induced by the symmetry axes of a face ($r$-gon) of the tessellation. Each triangle has internal angles $\pi/2$, $\pi/r$ and $\pi/s$, and will from now on be referred to as a *fundamental triangle*. In Figure D.5(a) and Figure D.6(a) we draw a fundamental triangle of the $\{4,4\}$ and $\{8,4\}$ tessellations respectfully, with sides labelled by the reflections $a$, $b$ and $c$ which act on them, and which generate $G_{r,s}$. Note that the isometries $a^2$, $b^2$, $c^2$, $(ac)^2$, $(ab)^r$ and $(ca)^s$ are equivalent to doing nothing and, since these are the only relations satisfied by $G_{r,s}$, the group has presentation

$$G_{r,s} = \langle a,b,c | a^2 = b^2 = c^2 = (ac)^2 = (ab)^r = (bc)^s = e \rangle \tag{D.2}$$

where $e$ is the identity element. By fixing one fundamental triangle as a fundamental domain of $G_{r,s}$, every other fundamental triangle can be labelled uniquely by an element of $G_{r,s}$.

We will be constructing codes derived from $\{r,s\}$-tessellations of *closed* Euclidean and hyperbolic surfaces. The process of defining a closed surface is called *compactification*. A regular tessellation of a closed surface can be defined by a quotient group $G_{r,s}^H := G_{r,s}/H$, where $H$ is a finite index, normal subgroup of $G_{r,s}$ with no fixed points (see [44] for more details). Note that the generators of $H$ become relations in the presentation of $G_{r,s}/H$, so compactification can be interpreted as

adding additional relations into the presentation of the symmetry group of the tessellation of the hyperbolic plane. An important subgroup of $G_{r,s}$ is the proper symmetry group $G^+_{r,s}$ generated by double reflections, or *rotations*, $\rho = ab$ and $\sigma = bc$. This group has presentation

$$G^+_{r,s} = \langle \rho, \sigma | (\rho\sigma)^2 = \rho^r = \sigma^s = e \rangle \tag{D.3}$$

where $e$ is again the identity element. Regular tessellations of *orientable* closed surfaces can be constructed from a quotient group $G^{H+}_{r,s} := G^+_{r,s}/H$, where $H$ is a normal subgroup of $G^+_{r,s}$.

## D.4 Symmetry groups that admit subsystem hyperbolic codes

In Section 6.4 we introduced subsystem hyperbolic codes, which are derived from $\{2c,4\}$ tessellations of hyperbolic surfaces, where $c \in \mathbb{Z}^+$ and $c > 2$. In this section we will show how a subsystem hyperbolic code can be described in terms of the symmetry group of the tessellation from which it is derived. By doing so we will show what conditions must be satisfied by the compactification procedure for a $\{2c,4\}$ tessellation of a closed hyperbolic surface to be used for constructing a subsystem hyperbolic code.

Let us first consider some properties of the subsystem toric code in group theoretic terms. These properties will later be used as requirements for the subsystem hyperbolic codes we define. First, note that each triangle operator (gauge generator) of the subsystem toric code can be identified by a *pair* of fundamental triangles related by a $b$ reflection in $G_{4,4}$. In other words, each triangle operator is identified by a *left coset* of the subgroup $\langle b \rangle$ given by $g\langle b \rangle := \{g, gb\}$ for some $g \in G^H_{4,4}$, and thus each element $g \in G^H_{4,4}$ identifies a unique triangle operator (but not vice versa). For now we will consider only the Pauli type of each triangle operator, which can be either $Z$-type (blue) or $X$-type (red). We will call an assignment of a Pauli type to each triangle operator a *colouring*. For the subsystem toric code, note that

blue triangle operators are always mapped to red triangle operators by either an $a$ or $c$ reflection, and vice versa. We will make this property a requirement of our subsystem hyperbolic codes, and will call a colouring that satisfies this property a *valid* colouring.

Since each triangle operator can be identified by the coset $g\langle b\rangle$ of an element $g \in G^H_{r,s}$, and after identifying each *colour* of triangle operator with a different element of the cyclic group $\mathbb{Z}_2 = \mathbb{Z}/2\mathbb{Z}$, a colouring of the triangle operators can be achieved by defining an appropriate function $f : G^H_{r,s} \to \mathbb{Z}_2$. The constraint that either $a$ or $c$ reflections map a triangle operator to another of a different type, with $b$ reflections leaving it invariant, defines the image of the generators and identity element $e$ of $G^H_{r,s}$ by $f$ to be

$$\begin{aligned} f(a) = f(c) = 1, \\ f(b) = f(e) = 0. \end{aligned} \tag{D.4}$$

Since we require that, by definition of the code, the action of a reflection $a$, $b$ or $c$ should have the same effect on the colour of a triangle operator no matter which triangle operator we apply it to, this implies that $f(g_i g_j) = f(g_i) + f(g_j) \quad \forall g_i, g_j \in G^H_{r,s}$. This condition implies (from the definition of a homomorphism) that $f$ must extend to a *group homomorphism* from $G^H_{r,s}$ to $\mathbb{Z}_2$. For each triangle operator to be assigned a unique colour, we must also have that $f(r_i) = 0$ for each relation $r_i$ in the presentation of $G^H_{r,s}$. This latter condition is in fact also a necessary and sufficient condition for the function $f$ to extend to a homomorphism from $G^H_{r,s}$ to $\mathbb{Z}_2$ [151]. This constraint $f(r_i) = 0$ holds not just for the $\{4,4\}$ tiling, but also $\{r,s\}$ tilings for which $r$ and $s$ are even, since $(ab)^r = e$ and $(bc)^s = e$ are relations. The constraints do not hold if either $r$ or $s$ are odd. However, we also have the constraint $f(g_i) = 0$ on the generators $g_i$ of the normal subgroup $H$ defining the compactification (since these generators are relations in $G^H_{r,s}$) and, therefore, only a subset of the possible compactifications of these regular tessellations admit valid colourings.

We must also ensure that each triangle operator in a coloured tessellation commutes with every stabiliser, and that all stabilisers mutually commute (since by

definition $\mathcal{S}$ is abelian and the center of $\mathcal{G}$). We will now show that this condition further restricts us to tessellations where $s = 4$ faces meet at each vertex. For regular tessellations of closed Euclidean or hyperbolic surfaces, we are already restricted to $s \geq 3$, and we already require that $s$ be even to ensure a valid colouring. For all $s \in \{6, 8, 10, \ldots\}$ we see that each triangle operator anti-commutes with the stabiliser (of the opposite Pauli-type) belonging to the face related to it by a $(bc)^3$ rotation, since it overlaps with this stabiliser on only a single qubit. On the other hand, for $s = 4$, it can be directly verified that each triangle operator commutes with all stabilisers, since each triangle operator overlaps on either zero or two qubits with stabilisers of the opposite Pauli type. Since stabilisers are products of non-overlapping triangle operators, all stabilisers must also mutually commute. We are therefore restricted to tessellations with $s = 4$ faces meeting at each face and with $r = 2c$ sides to each face, and for which $f(g_i) = 0$ for each generator $g_i$ of the normal subgroup $H$ defining the compactification.

## D.5 Group theoretic condition for consistent scheduling

In Section 6.3 of the main text, we showed that any translationally invariant schedule for the subsystem toric code assigns the same schedule to each triangle operator with the same *label*, where a label is an assignment of an element of the cyclic group $\mathbb{Z}_4$ to each triangle operator as shown in Figure D.5(b). We will now describe this labelling of the triangle operators of the subsystem toric code in terms of the proper symmetry group $G_{r,s}^{H+}$ of orientation-preserving symmetries of the lattice, generated by the rotations $\rho$ and $\sigma$ (shown in Figure D.5(a)). First note that, after choosing any triangle operator to be the fundamental domain, each triangle operator is now identified by a unique element in $G_{r,s}^{H+}$, and we will denote by $T_g$ the triangle operator identified by $g \in G_{r,s}^{H+}$. A labelling of the triangle operators is then defined by a function $h : G_{r,s}^{H+} \to \mathbb{Z}_4$. Note that, for the labelling of the subsystem toric code in Figure D.5(b), applying either a $\rho$ or $\sigma$ rotation to *any* triangle operator adds one (modulo 4) to the label. Using similar arguments to those given in Appendix D.4

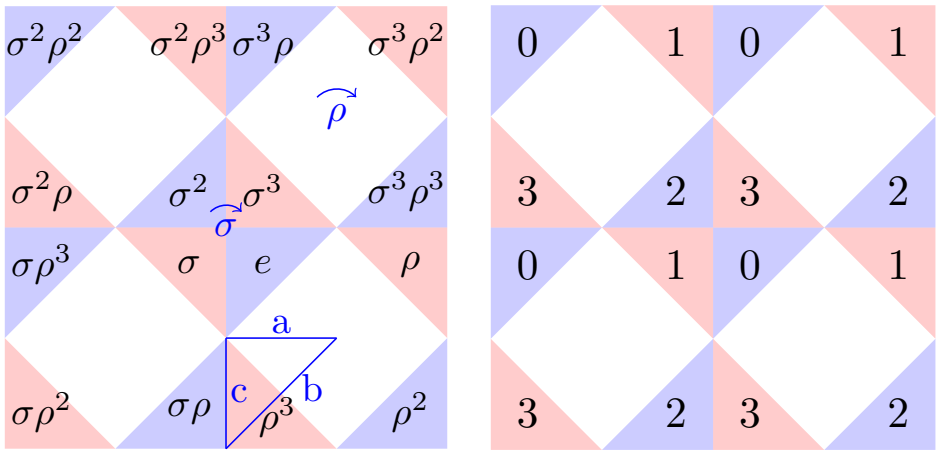



**Figure D.5:** An $L = 2$ subsystem surface code. (a) After associating a triangle operator with the identity element $e$, every triangle operator is in one-to-one correspondence with an element of the proper symmetry group $G^{H+}_{4,4}$ of the tessellation. In blue we have labelled a fundamental triangle with sides $a$, $b$ and $c$, as well as the rotations $\rho = ab$ and $\sigma = bc$. (b) Each triangle operator can be labelled with an element of the cyclic group $\mathbb{Z}_4$ using the homomorphism $h(\rho) = h(\sigma) = 1$ from $G^{H+}_{4,4}$ to $\mathbb{Z}_4$.

for valid colourings, we see that the function $h$ must extend to a homomorphism $h : G^{H+}_{r,s} \to \mathbb{Z}_4$ with

$$h(\rho) = h(\sigma) = 1. \tag{D.5}$$

We can generalise a translationally symmetric schedule of the subsystem toric code to subsystem hyperbolic codes by first labelling the triangle operators of a subsystem hyperbolic code in such a way that the neighbourhood of each triangle operator is the same as it would be in the subsystem toric code, and then apply the same schedule to all triangle operators with the same label in the subsystem hyperbolic code. The *neighbourhood* of a triangle operator $T$ is the relative position and label of the triangle operators that overlap with $T$ on at least one qubit (each of which we call a *neighbour*). We see from Figure D.5(b) that each triangle operator $T_g$ in the subsystem toric code has seven neighbours: $T_{g\sigma}$, $T_{g\sigma^2}$, $T_{g\sigma^3}$, $T_{g\rho}$, $T_{g\rho\sigma}$, $T_{g\rho^{-1}}$ and $T_{g\rho^{-1}\sigma^{-1}}$. In the toric code, exactly three of these neighbours overlap on a vertex of the $\{4,4\}$ tessellation. To ensure this remains the case for the hyperbolic tessellations, it is necessary to require that $s = 4$, which is by definition a property of

