UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Daglas Aitsen

Search for Good Quantum Error-Correcting Codes

Bachelor's Thesis (9 ECTS)

Supervisor: Irina Bocharova, PhD

Tartu 2025

Search for Good Quantum Error-Correcting Codes

Abstract:

Quantum error correction codes are developed to correct errors that occur in quantum data. However, their design is complicated by the fundamental constraints of quantum mechanics. This thesis explores how to adapt quasi-cyclic low-density parity-check (QC-LDPC) codes, used in classical communications, for quantum error correction. Since a small girth in an LDPC code is linked to poor decoding performance, this thesis focuses on constructing quantum QC-LDPC codes with large girth. To achieve this, we present two methods: one based on minimal-weight codewords and another based on multiplicity constraints to avoid short cycles. Using these approaches, we construct quantum QC-LDPC codes with girth 8. The construction methods offer a more flexible alternative to existing techniques, enabling the design of large-girth codes with a wider range of structures and sizes.

Keywords: Coding theory, quantum error correction, girth, QQC-LDPC codes

CERCS: P170 Computer science, numerical analysis, systems, control

Heade kvantveaparanduskoodide otsing

Lühikokkuvõte:

Kvantveaparanduskoodid aitavad tuvastada ja parandada kvantarvutites esinevaid vigu, kuid nende konstrueerimine on keeruline kvantmehaanika piirangute tõttu. Käesolevas töös uuritakse, kuidas konstrueerida häid kvantveaparanduskoode klassikalises sidetehnikas kasutatavatest kvaasitsüklilistest madala tihedusega paarsuskontrolli (QC-LDPC) koodidest. Kuna väike LDPC koodi ümbermõõt langetab koodi dekodeerimise efektiivsust, keskendub töö suure ümbermõõduga koodide leidmisele. Töös esitatakse kaks meetodit QC-LDPC kvantkoodide konstrueerimiseks: esimene kasutab minimaalse kaaluga koodisõnu, teine aga mitmekordsustingimusi, et vältida lühikeste tsüklite tekkimist. Esitatud lähenemiste abil konstrueeriti kvantkoodid, mille ümbermõõt on 8. Pakutavad meetodid on võrreldes olemasolevate tehnikatega paindlikumad, võimaldades luua erineva struktuuri ja suurusega kvantkoode.

Võtmesõnad: Kodeerimisteooria, kvantveaparandus, ümbermõõt, QQC-LDPC koodid

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

Introduction 4

1 Elements of Classical Coding Theory 5
- 1.1 Linear Codes 5
- 1.2 LDPC Codes 8
  - 1.2.1 Graph Representation 9
  - 1.2.2 Iterative Decoding 10
- 1.3 QC-LDPC Codes 10

2 Elements of Quantum Coding Theory 13
- 2.1 Quantum Information 13
- 2.2 Examples of Quantum Codes 15
- 2.3 Stabilizer Codes 19
  - 2.3.1 Stabilizers 19
  - 2.3.2 Stabilizer Codes 20
  - 2.3.3 CSS Codes 21

3 Construction of QQC-LDPC Codes 23
- 3.1 Problem Statement 23
- 3.2 Methodology 24
  - 3.2.1 Codeword-Based Construction 24
  - 3.2.2 Construction via Multiplicity Constraints 26
- 3.3 Results 32

4 Discussion 34
- 4.1 Comparison 34
- 4.2 Limitations 35
- 4.3 Future Work 35

Conclusion 37

References 39

Appendix 40
- I. Sum-Product Algorithm 40
- II. Licence 42

Introduction

The idea of quantum computers was first proposed by Richard P. Feynman in 1982 [1], who suggested they could simulate the physical world more effectively than classical machines. Today, quantum computing is an active research field with potential to solve problems beyond classical capabilities, such as breaking RSA encryption using Shor's algorithm [2, 3], speeding up database search [4], simulating drug metabolism [5], and reducing carbon dioxide [6, 7].

Quantum computers are inherently noisy and error-prone compared to classical systems [8], limiting the size of reliable circuits and making quantum error correction essential. A promising approach is to adapt efficient classical error correction techniques to the quantum domain.

Quasi-cyclic low-density parity-check (QC-LDPC) codes are a structured subclass of LDPC codes, known for their sparse parity-check matrices [9]. A critical design parameter is the girth—the length of the shortest cycle in the Tanner graph—as small cycles degrade decoding performance [10]. While large-girth QC-LDPC codes show promise for quantum error correction, not all classical designs translate directly, and quantum-specific constraints complicate construction.

