Decoding Schemes for Foliated Sparse Quantum Error Correcting Codes

A. Bolt, D. Poulin, and T. M. Stace

1ARC Centre for Engineered Quantum System, Department of Physics, University of Queensland, Brisbane, QLD 4072, Australia
2Département de Physique, Université de Sherbrooke, Québec, Canada

(Dated: October 30, 2018)

Foliated quantum codes are a resource for fault-tolerant measurement-based quantum error correction for quantum repeaters and for quantum computation. They represent a general approach to integrating a range of possible quantum error correcting codes into larger fault-tolerant networks. Here we present an efficient heuristic decoding scheme for foliated quantum codes, based on message passing between primal and dual code sheets. We test this decoder on two different families of sparse quantum error correcting code: turbo codes and bicycle codes, and show reasonably high numerical performance thresholds. We also present a construction schedule for building such code states.

Quantum information processing (QIP) requires that the computational process must be performed with high fidelity. In a noisy environment this will require quantum error correction (QEC) [1, 2]. Depending on the computational model, this noise manifests in different ways. A conceptually and technologically important step in the project to build quantum computers was the observation by Raussendorf et al. [3, 4] that highly-entangled cluster states, are universal resource states with which to perform a quantum computation. In cluster-state-based computation, the computation is driven forward by a series of measurements.

Subsequently, Raussendorf et al. [5–8] proposed a method of fault-tolerant quantum computation utilising cluster states. In this scheme a 3D cluster state lattice is constructed, which can be viewed as a foliation of Kitaev’s surface code [9, 10]. Alternating sheets within this foliated structure correspond to primal or dual surface codes. Measurements on the bulk qubits generate correlations between corresponding logical qubits on the boundary of the lattice. In these schemes, errors are partially revealed through parity checks operators, which can be determined from the outcomes of single qubit measurements.

Raussendorf’s 3D measurement-based computation scheme has proved important for the practical development of quantum computers [11–14], due to its high fault-tolerant computational error thresholds \( \lesssim 1\% [10] \). Furthermore, the robustness to erasure errors [15–17] makes these schemes attractive for various architectures, including optical networks [18]. This high threshold is a result of the underlying surface code, which itself has a high computational error threshold \( \sim 10\% [9, 10, 15, 16] \).

Fault-tolerant, measurement-based quantum computation is achieved, in part, by ‘braiding’ defects within the foliated cluster, to generate subgroup of the Clifford group. By virtue of their topologically protected nature, braiding operations can be made extremely robust, and so can be used to distill magic states. Together, these resources allows for universal quantum computation [7, 8, 19].

In an earlier paper, we showed that all Calderbank-Steen-Shor (CSS) codes can be clusterised using a larger cluster state resource [20]. These cluster state codes can be foliated as a generalisation of Raussendorf’s 3D lattice [20]. In that work we also demonstrated the performance of a turbo code with a heuristic decoder that we have developed.

In this paper we present a detailed description of the decoding algorithm, and we apply the decoder to two classes of foliated CSS codes: serial turbo codes [21–24] and bicycle codes. In contrast to the surface code, these code families have finite rate. This allows for a much lower overhead of physical qubits to encoded qubits as the size of the code is increased, as compared to surface codes. In both cases, the decoder on the complete foliated cluster state uses a soft-input soft-output (SISO) decoder within each sheet of the code as a subroutine, followed by an exchange of marginal information between neighbouring primal and dual code sheets. Iterating these steps yields an error pattern consistent with the error syndrome.

We present Monte Carlo simulations of the error-correcting performance using this decoder, assuming independently distributed Pauli X and Z errors. We analyse the code performance in terms of both the Bit Error Rate (BER) and Word Error Rate (WER). Our numerical results, simulating uncorrelated Pauli noise errors, indicate that the codes exhibit reasonably high (pseudo-)thresholds in the order of a few percent.

In section II we review the clusterization of CSS codes. Section III reviews the foliation of clusterised codes and presents a general decoding approach. In sections IV we review the construction of decoding trellises for convolutional codes, which are a resource for the convolutional decoding. In section V we develop decoding trellises for clusterised convolutional codes within a larger foliated code. In section VI we present a decoding algorithm for foliated convolutional codes. This is a subroutine for the foliated turbo decoder. In section VII we
present the decoding scheme for foliated turbo codes and display our numerical results of simulated trials. Section VII presents a decoding algorithm for foliated bicycle codes, and presents numerical results. In Section VIII we analyse the construction of clustered convolutional and turbo codes from an architectural viewpoint. We investigate fault-tolerant constructions of foliated turbo codes.

I. CLUSTERISED CODES

We begin by reiterating some definitions, in order to set notation for what follows.

A cluster state is defined on a collection of qubits located at the vertices of a graph [3, 4, 25]. A qubit at vertex v carries with it an associated cluster stabiliser $K_v = X_v (\otimes_{N_v} Z) \equiv X_v Z_{N_v}$, acting on it and its neighbours, $N_v$. The cluster state is the $+1$ eigenstate of the stabilisers $K_v$. For example, in Fig. 1(a), there is a cluster stabiliser $X_{a1} Z_{c1} Z_{c2} Z_{a5} Z_{c6}$ associated to the ancilla qubit $a_1$. Operationally, such a state can be produced with single and two qubit gates: each qubit is prepared in a $+1$ eigenstate of $X$, and then C-PHASE gates are applied to pairs of qubits that share an edge in the graph, e.g. in Fig. 1b, between qubit $c_2$ and its graph neighbours $a_1$ and $a_2$.

A stabiliser quantum code is defined as a set of mutually commuting hermitian operators, whose simultaneous $+1$ eigenstates define valid code states. A CSS code is one for which the generators for $S$ can be partitioned into a set of operators for $X$-like stabilisers, $S_X$, which are products of Pauli $X$ operators acting on subsets of the code qubits, and a set of stabilisers for $Z$-like stabilisers, $S_Z$, which are products of Pauli $Z$ operators, i.e. $S = (S_X \cup S_Z)$.

An $[[n, k, d]]$ CSS code can be generated from a larger progenitor cluster state [23]. The progenitor cluster state is the cluster state associated with the Tanner graph of $S_Z$ [20], i.e. a bipartite graph $G = (V, E)$ whose vertices $V$ are labeled as code qubits $c$, or ancilla qubits $a$. Each ancilla qubit $a$ is associated to a stabiliser $Z_{N_a} \in S_Z$, so that $|V| = n + |S_Z|$. $E$ contains the graph edge $(c, a)$ if $[H_Z]_{c, a} = 1$, where $H_Z$ is the parity check matrix. The logical X codestate of the CSS codes is obtained by measuring the ancilla qubits of the progenitor cluster state in the $X$ basis [20].

Examples of clustered CSS codes are shown in Fig. 1. These are the Steane 7 qubit code, 9 qubit Shor code and a 13 qubit surface code with $Z$ stabilisers generated by

$$S_{Z}^{\text{Steane}} = \{ Z_{c_1} Z_{c_2} Z_{c_6} Z_{c_7}, Z_{c_2} Z_{c_3} Z_{c_4} Z_{c_7}, Z_{c_1} Z_{c_5} Z_{c_6} Z_{c_7} \},$$

$$S_{Z}^{\text{Shor}} = \{ Z_{c_1} Z_{c_2}, Z_{c_2} Z_{c_3}, Z_{c_1} Z_{c_5}, Z_{c_1} Z_{c_5} Z_{c_6} \},$$

$$S_{Z}^{\text{Surf}} = \{ Z_{c_1} Z_{c_3} Z_{c_6}, Z_{c_2} Z_{c_4} Z_{c_5} Z_{c_7}, Z_{c_1} Z_{c_5} Z_{c_6}, Z_{c_6} Z_{c_9} Z_{c_{11}}, Z_{c_1} Z_{c_3} Z_{c_{12}}, Z_{c_1} Z_{c_5} Z_{c_{13}} \}.$$

For each of the $Z$-like stabilisers of a code, an ancillary qubit (red squares) is built into a cluster fragment with the associated code qubits (blue circles) in the stabiliser. For instance in the figure each ancillary qubit, $a_k$, is associated with the $i$th stabilizer element of $S_Z$. This construction holds for all CSS codes [20].

CSS codes detect $X$ and $Z$ errors independently. Each error type may be corrected independently, although this disregards potentially useful correlations, if such exist. We will assume independent $X$ and $Z$ Pauli noise, and in what follows, we describe the process for detecting and correcting $Z$ errors using $S_X$ syndrome information. The dual problem, using $S_Z$ syndrome information to correct $X$ errors proceeds in exact analogy.

We note that for every CSS code, there is a dual CSS code. The dual code is generated by exchanging $X$ and $Z$ operators in the stabilisers and logical operators. That is an $X$-like stabiliser in the primal code transforms to a $Z$-like stabiliser in the dual code, and vice versa. Following the prescription above, a dual CSS code also has a progenitor cluster state, i.e. the dual code can also be clusterized in the same way. If the code is self dual (e.g. the Steane code), then the corresponding primal and dual cluster states are identical.

II. FOLIATED CODES

In this section we review the general construction of foliated codes as an extension of Raussendorf’s 3D cluster state construction [3, 4, 25]. Section II A covers the generation of foliated codes from the cluster state resources in section II B and section II C outlines a heuristic decoding approach which is suitable for general CSS codes. Specific implementations for convolutional, turbo and bicycle codes, which are all examples of low-density parity check (LDPC) codes, are covered in later sections.

A. Foliated Code Construction

The general foliated construction consists of alternating sheets of primal and dual clusterised codes as defined
in section I which are ‘stacked’ together [20]. The stacking of code sheets simply amounts to the introduction of additional cluster bonds (i.e. c-phase gates) between corresponding code qubits on neighbouring sheets. This is depicted in Fig. 2 for a specific code example (the Steane code of fig. 1).

Suppose a CSS code has an X-like stabiliser, $\hat{s}$, with support on code qubits $c_{h,i}$, indicated by the support vector $\hat{h} = \{h_1, h_2, \ldots \}$, i.e. $\hat{s} = X_{c_{h,1}} \otimes X_{c_{h,2}} \ldots \otimes X_{c_{h,c}}$. In the foliated construction, there is a corresponding parity check operator centred on sheet $m$, given by

$$\hat{P}_{h,m} = X_{a_{h,m-1}} X_{a_{h,m}} X_{a_{h,m+1}}, \quad (2)$$

where $a_{h,m+1}$ is the ancilla qubit associated with the dual code stabiliser $Z_{c_{h,m+1}}$, acting on sheet $m \pm 1$.

Given that each code sheet is derived from an underlying CSS code with logical operators $X_L$ and $Z_L$, we can define corresponding logical operators within each code sheet, $X_{L,m}$ and $Z_{L,m}$. After the code is foliated, the logical operators in each code sheet commute with the parity check operators [20], i.e. $[\hat{P}_{h,m}, Z_{L,m}] = 0$.

A reason for considering this construction is the observation that the foliated code cluster state provides a resource for error tolerant entanglement sharing. This generalises one of the major insights of Raussendorf et al. [3], in which is was shown that after foliating $L$ surface code sheets, the resulting 3-dimensional cluster state (defined on a cubic lattice) served as a resource for fault-tolerantly creating an entangled Bell pair of surface code logical qubits between the first and last sheet (labelled by the index $m = 1$ and $L$). Starting from the 3-dimensional foliated surface code cluster, this long range entanglement is generated by measuring each of the bulk physical qubits (i.e. all ancilla qubits, and all code qubits in sheets $m = 2$ to $L - 1$) in the $X$ basis. Formally, this is shown by noting that after the bulk qubit measurements, the remaining physical qubits (which are confined to sheets 1 and $L$) are stabilised by the operators $X_{c_{1,1}} \otimes X_{L,1} \otimes Z_{L,1} \otimes Z_{c_{1,1}}$ up to Pauli frame corrections that depend on the specific measurement outcomes on the bulk qubits. The underlying surface code makes the protocol described therein robust against pauli errors on the bulk qubits.

As discussed in [20], this property is respected for any foliated CSS code. That is, measurements on the bulk qubits project the logical qubits encoded within the end sheets into an entangled logical state. This is verified by checking that $X_{L,1} \otimes X_{L,1} \otimes Z_{L,1} \otimes Z_{L,1}$ and $X_{c_{1,1}} \otimes X_{c_{1,1}} \otimes Z_{c_{1,1}} \otimes Z_{c_{1,1}}$ are in the stabiliser group of the cluster state after bulk measurements are completed.

For an underlying $[[n, k, d]]$ code, there are weight-$d$ undetectable error chains on the foliated cluster, as in [3–8 11]. Since the structure of the code in the direction of foliation is a simple repetition, it follows that the foliated cluster inherits the distance of the underlying code.

Fig. 2 shows an example of the cluster state for a foliated Steane code. Alternating sheets of the primal Steane code cluster state (shown in Fig. 1) and its self-dual are stacked together, with additional cluster bonds (green streaked lines) extending between corresponding code qubits in each sheet; operationally these correspond to c-phase gates between code qubits. The Steane code is self-dual, so that primal and dual cluster sheets are identical. In this example, the end faces are indexed by $m = 1$ and $m = L = 3$. Bulk qubits include all qubits in the $m = 2$ sheet, and the ancilla qubits in the end faces. The product of cluster stabilisers centred on the labelled qubits (connected by dashed red lines) produces the parity check operator $\hat{P}_{1,m=2}$ in eq. (3).