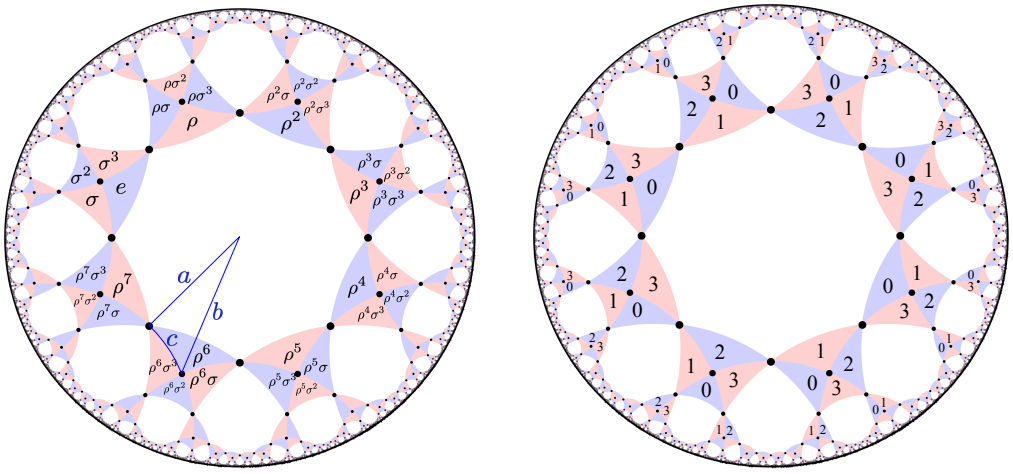



**Figure D.6:** The $\{8,4\}$ subsystem hyperbolic code. (a) Each triangle operator can be uniquely identified with an element of the proper symmetry group $G^{H+}_{8,4}$ of the lattice (after identifying a triangle operator with the identity element $e$). We have labelled a fundamental triangle in blue. (b) Each triangle operator can be labelled with an element of $\mathbb{Z}_4$ using the homomorphism $h(\rho) = h(\sigma) = 1$ from $G^{H+}_{8,4}$ to $\mathbb{Z}_4$. The neighbourhood of each triangle operator (the labels and relative locations of triangle operators it overlaps with) is the same as in the toric code.

our subsystem hyperbolic codes. Setting $s = 4$ alone is not sufficient, since we must now also ensure that the entire neighbourhood (all *seven* neighbours) of each triangle operator with a given label in the lattice remains identical to that of a triangle operator with the same label in the toric code. The relative labels of these seven neighbours is determined by the homomorphism $h : G^{H+}_{r,s} \to \mathbb{Z}_4$ defined in Equation (D.5).

Therefore, a hyperbolic tessellation is schedulable if its proper symmetry group admits the homomorphism $h$ as defined in Eq. (D.5), which is the case if and only if $h(r_i) = 0$ for each relation $r_i$ in the presentation of $G^{H+}_{r,s}$. This condition is met for subsystem hyperbolic codes derived from the subset of closed $\{4c,4\}$ tessellations (where $c \in \mathbb{Z}^+$), for which the generators $g_i$ of the normal subgroup $H$ defining the compactification satisfies $h(g_i) = 0$. As an example, consider the quotient group for a distance $L$ toric code which has presentation

$$G^{H+}_{4,4} = \langle \rho, \sigma | \rho^4 = \sigma^4 = (\rho\sigma)^2 = (\rho\sigma^{-1})^L = (\sigma^{-1}\rho)^L = e \rangle \tag{D.6}$$

from which it is clear that each relation $r_i$ satisfies $h(r_i) = 0$.

For schedulable subsystem hyperbolic codes, we can use the very efficient measurement schedule of Ref. [36] (which is translationally invariant for the subsystem toric code) for each triangle operator, which requires only four time steps (one time step is the duration of a CNOT gate) to measure all $X$ and $Z$ check operators. Note that subsystem hyperbolic codes which do *not* satisfy these constraints will still admit a measurement schedule, but such a schedule may be considerably less efficient and also more difficult to construct.

Given the map $m : \mathbb{Z}_4 \to \mathbb{Z}_2$ defined by $m(x) = x \mod 2$ assigning a colour to a label, we see that $f(g) = m(h(g)) \quad \forall g \in G^{H+}_{r,s}$, where $f$ is defined in Eq. (D.4), and hence every schedulable code is colourable (but not vice versa, as exemplified by the $\{6,4\}$ tessellation for which $\rho^6$ is a relation yet $h(\rho^6) \neq 0$).

There is another way of interpreting the scheduling: Consider the graph which is generated by the rotation subgroup $\langle \rho, \sigma \rangle$. this group acts regularly between the triangles of the subsystem code, so there is a one-to-one map between them. The labeling is a coloring of the Cayley graph of this group (each vertex of this Cayley graph corresponds to a triangle). This coloring is achieved by a "covering" of the cycle graph with 4 vertices (Cayley graph of $\mathbb{Z}_4$) since this is clearly 4-colourable. More generally, we can consider normal subgroups $N$ of the group as long as this normal subgroup does not contain $\rho$ or $\sigma$. The number of colours in this case is the index of $N$ in $G$.

The dual semi-hyperbolic tessellations used for constructing the subsystem semi-hyperbolic codes do not have a group structure, so they cannot be labelled using the homomorphism of Equation (D.5) alone. However, we now show that, given a schedulable $\{4c,4\}$ tessellation, the corresponding dual semi-hyperbolic tessellation derived from it is also schedulable. Take a schedulable $\{4c,4\}$ tessellation $V$, where we have already labelled each corner in the tessellation with an element of $\mathbb{Z}/4\mathbb{Z}$. Now consider its dual tessellation $V^*$, constructed by exchanging vertices and faces in the Hasse diagram of the tessellation [40]. Each corner in $V$ is identified by a face and vertex, and so each corner in $V$ is in one to one correspondence with a corner in

$V^*$ (where the face and vertex are exchanged). We give each corner in $V^*$ the same label as the corner in $V$ that it is in one to one correspondence with. This constitutes a valid labelling of $V^*$, since each pair of corners related by $\rho$ ($\sigma$) in $V$ are related by $\sigma$ ($\rho$) in $V^*$, and $h(\rho) = h(\sigma)$ in Equation (D.5). We now construct a semi-hyperbolic tessellation $V_l^*$ by tiling each face of $V^*$ with an $l \times l$ square lattice. Note that the corners of each face in $V^*$ are already labelled, so we can label $V_l^*$ just by labelling the new corners introduced by the $l \times l$ square tiling of each face. Corners related by a $\sigma$ rotation in $V^*$ are still related by a $\sigma$ rotation in $V_l^*$. Corners related by a $\rho$ rotation in $V^*$ are now related by a $(\rho\sigma^{-1})^{l-1}\rho$ translation in $V_l^*$. However, now treating $h$ as a function not a homomorphism, note that $h(\rho) = h((\rho\sigma^{-1})^{l-1}\rho)$, so the original labels retained from $V^*$ remain valid. We can therefore label the new corners in the square $l \times l$ tilings in $V_l^*$ in a way that is consistent with the corners already labelled. We now take the dual of $V_l^*$ to obtain $V_l$, preserving the labels of each corner when taking the dual as before. The tessellation $V_l$ is now used to derive a subsystem semi-hyperbolic code, and we have demonstrated that $V_l$ is schedulable if $V$ is schedulable.

## D.6 Subsystem semi-hyperbolic and subsystem toric code comparison

A quantum code derived from a $\{r,s\}$-tessellation satisfies [44]

$$\frac{k}{n} = 1 - \frac{2}{s} - \frac{2}{r} + \frac{2}{n} \tag{D.7}$$

where $n$ is the number of physical qubits and $k$ is the number of logical qubits. A semi-hyperbolic code derived from such a code has $l^2 n$ qubits, where $l$ is the dimension of the lattice tiling each face in the semi-hyperbolic code. Therefore, the number of data qubits (excluding ancillas) in a subsystem $\{8,4\}$-semi-hyperbolic code with $k$ logical qubits is $6(k-2)l^2$. To compare the performance of subsystem semi-hyperbolic codes with subsystem toric codes, we will compare each semi-hyperbolic code to multiple independent copies of a toric code with the same rate

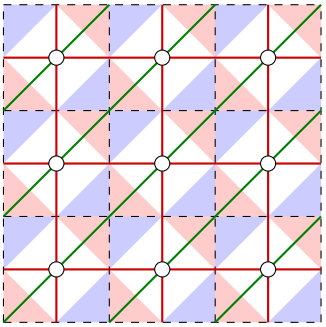


**Figure D.7:** The subsystem toric code. The black dashed lines are edges of the $\{4,4\}$ tessellation from which the subsystem toric code is derived. The edges in the $X$-type matching graph are the union of the solid red and green lines, and vertices in the matching graph are denoted by circles. Each edge in the $X$-type matching graph corresponds to a data qubit, and each face corresponds to a $Z$-type triangle operator. The solid red lines are the edges of the matching graph for the standard surface code derived from the same $\{4,4\}$ tessellation. Opposite sides are identified.

$k/n$, such that we can compare the performance keeping $k$ and $n$ fixed. Since the rate of a subsystem toric code with distance $L$ is $2/(3L^2)$, we compare our subsystem semi-hyperbolic $\{8,4\}$ codes with copies of a toric code with distance close to

$$L = 2l\sqrt{1-\frac{2}{k}} \tag{D.8}$$

where $k$ is the number of qubits in the $\{8,4\}$ semi-hyperbolic code and $l$ is the dimension of the lattice tiling each face in the semi-hyperbolic code. Note that the total number of qubits including ancillas $(1+4n_a/3)n$ is proportional to the number of data qubits $n$ with the same constant of proportionality for the subsystem toric, hyperbolic and semi-hyperbolic codes. Here, $n_a$ is the number of ancilla qubits used per triangle operator (we can always set $n_a = 1$, but for some schedules setting $n_a = 2$ can improve performance by parallelising the measurement schedule). Therefore Eq. (D.8) still holds once ancillas are taken into account.

## D.7 Distance of subsystem hyperbolic codes

We can determine the distance of the subsystem hyperbolic and semi-hyperbolic codes by considering their matching graphs. Each vertex in the $X$-type matching graph corresponds to an $X$ stabiliser, and there is an edge between each pair of

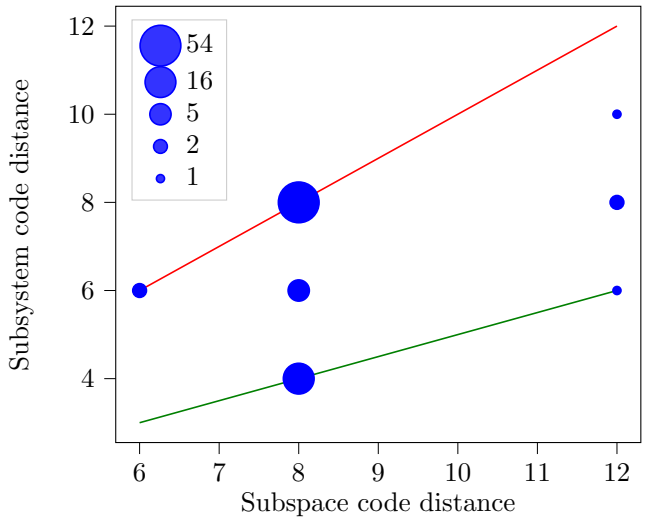



**Figure D.8:** For all $l = 2$ $\{8,4\}$ subsystem semi-hyperbolic codes we constructed, here we plot the distance of each code ($y$-axis) against the distance of the (subspace) semi-hyperbolic surface code derived from the same tessellation ($x$-axis), computed using the method in Ref. [77]. The size of each blue circle corresponds to the number of codes we found with the same ($x$,$y$) coordinate on the figure, and the number of codes for each size of circle is given in the legend.

stabilisers $u$ and $v$ for which a single $Z$ error on a data qubit anti-commutes with both $u$ and $v$. Each face in the $X$-type matching graph corresponds to a $Z$-type triangle operator. Each non-contractible closed loop in the $X$-type matching graph corresponds to a logical $Z$ operator. Therefore, the $Z$-distance of the code is determined by the shortest non-contractible closed loop in the $X$-type matching graph. A $Z$-type matching graph can be defined analogously for $Z$-type stabilisers and so the $X$ distance of the code is determined by the shortest non-contractible closed loop in the $Z$-type matching graph.

For the subsystem toric, hyperbolic and semi-hyperbolic codes we construct, the $X$-type matching graph is isomorphic to the $Z$-type matching graph, since the $Z$-type matching graph can be obtained from the $X$-type matching graph (and vice versa) by a single rotation that is also a symmetry of the tessellation from which the code is derived. Therefore, the $Z$ and $X$ distances are the same for these codes.