The goal of this thesis is to develop new methods for constructing quantum QC-LDPC (QQC-LDPC) codes by addressing the following research question:

- How can we systematically construct large-girth quantum LDPC codes with various quasi-cyclic structures?

To answer this, we propose algorithmic techniques to generate candidate large-girth QQC-LDPC codes. Our main contribution is a demonstrated construction framework yielding new codes with girth metrics comparable to those in prior literature.

Chapters 1 and 2 provide the theoretical background, beginning with linear codes and QC-LDPC codes, followed by quantum codes, stabilizer codes, and CSS formalism. Chapter 3 presents our code search methods and results. Finally, we compare the proposed construction approaches and discuss their limitations.

While polishing the final version of this thesis, ChatGPT[1] was used to improve wording—such as rephrasing sentences, fixing grammar, or finding synonyms. It was used sparingly, only for one to a few sentences at a time, and never for longer sections spanning multiple paragraphs or pages.

[1]OpenAI (2025). ChatGPT 4o: https://chat.openai.com

1 Elements of Classical Coding Theory

Coding theory focuses on techniques for encoding information to detect and correct errors during transmission or storage, ensuring reliable communication even in noisy environments [11]. Classical coding theory provides the foundation for quantum error correction. Linear codes and in particular low-density parity-check (LDPC) codes are significant in this context due to their efficient encoding and decoding properties [12, 9].

This chapter introduces key concepts from classical coding theory that are essential for understanding quantum error correction codes. We first discuss linear codes, including the generator and parity-check matrices, and minimum Hamming distance. Next, we explore LDPC codes, highlighting their graph-based representation and iterative decoding algorithms. Finally, we introduce QC-LDPC codes.

1.1 Linear Codes

Linear codes are a fundamental class of error-correcting codes used to detect and correct errors in transmitted data. They work by transforming each message into a longer vector called a *codeword* through the addition of parity information. This redundancy allows a receiver to detect and correct errors introduced during transmission through a noisy communication channel. The encoding is performed by an encoder, and a corresponding decoder attempts to reconstruct the original message from the received codeword.

The key idea is to represent messages as codewords that belong to a linear subspace of a finite-dimensional vector space. This subspace structure ensures that any linear combination of codewords remains within the set of valid codewords.

Definition 1.1. A linear $[n, k]_q$ code C of length n and dimension k over the finite field $\mathbb{F}_q$ is a k-dimensional subspace of $\mathbb{F}_q^n$.

If $q = 2$, the code is called a binary linear code, which is generally the case for classical computers that use bits. Each codeword in C is a binary vector of length n, and C has 2^k distinct codewords, since a k-dimensional subspace of $\mathbb{F}_2^n$ has 2^k elements.

Definition 1.2. Code rate of the linear $[n, k]$-code is defined as $R = \frac{k}{n}$.

Definition 1.3. A generator matrix G of an $[n, k]_q$ linear code is a $k \times n$ matrix whose rows form a basis for the subspace C.

Every codeword $\mathbf{c} \in C$ can be expressed as

$$\mathbf{c} = \mathbf{m} \cdot G,$$

where $\mathbf{m} \in \mathbb{F}_q^k$ is a message vector of length k, and the multiplication is performed over $\mathbb{F}_q$. This means we can turn any message vector into a codeword by multiplying it with the generator matrix.

Definition 1.4. The parity-check matrix H of an $[n, k]_q$ linear code C is an $r \times n$ matrix, where $r = n - k$, that satisfies

$$H \cdot \mathbf{c}^T = \mathbf{0}$$

for all codewords $\mathbf{c} \in C$.

The generator matrix G and the parity-check matrix H of a linear code satisfy

$$GH^T = 0.$$

The generator matrix G is used to encode messages by turning a message vector $\mathbf{m}$ into a codeword $\mathbf{c} = \mathbf{m}G$, which is then sent over a noisy channel. The parity-check matrix H is used in decoding to check for errors. If a received vector $\mathbf{r}$ does not satisfy $H \cdot \mathbf{r}^T = \mathbf{0}$, then $\mathbf{r}$ is not a valid codeword, and an error is assumed to have occurred.

Definition 1.5. Let $C \subset \mathbb{F}_q^n$ be a linear code. The dual code of C, denoted by $C^\perp$, is defined as

$$C^\perp = \left\{ x \in \mathbb{F}_q^n \,\middle|\, (x, c) = 0 \text{ for all } c \in C \right\},$$

where $(x, c) = \sum_{i=1}^n x_i c_i$ is the scalar product over $\mathbb{F}_q$.