An example of a parity check operator centred on sheet $m = 2$ is

$$\hat{P}_{1,m=2} = X_{a_{1,1}} X_{c_{1,2}} X_{c_{2,2}} X_{c_{6,2}} X_{c_{7,2}} X_{a_{1,3}}, \quad (3)$$

is depicted in Fig. 2. For this example, there are two other parity check operators centred on sheet $m = 2$, associated to the other ancilla qubits therein. The logical operators for the Steane code pictured in fig. 1 are $Z_{L,1}^{\text{Steane}} = Z_{c_{1}} Z_{c_{2}} Z_{c_{3}}$ and $X_{L,1}^{\text{Steane}} = X_{c_{1}} X_{c_{2}} X_{c_{3}}$. By inspection, the parity check operator $\hat{P}_{1,m}$ commutes with $Z_{L,1}^{\text{Steane}}$ on sheet $m$.

Fig. 2 also illustrates a minimal example of entanglement sharing between the end sheets of the foliated construction, for $L = 3$ code sheets. After the foliated cluster state is formed, $X$ measurements on the bulk qubits (all qubits in the dual sheet shown, and the ancilla qubits in the end primal sheets) leaves the remaining physical qubits (the code qubits in the primal end sheets) stabilised by the operators $X_{L,1}^{\text{Steane}} \otimes X_{L,3}^{\text{Steane}}$ and $Z_{L,1}^{\text{Steane}} \otimes Z_{L,3}^{\text{Steane}}$.

Additional examples of the Shor code and surface code are presented in [20].

B. Decoding Approach

Errors in the foliated cluster are detected by parity check operators: a $Z$ error will flip one or more parity
FIG. 3: A schematic flow of beliefs (i.e. marginal probabilities) in the heuristic decoder for a generic foliated cluster code. Code qubits are shown as circles, ancilla qubits as rectangles within each code sheet. Cluster bonds are shown as black (green) lines within (between) code sheets (as in fig. 2). The decoding sheet consists of code qubits $c_{m}$ in sheet $m$, and the ancilla in the adjacent sheets, $a_{m±1}$. At the left, one stabiliser, $S_{j, m}$, with support on code qubits $c_{j, 1,...,4, m}$ in sheet $m$ is shown (shaded), and forms a parity check operator with the associated ancilla qubits $a_{j, m±1}$ (connected by dashed red lines). Step 1: on each decoding sheet $m$, the soft-input soft-output (SISO) decoder receives syndrome data, $\vec{S}_{m}$ and input error marginals $P_{m}(c_{m})$ on the code qubits (thick purple arrows) and adjacent ancilla qubits $P_{m±1}(a_{m})$ (thin orange arrows) to compute updated error marginals $P'_{m}$ for code and ancilla qubits. This step can be processed in parallel across all sheets, since there are no cross-dependencies. Note that after Step 1 there are two marginals associated to each adjacent ancilla: e.g. $P'_{m}(a_{j, m±1})$ (solid orange arrow emerging from the decoder) and $P'_{m±2}(a_{j, m±1})$ (dotted orange arrow emerging from the decoder), which are computed from decoders on sheets $m$ and $m±2$ respectively. If these disagree, $P'_{m}(a_{j, m±1}) \neq P'_{m±2}(a_{j, m±1})$, the value are exchanged in Step 2. After marginal exchange, the process is iterated (Step 3), until marginals converge. After marginals on the ancilla have converged sufficiently, maximum likelihood decoding is performed within each code sheet (Step 4, not shown).
may work well in many cases \cite{22,30}. The overall foliated decoder calls a Soft-Input Soft-Output\(^1\) (SISO) decoder for each of the underlying primal and dual CSS codes, as a subroutine. The SISO decoder calculates the probability of a Pauli error, \(\sigma\), on qubit \(q \in \{a_k,c_j\}\), \(P(\sigma_q|\vec{S}_{\text{CSS}})\), given a physical error model and syndrome data for the CSS code, \(\vec{S}_{\text{CSS}}\), which may itself be unreliable, due to errors on the ancilla qubits. Using such a decoder it is possible to assign a probability of failure to a parity check to account for errors on ancilla qubits, \(P(\sigma_q) = \sum_{\vec{S}_{\text{CSS}}} P(\sigma_q|\vec{S}_{\text{CSS}})P(\vec{S}_{\text{CSS}})\).

For the foliated case, consider a parity check operator given by eq. (2). A non-trivial syndrome can arise because errors on the corresponding ancilla qubits, \(a_g\) in adjacent dual sheets \(m \pm 1\).

In the case where dual-sheet ancilla qubits are error-free the decoding problem reduces to a series of independent CSS decoders using perfect syndrome extraction. However, errors on the ancillas mean that the in-sheet syndrome is unreliable. To account for the dual-sheet ancilla errors we embed the CSS decoder in a belief propagation (BP) routine, as iterated in the following scheme. Steps 1, 2 and 3 are illustrated in fig. 3.

0. A decoding sheet centred on sheet \(m\) is defined by a set of qubits, \(q\), which contain the code qubits \(c_{j,m}\) within the sheet and the ancilla qubits \(a_{k,m \pm 1}\) on neighbouring sheets. The input to the decoder is the syndrome data \(\vec{S}_m\) on sheet \(m\), and a prior probability distribution, \(P_0^m(a_{k,m \pm 1})\), for the marginals on the ancilla qubits associated to the decoding sheet \(m\).

1. The SISO decoder on sheet \(m\) computes marginal error probabilities for the code qubits in the sheet, \(P_1^m(c_{j,m})\), assuming marginals on the ancilla in sheet \(m \pm 1\), and updated marginals on the ancilla qubits \(P_1^m(a_{k,m})\). This step can be parallelised across all sheets.

2. The assumed error model for ancilla qubits is updated using the results from step 1. \(P_1^m(a_{k,m \pm 1}) \rightarrow P'_m(a_{k,m \pm 1}) = P_{m \pm 2}(a_{k,m \pm 1}|\vec{S}_{m \pm 2})\), where \(P_{m \pm 2}(a_{k,m \pm 1})\) is the probability distribution found from a neighbouring decoding process. We refer to this update rule as an exchange of marginals.

3. Using the updated ancilla marginals from step 2, we iterate back to Step 1 until ancilla marginals converge sufficiently, i.e. \(P_1^m(a_{k,m \pm 1}) \approx P'_m(a_{k,m \pm 1})\), in which case we proceed to Step 4.

4. Use the converged marginals computed on the code and ancilla qubits, \(P_m(c_{j,m})\) and \(P_m(a_{k,m \pm 1})\), to calculate an error correction chain (parametrised by error correction binary support vector, \(\vec{c}\)) with (approximately) maximum likelihood \(P_m(\vec{c} | \vec{S}_m, P_m(c_{j,m}), P_m(a_{k,m \pm 1}))\), within each code sheet. This could employ a hard decoder.

In sections \ref{sec:conventional} \textendash \ref{sec:ldpc_bicycles} we describe specific soft decoding implementations for foliated convolutional, turbo, and LDPC bicycle codes. For convolutional and turbo codes we first introduce a trellis decoding framework, which is then adapted for use as a single layer decoder. This is then combined with the BP process above to generate a full decoding scheme. For bicycle codes we use belief propagation directly on the Tanner graph representation of the foliated code.

\section{Convolutional Trellis Construction}

In this section we review the construction of trellises as a tool for SISO decoding of convolutional codes \cite{22,23,31,32}. Section \ref{sec:construction} modifies this construction for use with single sheets of the foliated code, which is itself a subroutine in the full decoding of foliated convolutional codes.

Generators and stabilisers of convolutional codes are translations of some ‘seed’ generators or stabilisers which act over a sequence of frames. Each frame labels a contiguous block of \(n\) qubits. We assume here that the code has \(\tau\) frames, labelled by a frame index, \(t = 1, \ldots, \tau\). For a classical rate \(\frac{k}{n}\) code the generator matrix (for logical \(Z\) operators) has the form:

\[
\text{frame: } \quad \ldots \quad t-1 \quad t \quad t+1 \quad t+2 \quad \ldots
\]

\[
G^T = \begin{bmatrix}
G^{(0)} & G^{(2)} & \cdots & G^{(\nu_g)} & \cdots \\
\ldots & G^{(0)} & G^{(2)} & \cdots & G^{(\nu_g)} & \cdots \\
\ldots & \ldots & \ldots & \ldots & \ldots & \ldots
\end{bmatrix}_{k_T \times n_T}
\]

where \(G^{(i)}\) are binary-valued \(k \times n\) sub-matrices. We use the notation \(A^T\) to denote the transpose of the matrix \(A\); bold face matrices indicate generators acting on the entire set of physical qubits. All other elements in \(G\) are zero. Each \(G^{(i)}\) acts on a single frame of \(n\) physical bits, encoding \(k\) logical bits. The code is built from translations of the sub-matrices \([G^{(1)}, \ldots, G^{(\nu_g)}]\). Each component, \(G^{(j)}\), acts on a single frame of the code. The value of \(\nu_g\) is the codeword memory length, which counts the number of frames over which parity check operators have support.

Later we will discuss a specific example of a convolutional code that illustrates this construction. For
concreteness, we preempt that example by reference to fig. 7b, each row of which depicts a convolutional code with frames consisting of $n = 3$ qubits (blue circles), and with parity check operators (red squares) extending over $\nu_g = 3$ frames. Note that in this example, each parity check operator has support on 6 of the qubits (indicated by thick black lines) within the $\nu_g = 3$ contiguous frames.

Similarly, we define the parity check generator matrix

$$
H = \begin{bmatrix}
\ldots & H^{(1)} & \ldots & H^{(\nu_h)} & \ldots \\
\ldots & H^{(\nu_h)} & \ldots & H^{(1)} & \ldots \\
\ldots & H^{(1)} & \ldots & H^{(\nu_h)} & \ldots \\
\end{bmatrix}_{n_x \times n \tau},
$$

(5)

where $n_z = |S_Z|/\tau$ is the number of Z-like stabilisers per frame. The value of $\nu_h$ is the parity check memory length. Typically $\nu_h$ and $\nu_g$ are of similar size.

Codeword generators are expressed in the form $\hat{g} = Z^{\otimes \hat{g}}$, where $\hat{g} \in Z_{2^n}^\nu$ is in the row space of $G_Z$. We have introduced the notation $Z^{\otimes \hat{g}} = Z^{\nu_1} \otimes Z^{\nu_2} \ldots$ with $\nu_j \in Z_2$. Similarly the stabiliser generators formed from $H_Z$ are $\hat{h} = Z^{\otimes \hat{h}}$, where $\hat{h} \in Z_{2^n}^\tau$ is in the row space of $H_Z$.

The commutation relationships between generator and stabiliser matrices manifests as orthogonality conditions, i.e.

$$
\begin{bmatrix}
G_{Z}^T \\
H_{Z} \\
ISF_{Z}
\end{bmatrix}
\begin{bmatrix}
G_X \\
ISF_X^T \\
H_X^T
\end{bmatrix} = I_{n_x \times n \tau},
$$

(6)

where we have introduced the *Inverse Syndrome Formers* (ISFs). The ISF is useful determining an initial, valid decoding pattern, known as a *pure error*. Each row of $ISF_Z$ corresponds to an operator that commutes with all $X$-like stabiliser generators, $\hat{g} \in RowSpace(G_X)$, and with all but one of the parity check generators $\in RowSpace(H_X^T)$; there are as many ISF generators as there are parity check generators. The rows of $G_Z^T$ and $H_Z$ form an orthonormal set, but do not fully span $Z_{2^n}^\nu$. The $ISF_Z$ sub-matrix can be computed by finding an orthonormal completion of the rowspace of $G_Z^T$ and $H_Z$, e.g. using Gram-Schmidt. The matrix $ISF_Z$ is not unique: any orthogonal completion of the basis that satisfies $ISF_Z \cdot H_X^T = I$ will suffice. Similarly $ISF_X^T$ is generated from an orthonormal completion of the column space of $G_X$ and $H_X^T$.

Suppose some (unknown) pattern $\vec{e} \in Z_{2^n}^\tau$ of $Z$ errors gives rise to a syndrome that is revealed by the $X$-like parity checks, i.e. $\vec{S} = H_X \vec{e} \in Z_{2^n}^\nu$. We use $ISF_Z$ to generate a pure error, $\vec{E} = Z^{\otimes \vec{e}} \equiv Z^{\otimes (ISF_Z^T \cdot \vec{S})}$, based only on the syndrome data. The binary-valued support vector $\vec{e} \in Z_{2^n}^\tau$ corresponds to a possible error correction pattern that satisfies the syndrome $\vec{S}$, i.e. $\vec{S} = H_X \vec{e} \vec{e} = H_X \vec{e} \in Z_{2^n}^\nu$. Therefore $\vec{E}$ is a *valid* decoding pattern, but it is unlikely to be the most probable decoding, and is so unlikely to robustly correct the original error. However it defines a reference decoding from which the set of all valid decodings, $E$, can be enumerated through

$$
E = \{ \vec{E}_0 \hat{h} | \hat{h} \in S_Z, \vec{y} \text{ is a code word} \},
$$

$$
= \{ Z^{\otimes (\vec{e}^0 + \hat{h} \vec{y})} | \vec{y} \in RowSpace(H_Z), \vec{y} \in RowSpace(G_Z^T) \},
$$

(7)

where we take linear combinations of rows of $H_Z$ and $G_Z^T$ over $Z_2$. In what follows, we suppress the subscripts $X$ and $Z$.