We will now consider how the distance of a subsystem hyperbolic or semi-hyperbolic code compares to the distance of the subspace CSS (surface) code derived from the same tessellation. To do so, we will consider the structure of the matching

graph for both codes. The solid red lines in Figure D.7 form the edges of the $Z$-type matching graph for the toric code, and so the length of the shortest non-contractible loop in that graph is the $X$ distance of the toric code. We can obtain the $X$-type matching graph for the *subsystem* toric code derived from the same tessellation by adding in the green edges, also shown in Figure D.7, and keeping the same set of vertices. Each green edge in the subsystem toric code $X$-type matching graph is equivalent (up to a triangle operator) to a *pair* of red edges. Therefore, the distance between two vertices in the matching graph consisting only of red edges can *at most* be reduced by half by the inclusion of the green edges (and inclusion of the green edges cannot increase the distance between vertices).

For the subsystem hyperbolic and semi-hyperbolic codes, we again find that both the $Z$-type and $X$-type matching graphs can be constructed by adding additional edges to the $Z$-type matching graph $V_Z$ of the subspace codes derived from the tessellation, where each of these additional edges is equivalent to a pair of edges in $V_Z$. Therefore, the shortest non-contractible loop in either the $Z$-type or $X$-type matching graph for a subsystem hyperbolic or semi-hyperbolic code is between one and two times smaller than the shortest non-contractible loop in the $Z$-type matching graph of the subspace code derived from the same tessellation. Consequently, given a hyperbolic or semi-hyperbolic code with $X$ distance $d_X$, the distance $d$ of the subsystem hyperbolic or semi-hyperbolic code derived from the same tessellation is bounded by $d_X/2 \leq d \leq d_X$. Furthermore, the $X$ distance of hyperbolic codes we consider is always less than or equal to their $Z$ distance. Both the subsystem toric code and standard toric code have distance $d = L$, but for the subsystem hyperbolic and semi-hyperbolic codes we construct, the subsystem codes do have a reduced distance compared to surface codes derived from the same tessellation. This is shown in Figure D.8, which compares the distance of $l = 2$, $\{8,4\}$ subsystem semi-hyperbolic codes to the distance of the subspace semi-hyperbolic codes derived from the same tessellations. We see that the distance of each subsystem code can be reduced by up to $2\times$ relative to the subspace code derived from the same tessellation as expected, with some subsystem codes not suffering any reduction in distance.

## D.8 Scheduling from group homomorphisms

In Appendix D.5 we showed that an efficient syndrome measurement schedule for subsystem hyperbolic codes could be constructed if the orientation-preserving symmetry group $G^{H+}_{r,s}$ of the tessellation (generated by rotations $\rho$ and $\sigma$) admits a homomorphism $f : G^{H+}_{r,s} \to \mathbb{Z}_4$ to the cyclic group $\mathbb{Z}_4$, with $f$ defined by $f(\rho) = f(\sigma) = 1$. This homomorphism is a useful tool for scheduling subsystem hyperbolic codes for the same reason that translation invariance is useful for scheduling Euclidean surface codes: the problem of scheduling the entire code reduces to the problem of scheduling only a small number of stabilisers in a region of the tessellation.

While the homomorphism $f : G^{H+}_{r,s} \to \mathbb{Z}_4$ is a useful tool for scheduling the subsystem hyperbolic codes, such a homomorphism only exists for a subset of $\{r,s\}$ tessellations (for which four divides both $r$ and $s$). In this section we will look for homomorphisms from $G^{H+}_{r,s}$ to *any* cyclic group, in the hope that these homomorphisms will be a useful tool for scheduling subspace hyperbolic codes based on a wider range of tessellations, where each $Z$ stabiliser (plaquette) and $X$ stabiliser (site) is measured using a circuit with a single ancilla qubit. Each corner $C_g$ of a face of the tessellation is identified with an element $g \in G^{H+}_{r,s}$. By finding a homomorphism $f : G^{H+}_{r,s} \to \mathbb{Z}_n$ to a cyclic group $\mathbb{Z}_n$, we can label each corner uniquely with an element in $\mathbb{Z}_n$. The function $f$ is a homomorphism if and only if $f(r_i) = 0$ for each relation $r_i$ in the presentation of $G^{H+}_{r,s}$. The tessellation group $G^{H+}_{r,s}$ has presentation

$$G^{H+}_{r,s} := \langle \rho, \sigma | (\rho\sigma)^2 = \rho^r = \sigma^s = e \rangle \tag{D.9}$$

from which we see that $(\rho\sigma)^2$ is always a relation, and hence $f$ must always satisfy $f((\rho\sigma)^2) = 0$.

For the homomorphism $f : G^{H+}_{r,s} \to \mathbb{Z}_n$ to be useful for scheduling, we will require that it must satisfy a additional properties. Firstly, the homomorphism should not be defined by $f(\rho) = f(\sigma) = 0$, since this homomorphism does not give us any additional information. Secondly, the label of each corner $C_g$ should be different to the corner $C_{g\rho\sigma}$. This is because $C_g$ and $C_{g\rho\sigma}$ overlap on an edge $e$ in such a way that, if both corners had the same schedule, two CNOT gates applied to the qubit at $e$

would occupy the same time step.

We will assume that can have more than one ancilla for each stabiliser, to parallelise the measurement circuits. If we were instead to insist that only a single ancilla be used, then we must require that all corners belonging to the same vertex must have different labels. This is because these corners share an ancilla qubit on the vertex, and two CNOT gates cannot be applied to the ancilla qubit within the same time step. Furthermore, we would also require that all corners belonging to a face must have a different label, since only a single CNOT gate can be applied to the ancilla qubit in the centre of each face in each time step.

Therefore for each tessellation $\{r,s\}$, we will seek to find a cyclic group order $n$ and elements $x,y \in \mathbb{Z}_n$ such that the function defined by $f(\rho) = x, f(\sigma) = y$ extends to a homomorphism $f : G_{r,s}^{H+} \to \mathbb{Z}_n$. The restrictions on the relations in the presentation of $G_{r,s}^{H+}$, along with the additional three properties we have imposed, correspond to the following constraints on $x,y,n$:

$$\begin{aligned} rx &= 0 \mod n \\ sy &= 0 \mod n \\ 2(x+y) &= 0 \mod n \\ x+y &\neq 0 \mod n \end{aligned} \tag{D.10}$$

and if we could use only a single ancilla per stabiliser, then we would additionally have the constraints

$$\begin{aligned} \mathrm{lcm}(x,n) &= rx \\ \mathrm{lcm}(y,n) &= sy. \end{aligned} \tag{D.11}$$

For all $r,s \leq 10$ we have searched for all $n,x,y$ satisfying Eq. D.10 (for $n < 5\max(r,s)$) and list all the tessellations we found which admitted at least one such homomorphism in Table D.3.

While we have found homomorphisms to cyclic groups for many tessellations, we did not find any for the $\{5,5\}$ code, which has the desirable properties of being

| $r$ | $s$ | $n$ | $x$ | $y$ |
|---|---|---|---|---|
| 3 | 6 | 6 | 2 | 1 |
| 4 | 4 | 4 | 1 | 1 |
| 4 | 8 | 4 | 1 | 1 |
| 5 | 10 | 10 | 2 | 3 |
| 6 | 6 | 6 | 1 | 2 |
| 6 | 9 | 6 | 1 | 2 |
| 8 | 8 | 8 | 1 | 3 |
| 10 | 10 | 10 | 1 | 4 |

**Table D.3:** Solutions to Eq. D.10 for all $r,s \leq 10, r \leq s$. By symmetry, solutions for $r \geq s$ can be found by exchanging column $r$ with $s$ and column $x$ with $y$. For each tessellation $\{r,s\}$, we give the parameters $n,x,y$ defining only one homomorphisms $f : G_{r,s}^{H+} \to Z_n$ (the homomorphism which minimimises both $n$ and $x$). There are at least two solutions for each tessellation.

| Schedule | $p_{depol}^{th}$ | $p_{depol}^{th,*}$ |
|---|---|---|
| ZX | 0.666(1) | 0.666(1) |
| $Z^2X^2$ | 0.757(1) | 0.6587(9) |
| $Z^3X^3$ | 0.810(2) | 0.676(1) |
| $Z^4X^4$ | 0.811(2) | 0.669(2) |
| $Z^5X^5$ | 0.792(2) | 0.652(2) |
| $Z^{10}X^{10}$ | 0.522(2) | 0.493(1) |

**Table D.4:** Thresholds (in %) for the subsystem toric code for some balanced homogeneous schedules under the circuit-level depolarising noise model, each computed using the critical exponent method of Ref. [201] to analyse results from Monte Carlo simulations using subsystem toric codes with distances $L = 26,30,34,38,42,46$. Numbers in brackets are the $1\sigma$ uncertainties in the last digit. For each threshold, we keep the number of syndrome extraction rounds constant for all codes, always using at least 92 rounds to ensure boundary effects (in time) are small even for the largest codes. For the column with an asterisk, gauge fixing was not used when decoding.

self-dual and having low stabiliser weights. Therefore, an interesting question is whether there exist homomorphisms to groups that are not cyclic, and which contain a small number of elements, but otherwise satisfy the constraints of Equation (D.10). If such a homomorphism exists for tessellations such as $\{4,5\}$ and $\{5,5\}$, the trade off of circuit-level threshold and encoding rate for these codes may be very favourable.

| Schedule | $p_X^{th}$ | $p_X^{th,*}$ | $p_Z^{th}$ | $p_Z^{th,*}$ |
|---|---|---|---|---|
| ZX | 0.515(1) | | 0.515(1) | |
| $Z^2X^2$ | 0.5863(9) | | 0.5863(9) | |
| $Z^3X^3$ | 0.628(1) | | 0.628(1) | |
| $Z^4X^4$ | 0.631(2) | | 0.631(2) | |
| $Z^5X^5$ | 0.619(2) | | 0.619(2) | |
| $ZX^2$ | 0.3928(8) | 0.3928(8) | 0.749(1) | 0.625(3) |
| $ZX^3$ | 0.3236(9) | 0.3236(9) | 0.931(1) | 0.7234(9) |
| $ZX^5$ | 0.2449(5) | 0.2449(5) | 1.160(2) | 0.816(2) |
| $ZX^{10}$ | 0.1595(4) | 0.1595(4) | 1.430(3) | 0.902(2) |
| $Z^2X^{10}$ | 0.2394(5) | 0.2259(5) | 1.197(3) | 0.821(2) |
| $X$ | 0 | 0 | 2.2231(1) | 1.029(2) |

**Table D.5:** Thresholds (in %) for the subsystem toric code for various homogeneous schedules under the independent circuit-level noise model, each computed using the critical exponent method of Ref. [201] to analyse results from Monte Carlo simulations using subsystem toric codes with distances $L = 26, 30, 34, 38, 42, 46$. Numbers in brackets are the $1\sigma$ uncertainties in the last digit. For each threshold, we keep the number of syndrome extraction rounds constant for all codes, always using at least 92 rounds to ensure boundary effects (in time) are small even for the largest codes. For the final two columns (with asterisks in the title), gauge fixing was not used even when possible.

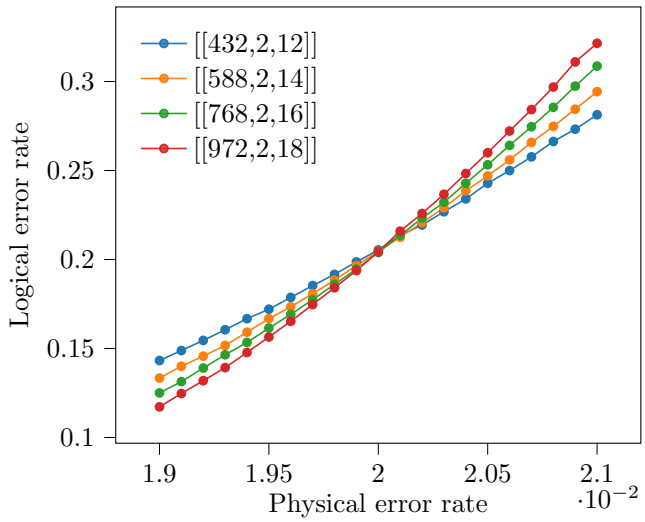



**Figure D.9:** Subsystem toric code threshold with a phenomenological noise model, and without using gauge fixing (triangular lattice matching graph). Using the critical exponent method we find a threshold of 0.02004(2).