We say that a code C is *self-dual* if $C^\perp = C$, and *dual-containing* if $C^\perp \subset C$. For a code C with generator matrix G and parity-check matrix H, the dual code $C^\perp$ has generator and parity-check matrices given by $G^\perp = H$ and $H^\perp = G$.

Syndrome. When a message is transmitted over a noisy channel, the received vector $\mathbf{r}$ may differ from the original codeword $\mathbf{c}$. To detect and locate these errors, the concept of the syndrome is used. The syndrome $\mathbf{s}$ of a received vector $\mathbf{r}$ is computed as

$$\mathbf{s} = H \cdot \mathbf{r}^T.$$

If $\mathbf{s} = \mathbf{0}$, then $\mathbf{r}$ is a valid codeword. Otherwise, the nonzero syndrome provides information about location of errors. The mapping from the syndrome to the error pattern can be precomputed and stored in a syndrome table.

Minimum Hamming Distance. A crucial parameter for assessing the error-correcting capability of a linear code is its minimum Hamming distance d_{min}. The Hamming

distance between two binary vectors $\mathbf{x}$ and $\mathbf{y}$ of equal length is the number of positions at which they differ. For a linear code C, the minimum Hamming distance d_{min} is defined as the smallest Hamming distance between any two distinct codewords in C:

$$d_{\text{min}} = \min_{\mathbf{c}_1, \mathbf{c}_2 \in C, \mathbf{c}_1 \neq \mathbf{c}_2} d_H(\mathbf{c}_1, \mathbf{c}_2).$$

Since C is a linear subspace, the minimum distance can also be interpreted as the minimum Hamming weight (i.e., the number of nonzero bits) of any nonzero codeword in C:

$$d_{\text{min}} = \min_{\mathbf{c} \in C, \mathbf{c} \neq \mathbf{0}} d_H(\mathbf{c}, \mathbf{0}).$$

The minimum distance d_{min} determines the guaranteed error-detecting and error-correcting capabilities of the code. Specifically, the code can detect up to $d_{\text{min}} - 1$ and can correct up to $t = \lfloor (d_{\text{min}} - 1)/2 \rfloor$ errors. For example, if $d_{\text{min}} = 3$, then the code can detect up to 2 errors and correct up to 1 error.

Repetition Code. One of the simplest examples of a linear error-correcting code is the repetition code. To correct errors that may occur during transmission, the message bits are repeated multiple times. If an error arises, it can be corrected using a majority vote.

Consider a repetition code of length 3, where each bit in the original message is duplicated three times. For example, if the original message is `101`, the transmitted message will be `111000111`. Suppose the receiver obtains the message `110000111`. Here, an error has occurred in the first bit of the original message, as the repeated bits are not identical. To correct the error, a majority vote can be taken from the received bits. Since the majority of bits in the group `110` are 1, we can infer that the original bit was 1, allowing us to decode the received message as `101`.

Single Parity-Check Codes. Another basic example of a linear code is the single parity-check code. In this code, an additional bit, known as the parity check bit, is appended to the message. The parity check bit is chosen such that the sum of all bits in the resulting codeword is even. The minimum Hamming distance of a single parity-check code is 2, which means it can detect the occurrence of an error but cannot correct it.

The generator matrix for a $[n, n-1]$ single parity-check code has the form:

$$G = \begin{bmatrix} 1 & 0 & \cdots & 0 & 1 \\ 0 & 1 & \cdots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 1 \end{bmatrix}.$$

The corresponding parity check matrix H would be:

$$H = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \end{bmatrix}.$$

Let us consider an example of encoding and decoding a single parity-check code of length $n = 4, k = n - 1 = 3$. Let the original message $\mathbf{m} = \texttt{101}$. Since the Hamming weight (number of 1s) of $\mathbf{m}$ is even, the parity bit will be $\texttt{0}$ (the Hamming weight of the codeword must be even), meaning the codeword $\mathbf{c} = \texttt{1010}$, where $\mathbf{c} = \mathbf{m} \cdot G$.

If an error, for example $\mathbf{e} = \texttt{0100}$, occurs during transmission, then the received vector will be $\mathbf{r} = \mathbf{c} + \mathbf{e} = \texttt{1110}$. Since the Hamming weight of this vector is odd ($\mathbf{r} \cdot H^T = 1$), we can conclude that an error has occurred. If the Hamming weight of the original message had been odd, then the parity bit would have been $\texttt{1}$ to ensure that the Hamming weight of the codeword remains even.