A valid decoding of the syndrome data may be written as $\vec{E}_0 Z^{\otimes \vec{p}}$, where $\vec{p} \equiv \hat{h} + \vec{y} \in RowSpace(H) \cup RowSpace(G^T) \subset Z_{2^n}^\tau$. That is we define

$$
\vec{p} = \sum_{i=1}^{k} \sum_{j=1}^{\tau} \vec{p}_{ij} G_{i+j} + \sum_{i=1}^{n} \sum_{j=1}^{\tau} \vec{p}_{ij} H_{i+j},
$$

(8)

where $A_i$ refers to the $i^{th}$ row of $A$.

Relative to the (easily found) pure error support vector, $\vec{e}^0$, the vector $\vec{p}$ parametrises all possible valid decoding through the coefficients $\vec{p}_{ij}^g$ and $\vec{p}_{ij}^h$. A good decoder will return optimal values of $\vec{p}_{ij}^g$ and $\vec{p}_{ij}^h$, corresponding to an element, $\vec{E} = Z^{\vec{p}+\vec{e}^0} \in E$, that has a high likelihood of correcting the original error. For low error rates, this amounts finding a $\vec{p}$ that minimises the Hamming weight of $\vec{p} + \vec{e}^0$.

For readers unfamiliar with this general construction, we provide a short example based on the 7-qubit Steane code in Appendix A.

### A. Seed Generators of Convolutional Codes

Finding an optimal $\vec{p}$ by enumerating over all $2^{(k+n_z)\tau}$ possible binary values of $\vec{p}_{ij}^g$ and $\vec{p}_{ij}^h$ becomes computationally intractable as the size of the code is increased. However the repeated structure of convolutional codes, in blocks of length $\nu$, allows for a simplification, using a decoding trellis. The trellis decoder reduces the search space to $\sim 2^{k+\nu+n_z+v_h}$, so that the decoding is linear in the code size $\tau$, albeit with potentially large prefactor depending on the total memory lengths $\nu_g$ and $\nu_h$.

As the number of encoded bits increases, the size of the generator matrix increases. We therefore use a more compact representation using transfer functions, which are polynomials in the *delay* operator, denoted by $D$. We interpret $D$ as a discrete generator of frame shifts, so that $D^n$ represents a ‘delay’ of $n$ frames. We also define the inverse shift generator, $\bar{D}$, which acts like a reverse translation such that $D^n \bar{D}^n = \delta_{ab}$. Detail of the construction are given in Appendix B.

Using this delay notation, we define the *seed matrix* of a rate $\frac{k}{n}$ convolutional code in terms of the sub-matrices, $\frac{2^{n_z}}{\bar{D}}$.
\[ G^{(i)} = \sum_{q=1}^{\nu_g} D^{q-1}G^{(q)}_{ij} \]

The utility of the delay operator notation is that (1) it enables us to write \( G^\top(D) \) in terms of a matrix that is independent of the number of frames, \( \tau \), and (2) it sets the degree of the polynomial entries.

Similarly
\[ H(D) = H^{(1)} + DH^{(2)} + \ldots + D^{\nu_h-1}H^{(\nu_h)} = \begin{bmatrix} h_{11}(D) & h_{12}(D) & \ldots & h_{1n}(D) \\ \vdots & \vdots & \ddots & \vdots \\ h_{n1}(D) & h_{n2}(D) & \ldots & h_{nn}(D) \end{bmatrix} \]

where \( h_{ij}(D) = \sum_{q=1}^{\nu_h} D^{q-1}H^{(q)}_{ij} \).

In this delay notation, eq. (6) becomes
\[ D^u \begin{bmatrix} G^\top_Z(D) \\ H_Z(D) \\ ISF_Z(D) \end{bmatrix} \equiv \begin{bmatrix} G_X(D) & ISF_X(D) & H^\top_X(D) \end{bmatrix} \bar{\nu}^u = \bar{z}_{n \times n} \delta_{uv} \]
FIG. 6: The decoding trellis for the convolutional code ineq. (14). Each transition corresponds to a different product of \( \hat{g} \) and \( \hat{h} \) at a given time-step. The memory state records the values of previous transitions. Each path corresponds to a certain product of \( \hat{g} \) and \( \hat{h} \) terms. The blue and red paths are the most likely paths for the syndromes in eq. (15) and eq. (16) respectively.

circuit diagram relating \( \vec{p}_i \) to \( \vec{l}_t \), for the self-dual rate convolutional code defined by

\[
G_Z(D) = \begin{bmatrix} \bar{D} & D & 1 \end{bmatrix},
\]

\[
H_Z(D) = \begin{bmatrix} 1 + D + D^2 & 1 + D^2 & 1 \end{bmatrix}, \quad (14)
\]

\[
\text{ISF}_Z(D) = \begin{bmatrix} D & D & 0 \end{bmatrix}.
\]

Reading off the highest powers of \( D \) in \( G_Z(D) \) and \( H_Z(D) \) respectively, we see that \( v_\bar{g} = 1 \) and \( v_h = 2 \), so that the memory state \( \alpha_t = \{i_t^{(g)}; i_t^{(h)}; j_t^{(h)}\} \) is a list with three entries. In the equivalent circuit depicted in Fig. 4, the memory is represented by black boxes.

Fig. 5 illustrates allowed transitions from state \( \alpha_t = \{1; 01\} \) to a new state \( \alpha_{t+1} \) at frame \( t \) for the convolutional code defined in eq. (14). Given that this code example is sufficiently simple, we show all possible choices for \( \vec{l}_t \) for this transition, i.e. \( \vec{l}_t = \{0; 0\} \), \( \vec{l}_t = \{0; 1\} \), \( \vec{l}_t = \{1; 0\} \) and \( \vec{l}_t = \{1; 1\} \). We also show the corresponding code-word block frame \( \vec{p}_t \), respectively (000), (111), (001) and (110) for each possible choice of \( \vec{l}_t \). Expanding Fig. 5 over several frames yields the trellis, shown in Fig. 6.

With the foregoing machinery in place, error correction works as follows. A set of errors, \( \vec{e} \), produces a syndrome \( \vec{S} \). From \( \vec{S} \) we use the ISF to calculate \( \vec{e}^0 \). Any path, \( \vec{p} = (\vec{p}_1, \vec{p}_2, ..., \vec{p}_t, ..., \vec{p}_T) \), through the trellis that begins in the trivial state \( \vec{p}_1 = (0; 0; 0; 0; 0; 0) \) and ends in the state \( \vec{p}_T = (0; 0; 0; 0; 0; 0) \) state corresponds to a valid decoding. The path \( \vec{p}_{\min} \) which minimises the Hamming weight of \( \vec{p}_{\min} + \vec{e}^0 \) is the most likely decoding solution. If no detectable errors occur so that \( \vec{e}^0 \) is trivial (i.e. if \( \vec{e}^0 = \{00...\} \), then the lowest weight path through the trellis has \( \vec{p}_{\min} = (000...) \), and \( \text{wt}(\vec{e}^0 + \vec{p}_{\min}) = 0 \). For nontrivial \( \vec{e}^0 \), finding this path is the core of the decoding problem.

Given a trellis, such as in Fig. 6, and priors for the qubit error marginals, a trellis decoding algorithm will return a suitable choice for \( \vec{p}_{\min} \). Several algorithms exist to identify (optimally or suboptimally) the most likely trellis path. These include the Viterbi algorithm [33] the Bahl, Cocke, Jelinek, Raviv (BCJR) algorithm [34] and the Benedetto algorithm [35].

The last of these algorithms is a maximum a posteriori (MAP) SISO decoder which is capable of dealing with parallel transitions, i.e. those for which several transitions between states \( \alpha_t \) and \( \alpha_{t+1} \) exist. Single layer foliated codes necessarily have parallel transitions in their trellis descriptions, as will be shown in section [V]. For this reason we will use the Benedetto algorithm as a SISO decoder for convolutional codes. This is discussed in detail in Section [V].

Examples

Before concluding this section, we give two illustrative examples using the trellis shown in Fig. 6. Because the examples are sufficiently simple, we find the lowest weight path through the trellis by inspection. In practice we use a modified version of Benedetto’s trellis algorithm [35] to determine the most likely path through the trellis (see [V]).

In the first example the error pattern, \( \vec{e} \), is a single error occurring on the first bit in frame \( t + 1 \), as indicated in the following equation:

\[
\text{frame:} \quad \ldots \ t - 2 \quad t - 1 \quad t \quad t + 1 \quad t + 2 \quad \ldots \\
\vec{e} = \ldots \quad 000 \quad 000 \quad 000 \quad 000 \quad \ldots \\
\vec{S} = \ldots \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad \ldots \\
\vec{e}^0 = \ldots \quad 000 \quad 000 \quad 110 \quad 110 \quad 110 \quad \ldots \quad (15) \\
\vec{p}_B = \ldots \quad 000 \quad 000 \quad 110 \quad 010 \quad 110 \quad \ldots \\
\vec{e}_{\min} = \vec{e}^0 + \vec{p}_B \\
\quad = \ldots \quad 000 \quad 000 \quad 000 \quad 100 \quad 000 \quad \ldots
\]

Given, \( \vec{e} \) we find \( \vec{S} \) and \( \vec{e}^0 \), also shown in eq. (15) (for this and the following example, we work through the calculation of \( \vec{S} \) and \( \vec{e}^0 \) from \( \vec{e} \) in Appendix D). Using \( \vec{e}^0 \), we find (by inspection for this example) the path \( \vec{p}_B \) through the trellis that minimises \( \text{wt}(\vec{e}^0 + \vec{p}_B) \equiv \text{wt}(\vec{e}_{\min}) = 1 \). This path is shown in blue in Fig. 6, the blue highlighted triples of binary values in the figure correspond, frame-by-frame, the triples in \( \vec{p}_B \). This path determines the optimal error correction procedure, which is to apply the correction operator \( Z^{\vec{e}_{\min}} \). In this example, \( \vec{e}_{\min} = \vec{e} \), so the correction would succeed since \( \vec{e}_{\min} \) and \( \vec{e} \) are logically equivalent, i.e. the correction \( \vec{e}_{\min} \) would return the system to the correct codeword.
In the second example the error pattern, \( \tilde{e} \), consists of two errors occurring on the first and second qubits in frame \( t + 1 \). The most likely path through the trellis, \( \tilde{p}_R \), is shown in red in Fig. 6. We have

\[
\begin{align*}
\text{frame: } \ldots & \text{ } t-2 \text{ } t-1 \text{ } t \text{ } t+1 \text{ } t+2 \ldots \\
\tilde{e} & = \ldots 000 \text{ } 000 \text{ } 110 \text{ } 000 \text{ } \ldots \\
\tilde{S} & = \ldots 0 \text{ } 0 \text{ } 1 \text{ } 0 \text{ } 0 \text{ } \ldots \\
\tilde{e}^0 & = \ldots 000 \text{ } 000 \text{ } 001 \text{ } 110 \text{ } 000 \text{ } \ldots \\
\tilde{p}_R & = \ldots 000 \text{ } 000 \text{ } 001 \text{ } 110 \text{ } 000 \text{ } \ldots \\
\tilde{e}_\text{min} & = \tilde{e}^0 + \tilde{p}_R \\
& = \ldots 000 \text{ } 000 \text{ } 001 \text{ } 000 \text{ } 000 \text{ } \ldots 
\end{align*}
\]

Again, the red highlighted triples of binary values in Fig. 6 correspond, frame-by-frame, the triples in \( \tilde{p}_R \) in eq. (16). The lowest weight recovery operation is a single error on the third qubit in frame \( t \). In this example, the product of the recovery operation and physical errors is \( \tilde{e}_\text{min} + \tilde{e} = \ldots 001 \text{ } 1100 \ldots \) is a nontrivial codeword of \( G^t_Z \) in eq. (14). That is the decoder fails in this example.

This second example is illustrative; because this is \( d = 3 \) code, adjacent errors are not expected to be corrected. However, if errors are sufficiently far apart (determined by the code memory length), then they behave as if they were independent, and so the convolutional code can decode many more errors than \( d/2 \) if they are sparsely distributed.

IV. FOLIATED CONVOLUTIONAL DECODING

In this section we build on the trellis construction in section III to operate on independent sheets of a foliated convolutional code, accounting for additional ancilla. Decoding can be performed using the algorithm in section III to operate on independent sheets of a foliated convolutional code; a specific parity check operator, \( \tilde{S} \), is indicated by the labelled vertices \( a_1, c_1, \ldots, c_6 \) and \( a_2 \), where the ancilla qubits are on the adjacent code sheets \( m \pm 1 \). Measuring all such operators associated to code sheet \( m \) yields the syndrome \( \tilde{S}_m \) for that sheet.

For the purposes of decoding, we introduce decoding sheets. A decoding sheet \( m \) is identified with the corresponding code sheet \( m \); it refers to the same code qubits, but includes virtual ancilla qubits associated to the neighbouring sheets.

The shaded ellipse in Fig. 7 illustrates the formation of a single decoding sheet for a foliated convolutional code; this decoding sheet, \( m \), is identified with the corresponding code sheet, \( m \), in Fig. 7. Independent decoding of each decoding sheet can be performed by creating a virtual code associated with a single sheet of the foliated structure. This code accounts for the ancillary qubits \( a_1 \) and \( a_2 \) in adjacent sheets by introducing virtual ancilla \( a''_1 \) and \( a''_2 \). There are \( 2n_x \) virtual ancilla associated to the ancilla on the adjacent code sheets, so the virtual code has rate \( \frac{k}{n+2n_x} \).