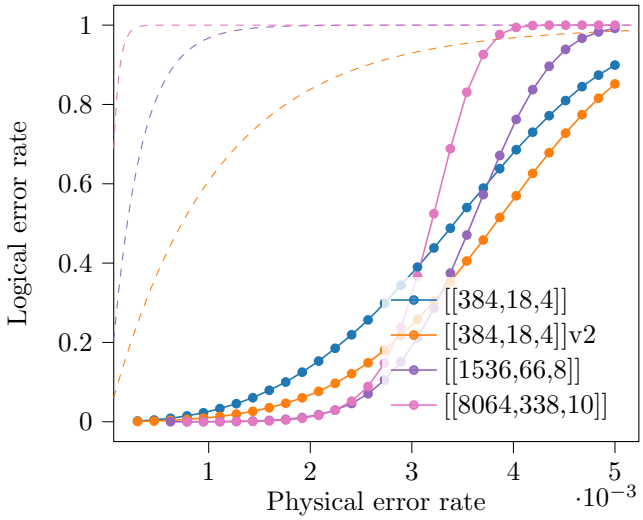



**Figure D.10:** Performance of the extremal subsystem $\{8,4\}$ $l = 2$ semi-hyperbolic codes under a circuit-level depolarising noise model. A homogeneous $(ZX)^{20}$ schedule is used for all codes, and the $y$ axis is the probability that at least one logical $Z$ error occurs. Dashed lines are the probability of a $Z$ error occurring on at least one of $k$ physical qubits without error correction under the same error model and for the same duration (80 time steps), with $k = 4$ (orange), $k = 8$ (purple) and $k = 10$ (pink).

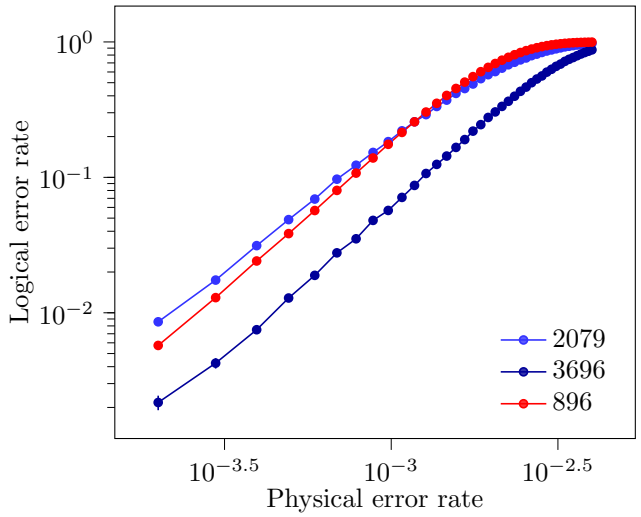



**Figure D.11:** Performance of a [[384,66,4]] $\{8,4\}$ subsystem hyperbolic code (red) compared to the $L = 3$ and $L = 4$ subsystem toric codes (shades of blue) using a $(ZX)^{10}$ schedule with the circuit-level depolarising error model. We use 33 independent copies of the subsystem toric codes to fix the number of logical qubits at $k = 66$. In the legend we give the number of physical qubits used, including ancillas.

## D.9 Additional numerical results

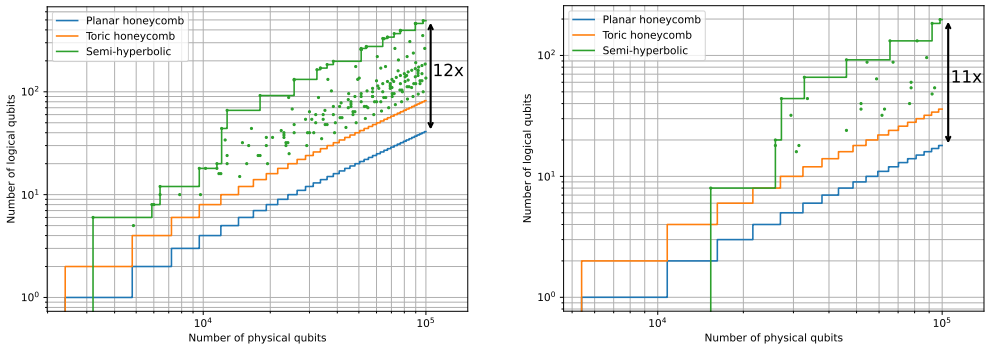
In this section we give some additional numerical results from simulations of the subsystem toric and semi-hyperbolic codes. In Table D.4, we give thresholds for the subsystem toric code code under a circuit-level depolarising noise model using gauge fixing with balanced schedules. In Table D.5, we give thresholds for the subsystem toric code under an independent circuit-level noise model using both balanced and unbalanced schedules. In Figure D.9 we plot the threshold for the subsystem surface code with a phenomenological noise model, which we find to be $0.02004(2)$ using the critical exponent method of Ref. [201]. In Figure D.10 we plot the threshold of the $l = 2$ $\{8,4\}$ subsystem semi-hyperbolic codes *without* adjusting for the number of logical qubits, unlike in the text. This is helpful to better understand the logical error rates of the codes themselves, but less so for understanding the threshold for the logical error rate per logical qubit, for which multiple independent copies of the smaller codes should be taken, as done in the main text. In Figure D.11 we compare the performance of a [[384,66,4]] $\{8,4\}$ subsystem hyperbolic code with 33 copies of $L = 3$ and $L = 4$ subsystem toric codes, all encoding 66 logical qubits. Since this hyperbolic code is quite small, its overhead $n/(kd^2) \approx 0.36$ is less favourable than that of the much larger [[8064,338,10]] subsystem semi-hyperbolic code analysed in Section 6.5 of the main text, for which $n/(kd^2) \approx 0.24$. However, as can be seen from Figure D.11, the [[384,66,4]] subsystem hyperbolic code still uses $2.3\times$ fewer physical qubits than the subsystem toric code to achieve the same logical error rate per logical qubit below a circuit-level depolarising physical error rate of $0.1\%$. Furthermore, it only requires 896 physical qubits to implement this subsystem hyperbolic code including ancillas, compared to the $18,816$ needed for the [[8064,338,10]] subsystem semi-hyperbolic code.

# Appendix E

# Hyperbolic Floquet

## E.1 Additional results

Figure E.1 gives the number of logical qubits that can be encoded with embedded distance at least 20 and 30 using semi-hyperbolic Floquet codes or honeycomb codes (these plots are slight variants of Figure 7.10).




**(a)** **(b)**

**Figure E.1:** The number of logical qubits that can be encoded using multiple copies of a semi-hyperbolic, toric honeycomb or planar honeycomb floquet code. In (a) the distance is required to be at least 20 for an EM3 noise model, whereas in (b) the distance is at least 30.

# Bibliography


[1] Scott Aaronson and Daniel Gottesman. “Improved simulation of stabilizer circuits”. In: *Physical Review A* 70.5 (Nov. 2004). ISSN: 1094-1622. DOI: 10.1103/physreva.70.052328. URL: http://dx.doi.org/10.1103/PhysRevA.70.052328.

[2] David Aasen, Zhenghan Wang, and Matthew B. Hastings. “Adiabatic paths of Hamiltonians, symmetries of topological order, and automorphism codes”. In: *Phys. Rev. B* 106 (8 Aug. 2022), p. 085122. DOI: 10.1103/PhysRevB.106.085122. URL: https://link.aps.org/doi/10.1103/PhysRevB.106.085122.

[3] Miguel Aguado and Guifré Vidal. “Entanglement Renormalization and Topological Order”. In: *Phys. Rev. Lett.* 100 (7 Feb. 2008), p. 070404. DOI: 10.1103/PhysRevLett.100.070404. URL: https://link.aps.org/doi/10.1103/PhysRevLett.100.070404.

[4] Ravindra K Ahuja et al. “Faster algorithms for the shortest path problem”. In: *Journal of the ACM (JACM)* 37.2 (1990), pp. 213–223.

[5] Google Quantum AI. “Suppressing quantum errors by scaling a surface code logical qubit”. In: *Nature* 614.7949 (Feb. 2023), pp. 676–681. ISSN: 1476-4687. DOI: 10.1038/s41586-022-05434-1. URL: https://doi.org/10.1038/s41586-022-05434-1.

[6] Panos Aliferis and Andrew W Cross. “Subsystem fault tolerance with the Bacon-Shor code”. In: *Physical review letters* 98.22 (2007), p. 220502.

[7] Panos Aliferis and John Preskill. "Fault-tolerant quantum computation against biased noise". In: *Phys. Rev. A* 78 (5 Nov. 2008), p. 052331. DOI: 10.1103/PhysRevA.78.052331. URL: https://link.aps.org/doi/10.1103/PhysRevA.78.052331.

[8] Zunaira Babar et al. "Fifteen Years of Quantum LDPC Coding and Improved Decoding Strategies". In: *IEEE Access* 3 (2015), pp. 2492–2519. DOI: 10.1109/ACCESS.2015.2503267.

[9] Dave Bacon. "Operator quantum error-correcting subsystems for self-correcting quantum memories". In: *Phys. Rev. A* 73 (1 Jan. 2006), p. 012340. DOI: 10.1103/PhysRevA.73.012340. URL: https://link.aps.org/doi/10.1103/PhysRevA.73.012340.

[10] Dave Bacon et al. "Sparse quantum codes from quantum circuits". In: *IEEE Transactions on Information Theory* 63.4 (2017), pp. 2464–2479.

[11] Paul Baireuther et al. "Machine-learning-assisted correction of correlated qubit errors in a topological code". In: *Quantum* 2 (Jan. 2018), p. 48. ISSN: 2521-327X. DOI: 10.22331/q-2018-01-29-48. URL: https://doi.org/10.22331/q-2018-01-29-48.

[12] F Barahona. "On the computational complexity of Ising spin glass models". In: *Journal of Physics A: Mathematical and General* 15.10 (Oct. 1982), p. 3241. DOI: 10.1088/0305-4470/15/10/028. URL: https://dx.doi.org/10.1088/0305-4470/15/10/028.

[13] Ben Barber et al. "A real-time, scalable, fast and highly resource efficient decoder for a quantum computer". In: (2023). arXiv: 2309.05558 [quant-ph].

[14] Andreas Bauer. "Topological error correcting processes from fixed-point path integrals". In: (2023). arXiv: 2303.16405 [quant-ph].

[15] Asmae Benhemou et al. "Minimising surface-code failures using a color-code decoder". In: (2023). arXiv: 2306.16476 [quant-ph].

[16] Charles H. Bennett et al. “Mixed-state entanglement and quantum error correction”. In: *Phys. Rev. A* 54 (5 Nov. 1996), pp. 3824–3851. DOI: 10.1103/PhysRevA.54.3824. URL: https://link.aps.org/doi/10.1103/PhysRevA.54.3824.

[17] Claude Berge. “Two theorems in graph theory”. In: *Proceedings of the National Academy of Sciences* 43.9 (1957), pp. 842–844.

[18] E. Berlekamp, R. McEliece, and H. van Tilborg. “On the inherent intractability of certain coding problems (Corresp.)” In: *IEEE Transactions on Information Theory* 24.3 (1978), pp. 384–386. DOI: 10.1109/TIT.1978.1055873.

[19] Piotr Berman et al. “The T-join Problem in Sparse Graphs: Applications to Phase Assignment Problem in VLSI Mask Layout”. In: *Algorithms and Data Structures*. Ed. by Frank Dehne et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 25–36. ISBN: 978-3-540-48447-9.

[20] Dolev Bluvstein et al. “A quantum processor based on coherent transport of entangled atom arrays”. In: *Nature* 604.7906 (2022), pp. 451–456. DOI: 10.1038/s41586-022-04592-6. URL: https://doi.org/10.1038/s41586-022-04592-6.