1.2 LDPC Codes

Low-density parity-check (LDPC) codes are a class of linear error-correcting codes whose parity-check matrices are sparse, meaning they contain only a small number of non-zero entries, or for binary linear codes, a small number of 1's. The sparsity of these matrices is important, since it allows for low-complexity iterative decoding of these codes [9, 13]. In addition, the decoding process of LDPC codes can be easily parallelized in computation [14], which makes them a good choice for modern communication applications. LDPC codes were first invented by R. Gallager in 1960s [9].

Definition 1.6. A binary linear $[n, k]$-code determined by a parity-check matrix H is called a (J, K)-regular LDPC code if each column of H contains J ones and each row contains K ones.

A LDPC code is called regular when it has the same number of ones in each column and row in its parity check matrix, otherwise the code is called irregular. The values of J and K are, in general, relatively small compared to n and k, resulting in a sparse parity-check matrix. The code rate for a LDPC code can be expressed in terms of the parameters J and K as

$$R \geq 1 - \frac{J}{K}.$$

To better understand the structure and decoding process of LDPC codes, they can be represented graphically.

1.2.1 Graph Representation

In 1981, M. Tanner introduced a graphical representation for LDPC codes, known as Tanner graphs [10]. Tanner graphs also help in designing and decoding LDPC codes, as the performance of iterative decoding is influenced by cycles in the graph.

Definition 1.7. The Tanner graph of a linear code determined by the parity-check matrix $H = (h_{ij})$, where $i = 1, \ldots, r$ and $j = 1, \ldots, n$, is a bipartite graph whose one set of nodes corresponds to the rows of H (check nodes) and the other set of nodes corresponds to the set of columns (variable nodes). A check node c_i is connected with a variable node v_j if $h_{ij} \neq 0$.

Let us consider a parity-check matrix of the $(J = 2, K = 4)$-regular LDPC code:

$$H = \begin{bmatrix} v_0 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & c_0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & c_1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & c_2 \\ 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & c_3 \end{bmatrix}.$$

Every row of the matrix H corresponds to a parity-check equation. Since a valid codeword $\mathbf{c} = (v_0, v_1, \ldots, v_7)$ must satisfy the constraint $H\mathbf{c}^T = 0$, these parity-check equations define a system of linear equations that all valid codewords must satisfy:

$$\begin{cases} v_1 + v_3 + v_4 + v_7 = 0, \\ v_0 + v_1 + v_2 + v_5 = 0, \\ v_2 + v_5 + v_6 + v_7 = 0, \\ v_0 + v_3 + v_4 + v_6 = 0. \end{cases}$$

This system ensures that every valid codeword satisfies the parity constraints of H. The Tanner graph for the parity-check matrix H is shown in Figure 1.

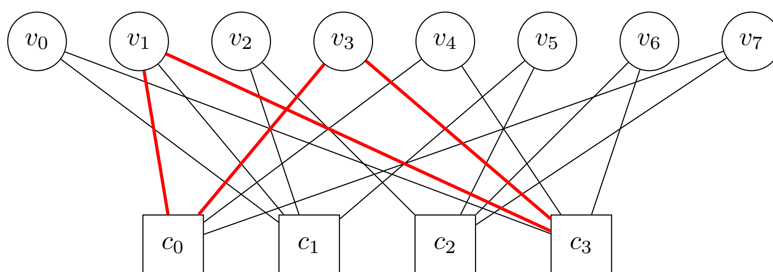

Figure 1. Tanner graph for the given LDPC code.

Definition 1.8. A cycle of length g in a graph is an alternating sequence of $g+1$ vertices v_i, where $i = 1, 2, \ldots, g+1$, and g edges e_i, where $i = 1, 2, \ldots, g$, such that $e_i \neq e_{i+1}$ and the first and last vertices coincide, i.e., $v_1 = v_{g+1}$. A cycle is called simple if all its vertices and edges are distinct, except for the first and last vertex, which coincide. The length of the shortest simple cycle is called the girth of the graph.

The girth of the Tanner graph shown in Figure 1 is 4, highlighted in red. The girth of the LDPC code is an important property that affects code decoding performance [10].

1.2.2 Iterative Decoding

LDPC codes can be decoded using iterative decoding. One of the most well-known and widely adopted algorithms in this category is the sum-product algorithm [15]. The algorithm uses probabilistic message passing between the variable and check nodes of the Tanner graph to iteratively refine the decoder estimate of the transmitted codeword.