We note that a physical ancilla qubit \( a_{j,m} \) is represented virtually in two different decoding sheets \( m \pm 1 \) (as \( a'_{j,m} \) and \( a''_{j,m} \)). This yields a consistency condition on ancilla marginals between neighbouring sheets, which we discuss later.

Within a decoding sheet (see Fig. 7), we begin by find-
ing an initial recovery operation, $\hat{E}^0$, which satisfies a received syndrome, $\hat{S}$. As with the previous section this can be achieved by using ISFs. Setting the final $2n_x$ qubits in a frame to correspond to virtual ancilla qubits, the seed generator is given by

$$\mathbf{G}^T (D) = \left[ \begin{array}{cc} G^T (D) & 0_{k \times n_x} \\ 0_{n_x \times n_x} & 0_{k \times n_x} \end{array} \right],$$

where $G^T (D)$ refers to the generator matrix of the base convolutional code, as exemplified in eq. (14). Similarly the seed generator matrix and ISF $\mathbf{P}$ are given by

$$\mathbf{P} = \mathbf{H}_X (D), \quad \mathbf{J} \mathbf{S} = \mathbf{P} (D), \quad \mathbf{J} = \mathbf{T} (D).$$

Additional pairs of gauge operators are generated from the degrees of freedom introduced by the extra ancilla qubits. The seed generators, stabilisers, ISFs and gauges $\mathbf{J}_Z$ and $\mathbf{J}_X$ satisfy the orthogonality relations

$$\mathbf{D}^T \left[ \begin{array}{c} G^T (D) \\ \mathbf{H}_Z (D) \\ \mathbf{J} \mathbf{S}_X (D) \\ \mathbf{J} (D) \end{array} \right] = \mathbf{I} \delta_{\mathbf{0}_n},$$

which implicitly defines $\mathbf{J}_X$. A valid choice of $\mathbf{J}_X$ is

$$\mathbf{J}_X = \left[ \begin{array}{cc} 0_{n_x \times n} & 0_{n_x \times n} \\ 0_{n_x \times n} & 0_{n_x \times n} \end{array} \right],$$

which commutes with $\mathbf{G}_Z$, $\mathbf{H}_Z$ and ISF $\mathbf{J}_X$.

Generally, $\mathbf{J}_Z$ depends on the details of the code. Each $\mathbf{J} \mathbf{S}_X (D)$ is a set of operators which commute with $\mathbf{G}_X$, $\mathbf{P}_X$ and ISF $\mathbf{J}_X$, but anti-commute with a single $X^{\otimes a} \in \mathcal{J}_X$.

As an example, foliating the code in eq. (14), we have

$$\mathbf{J}_Z (D) = \left[ \begin{array}{ccc} 1 + D & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{array} \right].$$

Because $\mathbf{J}_Z$ commutes with the stabilisers, they correspond to undetectable error patterns within the sheet. As an aside, using the Raussendorf lattice as an example, these would correspond to error chains that pass through a sheet in the ‘time’-like direction of foliation: they leave no syndrome data within the sheet (but would be detected by other sheets in the 3D lattice).

The set of valid recovery operations is given by

$$\mathcal{E} = \{ Z^{x_0 + \hat{h} + \hat{g} + \hat{j}} | \hat{h} \in \text{RowSpace} (\mathbf{H}_Z), \hat{g} \in \text{RowSpace} (\mathbf{G}_Z), \hat{j} \in \text{RowSpace} (\mathbf{J}_Z) \},$$

where

$$\mathbf{J}_Z = \left[ \begin{array}{cccc} J^{(1)} & \ldots & J^{(v_x)} & \ldots \\ \ldots & J^{(1)} & \ldots & J^{(v_x)} & \ldots \\ \ldots & \ldots & J^{(1)} & \ldots & J^{(v_x)} \end{array} \right]_{2n \times n}$$

by analogy with eq. (4). In the example in eq. (20),

$$J^{(1)} = \left[ \begin{array}{cc} 1 & 1 \\ 0 & 0 \end{array} \right], \quad J^{(2)} = \left[ \begin{array}{cc} 0 & 1 \\ 0 & 0 \end{array} \right].$$

Given $\mathbf{H}_Z$, $\mathbf{G}^T (D)$ and $\mathbf{J}_Z$, a trellis may be constructed, closely mirroring the trellis construction in section III. The main difference is that the decoding trellis must be modified to account for $\mathbf{J}_Z$ terms, which represent the virtual ancilla in each sheet. This give rise to larger memories and more allowed transitions. The $\mathbf{J}_Z$ terms are modified to include $\mathbf{J}^{(J)}$ components.

Once the trellis is formed, a soft-input hard-output decoder will take as input marginal error probabilities on the qubits, and return $\hat{h}$, $\hat{g}$ and $\hat{j}$, which correspond to a specific, maximum likelihood error pattern consistent with the syndrome data on sheet $m$. More suited to our goal is a soft-input soft-output (SISO) decoder which returns updated a posteriori probabilities for qubit error marginals. We discuss such a decoder in Section V.

B. Ancilla Marginal Exchange for Consistency Between Decoding Sheets

When decoding is performed on each convolutional code sheet, a given ancilla qubit, $a_{k,m}$, is associated to corresponding virtual ancilla, $a'_{k,m}$ and $a''_{k,m}$, from each of the neighbouring decoding sheets $m + 1$ and $m - 1$. For consistency, we require the SISO marginal a posteriori error probabilities on $a'_{k,m}$ and $a''_{k,m}$ to match, i.e.

$$P(a'_{k,m}) = P(a''_{k,m}).$$

Independent SISO decoding of adjacent decoding sheets does not automatically respect this constraint, so we perform sequential rounds of independent decoding, and iteratively exchange marginals, $P(a'_{k,m}) \leftrightarrow P(a''_{k,m})$ between each round, until the marginals are satisfactorily converged.

This approach is similar to belief propagation on a modified Tanner graph, with a message passing schedule which gives priority to intra-sheet messages over that of inter-sheet messages [26, 36–38]. It is unlikely that the process we have proposed is optimal, but good numerical decoding results are still possible as shown in section VI for turbo codes.

In practical settings it may be desirable to terminate the iterative message passing after a fixed number of rounds. In this case, the marginals on the ancilla from adjacent sheet decoders may not have converged. One heuristic resolution to this is to average the inconsistent marginals on each ancilla, $P(a_{k,m}) := (P(a'_{k,m}) + P(a''_{k,m}))/2$, and then make a hard decoding choice by choosing the most likely error configuration at each ancilla. A final decoding round is performed on each sheet. This ensures that the decoding falls within the codespace.
V. TRELlis SISO Algorithm

So far, we have concentrated on constructing trellises for convolutional codes. As already mentioned, we then use the trellis to find optimal error configurations (i.e., a hard decoding) or assign error marginals (i.e., soft decoding) that are consistent with the syndrome data.

In this section we present a modified MAP SISO trellis decoder, following the approach of Benedetto et al. [35] on classical convolutional codes. The modified algorithm compiles codewords and stabilisers against a trial error pattern, \( \bar{e}_t \), determined from the syndrome data, and calculates marginal error probabilities consistent with it.

The algorithm consists of three stages: (1) a forward pass, which passes through the trellis starting from the first frame, (2) a backward pass, which passes through the trellis starting from the last frame to right to left, and (3) a local update which determines marginals using information from forward and backward passes.

In the rest of this section, we detail the core of the SISO algorithm. As in prior sections, the trellis memory states at frame \( t \) are \( \alpha_t \). The initial error pattern \( \bar{e}_0 \) is calculated from the syndrome, \( \bar{S} \), and the ISF. The physical operations associated with a transition through the trellis are denoted \( \bar{p}_t \).

A. Forward Pass Algorithm

The forward pass algorithm assigns a probability for each memory state, \( A(\alpha_t|\bar{S}_{t\leq t}) \), as we pass through the trellis starting from the first frame, \( t = 1 \), (i.e., the leftmost frame in the presentation of Fig. 6) and traversing to \( t = \tau \) (right). Here \( \bar{S}_{t\leq t} \) is a shorthand notation which indicates that only syndrome information up to frame \( t \) is used to calculate likelihoods. At each frame, there are \( 2^{(k\nu_f+n\nu_l)} \) memory states to store. Note that for decoding sheets in the foliated construction a memory term \( \nu_f \) must be incorporated for the \( J_2 \) guages.

We assign initial probabilities \( A(\alpha_1 = \bar{0}) = 1 \) and \( A(\alpha_1 \neq \bar{0}) = 0 \) to memory states at frame \( t = 1 \). The forward pass algorithm then computes probabilities for subsequent memory states as

\[
A(\alpha_{t+1}|\bar{S}_{t\leq t}) = \sum_{\bar{l}_t} A(\alpha_t|\bar{S}_{t\leq t-1}) \Pr(\bar{p}_t + \bar{e}_t^0) \Pr(\bar{l}_t).
\]

(23)

where \( \bar{p}_t = U_p(\alpha_t, \bar{l}_t) \) is given by eq. (12), and we recall from eq. (13) that \( \alpha_t \) on the RHS of eq. (23) is determined by \( \alpha_{t+1} \) and \( \bar{l}_t \). \( \Pr(\bar{p}_t + \bar{e}_t^0) \) is the a priori probability of the error pattern \( \bar{p}_t + \bar{e}_t^0 \), which depends on the details of the prior error mode; for our purposes this is just an i.i.d. error process on each of the physical qubits, so that \( \Pr(\bar{p}_t + \bar{e}_t^0) \) is given by the binomial formula. \( \Pr(\bar{l}_t) \) is the a priori probability of undetectable error processes (generated by eq. (8)). For convolutional codes, we take \( \Pr(\bar{l}_t) \) to be a constant (so that it factors out of \( A \)), however it will become important when we consider turbo codes, in which physical errors in the inner code affect logical priors in the outer code.

B. Backward Pass Algorithm

The backward pass algorithm is identical to the forward pass, but working now from the last frame, \( t = \tau \) back to the first, \( t = 1 \). The update rule is given similarly,

\[
B(\alpha_t|\bar{S}_{t\geq t+1}) = \sum_{\bar{l}_t} B(\alpha_{t+1}|\bar{S}_{t\geq t+1}) \Pr(\bar{p}_t + \bar{e}_t^0) \Pr(\bar{l}_t).
\]

(24)

The initial conditions are set as \( B(\alpha_t = \bar{0}) = 1 \) and \( B(\alpha_t \neq \bar{0}) = 0 \), where \( \tau \) is the final frame.

C. Local Update Algorithm

The local update calculates the likelihoods of physical error patterns, \( \bar{e}_{p,t} = \bar{p}_t + \bar{e}_t^0 \in \mathbb{Z}_2^n \), at frame \( t \), given a valid error configuration \( \bar{e}_t^0 \) (which is itself derived directly from the syndrome data). That is, we calculate the marginals \( P(\bar{e}_{p,t}|\bar{S}) \), and the marginals over logical bits \( P(\bar{e}_{l,t}|\bar{S}) \), where \( \bar{e}_l \in \mathbb{Z}_2^{k+n_2} \) is a specification of the logical states at frame \( t \). These marginals depend in turn on the marginal beliefs of memory states computed in the forward, \( A \), and backward, \( B \), pass algorithms. As inputs the decoder uses an initial prior distribution, \( \Pr(\bar{e}_{p,t}) \), for the error configuration \( \bar{e}_{p,t} \), and the logical error patterns, \( \Pr(\bar{l} = \bar{e}_{l,t}) \), as well as the syndrome, and computes the marginals

\[
P(\bar{e}_{p,t}|\bar{e}_t^0) = N_p \sum_{\bar{l}_t, \alpha_t} A(\alpha_t) B(\alpha_{t+1}) \Pr(\bar{p}_t + \bar{e}_t^0) \Pr(\bar{l}_t),
\]

(25)

\[
P(\bar{e}_{l,t}|\bar{e}_t^0) = N_l \sum_{\bar{p}_t = U_p(\alpha_t, \bar{e}_l,t)} A(\alpha_t) B(\alpha_{t+1}) \Pr(\bar{p}_t + \bar{e}_t^0) \Pr(\bar{l}_t),
\]

(26)

where \( N_p \) and \( N_l \) are normalization constants chosen to ensure that \( \sum_{\bar{e}_{p,t}} P(\bar{e}_{p,t}|\bar{e}_t^0) = 1 \) and \( \sum_{\bar{e}_{l,t}} P(\bar{e}_{l,t}|\bar{e}_t^0) = 1 \). Again, we recall from eq. (13) that \( \alpha_t \) on the RHS of eq. (23) is determined by \( \alpha_{t+1} \) and \( \bar{l}_t \).

Equations 23 to 26 are the ingredients for the sum-product belief propagation algorithm: the Forward and Backward Pass algorithms each run independently, and then the Local Update calculates marginals for each of the \( 2^n \) possible error configuration over each of the \( \tau \) frames. Storing this information requires memory \( \sim \tau 2^n \).
VI. FOLIATED TURBO CODES

Our main motivation for studying turbo codes is to demonstrate the foliated construction and BP decoder in an extensible, finite-rate code family. Practically, these and other finite-rate codes may have applications in fault-tolerant quantum repeaters networks [39][42], where local nodes create optimal clusterised codes to reduce resource overheads or error tolerance [43], however we do not address these applications here.

A. Turbo Code Construction

A turbo codes is essentially a concatenation of two convolutional codes, albeit with an interleaver between them. When convolutional codes fail, they tend to produce bursts of errors on logical bits. Turbo codes address this by concatenating encoded (qu)bits from the inner convolutional code into widely separated logical (qu)bits in the outer code. The interleaver is simply a permutation, \( \Pi \), on the inner logical qubits, and serves to transform a local burst of errors from the inner decoder into widely dispersed (and thus approximately independent) errors that the outer decoder is likely to correct.