[21] Thomas C. Bohdanowicz. “Quantum Constructions on Hamiltonians, Codes, and Circuits”. PhD dissertation. California Institute of Technology, 2021. DOI: 10.7907/pxp0-aw46.

[22] H Bombin and M A Martin-Delgado. “Quantum measurements and gates by code deformation”. In: *Journal of Physics A: Mathematical and Theoretical* 42.9 (Feb. 2009), p. 095302. DOI: 10.1088/1751-8113/42/9/095302. URL: https://dx.doi.org/10.1088/1751-8113/42/9/095302.

[23] H. Bombin. “Topological subsystem codes”. In: *Phys. Rev. A* 81 (3 Mar. 2010), p. 032301. DOI: 10.1103/PhysRevA.81.032301. URL: https://link.aps.org/doi/10.1103/PhysRevA.81.032301.

[24] H. Bombin and M. A. Martin-Delgado. "Optimal resources for topological two-dimensional stabilizer codes: Comparative study". In: *Phys. Rev. A* 76 (1 July 2007), p. 012305. DOI: 10.1103/PhysRevA.76.012305. URL: https://link.aps.org/doi/10.1103/PhysRevA.76.012305.

[25] H. Bombin and M. A. Martin-Delgado. "Topological Quantum Distillation". In: *Phys. Rev. Lett.* 97 (18 Oct. 2006), p. 180501. DOI: 10.1103/PhysRevLett.97.180501. URL: https://link.aps.org/doi/10.1103/PhysRevLett.97.180501.

[26] H. Bombin et al. "Strong Resilience of Topological Codes to Depolarization". In: *Phys. Rev. X* 2 (2 Apr. 2012), p. 021004. DOI: 10.1103/PhysRevX.2.021004. URL: https://link.aps.org/doi/10.1103/PhysRevX.2.021004.

[27] Héctor Bombin et al. "Logical Blocks for Fault-Tolerant Topological Quantum Computation". In: *PRX Quantum* 4 (2 Apr. 2023), p. 020303. DOI: 10.1103/PRXQuantum.4.020303. URL: https://link.aps.org/doi/10.1103/PRXQuantum.4.020303.

[28] Héctor Bombín. "Gauge color codes: optimal transversal gates and gauge fixing in topological stabilizer codes". In: *New Journal of Physics* 17.8 (Aug. 2015), p. 083002. DOI: 10.1088/1367-2630/17/8/083002. URL: https://dx.doi.org/10.1088/1367-2630/17/8/083002.

[29] Juan Pablo Bonilla Ataides et al. "The XZZX surface code". In: *Nature Communications* 12.1 (Apr. 2021), p. 2172. ISSN: 2041-1723. DOI: 10.1038/s41467-021-22274-1. URL: https://doi.org/10.1038/s41467-021-22274-1.

[30] Burak BoyacÄ±, Thu Huong Dang, and Adam N. Letchford. "On matchings, T-joins, and arc routing in road networks". In: *Networks* 79.1 (2022), pp. 20–31. DOI: https://doi.org/10.1002/net.22033. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/net.22033. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/net.22033.

[31] Sergey Bravyi, Matthew B Hastings, and Frank Verstraete. "Lieb-Robinson bounds and the generation of correlations and topological quantum order". In: *Physical review letters* 97.5 (2006), p. 050401. DOI: 10.1103/PhysRevLett.97.050401.

[32] Sergey Bravyi and Alexei Kitaev. "Universal quantum computation with ideal Clifford gates and noisy ancillas". In: *Phys. Rev. A* 71 (2 Feb. 2005), p. 022316. DOI: 10.1103/PhysRevA.71.022316. URL: https://link.aps.org/doi/10.1103/PhysRevA.71.022316.

[33] Sergey Bravyi, David Poulin, and Barbara Terhal. "Tradeoffs for reliable quantum information storage in 2D systems". In: *Physical review letters* 104.5 (2010), p. 050503.

[34] Sergey Bravyi, Martin Suchara, and Alexander Vargo. "Efficient algorithms for maximum likelihood decoding in the surface code". In: *Phys. Rev. A* 90 (3 Sept. 2014), p. 032326. DOI: 10.1103/PhysRevA.90.032326. URL: https://link.aps.org/doi/10.1103/PhysRevA.90.032326.

[35] Sergey Bravyi et al. "High-threshold and low-overhead fault-tolerant quantum memory". In: (2023). arXiv: 2308.07915 [quant-ph].

[36] Sergey Bravyi et al. "Subsystem surface codes with three-qubit check operators". In: *Quantum Inf. Comput.* 13.11-12 (2013), pp. 963–985. DOI: 10.26421/QIC13.11-12-4. URL: https://doi.org/10.26421/QIC13.11-12-4.

[37] Sergey B. Bravyi and Alexei Yu. Kitaev. "Fermionic Quantum Computation". In: *Annals of Physics* 298.1 (2002), pp. 210–226. ISSN: 0003-4916. DOI: https://doi.org/10.1006/aphy.2002.6254. URL: https://www.sciencedirect.com/science/article/pii/S0003491602962548.

[38] Nikolas P Breuckmann et al. "Hyperbolic and semi-hyperbolic surface codes for quantum storage". In: *Quantum Science and Technology* 2.3 (Aug. 2017), p. 035007. DOI: 10.1088/2058-9565/aa7d3b. URL: https://dx.doi.org/10.1088/2058-9565/aa7d3b.

[39] Nikolas P Breuckmann et al. “Hyperbolic and semi-hyperbolic surface codes for quantum storage”. In: *Quantum Science and Technology* 2.3 (Aug. 2017), p. 035007. DOI: 10.1088/2058-9565/aa7d3b. URL: https://dx.doi.org/10.1088/2058-9565/aa7d3b.

[40] Nikolas P. Breuckmann. “PhD thesis: Homological Quantum Codes Beyond the Toric Code”. In: (2018). arXiv: 1802.01520 [quant-ph].

[41] Nikolas P. Breuckmann and Simon Burton. “Fold-Transversal Clifford Gates for Quantum Codes”. In: (2022). arXiv: 2202.06647 [quant-ph].

[42] Nikolas P. Breuckmann and Jens N. Eberhardt. “Balanced Product Quantum Codes”. In: *IEEE Transactions on Information Theory* 67.10 (2021), pp. 6653–6674. DOI: 10.1109/TIT.2021.3097347.

[43] Nikolas P. Breuckmann and Jens Niklas Eberhardt. “Quantum Low-Density Parity-Check Codes”. In: *PRX Quantum* 2 (4 Oct. 2021), p. 040101. DOI: 10.1103/PRXQuantum.2.040101. URL: https://link.aps.org/doi/10.1103/PRXQuantum.2.040101.

[44] Nikolas P. Breuckmann and Barbara M. Terhal. “Constructions and Noise Threshold of Hyperbolic Surface Codes”. In: *IEEE Transactions on Information Theory* 62.6 (2016), pp. 3731–3744. DOI: 10.1109/TIT.2016.2555700.

[45] Benjamin J. Brown, Naomi H. Nickerson, and Dan E. Browne. “Fault-tolerant error correction with the gauge color code”. In: *Nature Communications* 7.1 (July 2016), p. 12302. ISSN: 2041-1723. DOI: 10.1038/ncomms12302. URL: https://doi.org/10.1038/ncomms12302.

[46] Natalie C Brown, Michael Newman, and Kenneth R Brown. “Handling leakage with subsystem codes”. In: *New Journal of Physics* 21.7 (2019), p. 073055.

[47] A. R. Calderbank and Peter W. Shor. “Good quantum error-correcting codes exist”. In: *Phys. Rev. A* 54 (2 Aug. 1996), pp. 1098–1105. DOI: 10.1103/

PhysRevA.54.1098. URL: https://link.aps.org/doi/10.1103/PhysRevA.54.1098.

[48] Christopher Chamberland et al. "Topological and Subsystem Codes on Low-Degree Graphs with Flag Qubits". In: *Phys. Rev. X* 10 (1 Jan. 2020), p. 011022. DOI: 10.1103/PhysRevX.10.011022. URL: https://link.aps.org/doi/10.1103/PhysRevX.10.011022.

[49] Rui Chao et al. "Optimization of the surface code design for Majorana-based qubits". In: *Quantum* 4 (Oct. 2020), p. 352. ISSN: 2521-327X. DOI: 10.22331/q-2020-10-28-352. URL: https://doi.org/10.22331/q-2020-10-28-352.

[50] Edward H Chen et al. "Calibrated decoders for experimental quantum error correction". In: *arXiv preprint arXiv:2110.04285* (2021).

[51] Jinghu Chen et al. "Reduced-complexity decoding of LDPC codes". In: *IEEE Transactions on Communications* 53.8 (2005), pp. 1288–1299. DOI: 10.1109/TCOMM.2005.852852.

[52] Richard Cleve and Daniel Gottesman. "Efficient computations of encodings for quantum error correction". In: *Phys. Rev. A* 56 (1 July 1997), pp. 76–82. DOI: 10.1103/PhysRevA.56.76. URL: https://link.aps.org/doi/10.1103/PhysRevA.56.76.

[53] J. Conrad et al. "The small stellated dodecahedron code and friends". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 376.2123 (2018), p. 20170323. DOI: 10.1098/rsta.2017.0323. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2017.0323. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rsta.2017.0323.

[54] Ben Criger and Imran Ashraf. "Multi-path summation for decoding 2D topological codes". In: *Quantum* 2 (2018), p. 102.

[55] Poulami Das et al. “AFS: Accurate, Fast, and Scalable Error-Decoding for Fault-Tolerant Quantum Computers”. In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2022, pp. 259–273. DOI: 10.1109/HPCA53966.2022.00027.

[56] Margarita Davydova, Nathanan Tantivasadakarn, and Shankar Balasubramanian. “Floquet Codes without Parent Subsystem Codes”. In: *PRX Quantum* 4 (2 June 2023), p. 020341. DOI: 10.1103/PRXQuantum.4.020341. URL: https://link.aps.org/doi/10.1103/PRXQuantum.4.020341.

[57] Margarita Davydova et al. “Quantum computation from dynamic automorphism codes”. In: (2023). arXiv: 2307.10353 [quant-ph].

[58] RD Delaney et al. “Superconducting-qubit readout via low-backaction electro-optic transduction”. In: *Nature* 606.7914 (2022), pp. 489–493. DOI: 10.1038/s41586-022-04720-2. URL: https://doi.org/10.1038/s41586-022-04720-2.

[59] Nicolas Delfosse. “Tradeoffs for reliable quantum information storage in surface codes and color codes”. In: *2013 IEEE International Symposium on Information Theory*. 2013, pp. 917–921. DOI: 10.1109/ISIT.2013.6620360.

[60] Nicolas Delfosse and Naomi H. Nickerson. “Almost-linear time decoding algorithm for topological codes”. In: *Quantum* 5 (Dec. 2021), p. 595. ISSN: 2521-327X. DOI: 10.22331/q-2021-12-02-595. URL: https://doi.org/10.22331/q-2021-12-02-595.

[61] Nicolas Delfosse and Adam Paetznick. “Spacetime codes of Clifford circuits”. In: (2023). arXiv: 2304.05943 [quant-ph].

[62] Nicolas Delfosse and Jean-Pierre Tillich. “A decoding algorithm for CSS codes using the X/Z correlations”. In: *2014 IEEE International Symposium on Information Theory*. 2014, pp. 1071–1075. DOI: 10.1109/ISIT.2014.6874997.

[63] Nicolas Delfosse and Gilles Zémor. "Linear-time maximum likelihood decoding of surface codes over the quantum erasure channel". In: *Phys. Rev. Res.* 2 (3 July 2020), p. 033042. DOI: 10.1103/PhysRevResearch.2.033042. URL: https://link.aps.org/doi/10.1103/PhysRevResearch.2.033042.

[64] Eric Dennis et al. "Topological quantum memory". In: *Journal of Mathematical Physics* 43.9 (Aug. 2002), pp. 4452–4505. ISSN: 0022-2488. DOI: 10.1063/1.1499754. eprint: https://pubs.aip.org/aip/jmp/article-pdf/43/9/4452/8171926/4452_1_online.pdf. URL: https://doi.org/10.1063/1.1499754.