The decoding process begins with the initialization of messages based on the received channel values. Each variable node sends an initial message to its connected check nodes representing its belief about the bit being 0 or 1. During each iteration, check nodes update and send messages back to variable nodes by aggregating the incoming beliefs from other connected variable nodes—this is known as the horizontal step. Then, variable nodes use these messages to update their own beliefs and send revised messages back to the check nodes—this is the vertical step. Finally, hard decisions are made on each bit based on the updated beliefs, and the syndrome is computed. If the syndrome equals zero, the decoding is successful and terminates early; otherwise, the algorithm continues until a predefined maximum number of iterations is reached. A detailed formulation of the algorithm is provided in Appendix 4.3.

The decoding performance of the sum-product algorithm is influenced by the structure of the Tanner graph [10]. Specifically, short cycles—small loops in the graph—are harmful because they cause the same piece of information to be repeatedly passed between nodes. This can lead to the reinforcement of incorrect decisions and a higher probability of decoding failure. Therefore, when designing LDPC codes, we should look for graphs with large girth to reduce the bad effects of short cycles and preserve the effectiveness of iterative decoding.

1.3 QC-LDPC Codes

In practice, short LDPC codes are rarely used; instead, codes with lengths of hundreds of thousands of bits are preferred. Storing these large parity-check matrices is inefficient, and the decoding process is computationally complex. In the early 2000s, a

new structured class of LDPC codes known as quasi-cyclic (QC-LDPC) codes was introduced [16].

QC-LDPC codes can be obtained from existing LDPC codes by a process called lifting. Lifting extends the base parity-check matrix by substituting sparse permutation or zero matrices into it.

Definition 1.9. A permutation matrix is a square binary matrix where each row and each column contains a single 1, while all other elements are 0.

The permutation matrix is an orthogonal matrix with a determinant of either ± 1, depending on the parity of the permutation.

Definition 1.10. A circulant matrix is a square matrix in which each row is a cyclic shift of the preceding row.

For example, the matrices P_M^i below are order $M = 3$ and power $i = 0, 1, 2$ circulant permutation matrices.

$$P_3^0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}; \quad P_3^1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}; \quad P_3^2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Definition 1.11. A linear code is called quasi-cyclic if, for some l, a cyclic shift of a codeword by l positions results in another codeword of the same code.

Moving forward, LDPC codes determined by block-circulant parity-check matrices are referred to as QC-LDPC codes.

The base matrix of a QC-LDPC code is a binary matrix that defines the structure of the code by indicating the positions of the non-zero submatrices in the final parity-check matrix. A 1 in the base matrix marks where a circulant matrix will be placed, and a 0 marks where a zero matrix will go. The labeled matrix is derived from the base matrix by replacing each 1 with a non-negative integer. The elements in the labeled matrix denote the cyclic shift of the circulant matrix that replaces them in the lifting process. The element -1 indicates a zero matrix. Thus, to define a LDPC code, we can simply specify the non-zero positions and their corresponding cyclic shifts in the labeled matrix. This approach allows for compact representation of long codes.

Example of QC-LDPC Code

Let us consider a binary 3×4 base matrix B of a QC-LDPC code. For example,

$$B = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}.$$

To define the full parity-check matrix, we use a labeled matrix $H_B = \{\mu_{ij}\}$, which is based on B. Let the lifting factor be $M = 3$. The entries of H_B are taken from $\mathbb{Z}/M\mathbb{Z} \cup \{-1\}$. An example labeling for base matrix B could be:

$$H_B = \begin{bmatrix} 0 & 2 & 1 & 2 \\ -1 & 1 & -1 & 0 \\ 0 & -1 & 2 & -1 \end{bmatrix},$$

where the value -1 corresponds to the zero sub-matrix, while degrees $\mu_{ij} = 0, 1, 2$ correspond to cyclic shifts of the $M \times M$ circular permutation matrix P_3. This labeled matrix defines a $[4M, 3M]$-code.

The binary parity-check matrix that we obtain after extending the labeled matrix H_B by lifting is as follows:

$$H(P_3) = \begin{bmatrix} P_3^0 & P_3^2 & P_3^1 & P_3^2 \\ 0 & P_3^1 & 0 & P_3^0 \\ P_3^0 & 0 & P_3^2 & 0 \end{bmatrix} = \left[\begin{array}{ccc|ccc|ccc|ccc} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right].$$

Classical coding theory is the foundation for understanding and building quantum error-correcting codes. In this chapter, we explored linear codes, LDPC codes, and their structured variants like QC-LDPC codes. These classical constructs form the basis for many quantum counterparts. To apply them in the quantum setting, we now need to understand how quantum information is represented and how errors occur in quantum systems. This brings us to the next chapter, where we explore the basics of quantum computing and how classical codes can help us build quantum ones.