For the numerical results we present in this section, we choose \( \Pi \) to be a completely random permutation on the inner code. This is a conventional choice for benchmarking turbo codes, however a completely random permutation leads to highly delocalised encodings. This may be undesirable in the quantum setting, and the optimal choice of interleavers was discussed in [44][45]. In the context of constructing clusterised codes, the interleaver choice will affect the weight of stabilisers. In section VII we return to this issue, and show that by choosing a structured interleaver, we reduce the weight of stabilisers substantially. This reduces the weight of correlated errors that build up during the systematic construction of the cluster state resource.

Turbo codes are generated using underlying convolutional codes. We use two different convolutional codes, one with \( d = 3 \), which we refer to as the C3 family of codes, and one with \( d = 5 \), which we refer to as the C5 family [46]. When embedded as clustered codes the distance of these codes are reduced to 2 and 3 respectively. This represents the effective code distance, \( d_{eff} \), of a single clusterised code sheet, which forms part of the larger foliated structure. While the effective distances are diminished for codes acting within a single sheet, the distance of the foliated convolutional codes remain 3 and 5 respectively. This is because some error patterns which are undetectable within a single sheet decoder are detectable by neighbouring layers, as discussed in section IV.

At the end of this section, we present numerical results about the decoding performance of two families of foliated turbo codes, T9 and T25 codes. T9 codes are generated from the concatenation and interleaving of two C3 codes; similarly the T25 code is formed from the concatenation and interleaving of two C5 codes. The distances of these turbo codes are \( d_{T9} = 9 \) and \( d_{T25} = 25 \) respectively.

B. Turbo Code Decoders

We now develop a decoding method for foliated turbo codes based on the trellis construction methods outlined in section IV and the SISO decoder in section V. The trellis construction and SISO decoder allow for the decoding of foliated convolutional codes by iteratively decoding individual sheets followed by a series of marginal exchanges on ancilla qubits.
Turbo codes consist of an interleaved concatenation of two convolutional codes. The decoding approach is shown schematically in Fig. 8. Using an a priori distribution of qubit error states, the decoder is implemented and marginal values for qubits \( P(q_l) \) and logical qubits \( P(l_j) \) are calculated for each decoding sheet in the foliated code. Marginals are then exchanged between the shared ancilla qubits in neighbouring layers and used as prior values for a new round of trellis decoding. This process is applied iteratively.

Within a sheet decoder (i.e., red boxes in Fig. 8), the logical marginals, \( P(l_j) \), are deinterleaved (\( \Pi^{-1} \)) and used as priors for the outer decoder, \( P(q_0) \). The same process of ancilla marginal exchange is performed and the decoding process is iterated. Finally the qubit marginals from the outer decoder, \( P(q_l) \), are interleaved (\( \Pi \)) and used as logical priors, \( P(l_l) \) for the inner code.

### C. Numerical Results for Turbo Codes

As noted earlier, \( X \) errors on the foliated cluster commute with parity check measurements. Thus, for our simulations we assume a phenomenological error model in which uncorrelated \( Z \) errors are distributed independently across the cluster with probability \( p \). The decoder performance is quantified in terms of both word error rate (WER), which is the probability of one or more logical errors across all \( k \) encoded qubits, and the bit error rate (BER) which is the probability of an error in any of the encoded qubits. These are defined formally in eqs. (28) and (30).

One common approach to numerically evaluating code performance curves is to sample error patterns, \( \vec{e} \), use the decoder to find a recovery operation \( \vec{e}_{\text{rec}} \), and then test for success or failure of the decoder with respect to the specific error sample. The decoder is successful if \( \vec{e}_{\text{rec}} = \mathbf{G} (\vec{z} + \vec{e}_{\text{rec}}) = 0 \), and unsuccessful if not. If the decoder fails on any logical qubit, this constitutes a Word Error; the Hamming weight of \( \vec{e}_{l} \) counts the number of logical Bit Errors.

One approach to generating code performance curves is to fix the error rate per physical qubit, \( p \), then generate \( N_{\text{trials}} \) error configurations at that error rate. The decoder will fail on some number of those trials, and then the WER is a function of the error rate, given by

\[
\text{WER}(p) = \frac{\#\text{Word Failures}}{N_{\text{trials}}} |_{p}.
\]

(27)

The error rate is then incremented, \( p \rightarrow p' \), and new trials are run for the new value of \( p \).

Similarly, we define the BER (at a given error rate, \( p \), per physical qubit) to be

\[
\text{BER}(p) = \frac{\#\text{Bit Failures}}{k N_{\text{trials}}} |_{p'}.
\]

(28)

In the numerical results reported here, we employ binomial sampling, in which we sample over a fixed number of errors, \( j = 0, 1, 2, \ldots \), and compute the failure probability for each \( j \)

\[
P_{\text{Word}}(j) = \frac{\#\text{Word Failures}}{N_{\text{trials}}} |_{\text{fixed } j}
\]

(29)

for a suitable range of values of \( j \). We then use the binomial formula to relate WER(\( p \)) to \( P_{\text{Word}}(j) \):

\[
\text{WER}(p) = \sum_{j=0}^{n} P_{\text{Word}}(j) \binom{n}{j} p^j (1 - p)^{n-j}.
\]

(30)

In practice the upper limit of the sum can be truncated to much less than \( n \).

Similarly, we define

\[
P_{\text{Bit}}(j) = \frac{\#\text{Bit Failures}}{k N_{\text{trials}}} |_{\text{fixed } j},
\]

(31)

so that the BER is given by

\[
\text{BER}(p) = \sum_{j=0}^{n} P_{\text{Bit}}(j) \binom{n}{j} p^j (1 - p)^{n-j}.
\]

(32)

In what follows we present numerical results for code performance as a function of the code size \( k = nr \), and for several different foliation depths, where we vary the number of code sheets, \( L \). We note that the case \( L = 1 \) is a special case: it corresponds to decoding a single clusterised code sheet, in which errors may also occur on the ancilla qubits (i.e., the red squares in Fig. 1). This is equivalent to decoding the base (i.e., unclusterised) code, but including faulty syndrome measurements.

As an aside, we validate this binomial sampling method by comparing the numerical results of eq. (29) with numerical results generated by the conventional sampling approach, eq. (30). Fig. 9 contains a series of WER trials generated using the conventional sampling with for the case of \( k = 40 \) and \( L = 6 \) (second panel, LHS), shown as points with error bars, using \( N_{\text{trials}} = 10^3 \) for the smallest value of \( p \). These points are in close agreement with the data generated through binomial sampling (solid lines).

### 1. Numerical Results for T9 Turbo Codes

Fig. 9 shows the performance of the \([n, k = n/16, 9]\) T9 self-dual foliated turbo code. We see that for a given foliation depth, \( L \), the performance degrades with the size of the code, \( k \). This indicates that the foliated T9 code is a poorly performing code.

This is explained by considering the effective distance of the clusterised, and then foliated T9 code. The T9 code is generated using two constituent rate \( r = 1/3 \) C3 codes. By construction these codes have a distance of 3, however when clusterised into a single code sheet (which is equivalent to accounting for noisy stabiliser measurements that generate faulty syndrome data) the distance
is reduced to 2. In this setting the seed generator and stabilizer are
\[ G = \begin{bmatrix} D & D & 1 & 0 & 0 \\ 1 + D + D^2 & 1 + D^2 & 1 & 1 & 1 \end{bmatrix}, \quad (33) \]
\[ H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \quad (34) \]
respectively. (Here, we use the notation that qubits the left of the vertical bar are code qubits in the C3 code, and those to the right of the vertical bar are ancilla qubits associated to code stabilizer measurements; these correspond to quintuplets of qubits in each frame of the shaded code sheet in Fig. 7b.)

For the C3 code, an example of an undetectable weight 2 error pattern error pattern in a single code sheet is \( e^* = [0 \ 0 \ 1 \ 1 \ 0] \), which has support on one of the ancilla. Since the distance of the clusterised code sheet is 2, it is reduced to an error detecting code within the sheet. We note that if this specific error pattern were isolated within a larger foliated structure, it would be detected by parity checks in the adjacent sheets, which have support on the affected ancilla qubit.

Given the reduced distance of the clusterised code, we do not necessarily expect this code to perform well in the foliated regime. This is borne out in the numerical results: for a given foliation depth, \( L \); the T9 code has no threshold.

Though this negative result is unsurprising given the foregoing discussion on the clusterised code distance, we show this as an example of a poorly-performing foliated code. The simplest way to rectify this issue is to increase the underlying code distance, which we do in the next example.

2. Numerical Results for T25 Turbo Codes

Fig. 10 shows the performance of the \([n, k = n/16, 25]\) T25, self-dual foliated turbo code.

For each \( L \), there is a threshold error rate around \( p \sim 2\% \), below which the code performance improves with code length (up to at least 160 encoded logical qubits per code sheet, encoded into 4160 physical qubits per sheet), consistent with (pseudo-)threshold behaviour seen in turbo codes [23]. As \( L \) increases, the threshold decreases, more pronouncedly for the WER than the BER. The range of \( k \) and \( L \) that we can simulate is limited by computational time, so we cannot explore the asymptotic performance for large \( L \). Nevertheless, numerics indicate that foliated turbo codes perform quite well for moderate depth foliations.

We note that the foliated construction transforms a clusterised code into a fault tolerant resource state, but with a reduced threshold. This is seen in Fig. 10 in which the threshold is seen to reduce with the number of sheets in the foliation. A similar effect is also seen in Raussendorf’s foliated surface-code construction, in which the fault-tolerant threshold \( k \lesssim 1\% \) is smaller than the \( \sim 11\% \) threshold (assuming perfect stabiliser measurements) for the surface code on which it is based.

One important difference between the surface code and a turbo code family is that the code distance is fixed in the latter, whereas it grows in the former. As a result, we do not expect this threshold behaviour to survive for large foliation depths: this would be analogous to the degradation in performance of Raussendorf’s construction if the transverse code size were held fixed, while the foliation depth were increased. In the numerical results shown, the code distance is fixed at \( d = 25 \), and so we expect that for foliation depths \( L \gg d \), the threshold will disappear; consequently the code is more correctly described as having a pseudo-threshold. Nevertheless, there may be applications where this is sufficient for practical purposes.
To create the generator matrix $k$ rows are removed from $H$. This generates a $[[2m, k, d \approx 2w]]$ quantum code. Here the distance is only approximately $2w$ and will depend on the rows removed from $C$ and the construction of $C$ itself.

We can separate the code into two Tanner graph representations, corresponding to $X$ and $Z$ stabilizers. Since bicycle codes are self-dual the Tanner graphs will be identical in both cases.

In the foliated setting stabilisers are parity check operators of the form given in eq. (2). The code can be separated into two Tanner graphs, one which contains qubits within in the primal sub-lattice, and one which contains on qubits in the dual sub-lattice. For example the primal Tanner graph contains the code qubits in odd sheets $2m + 1$ and ancilla qubits in even sheets $2m$. Primal parity checks are parity check operators which are centred on the sheets $2m + 1$.

### B. Decoding

Bicycle decoding is typically performed using belief propagation on the Tanner graph representation of the code. A variable node corresponds to a given physical qubit; and records the likelihood of all possible errors on that qubit. A factor node corresponds to a given stabiliser, which constrains the possible error states of the connected variable (qubit) nodes.

A factor graph $G = (V, E)$ is a bipartite graph defined by the set, $V = A \cup I$, of variable nodes $I = \{q_1, q_2, \ldots, q_n\}$, and factor nodes, $A = \{a_1, a_2, \ldots, a_{(n-k)}/2\}$, and edges, $E$, between variable and factor nodes, $E = \{(q, a) | a \in A, q \in N(a)\}$, where $N(a)$ is the set of all variables which appear in constraint $a$.

The belief propagation algorithm calculates marginal distributions for the possible error states of each qubit by using repeated message passing. A belief, $b_i(\varepsilon_j)$ represents the probability that qubit $q_i$ has suffered error $\varepsilon_j$; the index $j$ enumerates over possible errors in the error model. This belief is calculated from messages, $m_{a \rightarrow q}(\varepsilon_j) \in [0, 1]$, in which factor nodes, $a$, report a marginal probability that node $q$ has suffered error $\varepsilon_j$

$$b_i(\varepsilon_j) = \frac{1}{N_i} \prod_{a \in N(q_i)} m_{a \rightarrow q}(\varepsilon_j),$$

where $N_i$ is a normalization condition to ensure $\sum_{\varepsilon_j} b_i(\varepsilon_j) = 1$ at each $q_i$.

To calculate $b_i(\varepsilon_j)$, we also pass messages, $m_{q \rightarrow a}(\varepsilon_j) \in [0, 1]$ from qubit nodes to check nodes, reporting the likelihood that $q$ is subject to error $\varepsilon_j$. The values of messages in both directions are determined by iterating over the
following consistency conditions

\[
m_{a \rightarrow q}(\varepsilon_j) = \sum_{\tilde{e}_p : p \in N(a) \setminus q} f_a(\varepsilon_p | \varepsilon_j \text{ on } q) \prod_{r \in N(a) \setminus q} m_{r \rightarrow a}(\varepsilon_{p,r}), \quad (37)
\]

\[
m_{q \rightarrow a}(\varepsilon_j) = \prod_{b \in N(q) \setminus a} m_{b \rightarrow q}(\varepsilon_j).
\]

Here we sum over all possible configurations of errors, \(\tilde{e}_p\), over the neighbours of \(a\) (excluding the target qubit \(q\)), and \(f_a(\varepsilon_p | \varepsilon_j \text{ on } q) \in [0,1]\) are constraint functions that return the \(a\) priori likelihood of the error configuration \(\varepsilon_p\) given that qubit \(q\) is subject to error \(\varepsilon_j\), and \(\varepsilon_{p,r}\) is the restriction of the error configuration \(\varepsilon_p\) to qubit \(r\). The function \(f\) serves two purposes: it vanishes on error configurations that are inconsistent with syndrome data, and otherwise returns the likelihood of a valid error configuration. For our numerical simulations, we will assume independently distributed errors, which implicitly defines \(f\). We note in passing that \(f\) may be tailored to correlated error models if necessary.