[65] Charles Derby et al. "Compact fermion to qubit mappings". In: *Phys. Rev. B* 104 (3 July 2021), p. 035118. DOI: 10.1103/PhysRevB.104.035118. URL: https://link.aps.org/doi/10.1103/PhysRevB.104.035118.

[66] David P DiVincenzo and Firat Solgun. "Multi-qubit parity measurement in circuit quantum electrodynamics". In: *New Journal of Physics* 15.7 (July 2013), p. 075001. DOI: 10.1088/1367-2630/15/7/075001. URL: https://dx.doi.org/10.1088/1367-2630/15/7/075001.

[67] Arpit Dua et al. "Clifford-deformed Surface Codes". In: *arXiv preprint arXiv:2201.07802* (2022).

[68] Arpit Dua et al. "Engineering Floquet codes by rewinding". In: (2023). arXiv: 2307.13668 [quant-ph].

[69] Guillaume Duclos-Cianci and David Poulin. "A renormalization group decoding algorithm for topological quantum codes". In: *2010 IEEE Information Theory Workshop*. 2010, pp. 1–5. DOI: 10.1109/CIG.2010.5592866.

[70] Guillaume Duclos-Cianci and David Poulin. "Fast decoders for topological quantum codes". In: *Physical review letters* 104.5 (2010), p. 050504.

[71] Guillaume Duclos-Cianci and David Poulin. "Fault-tolerant renormalization group decoder for abelian topological codes". In: *arXiv preprint arXiv:1304.6100* (2013).

[72] Jack Edmonds. “Maximum matching and a polyhedron with 0, 1-vertices”. In: *Journal of research of the National Bureau of Standards B* 69.125-130 (1965), pp. 55–56.

[73] Jack Edmonds. “Paths, Trees, and Flowers”. In: *Canadian Journal of Mathematics* 17 (1965), pp. 449–467. DOI: 10.4153/CJM-1965-045-4.

[74] Jack Edmonds and Ellis L Johnson. “Matching, Euler tours and the Chinese postman”. In: *Mathematical programming* 5 (1973), pp. 88–124.

[75] Laird Egan et al. “Fault-Tolerant Operation of a Quantum Error-Correction Code”. In: *arXiv preprint arXiv:2009.11482* (2020).

[76] Javan Erfanian, Subbarayan Pasupathy, and Glenn Gulak. “Reduced complexity symbol detectors with parallel structure for ISI channels”. In: *IEEE Transactions on Communications* 42.234 (1994), pp. 1661–1671.

[77] Jeff Erickson and Kim Whittlesey. “Greedy optimal homotopy and homology generators”. In: *SODA*. Vol. 5. 2005, pp. 1038–1046.

[78] Ethan Fetaya. “Bounding the distance of quantum surface codes”. In: *Journal of Mathematical Physics* 53.6 (June 2012), p. 062202. ISSN: 0022-2488. DOI: 10.1063/1.4726034. eprint: https://pubs.aip.org/aip/jmp/article-pdf/doi/10.1063/1.4726034/15812523/062202_1_online.pdf. URL: https://doi.org/10.1063/1.4726034.

[79] Marc PC Fossorier, Miodrag Mihaljevic, and Hideki Imai. “Reduced complexity iterative decoding of low-density parity check codes based on belief propagation”. In: *IEEE Transactions on communications* 47.5 (1999), pp. 673–680.

[80] Austin G. Fowler. “Minimum weight perfect matching of fault-tolerant topological quantum error correction in average $O(1)$ parallel time”. In: (2013). arXiv: 1307.1740 [quant-ph].

[81] Austin G. Fowler. “Optimal complexity correction of correlated errors in the surface code”. In: (2013). arXiv: 1310.0863 [quant-ph].

[82] Austin G. Fowler and Craig Gidney. "Low overhead quantum computation using lattice surgery". In: (2019). arXiv: 1808.06709 [quant-ph].

[83] Austin G. Fowler, Adam C. Whiteside, and Lloyd C. L. Hollenberg. "Towards Practical Classical Processing for the Surface Code". In: *Phys. Rev. Lett.* 108 (18 May 2012), p. 180501. DOI: 10.1103/PhysRevLett.108.180501. URL: https://link.aps.org/doi/10.1103/PhysRevLett.108.180501.

[84] Austin G. Fowler et al. "Surface Code Quantum Communication". In: *Phys. Rev. Lett.* 104 (18 May 2010), p. 180503. DOI: 10.1103/PhysRevLett.104.180503. URL: https://link.aps.org/doi/10.1103/PhysRevLett.104.180503.

[85] Austin G. Fowler et al. "Surface codes: Towards practical large-scale quantum computation". In: *Phys. Rev. A* 86 (3 Sept. 2012), p. 032324. DOI: 10.1103/PhysRevA.86.032324. URL: https://link.aps.org/doi/10.1103/PhysRevA.86.032324.

[86] Keisuke Fujii and Yuuki Tokunaga. "Error and loss tolerances of surface codes with general lattice structures". In: *Phys. Rev. A* 86 (2 Aug. 2012), p. 020303. DOI: 10.1103/PhysRevA.86.020303. URL: https://link.aps.org/doi/10.1103/PhysRevA.86.020303.

[87] Zvi Galil. "Efficient algorithms for finding maximum matching in graphs". In: *ACM Computing Surveys (CSUR)* 18.1 (1986), pp. 23–38.

[88] Craig Gidney. "A Pair Measurement Surface Code on Pentagons". In: (2022). arXiv: 2206.12780 [quant-ph].

[89] Craig Gidney. "Cleaner magic states with hook injection". In: (2023). arXiv: 2302.12292 [quant-ph].

[90] Craig Gidney. *Decorrelated Depolarization*. https://algassert.com/post/2001. Accessed: 2021-10-04. 2020.

[91] Craig Gidney. "Stim: a fast stabilizer circuit simulator". In: *arXiv preprint arXiv:2103.02202* (2021).

[92] Craig Gidney. "Stim: a fast stabilizer circuit simulator". In: *Quantum* 5 (July 2021), p. 497. ISSN: 2521-327X. DOI: 10.22331/q-2021-07-06-497. URL: https://doi.org/10.22331/q-2021-07-06-497.

[93] Craig Gidney and Dave Bacon. "Less Bacon More Threshold". In: (2023). arXiv: 2305.12046 [quant-ph].

[94] Craig Gidney and Martin Ekerå. "How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits". In: *Quantum* 5 (Apr. 2021), p. 433. ISSN: 2521-327X. DOI: 10.22331/q-2021-04-15-433. URL: https://doi.org/10.22331/q-2021-04-15-433.

[95] Craig Gidney, Michael Newman, and Matt McEwen. "Benchmarking the Planar Honeycomb Code". In: *Quantum* 6 (Sept. 2022), p. 813. ISSN: 2521-327X. DOI: 10.22331/q-2022-09-21-813. URL: https://doi.org/10.22331/q-2022-09-21-813.

[96] Craig Gidney, Michael Newman, and Matt McEwen. *Generation/Simulation Tools for the Planar Honeycomb Code*. https://github.com/Strilanc/honeycomb-boundaries. 2022. URL: https://github.com/Strilanc/honeycomb-boundaries.

[97] Craig Gidney et al. "A Fault-Tolerant Honeycomb Memory". In: *Quantum* 5 (Dec. 2021), p. 605. ISSN: 2521-327X. DOI: 10.22331/q-2021-12-20-605. URL: https://doi.org/10.22331/q-2021-12-20-605.

[98] Daniel Gottesman. *Stabilizer codes and quantum error correction*. California Institute of Technology, 1997.

[99] Jérémie Guillaud and Mazyar Mirrahimi. "Repetition Cat Qubits for Fault-Tolerant Quantum Computation". In: *Phys. Rev. X* 9 (4 Dec. 2019), p. 041053. DOI: 10.1103/PhysRevX.9.041053. URL: https://link.aps.org/doi/10.1103/PhysRevX.9.041053.

[100] Jeongwan Haah and Matthew B. Hastings. "Boundaries for the Honeycomb Code". In: *Quantum* 6 (Apr. 2022), p. 693. ISSN: 2521-327X. DOI: 10.

22331/q-2022-04-21-693. URL: https://doi.org/10.22331/q-2022-04-21-693.

[101] John H. Halton. "On the thickness of graphs of given degree". In: *Information Sciences* 54.3 (1991), pp. 219–238. ISSN: 0020-0255. DOI: https://doi.org/10.1016/0020-0255(91)90052-V. URL: https://www.sciencedirect.com/science/article/pii/002002559190052V.

[102] Matthew B. Hastings and Jeongwan Haah. "Dynamically Generated Logical Qubits". In: *Quantum* 5 (Oct. 2021), p. 564. ISSN: 2521-327X. DOI: 10.22331/q-2021-10-19-564. URL: https://doi.org/10.22331/q-2021-10-19-564.

[103] Matthew B. Hastings, Jeongwan Haah, and Ryan O'Donnell. "Fiber Bundle Codes: Breaking the N1/2 Polylog(n) Barrier for Quantum LDPC Codes". In: *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2021. Virtual, Italy: Association for Computing Machinery, 2021, pp. 1276–1288. ISBN: 9781450380539. DOI: 10.1145/3406325.3451005. URL: https://doi.org/10.1145/3406325.3451005.

[104] Allen Hatcher. *Algebraic Topology*. Cambridge University Press, 2002.

[105] Vojt ěch Havlí ček, Matthias Troyer, and James D. Whitfield. "Operator locality in the quantum simulation of fermionic models". In: *Phys. Rev. A* 95 (3 Mar. 2017), p. 032332. DOI: 10.1103/PhysRevA.95.032332. URL: https://link.aps.org/doi/10.1103/PhysRevA.95.032332.

[106] Oscar Higgott. "PyMatching: A Python Package for Decoding Quantum Codes with Minimum-Weight Perfect Matching". In: *ACM Transactions on Quantum Computing* 3.3 (June 2022). ISSN: 2643-6809. DOI: 10.1145/3505637. URL: https://doi.org/10.1145/3505637.

[107] Oscar Higgott. "PyMatching: A Python package for decoding quantum codes with minimum-weight perfect matching". In: *ACM Transactions on Quantum Computing* 3.3 (2022), pp. 1–16.

[108] Oscar Higgott and Nikolas P Breuckmann. *Ancillary data for hyperbolic and semi-hyperbolic Floquet codes*. https://github.com/oscarhiggott/hyperbolic-floquet-data. 2023. URL: https://github.com/oscarhiggott/hyperbolic-floquet-data.

[109] Oscar Higgott and Nikolas P Breuckmann. *Quantum computing error correction method, code, and system*. US Patent App. 17/444,943. Mar. 2023.

[110] Oscar Higgott and Nikolas P Breuckmann. "Subsystem codes with high thresholds by gauge fixing and reduced qubit overhead". In: *arXiv preprint arXiv:2010.09626* (2020).

[111] Oscar Higgott and Nikolas P. Breuckmann. "Constructions and performance of hyperbolic and semi-hyperbolic Floquet codes". In: (2023). arXiv: 2308.03750 [quant-ph].

[112] Oscar Higgott and Nikolas P. Breuckmann. "Subsystem Codes with High Thresholds by Gauge Fixing and Reduced Qubit Overhead". In: *Phys. Rev. X* 11 (3 Aug. 2021), p. 031039. DOI: 10.1103/PhysRevX.11.031039. URL: https://link.aps.org/doi/10.1103/PhysRevX.11.031039.

[113] Oscar Higgott and Craig Gidney. "Sparse Blossom: correcting a million errors per core second with minimum-weight matching". In: (2023). arXiv: 2303.15933 [quant-ph].

[114] Oscar Higgott et al. "Improved Decoding of Circuit Noise and Fragile Boundaries of Tailored Surface Codes". In: *Phys. Rev. X* 13 (3 July 2023), p. 031007. DOI: 10.1103/PhysRevX.13.031007. URL: https://link.aps.org/doi/10.1103/PhysRevX.13.031007.

[115] Oscar Higgott et al. "Improved decoding of circuit noise and fragile boundaries of tailored surface codes". In: (2023). arXiv: 2203.04948 [quant-ph].