To begin the iterative message passing, we initialise the messages on the RHS of eq. (37) using the \(a\) priori error model

\[
m_{q \rightarrow a}(\varepsilon_j) = \Pr(\varepsilon_j).
\]

Belief propagation is exact on tree graphs, allowing for the factorisation of complete probability distribution into marginals over elements, and Equations (37) and (38) naturally terminate at the leaves of the tree. For loopy graphs, the iterative message passing does not terminate in a fixed number of steps; rather we test for convergence of the messages to a fixed point. In some cases, particularly at higher error rates, the message passing may not converge; in this case we simply register a decoding failure on the subset of logical qubits that are affected. Further, the presence of many short cycles may lead to poor performance of the decoder. In the foliated bicycle code, there are numerous short, inter-sheet, graph cycles, however in the numerical results we present in the next subsection, we see empirically that the decoder is effective nonetheless.

Finally, we note that the message passing algorithm described here can be applied directly to the full foliated code. Here, the marginal exchange between sheets happens concurrently with intra-sheet message passing, i.e. eq. (38) performs both inter-sheet marginal exchange and intra-sheet message passing.

C. Numerical Results for Bicycle Codes

We analyse the performance of the codes as a function of the code size \(k = nr\), and the number of foliated layers, \(L\). Fig. 11 shows the performance of a \(d = 26, r = 1/16\), bicycle code, based on Monte Carlo simulations of errors. We use the same binomial sampling schedule as described in Section VI C.

For each \(L\), there is a threshold error rate around \(p \sim 4.5\%\), below which the code performance improves with code length (up to at least 40 encoded logical qubits per code sheet). As with the Turbo codes, as \(L\) increases, the performance increase gained from larger codes is diminished.

This result shows that LDPC bicycle codes are potentially promising codes for foliating.

VIII. CLUSTERISED CODE ARCHITECTURE

In this section we analyse the gate based implementation of cluster state resources for clusterised convolutional and turbo codes. The faulty implementation of cluster bonds (c-phase gates) causes correlated errors to arise during the construction of the cluster state resource. It is important to model these types of errors and ensure that the decoding process is fault-tolerant.
A. Schedule for Cluster-state construction

Cluster state construction requires the preparation of resource $|+\rangle$ qubits and the implementation of C-PHASE gates, $\Lambda(a, b)$, between pairs of qubits. The gates must be implemented over a series of time steps so that during any given time step, $T_j$, any qubit is addressed by at most one phase gate. To generate clusterised codes phase, gates are implemented between ancilla qubits and code qubits according to the Tanner graph of the $S_Z$ stabilisers. The number of ancilla qubits is $\lfloor S_Z \rfloor$. The minimum number of time steps required to implement pairwise C-PHASE gates between each ancilla and the code qubits is proportional to the weight of the largest stabiliser.

The bonds in a clusterized convolutional code can be characterised by a series of qubit pair operations $\Lambda_{T_m}(a_{i,j}, c_{i',j'})$, where $a_{i,j}$ refers to the $i$th ancilla qubit in frame $j$, $c_{i',j'}$ refers to the $i'$th code qubit in frame $j'$ and $T_m$ is a ‘time’ index in the cluster construction schedule.

As an example consider the C3 code defined in equations \eqref{eq:C3}. The parity check operators are weight 6, and the corresponding ancilla qubit are represented by square vertices in the figure. As a result, this clusterised code may be implemented in 6 time steps, $T_1, ..., T_6$. A possible schedule for implementing C-PHASE gates is shown in Fig. 12. Thick, red lines indicate C-PHASE to implement at the corresponding time $T_j$; fine grey lines indicate previously implemented gates. Implied, but not shown are simultaneous translations of these gates across all frames, indexed by $... , t-1, t, t+1 , ...$.

For the case of Turbo codes, the number of cluster bonds between an ancilla qubit for an outer parity check depends on the weight of the outer convolutional parity check, the weight of the inner generator, and the choice of interleaver. The outer parity check is formed by encoding using the inner code generator. In the case of a random interleaver and a large code each bit within the convolutional parity check is distant from each other bit. As a result the total weight for the outer parity check will be $\text{wt}(H_{\text{outer}}) \times \text{wt}(G_{\text{inner}})$.

As we discussed in section \[1\] the choice of interleaver has an effect on the weight of the code stabilisers. In the T9 and T25 clusterised turbo codes, a completely random interleaver will generate stabilisers with weights up 18\footnote{\text{wt}(G_{\text{inner}}) \times \text{wt}(H_{\text{outer}}) = 3 \times 6.} and 98\footnote{\text{wt}(G_{\text{inner}}) \times \text{wt}(H_{\text{outer}}) = 7 \times 14.} respectively. In what follows, we describe more structured interleavers that reduce these weights to 10 and 26 respectively\footnote{Note that we have not done threshold simulations for these interleavers; the numerical results presented in Fig. 10 may depend on the choice of interleaver.}

An interleaver that achieves these lower weight stabilisers is one which permutes the order of bits $\vec{p}$ such that in a block of $f$ frames, the bitbit within each frame is mapped to a single contiguous block $\vec{b}_i$ by the permutation; the second bit within each frame is mapped to another, well spaced block, $\vec{b}_i$, and so on. This is illustrated in Fig. 13. The input sequence over $\tau$ frames is

$$\vec{p} = ((p_1^1, p_2^1, ..., p_n^1), (p_1^2, ..., p_n^2), ..., (p_1^n, ..., p_n^n))$$

\begin{equation}
\equiv (\vec{p}_1, \vec{p}_2, ..., \vec{p}_\tau)
\end{equation}

where $\vec{p}_i = (p_1^i, p_2^i, ..., p_n^i)$ is the input vector over frame $t$, with $n$ physical (qu)bits. The interleaver, $\Pi$, applies a permutation on $\vec{p}$ such that

$$\Pi \vec{p} = ((p_1^1, p_2^1, ..., p_n^1), (p_1^2, ..., p_n^2), ..., (p_1^n, ..., p_n^n))$$

\begin{equation}
= (\vec{b}_1, \vec{b}_2, ..., \vec{b}_n)
\end{equation}

We call this interleaver a transpose interleaver\footnote{if we write $\vec{p}$ as a $\tau \times n$ matrix where the $\vec{p}_i$ are row vectors, then $\Pi \vec{p}$ is the $n \times \tau$ matrix transpose.}. This interleaver does not disperse bits as widely throughout the bitstream as a completely random interleaver, however it does generate inner parity check stabilisers which have significantly lower rate than $\text{wt}(H_{\text{outer}}) \times \text{wt}(G_{\text{inner}})$.

Using a transpose interleaver, the cluster construction schedule is shown in Fig. 14 for the T9 code. In this example, there are two classes of parity check operators for
each frame: square vertices correspond to weight 6 stabilisers, and the diamond vertices correspond to weight 10 stabilisers. As a result, this clustered code may be implemented in 10 time steps, \( T_1, \ldots, T_{10} \).

Also using a transpose interleaver, the T25 code can be implemented in 26 time steps, and has maximum weight 26 stabilisers. We present the cluster-state construction schedule and interleaver details in appendix \( \text{E} \).

To generate the cluster resource for a foliated code each sheet can be generated independently using a suitable schedule for the corresponding primal and dual clusterised codes. An additional two time-steps are then required to connect the code qubits of neighbouring sheets to build the fully foliated network.

### B. Error propagation during cluster-state construction schedule

During the construction of cluster states errors may accumulate on individual qubits, or be caused by faulty gate implementation between qubits. We simplify the analysis of these errors by restricting our analysis to \( \Pi \) and \( \Pi \) Pauli errors. A \( \Pi \) error commutes with \( \Pi \)-phase gates, however an \( \Pi \) error does not, and will propagate a \( \Pi \) error to the neighbouring qubit.

Consider the case where an \( \Pi \) error occurs on an ancilla qubit, \( a_k \), at some time, \( T_e \), in the construction schedule. Subsequent \( \Pi \)-phase gates will generate \( \Pi \) errors on all code qubits \( \epsilon \in N(a_k) \) subject to gates \( \Pi_{T_e}(\epsilon, a_k) \) where \( T_k > T_e \). Note that \( \bigotimes_{\epsilon \in N(a_k)} Z_{\epsilon} \) is a stabiliser, so the this error pattern is equivalent \( \Pi \) errors on qubits \( \epsilon \) subject to gates \( \Pi_{T_k}(\epsilon, a_k) \) where \( T_k \leq T_e \). As a result, the maximum number of \( \Pi \) errors arising during the cluster construction is equal to half the weight of the stabiliser. For this reason codes with low weight parity checks are desirable.

The choice of time ordering for gates affects the types of correlated error patterns that arise during construction. Depending on the choice of code, some time orderings may be more favourable than others, producing error patterns which are more likely to be corrected. One criterion that should be met is that any single physical error during the cluster construction should lead to a correctable (i.e. decodable) correlated error pattern after the \( \Pi \) gates have been made. If the number of correlated errors is less than half the code distance, \( w_{\text{max}} < d/2 \), then this condition is always met. On the other hand, if the number of errors which are propagated is larger than \( d/2 \), then we must verify explicitly that the resulting error pattern is correctable.

As an example, the schedule for constructing the \( d = 9 \) T9 code, which is shown in Fig. 14 using the transpose interleaver. This code has stabilisers of weight 10 associated to the ancillae indicated by diamonds. As a result, an \( \Pi \) error midway through the cluster construction could result in a pattern of up to \( w_{\text{max}} = 5 \) correlated \( \Pi \) errors on code qubits adjacent to the diamond ancillae. In this case, even though \( w_{\text{max}} > d/2 \), we have checked that the decoder correctly corrects all such errors arising in the schedule in Fig. 14.

Similarly, for the \( d = 25 \) T25 code, a maximum stabilizer weight of 26 can be achieved using the transpose interleaver. A physical error during cluster construction could cause correlated error of weight \( w_{\text{max}} = 13 \). Again, in this case, even though \( w_{\text{max}} > d/2 \), we have checked that the schedule for cluster construction (listed in Appendix \( \text{E} \)) produces an error pattern which is correctly decoded.

![FIG. 14: A suitable time ordering for implementing \( \Pi \)-phase gates between ancilla qubits and code qubits in a clusterised T9 turbo code, which is a concatenation of two C3 convolutional codes. The gates for 3 inner seed stabilisers (square ancillas) and one outer seed stabiliser (diamond ancilla) are shown. All other stabilisers are translations of these. Each qubit is acted on by at most one gate at each time step. Thick, red lines indicate \( \Pi \)-phase to implement at the corresponding time \( T_j \), fine grey lines indicate previously implemented gates.](image)
IX. CONCLUSION

In conclusion, we have shown how to clusterize arbitrary CSS codes. We have shown how to foliate clusterized codes, generalising Raussendorf’s 3D foliation of the surface code. We have described a generic approach to decoding errors that arise on the foliated cluster using an underlying soft decoder for the CSS code as a subroutine in a BP decoder, and applied it to error correction by means of a foliated turbo code. We have also shown how decoding can be performed in the case of foliated bicycle codes. These construction may have applications where codes with finite rate are useful, such as long-range quantum repeater networks.

We have exemplified the foliated construction with several code families, namely the T9 and T25 codes, and a the LDPC Bicycle code. We believe this is the first example of a finite rate generalisation of a sparse cluster-state code with pseudo-threshold behaviour (i.e. up to moderately large code sizes). The T25 and the Bicycle code both exhibit threshold-like behaviour, even with moderately large code sizes. The T25 and the Bicycle code families, namely the T9 and T25 codes, and a general code families, namely the T9 and T25 codes, and a

An important direction for future work is the analysis of error models that take into account the cluster state construction schedule in section VIII B the correlated error patterns that arise during construction. Another direction that will be important for repeater application is the tolerance of the foliated construction and decoding process to erasure errors. This is likely to depend on the percolation threshold in the corresponding Tanner graph, as discussed in [10]. Finally, developing code deformation protocols for performing gates within the general foliated architecture is an important direction for future research.

Acknowledgments

This work was funded by ARC Future Fellowship FT140100952 and the ARC Centre of Excellence for Engineered Quantum Systems CE110001013, NSERC and the Canadian Institute for Advanced Research (CIFAR). We thank Sean Barrett, Andrew Doherty, Guillaume Duclos-Cianci, Naomi Nickerson, Terry Rudolph, Clemens Mueller and Stephen Bartlett for helpful conversations.

Appendix A: Example of Inverse Syndrome Formers

We provide an example of the construction of an Inverse Syndrome Former (ISF) and the corresponding pure errors, using the example of the 7-qubit Steane code. asad

The Steane code is self dual, so that the generator, \(G\), parity check, \(H\) and ISF matrices are the same for the \(X\)- and \(Z\)-like operators, so we will drop Pauli labels. The parity check matrix \(H\) for the Steane code is given by the binary support vectors corresponding to the stabilisers defined in \(S^\text{Steane}_2\) in eq. (1), and \(G_T = \{1,1,1,1,1,1,1\}\). The ISF is chosen to satisfy eq. (6). We group \(G\), \(H\) and one possible choice of ISF as sub-matrices in a composite, square matrix

\[
\begin{bmatrix}
G_T & H & ISF
\end{bmatrix} = \begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0
\end{bmatrix}. \tag{A1}
\]

It is straightforward to check that,

\[
\begin{bmatrix}
G_T & H & ISF
\end{bmatrix} . \begin{bmatrix}
G & ISF_T & H_T
\end{bmatrix} = I_{7 \times 7}, \tag{A2}
\]

consistent with eq. (6), i.e. \(ISF_T\) is a pseudo-inverse to \(H\).

A given error chain, \(\vec{e}\), yields a syndrome \(\vec{S} = H \vec{e} \in \mathbb{Z}^2_3\), from which we can compute a pure error \(\vec{e}^0 = ISF_T \cdot \vec{S}\). Since \(H . ISF^T = I\) the pure error satisfies \(H . \vec{e}^0 = H . ISF^T . \vec{S} = \vec{S}\), i.e. it has the same syndrome as the actual error. Equivalently, \(\vec{e} + \vec{e}^0\) is a logical operator on the code space.

Appendix B: Transfer Function Notation

Transfer functions are a convenient method of expressing codes families which have a regular structure but an arbitrary length. Convolutional codes are a family of codes which are often represented by transfer functions. In this appendix we show the relationship between the full matrix expressions for the generators, \(G\), and the seed generators, \(G\).

A \(k \times 1\) vector of logical bits is expressed by vector \(L^T = [l_1, l_2, \ldots, l_k]\). The generator matrix \(G\) can be represented using a finite dimensional seed generator with delay operations. We introduce \(D\) and \(\overline{D}\) which are \(f \times f \) and \(f \times f\) matrices defined by \(\overline{D} = \begin{bmatrix} D^{0} & D^{1} & \cdots & D^{k-1} \end{bmatrix}\) and \(\overline{D} = \begin{bmatrix} D^{0} & D^{1} & \cdots & D^{k-1} \end{bmatrix}\) for a rate \(\frac{1}{2}\) code. The operators \(D\) and \(\overline{D}\) satisfy \(D^i \times D^j = \delta_{ij}\). Here \(D\) is the usual delay operator and \(\overline{D}\) is a mnemonic for the inverse of \(D^i\). Then we have

\[
\overline{D} \times D = \begin{bmatrix}
\overline{D}^0 D^{0} & \cdots & \overline{D}^0 D^{k-1} \\
\overline{D}^1 D^{0} & \cdots & \overline{D}^1 D^{k-1} \\
\vdots & \ddots & \vdots \\
\overline{D}^{k-1} D^{0} & \cdots & \overline{D}^{k-1} D^{k-1}
\end{bmatrix}. \tag{B1}
\]

We write \(GL = \overline{D} DGL\) from which we can derive finite size seed generator.
As an illustration consider a rate \( \frac{1}{3} \) code with generator matrix
\[
G^T = \begin{bmatrix}
111 & 100 & 110 \\
111 & 100 & 110 \\
111 & 100 & 110
\end{bmatrix}.
\] (B2)

We can express the encoding as
\[
GL = \bar{D} \left[ \begin{bmatrix} I_f & I_f & D \end{bmatrix} \right] \times G \times \begin{bmatrix} l_1 \\ l_2 \\ \vdots \\ l_k \end{bmatrix},
\] (B3)

\[
= \bar{D} \begin{bmatrix}
1+D+D^2 & D+D^2+D^2 & D^2+D^3+D^4 \\
1+D^2 & 1+D & D \\
1 & D & D^2 \\
1 & D & D^4
\end{bmatrix}
\begin{bmatrix} l_1 \\ l_2 \\ \vdots \\ l_k \end{bmatrix}
\]

\[
= \bar{D} \begin{bmatrix}
(1 + D + D^2)(l_1 + l_2 D + \ldots + l_k D^{k-1}) \\
(1 + D^2)(l_1 + l_2 D + \ldots + l_k D^{k-1}) \\
(1)(l_1 + l_2 D + \ldots + l_k D^{k-1}) \\
1+D+D^2
\end{bmatrix}
\]

\[
= \bar{D} \begin{bmatrix}
1 + D + D^2 \\
1 + D^2 \\
1
\end{bmatrix} \sum_{i=0}^{k-1} D^i l_{i+1},
\]

\[
= \bar{D} G \sum_{i=0}^{k-1} D^i l_{i+1},
\] (B4)

where \( \bar{D} \) is a \( fn \times f \) matrix, with \( f = 3 \). Here we have \( D^0 \equiv 1 \). In the last line we define the seed generator, \( G \), which is a \( 3 \times 1 \) matrix defined in terms of delay operators.

To output the physical qubits from this seed generator a string of logical input bits \( L \) is multiplied by \( \Delta(D) = [1 \ D \ D^2 \ldots] \). In our working example we have
\[
\sum_{i=0}^{k-1} D^i l_{i+1} \equiv \Delta(D) \times L.
\] (B5)

The code qubits can be determined by multiplying this expression by \( G \). For the more general case of a rate \( \frac{2}{3} \) code takes \( b \) logical inputs and produces \( f \) physical outputs at each frame. For an encoding operation we have
\[
c = GL,
\]

\[
= \bar{D} D G L,
\]

\[
= \bar{D} G \Delta L,
\] (B6)

where \( |c| = fn \times 1 \), \( |\bar{D}| = fn \times f \), \( |D| = f \times fn \), \( |G| = f \times b \) and \( |\Delta| = b \times bn \).

**Appendix C: Transfer Function Manipulation**

Now that we have established the implementation of transfer functions as a description of convolutional codes consider the problem of determining the ISF. The standard approach takes a pseudo inverse of the seed generator matrix. For a rate \( \frac{2}{3} \) code the generator matrix has size \( nb \times nf \), where \( n \) is the number of frames. We have the property
\[
G^{-1} \times G = I_{nb \times nb}.
\] (C1)

To express this in terms of transfer function notation we have
\[
I_{nb \times nb} = \bar{D} G^{-1} G D \bar{D},
\]

\[
= \bar{D} G^{-1} G(D) \begin{bmatrix}
1 & D & \ldots & D^{k-1} & 0 & \ldots & 0 \\
0 & 0 & \ldots & 1 & D & \ldots & D^{k-1}
\end{bmatrix},
\]

\[
= \Delta^T(\bar{D})G^{-1}(\bar{D})G(D)\Delta(D).\]
(C2)

We can use the identity
\[
\bar{D}^j D^j = \delta^{ij} = \bar{D}^0 D^{-i-j},
\] (C3)

to express functions of \( \bar{D} \) in terms of \( D \). Note that the order of operations must be performed so that all \( D \) follow \( \bar{D} \) terms. From eq. \( \text{(C2)} \) we have
\[
I_{nb} = \bar{D}^0 \Delta^T(\bar{D}^{-1})G^{-1}(\bar{D}^{-1})G(D)\Delta(D),
\]

\[
= \bar{D}^0 \Delta D \Delta^T(\bar{D}^{-1})G^{-1}(\bar{D}^{-1})G(D)\Delta(D)\Delta^T(\bar{D}),
\]

\[
= \bar{D}^0 G^{-1}(\bar{D}^{-1})G(D),
\]

\[
= G^{-1}(\bar{D})G(D).
\] (C4)

The pseudo inverse of \( G(D) \) is \( G(D^{-1}) \). To calculate this we make a small alteration to \( G(\bar{D}) \) by substituting the terms \( D^i \) for \( \bar{D}^{-1} \).

We now work through an example to demonstrate this approach. Consider the case of a rate \( \frac{2}{3} \) convolutional code where we wish to find its parity checks matrix. One of the generators is taken from our working example and the second generator input is \( [D, D, 1] \). We have
\[
[G^T I] = \begin{bmatrix}
D & 1 & 0 \\
1 + D + D^2 & 1 + D^2 & 1 \\
1 + D & 0 & 1 \\
0 & 0 & 1
\end{bmatrix},
\] (C5)

has a pseudo inverse of
\[
[I G^{-1}] = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
1 + D & D & 1 + D + D^2 \\
1 + D & D & 1 + D^2
\end{bmatrix}.
\] (C6)

To satisfy the condition \( G^{-1} G = I \) we note that the product of the first column of \( G^{-1} \) and the first row of \( G \) must be 1. The same is true for the second column and second row. The product of the third column of \( A \) and the generators must be zero. Since \( HG = 0 \), and the number of
parity checks is \( n - k = 1 \), this means the third column must be equivalent to \( H^T \). Expressed in transfer function form this gives us

\[
H(D^{-1}) = \begin{bmatrix}
1 + D + D^2 & 1 + D^2 & D^2
\end{bmatrix}. \tag{C7}
\]

It follows that

\[
H(D) = \begin{bmatrix}
1 + D^{-1} + D^{-2} & 1 + D^{-2} & D^{-2}
\end{bmatrix}, \tag{C8}
\]

\[
= \begin{bmatrix}
1 + D + D^2 & 1 + D^2 & 1
\end{bmatrix} \times D^{-2}.
\]

We recognise that the parity check is equivalent to the first seed generator shifted by 2 frames. If we compensate for the shift then the two are equivalent \( (H_X = H_Z) \) since this convolutional code is self-dual. Performing an inverse on eq. \((\text{C6})\) we should reclaim the original seed generators as well as the inverse syndrome former.

\[
\begin{bmatrix}
G_0^T(D^{-1})
\end{bmatrix} \begin{bmatrix}
I
\end{bmatrix} = \begin{bmatrix}
1 + D & 1 + D & 1 + D^2 + D^2 & 1 & 0 & 0
\end{bmatrix}, \tag{C9}
\]

\[
\begin{bmatrix}
I
\end{bmatrix} \begin{bmatrix}
G(D)
\end{bmatrix} \text{ISF} = \begin{bmatrix}
1 & 0 & 0 & 0 & 1 & 0
0 & 1 & 0 & 1 & 0 & 0
0 & 0 & 1 & 1 & 0 & 1
\end{bmatrix}. \tag{C10}
\]

The inverse syndrome former ISF is given by

\[
\text{ISF}(D) = \begin{bmatrix}
D & 1 + D & 0
\end{bmatrix}. \tag{C11}
\]

The product of \( \text{ISF}(D) \) and \( H(D) \) is exactly 1. We can express this as:

\[
\text{ISF}(D^j) \times H(D^j) = \delta_{ij}. \tag{C12}
\]

For an arbitrary syndrome we can use the ISF to generate an error pattern which satisfies the syndrome.

Appendix D: Syndrome and Error pattern
Calculations

Here we show how to calculate the syndrome, \( \tilde{S} \), and initial error pattern, \( \varepsilon^0 \), in the examples given in section \([\text{III}]\). The results appearing in eq. \((15)\) are calculated using \( H, \varepsilon \), and the ISF. The complete list of stabilizers is given by translations of the seed stabilizer by \( D^j \). The index, \( j \), gives the \( j \)th row of the parity check matrix. We have, for all \( j \in \{1, ..., \tau\} \),

\[
H_j = \begin{bmatrix}
1 + D + D^2 & 1 + D^2 & 1
\end{bmatrix} \times D^j,
\]

\[
\text{ISF} = \begin{bmatrix}
D & D & 0
\end{bmatrix}.
\]

For reference, we also write \( H \) in the less compact but more direct notation of eq. \((5)\), using the colours to match terms above with locations of 1’s in the first row below; subsequent rows are translations of the top row.

\[
H = \begin{bmatrix}
\ldots & 000 & 111 & 100 & 110 & 000 & \ldots \\
\ldots & 000 & 111 & 100 & 110 & 000 & \ldots \\
\ldots & 000 & 111 & 100 & 110 & 000 & \ldots \\
\end{bmatrix}_{n \times \tau \times n \tau}.
\]

For the blue path in the example of eq. \((15)\), the error pattern, expressed in delay notation is

\[
\vec{e} = \begin{bmatrix}
D & 0 & 0
\end{bmatrix} \times D^t. \tag{D1}
\]

The \( j \)th element of the syndrome is then given by

\[
S_j = H_j \cdot \vec{e} = D^j(1 + D + D^2) \tilde{D}^t + 1,
\]

\[
= \delta_{j,t+1} + \delta_{j+1,t+1} + \delta_{j+2,t+1},
\]

\[
= \delta_{j,t+1} + \delta_{j,t} + \delta_{j,t-1},
\]

which corresponds to the syndrome, \( \tilde{S} \), listed in eq. \((15)\), with 1’s in the syndrome at frames \( t-1, t \) and \( t+1 \), and zero everywhere else.

In delay notation,

\[
\tilde{S} = \sum_j S_j D^j = D^{t+1} + D^t + D^{t-1}, \tag{D2}
\]

and then using the ISF and the syndrome we calculate the initial error pattern

\[
\vec{e}^0 = \text{ISF} \cdot \tilde{S},
\]

\[
= \begin{bmatrix}
D^t + D^{t+1} + D^{t+2} & D^t + D^{t+1} + D^{t+2} & 0
\end{bmatrix},
\]

which is expressed as the string of bits \( \ldots 110 110 110 \ldots \)

in eq. \((15)\), with the first triplet belonging to frame \( t \).