[116] Oscar Higgott et al. "Optimal local unitary encoding circuits for the surface code". In: *Quantum* 5 (Aug. 2021), p. 517. ISSN: 2521-327X. DOI: 10.

22331/q-2021-08-05-517. URL: https://doi.org/10.22331/q-2021-08-05-517.

[117] Dominic Horsman et al. "Surface code quantum computing by lattice surgery". In: *New Journal of Physics* 14.12 (Dec. 2012), p. 123011. DOI: 10.1088/1367-2630/14/12/123011. URL: https://dx.doi.org/10.1088/1367-2630/14/12/123011.

[118] Min-Hsiu Hsieh and Fran çois Le Gall. "NP-hardness of decoding quantum error-correction codes". In: *Phys. Rev. A* 83 (5 May 2011), p. 052331. DOI: 10.1103/PhysRevA.83.052331. URL: https://link.aps.org/doi/10.1103/PhysRevA.83.052331.

[119] Shilin Huang, Michael Newman, and Kenneth R. Brown. "Fault-tolerant weighted union-find decoding on the toric code". In: *Phys. Rev. A* 102 (1 July 2020), p. 012419. DOI: 10.1103/PhysRevA.102.012419. URL: https://link.aps.org/doi/10.1103/PhysRevA.102.012419.

[120] Adrian Hutter, James R Wootton, and Daniel Loss. "Efficient Markov chain Monte Carlo algorithm for the surface code". In: *Physical Review A* 89.2 (2014), p. 022326.

[121] Poolad Imany et al. "Quantum phase modulation with acoustic cavities and quantum dots". In: *Optica* 9.5 (May 2022), pp. 501–504. DOI: 10.1364/OPTICA.451418. URL: https://opg.optica.org/optica/abstract.cfm?URI=optica-9-5-501.

[122] Pavithran Iyer and David Poulin. "Hardness of Decoding Quantum Stabilizer Codes". In: *IEEE Transactions on Information Theory* 61.9 (2015), pp. 5209–5223. DOI: 10.1109/TIT.2015.2422294.

[123] Zhang Jiang et al. "Majorana Loop Stabilizer Codes for Error Mitigation in Fermionic Quantum Simulations". In: *Phys. Rev. Applied* 12 (6 Dec. 2019), p. 064041. DOI: 10.1103/PhysRevApplied.12.064041. URL: https://link.aps.org/doi/10.1103/PhysRevApplied.12.064041.

[124] Petar Jurcevic et al. “Demonstration of quantum volume 64 on a superconducting quantum computing system”. In: *Quantum Science and Technology* 6.2 (Mar. 2021), p. 025020. DOI: 10.1088/2058-9565/abe519. URL: https://doi.org/10.1088/2058-9565/abe519.

[125] Norbert Kalb et al. “Entanglement distillation between solid-state quantum network nodes”. In: *Science* 356.6341 (2017), pp. 928–932.

[126] Markus S. Kesselring et al. “Anyon condensation and the color code”. In: (2022). arXiv: 2212.00042 [quant-ph].

[127] A.Yu. Kitaev. “Fault-tolerant quantum computation by anyons”. In: *Annals of Physics* 303.1 (2003), pp. 2–30. ISSN: 0003-4916. DOI: https://doi.org/10.1016/S0003-4916(02)00018-0. URL: https://www.sciencedirect.com/science/article/pii/S0003491602000180.

[128] Alexei Kitaev. “Anyons in an exactly solved model and beyond”. In: *Annals of Physics* 321.1 (2006). January Special Issue, pp. 2–111. ISSN: 0003-4916. DOI: https://doi.org/10.1016/j.aop.2005.10.005. URL: https://www.sciencedirect.com/science/article/pii/S0003491605002381.

[129] Emanuel Knill and Raymond Laflamme. “Theory of quantum error-correcting codes”. In: *Phys. Rev. A* 55 (2 Feb. 1997), pp. 900–911. DOI: 10.1103/PhysRevA.55.900. URL: https://link.aps.org/doi/10.1103/PhysRevA.55.900.

[130] Alicia J Kollár, Mattias Fitzpatrick, and Andrew A Houck. “Hyperbolic lattices in circuit quantum electrodynamics”. In: *Nature* 571.7763 (2019), pp. 45–50. DOI: 10.1038/s41586-019-1348-3. URL: https://doi.org/10.1038/s41586-019-1348-3.

[131] Vladimir Kolmogorov. “Blossom V: a new implementation of a minimum cost perfect matching algorithm”. In: *Mathematical Programming Computation* 1.1 (July 2009), pp. 43–67. ISSN: 1867-2957. DOI: 10.1007/s12532-009-0002-8. URL: https://doi.org/10.1007/s12532-009-0002-8.

[132] Bernhard H Korte et al. *Combinatorial optimization*. Vol. 1. Springer, 2011.

[133] Sebastian Krinner et al. "Realizing repeated quantum error correction in a distance-three surface code". In: *Nature* 605.7911 (2022), pp. 669–674.

[134] Kao-Yueh Kuo and Chung-Chin Lu. "On the hardness of decoding quantum stabilizer codes under the depolarizing channel". In: *2012 International Symposium on Information Theory and its Applications*. 2012, pp. 208–211.

[135] Anthony Leverrier and Gilles Zémor. "Quantum Tanner codes". In: *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. 2022, pp. 872–883. DOI: 10.1109/FOCS54457.2022.00117.

[136] Muyuan Li, Daniel Miller, and Kenneth R Brown. "Direct measurement of Bacon-Shor code stabilizers". In: *Physical Review A* 98.5 (2018), p. 050301.

[137] Muyuan Li and Theodore 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)*. 2020, pp. 109–119. DOI: 10.1109/QCE49297.2020.00024.

[138] Muyuan Li et al. "2D Compass Codes". In: *Phys. Rev. X* 9 (2 May 2019), p. 021041. DOI: 10.1103/PhysRevX.9.021041. URL: https://link.aps.org/doi/10.1103/PhysRevX.9.021041.

[139] Ying Li. "A magic state's fidelity can be superior to the operations that created it". In: *New Journal of Physics* 17.2 (Feb. 2015), p. 023037. DOI: 10.1088/1367-2630/17/2/023037. URL: https://dx.doi.org/10.1088/1367-2630/17/2/023037.

[140] Daniel Litinski. "A game of surface codes: Large-scale quantum computing with lattice surgery". In: *Quantum* 3 (2019), p. 128.

[141] Namitha Liyanage et al. "Scalable Quantum Error Correction for Surface Codes using FPGA". In: (2023). arXiv: 2301.08419 [quant-ph].

[142] Seth Lloyd. "Universal Quantum Simulators". In: *Science* 273.5278 (1996), pp. 1073–1078. DOI: 10.1126/science.273.5278.1073. eprint: https://www.science.org/doi/pdf/10.1126/science.273.5278.1073. URL: https://www.science.org/doi/abs/10.1126/science.273.5278.1073.

[143] David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

[144] David JC MacKay and Radford M Neal. "Near Shannon limit performance of low density parity check codes". In: *Electronics letters* 32.18 (1996), pp. 1645–1646.

[145] J. F. Marques et al. "Logical-qubit operations in an error-detecting surface code". In: *Nature Physics* 18.1 (Jan. 2022), pp. 80–86. ISSN: 1745-2481. DOI: 10.1038/s41567-021-01423-9. URL: https://doi.org/10.1038/s41567-021-01423-9.

[146] Jiri Matoušek and Bernd Gärtner. *Understanding and using linear programming*. Vol. 33. Springer, 2007.

[147] Matt McEwen, Dave Bacon, and Craig Gidney. "Relaxing Hardware Requirements for Surface Code Circuits using Time-dynamics". In: *arXiv preprint arXiv:2302.02192* (2023).

[148] Kai Meinerz, Chae-Yeun Park, and Simon Trebst. "Scalable Neural Decoder for Topological Surface Codes". In: *Phys. Rev. Lett.* 128 (8 Feb. 2022), p. 080505. DOI: 10.1103/PhysRevLett.128.080505. URL: https://link.aps.org/doi/10.1103/PhysRevLett.128.080505.

[149] Mikhail V Menshikov. "Coincidence of critical points in percolation problems". In: *Soviet Mathematics Doklady*. Vol. 33. 1986, pp. 856–859.

[150] C. Monroe et al. "Large-scale modular quantum-computer architecture with atomic memory and photonic interconnects". In: *Phys. Rev. A* 89 (2 Feb. 2014), p. 022317. DOI: 10.1103/PhysRevA.89.022317. URL: https://link.aps.org/doi/10.1103/PhysRevA.89.022317.

[151] Lee Mosher. *When can a homomorphism be determined entirely by its generators*. Mathematics Stack Exchange. https://math.stackexchange.com/q/1402612 (version: 2015-08-19).

[152] Naomi H Nickerson, Ying Li, and Simon C Benjamin. "Topological quantum computing with a very noisy network and local error rates approaching one percent". In: *Nature communications* 4.1 (2013), p. 1756. DOI: 10.1038/ncomms2773. URL: https://doi.org/10.1038/ncomms2773.

[153] Naomi H. Nickerson, Joseph F. Fitzsimons, and Simon C. Benjamin. "Freely Scalable Quantum Technologies Using Cells of 5-to-50 Qubits with Very Lossy and Noisy Photonic Links". In: *Phys. Rev. X* 4 (4 Dec. 2014), p. 041041. DOI: 10.1103/PhysRevX.4.041041. URL: https://link.aps.org/doi/10.1103/PhysRevX.4.041041.

[154] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. 10th. USA: Cambridge University Press, 2011. ISBN: 1107002176.

[155] Adam Paetznick and Ben W Reichardt. "Universal fault-tolerant quantum computation with only transversal gates and error correction". In: *Physical review letters* 111.9 (2013), p. 090505.

[156] Adam Paetznick et al. "Performance of Planar Floquet Codes with Majorana-Based Qubits". In: *PRX Quantum* 4 (1 Jan. 2023), p. 010310. DOI: 10.1103/PRXQuantum.4.010310. URL: https://link.aps.org/doi/10.1103/PRXQuantum.4.010310.

[157] Pavel Panteleev and Gleb Kalachev. "Asymptotically Good Quantum and Locally Testable Classical LDPC Codes". In: *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*. STOC 2022. Rome, Italy: Association for Computing Machinery, 2022, pp. 375–388. ISBN: 9781450392648. DOI: 10.1145/3519935.3520017. URL: https://doi.org/10.1145/3519935.3520017.

[158] Pavel Panteleev and Gleb Kalachev. "Degenerate quantum LDPC codes with good finite length performance". In: *arXiv preprint arXiv:1904.02703* (2019).

[159] Pavel Panteleev and Gleb Kalachev. "Quantum LDPC Codes With Almost Linear Minimum Distance". In: *IEEE Transactions on Information Theory* 68.1 (2022), pp. 213–229. DOI: 10.1109/TIT.2021.3119384.

[160] Christopher A. Pattison, Anirudh Krishna, and John Preskill. "Hierarchical memories: Simulating quantum LDPC codes with local gates". In: (2023). arXiv: 2303.04798 [quant-ph].

[161] Christopher A. Pattison et al. "Improved quantum error correction using soft information". In: (2021). arXiv: 2107.13589 [quant-ph].

[162] Juan M Pino et al. "Demonstration of the trapped-ion quantum CCD computer architecture". In: *Nature* 592.7853 (2021), pp. 209–213. DOI: 10.1038/s41586-021-03318-4. URL: https://doi.org/10.1038/s41586-021-03318-4.

[163] David Poulin. "Stabilizer Formalism for Operator Quantum Error Correction". In: *Phys. Rev. Lett.* 95 (23 Dec. 2005), p. 230504. DOI: 10.1103/PhysRevLett.95.230504. URL: https://link.aps.org/doi/10.1103/PhysRevLett.95.230504.

[164] David Poulin and Yeojin Chung. "On the iterative decoding of sparse quantum codes". In: (2008). arXiv: 0801.1241 [quant-ph].

[165] Shruti Puri et al. "Bias-preserving gates with stabilized cat qubits". In: *Science Advances* 6.34 (2020), eaay5901. DOI: 10.1126/sciadv.aay5901. eprint: https://www.science.org/doi/pdf/10.1126/sciadv.aay5901. URL: https://www.science.org/doi/abs/10.1126/sciadv.aay5901.