The example using the red path, as shown in eq. \((16)\), uses the same code. As such the terms for \( H \) and ISF are the same as the previous example. The error pattern used in this example is

\[
\vec{e} = \begin{bmatrix}
D & D & 0
\end{bmatrix} \times D^t.
\]

This generates the \( j \)th element of the syndrome

\[
S_j = D^j(1 + D + D^2) \tilde{D}^t + 1 + D^t + D^{t+2} \tilde{D}^t + 1,
\]

\[
= D^{t+1} \tilde{D}^t + 1,
\]

\[
= \delta_{j+1,t+1},
\]

\[
= \delta_{j,t}.
\]

In delay notation, \( \tilde{S} = D^t \), and then using the ISF and the syndrome we calculate the initial error pattern. This agrees with the syndrome in eq. \((16)\). The initial error pattern is given by

\[
\vec{e}^0 = \text{ISF} \cdot \tilde{S},
\]

\[
= \begin{bmatrix}
D^{t+1} & D^{t+1} & 0
\end{bmatrix},
\]

which is expressed as the string of bits 110 in frame \( t+1 \) of in eq. \((16)\).

Appendix E: Schedule for constructing the T25 cluster code

The C5 code is a self-dual rate \( r = \frac{1}{3} \) code with stabilizers generated by

\[
H = \begin{bmatrix}
1 + D + D^2 & 1 + D^2 + D^3 & 1 + D^2 + D^3 \\
+ D^3 + D^4 & + D^5 & + D^4 + D^5
\end{bmatrix}. \tag{E1}
\]
The weight of this stabilizer is 14, and as such a clustered code can be constructed in 14 time-steps. Fig. [15] depicts the form of the cluster state used to construct the clustered C5 code. One possible scheduling for gate operations is recorded below using $\Lambda(i_{a}, j, c_{i}, j', T_{m})$ for gate operations, where $i$ refers to the $i$th ancilla qubit within a frame and $j$ refers to the $j$th frame of the code and $T_{m}$ is a time index. The terms $a$ and $c$ denote ancilla and code qubits respectively. The C5 code is a rate $\frac{5}{3}$ code and as such there are 3 code qubits and 1 ancilla qubit per frame. A possible scheduling for gate operations is given by

Order C3 =

\[
\begin{align*}
\Lambda_{T_{1}}(a_{1,t}, c_{1,t}), & \quad \Lambda_{T_{2}}(a_{1,t}, c_{3,t+4}), \\
\Lambda_{T_{3}}(a_{1,t}, c_{1,t+1}), & \quad \Lambda_{T_{4}}(a_{1,t}, c_{3,t+3}), \\
\Lambda_{T_{5}}(a_{1,t}, c_{3,t+2}), & \quad \Lambda_{T_{6}}(a_{1,t}, c_{1,t+3}), \\
\Lambda_{T_{7}}(a_{1,t}, c_{1,t+2}), & \quad \Lambda_{T_{8}}(a_{1,t}, c_{2,t+2}), \\
\Lambda_{T_{9}}(a_{1,t}, c_{2,t+3}), & \quad \Lambda_{T_{10}}(a_{1,t}, c_{3,t+5}), \\
\Lambda_{T_{11}}(a_{1,t}, c_{3,t}), & \quad \Lambda_{T_{12}}(a_{1,t}, c_{2,t+2}), \\
\Lambda_{T_{13}}(a_{1,t}, c_{2,t+5}), & \quad \Lambda_{T_{14}}(a_{1,t}, c_{1,t+4}).
\end{align*}
\] (E2)

Encoding these stabilizers with another C5 convolutional code produces a turbo code whose outer stabilizers are weight $7 \times 14 = 98$, where $7$ is the weight of the generators, $G_{C5}$. The weight of these stabilizers is very high.

To reduce the effective weight of these outer stabilizers it is possible to make a choice of interleaver such that the weight of these stabilizers is greatly reduced by taking operator products with inner stabilizers (whose form is identical to the scheduling outlined for the C3 code above). One such choice of interleaver is to generate independent inner encodings for set of qubits produced by the outer encoder, as determined by their position within a frame. For example encode all of the qubits which are in the first position in every frame, then encode all the qubits which are in the second position etc. This produces an inner encoding where logical qubits are more closely correlated than a random interleaver. The benefit is that an inner stabilizer can be generated with a weight of only 26, as compared to 98.

One possible scheduling for a weight 26 stabilizer produced by this choice of interleaver is given by

Order T25(1) =

\[
\begin{align*}
\Lambda_{T_{1}}(A_{1,t}, c_{2,t}), & \quad \Lambda_{T_{2}}(A_{1,t}, c_{2,t+5}), \\
\Lambda_{T_{3}}(A_{1,t}, c_{2,t+1}), & \quad \Lambda_{T_{4}}(A_{1,t}, c_{2,t+3}), \\
\Lambda_{T_{5}}(A_{1,t}, c_{3,t+2}), & \quad \Lambda_{T_{6}}(A_{1,t}, c_{3,t+5}), \\
\Lambda_{T_{7}}(A_{1,t}, c_{3,t}), & \quad \Lambda_{T_{8}}(A_{1,t}, c_{3,t+3}), \\
\Lambda_{T_{9}}(A_{1,t}, c_{2,t+2}).
\end{align*}
\] (E3)

Order T25(2) =

\[
\begin{align*}
\Lambda_{T_{2}}(A_{1,t}, c_{2,t}), & \quad \Lambda_{T_{3}}(A_{1,t}, c_{1,t}+5), \\
\Lambda_{T_{4}}(A_{1,t}, c_{3,t+5}), & \quad \Lambda_{T_{5}}(A_{1,t}, c_{2,t}+1), \\
\Lambda_{T_{6}}(A_{1,t}, c_{2,t}+3), & \quad \Lambda_{T_{7}}(A_{1,t}, c_{3,t}), \\
\Lambda_{T_{8}}(A_{1,t}, c_{2,t}+5), & \quad \Lambda_{T_{9}}(A_{1,t}, c_{1,t}+4), \\
\Lambda_{T_{10}}(A_{1,t}, c_{2,t}+2). & \quad \Lambda_{T_{11}}(A_{1,t}, c_{2,t}+2).
\end{align*}
\] (E4)

Order T25(3) =

\[
\begin{align*}
\Lambda_{T_{3}}(A_{1,t}, c_{2,t}), & \quad \Lambda_{T_{4}}(A_{1,t}, c_{2,t}+5), \\
\Lambda_{T_{5}}(A_{1,t}, c_{3,t}), & \quad \Lambda_{T_{6}}(A_{1,t}, c_{2,t}+1), \\
\Lambda_{T_{7}}(A_{1,t}, c_{3,t}+2), & \quad \Lambda_{T_{8}}(A_{1,t}, c_{3,t}+6), \\
\Lambda_{T_{9}}(A_{1,t}, c_{3,t}), & \quad \Lambda_{T_{10}}(A_{1,t}, c_{3,t}+6), \\
\Lambda_{T_{11}}(A_{1,t}, c_{2,t}+3), & \quad \Lambda_{T_{12}}(A_{1,t}, c_{1,t}+2), \\
\Lambda_{T_{13}}(A_{1,t}, c_{1,t}+3). & \quad \Lambda_{T_{14}}(A_{1,t}, c_{2,t}+3).
\end{align*}
\] (E5)

The scheduling has been split into three components, (1), (2) and (3), referring to each of the outputs bitstreams from the outer encoder. The frames indices $t$, $t'$ and $t''$ refer to frames which are removed from each other, such that each code qubit in the scheduling is uniquely identified by the scheme. Finally we have substituted $a$ with $A$ to refer to the outer ancilla qubits.

[1] P. Shor, Phys. Rev. A. 52-4, R2493 (1995).
[2] A. M. Steane, Phys. Rev. Lett. 77, 793 (1996).
[3] R. Raussendorf and H. J. Briegel, Phys. Rev. Lett. 86, 5188 (2001).
[4] R. Raussendorf, D. E. Browne, and H. J. Briegel, Phys. Rev. A. 68, 022312 (2003).
[5] R. Raussendorf, S. Bravyi, and J. Harrington, Phys. Rev. A. 71, 062313 (2005).
[6] R. Raussendorf, J. Harrington, and K. Goyal, Annals of Physics 321-9, 2242 (2006).
[7] R. Raussendorf and J. Harrington, Phys. Rev. Letters 98, 190504 (2007).
[8] R. Raussendorf, J. Harrington, and K. Goyal, New Journ. Phys. 9, 199 (2007).
[9] A. Kitaev, Annals of Physics 303-1, 2 (1997).
[10] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill, Journ. Math. Phys. 43-9, 4452 (2002).
[11] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland, Phys. Rev. A 86, 032324 (2012), URL [https://link.aps.org/doi/10.1103/PhysRevA.86.032324](https://link.aps.org/doi/10.1103/PhysRevA.86.032324)
[12] J. Martinis, R. Barends, J. Kelly, A. Veitia, D. Sank, E. Jeffrey, T. White, J. Mutus, A. Fowler, B. Campbell,
et al., Nature, 508 (2014).
[13] D. Ristè, S. Poletto, M. Z. Huang, A. Bruno, V. Vesterinen, O. P. Saira, and L. DiCarlo, Nature Communications 6, 6983 EP (2015), URL http://dx.doi.org/10.1038/ncomms7983.
[14] J. M. Gambetta, J. M. Chow, and M. Steffen, npj Quantum Information 3, 2 (2017).
[15] T. M. Stace and S. D. Barrett, Phys. Rev. A. 81, 022317 (2010).
[16] S. D. Barrett and T. M. Stace, Phys. Rev. Lett. 105, 200502 (2010), URL https://link.aps.org/doi/10.1103/PhysRevLett.105.200502.
[17] G. Duclos-Cianci and D. Poulin, Phys. Rev. Lett. 104, 050504 (2010), URL https://link.aps.org/doi/10.1103/PhysRevLett.104.050504.
[18] T. Rudolph, APL Photonics 2, 030901 (2017).
[19] S. Bravyi and A. Kitaev, Physical Review A 71, 022316 (2005).
[20] A. Bolt, G. Duclos-Cianci, D. Poulin, and T. M. Stace, Phys. Rev. Lett. 117, 070501 (2016).
[21] C. Berrou, N. Ecole, A. Glavieux, and P. Thitimajshima, in Communications. ICC ’93 Geneva. Technical Program, Conference Record, IEEE International Conference on (vol 2) (1993).
[22] R. McEliece, D. MacKay, and J. Cheng, IEEE Journal on Selected Areas in Communications 16, 140 (1998).
[23] D. Poulin and H. Tillich, J.-P. Ollivier, IEEE Transactions on Information Theory 55 No. 6, 2776 (2009).
[24] H. Utby, Master’s thesis, University of Bergen (2006).
[25] H. Briegel, D. Browne, W. Dur, R. Raussendorf, and V. d. Nest, Nature Physics 5, 19 (2009).
[26] R. Tanner, IEEE Trans. Inform. Theory 27-5 (1981).
[27] H. Ollivier and J.-P. Tillich, Phys. Rev. Lett. 91, 177902 (2003).
[28] A. Kitaev, Annals of Physics 303-1, 2 (2003).
[29] J.-P. Tillich and G. Zemor, in Information Theory, IEEE International Symposium on (2009).
[30] D. Poulin and Y. Chung, Quantum Information & Computation, 8-10 (2008).
[31] D. Forney, Ph.D. thesis, Massachusetts Institute of Technology (1965).
[32] H. Sadjadpour, Proc. SPIE 4045, Digital Wireless Communication 2, 73 (2000).
[33] A. Viterbi, IEEE Trans. Inf. Theory 13 (1967).
[34] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, IEEE Trans. Inf. Theory 20, 284 (1974).
[35] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, TDA Progress Report. 42-124. (1996).
[36] D. MacKay and R. Neal, Electronic Lett. 33 (1997).
[37] D. Mackay, G. Mitchison, and P. McFadden, IEEE Transactions on information theory 50-10, 2315 (2004).
[38] J. Pearl, in Proceedings of the 7th Conference of the Cognitive Science Society, University of California. (1985).
[39] Y. Li, S. D. Barrett, T. M. Stace, and S. C. Benjamin, Phys. Rev. Lett 105, 250502 (2010).
[40] L.-M. Duan, M. Lukin, J. Cirac, and P. Zoller, Nature 414, 413 (2001).
[41] S. C. Benjamin, J. Eisert, and T. M. Stace, New Journal of Physics 7, 194 (2005), URL http://stacks.iop.org/1367-2630/7/i=1/a=194.
[42] S. D. Barrett, P. P. Rohde, and T. M. Stace, New Journal of Physics 12, 093032 (2010), URL http://stacks.iop.org/1367-2630/12/i=9/a=093032.
[43] T. Satoh, K. Ishizaki, S. Nagayama, and R. Van Meter, Phys. Rev. A. 93, 032302 (2016).
[44] H. Sadjadpour, N. Sloane, M. Salehi, and G. Nebe, Selected Areas in Communications, IEEE Journal on. v19. (2001).
[45] S. Vaﬁ and T. Wysocki, in Proceedings of the 6th Australian Communications Theory Workshop (2005).
[46] D. Forney, M. Grassl, and S. Guha, IEEE Transactions on information theory 53 - 3, 865 (2007).
[47] B. Frey and D. MacKay, Advances in Neural Infromation Processing Systems (1998).
[48] M. Hagiwara and H. Imai, in Information Theory. IEEE International Symposium on (2007).
[49] D. Mackay, IEEE Transactions on Information Theory 45 No. 2, 399 (1999).