[166] Armanda O. Quintavalle, Paul Webster, and Michael Vasmer. "Partitioning qubits in hypergraph product codes to implement logical gates". In: (2022). arXiv: 2204.10812 [quant-ph].

[167] Joshua Ramette et al. “Fault-Tolerant Connection of Error-Corrected Qubits with Noisy Links”. In: (2023). arXiv: 2302.01296 [quant-ph].

[168] Andreas Reiserer and Gerhard Rempe. “Cavity-based quantum networks with single atoms and optical photons”. In: *Rev. Mod. Phys.* 87 (4 Dec. 2015), pp. 1379–1418. DOI: 10.1103/RevModPhys.87.1379. URL: https://link.aps.org/doi/10.1103/RevModPhys.87.1379.

[169] Joschka Roffe et al. “Decoding across the quantum low-density parity-check code landscape”. In: *Phys. Rev. Res.* 2 (4 Dec. 2020), p. 043423. DOI: 10.1103/PhysRevResearch.2.043423. URL: https://link.aps.org/doi/10.1103/PhysRevResearch.2.043423.

[170] Rishabh Sahu et al. “Quantum-enabled operation of a microwave-optical interface”. In: *Nature communications* 13.1 (2022), p. 1276. DOI: 10.1038/s41467-022-28924-2. URL: https://doi.org/10.1038/s41467-022-28924-2.

[171] Jacob T. Seeley, Martin J. Richard, and Peter J. Love. “The Bravyi-Kitaev transformation for quantum computation of electronic structure”. In: *The Journal of Chemical Physics* 137.22 (2012), p. 224109. DOI: 10.1063/1.4768229.

[172] Peter W. Shor. “Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer”. In: *SIAM Review* 41.2 (1999), pp. 303–332. DOI: 10.1137/S0036144598347011. eprint: https://doi.org/10.1137/S0036144598347011. URL: https://doi.org/10.1137/S0036144598347011.

[173] Adam Siegel et al. “Adaptive surface code for quantum error correction in the presence of temporary or permanent defects”. In: *Quantum* 7 (July 2023), p. 1065. ISSN: 2521-327X. DOI: 10.22331/q-2023-07-25-1065. URL: https://doi.org/10.22331/q-2023-07-25-1065.

[174] Jozef Širáň. "Triangle group representations and constructions of regular maps". In: *Proceedings of the London Mathematical Society* 82.3 (May 2001), pp. 513–532. ISSN: 0024-6115. DOI: 10.1112/plms/82.3.513. eprint: https://academic.oup.com/plms/article-pdf/82/3/513/4378132/82-3-513.pdf. URL: https://doi.org/10.1112/plms/82.3.513.

[175] Luka Skoric et al. "Parallel window decoding enables scalable fault tolerant quantum computation". In: (2023). arXiv: 2209.08552 [quant-ph].

[176] A. M. Steane. "Error Correcting Codes in Quantum Theory". In: *Phys. Rev. Lett.* 77 (5 July 1996), pp. 793–797. DOI: 10.1103/PhysRevLett.77.793. URL: https://link.aps.org/doi/10.1103/PhysRevLett.77.793.

[177] Ashley M Stephens. "Fault-tolerant thresholds for quantum error correction with the surface code". In: *Physical Review A* 89.2 (2014), p. 022321.

[178] L. J. Stephenson et al. "High-Rate, High-Fidelity Entanglement of Qubits Across an Elementary Quantum Network". In: *Phys. Rev. Lett.* 124 (11 Mar. 2020), p. 110501. DOI: 10.1103/PhysRevLett.124.110501. URL: https://link.aps.org/doi/10.1103/PhysRevLett.124.110501.

[179] Mark Steudtner and Stephanie Wehner. "Quantum codes for quantum simulation of fermions on a square lattice of qubits". In: *Phys. Rev. A* 99 (2 Feb. 2019), p. 022308. DOI: 10.1103/PhysRevA.99.022308. URL: https://link.aps.org/doi/10.1103/PhysRevA.99.022308.

[180] Armands Strikis, Simon C. Benjamin, and Benjamin J. Brown. "Quantum Computing is Scalable on a Planar Array of Qubits with Fabrication Defects". In: *Phys. Rev. Appl.* 19 (6 June 2023), p. 064081. DOI: 10.1103/PhysRevApplied.19.064081. URL: https://link.aps.org/doi/10.1103/PhysRevApplied.19.064081.

[181] Martin Suchara, Sergey Bravyi, and Barbara Terhal. "Constructions and noise threshold of topological subsystem codes". In: *Journal of Physics A: Mathematical and Theoretical* 44.15 (Mar. 2011), p. 155301. DOI: 10.1088/

1751-8113/44/15/155301. URL: https://dx.doi.org/10.1088/1751-8113/44/15/155301.

[182] Joseph Sullivan, Rui Wen, and Andrew C. Potter. "Floquet codes and phases in twist-defect networks". In: (2023). arXiv: 2303.17664 [quant-ph].

[183] Neereja Sundaresan et al. "Demonstrating multi-round subsystem quantum error correction using matching and maximum likelihood decoders". In: *Nature Communications* 14.1 (May 2023). DOI: 10.1038/s41467-023-38247-5. URL: https://doi.org/10.1038%2Fs41467-023-38247-5.

[184] Xinyu Tan et al. "Scalable surface code decoders with parallelization in time". In: *arXiv preprint arXiv:2209.09219* (2022).

[185] Google Quantum AI Team. *Data for "Suppressing quantum errors by scaling a surface code logical qubit"*. Zenodo, July 2022. DOI: 10.5281/zenodo.6804040. URL: https://doi.org/10.5281/zenodo.6804040.

[186] Barbara M. Terhal. "Quantum error correction for quantum memories". In: *Rev. Mod. Phys.* 87 (2 Apr. 2015), pp. 307–346. DOI: 10.1103/RevModPhys.87.307. URL: https://link.aps.org/doi/10.1103/RevModPhys.87.307.

[187] Giacomo Torlai and Roger G. Melko. "Neural Decoder for Topological Codes". In: *Phys. Rev. Lett.* 119 (3 July 2017), p. 030501. DOI: 10.1103/PhysRevLett.119.030501. URL: https://link.aps.org/doi/10.1103/PhysRevLett.119.030501.

[188] Alex Townsend-Teague, Julio Magdalena de la Fuente, and Markus Kesselring. "Floquetifying the Colour Code". In: (2023). arXiv: 2307.11136 [quant-ph].

[189] Maxime A. Tremblay, Nicolas Delfosse, and Michael E. Beverland. "Constant-Overhead Quantum Error Correction with Thin Planar Connectivity". In: *Phys. Rev. Lett.* 129 (5 July 2022), p. 050504. DOI: 10.1103/PhysRevLett.129.050504. URL: https://link.aps.org/doi/10.1103/PhysRevLett.129.050504.

[190] Hai-Tao Tu et al. “High-efficiency coherent microwave-to-optics conversion via off-resonant scattering”. In: *Nature Photonics* 16.4 (2022), pp. 291–296. DOI: 10.1038/s41566-022-00959-3. URL: https://doi.org/10.1038/s41566-022-00959-3.

[191] David K Tuckett et al. “Tailoring surface codes for highly biased noise”. In: *Physical Review X* 9.4 (2019), p. 041031.

[192] David K. Tuckett, Stephen D. Bartlett, and Steven T. Flammia. “Ultrahigh Error Threshold for Surface Codes with Biased Noise”. In: *Phys. Rev. Lett.* 120 (5 Jan. 2018), p. 050505. DOI: 10.1103/PhysRevLett.120.050505. URL: https://link.aps.org/doi/10.1103/PhysRevLett.120.050505.

[193] David K. Tuckett et al. “Fault-Tolerant Thresholds for the Surface Code in Excess of 5% under Biased Noise”. In: *Phys. Rev. Lett.* 124 (13 Mar. 2020), p. 130501. DOI: 10.1103/PhysRevLett.124.130501. URL: https://link.aps.org/doi/10.1103/PhysRevLett.124.130501.

[194] David Kingsley Tuckett. “Tailoring surface codes: Improvements in quantum error correction with biased noise”. (qecsim: https://github.com/qecsim/qecsim). PhD thesis. University of Sydney, 2020. DOI: 10.25910/x8xw-9077.

[195] W. T. Tutte. “The Factorization of Linear Graphs”. In: *Journal of the London Mathematical Society* s1-22.2 (1947), pp. 107–111. DOI: https://doi.org/10.1112/jlms/s1-22.2.107. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/jlms/s1-22.2.107. URL: https://londmathsoc.onlinelibrary.wiley.com/doi/abs/10.1112/jlms/s1-22.2.107.

[196] Javier Valls et al. “Syndrome-Based Min-Sum vs OSD-0 Decoders: FPGA Implementation and Analysis for Quantum LDPC Codes”. In: *IEEE Access* 9 (2021), pp. 138734–138743. DOI: 10.1109/ACCESS.2021.3118544.

[197] Frank Verstraete and J Ignacio Cirac. "Mapping local Hamiltonians of fermions to local Hamiltonians of spins". In: *Journal of Statistical Mechanics: Theory and Experiment* 2005.09 (2005), P09012. DOI: 10.1088/1742-5468/2005/09/P09012.

[198] Christophe Vuillot. "Planar Floquet Codes". In: (2021). arXiv: 2110.05348 [quant-ph].

[199] Christophe Vuillot and Nikolas P. Breuckmann. "Quantum pin codes". In: *IEEE Transactions on Information Theory* 68.9 (2022), pp. 5955–5974. DOI: 10.1109/TIT.2022.3170846.

[200] Christophe Vuillot et al. "Code deformation and lattice surgery are gauge fixing". In: *New Journal of Physics* 21.3 (2019), p. 033028.

[201] Chenyang Wang, Jim Harrington, and John Preskill. "Confinement-Higgs transition in a disordered gauge theory and the accuracy threshold for quantum memory". In: *Annals of Physics* 303.1 (2003), pp. 31–58. ISSN: 0003-4916. DOI: https://doi.org/10.1016/S0003-4916(02)00019-2. URL: https://www.sciencedirect.com/science/article/pii/S0003491602000192.

[202] David S Wang, Austin G Fowler, and Lloyd CL Hollenberg. "Surface code quantum computing with error rates over 1%". In: *Physical Review A* 83.2 (2011), p. 020302.

[203] Mark A. Webster, Armanda O. Quintavalle, and Stephen D. Bartlett. "Transversal Diagonal Logical Operators for Stabiliser Codes". In: (2023). arXiv: 2303.15615 [quant-ph].

[204] James R Wootton and Daniel Loss. "High threshold error correction for the surface code". In: *Physical review letters* 109.16 (2012), p. 160503.

[205] Yue Wu. *Fusion Blossom*. https://github.com/yuewuo/fusion-blossom. 2022.

[206] Yue Wu, Namitha Liyanage, and Lin Zhong. "An interpretation of Union-Find Decoder on Weighted Graphs". In: (2022). arXiv: 2211.03288 [quant-ph].

[207] Yue Wu and Lin Zhong. "Fusion Blossom: Fast MWPM Decoders for QEC". In: (2023). arXiv: 2305.08307 [quant-ph].

[208] Qian Xu et al. "Constant-Overhead Fault-Tolerant Quantum Computation with Reconfigurable Atom Arrays". In: (2023). arXiv: 2308.08648 [quant-ph].

[209] Theodore J Yoder. "Universal fault-tolerant quantum computation with Bacon-Shor codes". In: *arXiv preprint arXiv:1705.01686* (2017).

[210] Zhehao Zhang, David Aasen, and Sagar Vijay. "The X-Cube Floquet Code". In: (2022). arXiv: 2211.05784 [quant-ph].

[211] Na Zhu et al. "Waveguide cavity optomagnonics for microwave-to-optics conversion". In: *Optica* 7.10 (Oct. 2020), pp. 1291–1297. DOI: 10.1364/OPTICA.397967. URL: https://opg.optica.org/optica/abstract.cfm?URI=optica-7-10-1291.