2 Elements of Quantum Coding Theory

Quantum error correction is a critical component in the development of fault-tolerant quantum computing. It enables the preservation of quantum information in the presence of noise and operational imperfections. Largely, quantum error correction works in a similar way to classical error correction: by adding redundant information that can be used to detect and correct errors.

This chapter of the thesis provides the theoretical background needed for constructing quantum QC-LDPC codes. First, we discuss the basics of quantum computing: how quantum information is represented and how errors can be modeled. Additionally, we present examples of two foundational quantum error-correcting codes: the three-qubit bit-flip code and Shor's code. Lastly, we introduce stabilizer codes and also examine CSS codes, which allow us to use classical codes to construct quantum codes.

2.1 Quantum Information

In classical computing the most basic unit of information is called a bit. In quantum computing the basic unit of information is called a qubit. The classical binary bit can be in two basic states, either 1 or 0. A qubit can be in the state 0, the state 1, or a superposition of two states. The qubit $|v\rangle$ can be expressed as a linear combination of the basis states $|0\rangle$ and $|1\rangle$ as follows:

$$|v\rangle = \alpha|0\rangle + \beta|1\rangle,$$

where α and β are complex coefficients. The basis states are defined as:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Thus, the qubit $|v\rangle$ can also be represented in matrix form as:

$$|v\rangle = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}, \quad \text{where } \alpha, \beta \in \mathbb{C}.$$

For a superposition $|v\rangle = \alpha|0\rangle + \beta|1\rangle$ it must hold that $|\alpha|^2 + |\beta|^2 = 1$. If so, the qubit is measured as 0 with probability $|\alpha|^2$ and 1 with probability $|\beta|^2$.

Quantum errors. In classical systems, errors occur as bit-flips, where a 0 becomes a 1 or vice versa. Qubits, however, in addition to bit-flips, can also undergo a phase-flip or even a combination of both errors simultaneously.

These types of errors can be conveniently described using the Pauli matrices—a set of 2×2 unitary transformations. Operations on qubits must be unitary in order to preserve the normalization of quantum states. The Pauli matrices are:

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

Each of these matrices corresponds to a specific type of error:

- **Bit-flip (X)**: Changes the state $|0\rangle$ to $|1\rangle$ and vice versa.
- **Phase-flip (Z)**: Leaves $|0\rangle$ unchanged but multiplies $|1\rangle$ by -1, effectively flipping its phase.
- **Bit and Phase-flip (Y)**: Applies both a bit-flip and a phase-flip simultaneously.

For a quantum state $|\psi\rangle$, the errors are interpreted as transformations of the state vector v by multiplying it with one of the Pauli matrices:

- A bit-flip X results in $X|\psi\rangle$.
- A phase-flip Z results in $Z|\psi\rangle$.
- A bit and phase-flip Y results in $Y|\psi\rangle$.

For example for quantum state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ the bit flip would be the following

$$X|\psi\rangle = X(\alpha|0\rangle + \beta|1\rangle) = \alpha|1\rangle + \beta|0\rangle.$$

Any quantum noise sequence E can be expressed as a tensor product of Pauli matrices. For example,

$$E = X \otimes I \otimes Y \otimes Z \otimes X,$$

where X, Y, Z, and I are the Pauli matrices. For a given error E, the weight of E is defined as the number of components E_i in E that are not equal to the identity matrix I.

Constraints on quantum encoder and decoder

The application of classical error correction techniques, such as the repetition code, to quantum systems is constrained by the principles of quantum mechanics. One primary constraint is the *no-cloning theorem*, which states that you cannot create identical copies of an arbitrary quantum state. This means that it is impossible to directly replicate quantum information to detect and correct errors as one would in classical repetition codes. Another constraint is that measuring a quantum state collapses its superposition,

thereby destroying the original state. This makes it impossible for a quantum decoder to measure the received qubits directly to identify errors. Additionally, quantum error correcting codes must be able to not only correct bit-flip errors, but also correct phase-flip errors and combined errors.

Another example of a unitary transform is the Controlled-NOT (CNOT) gate, which plays a key role in many quantum circuits. The CNOT gate operates on two qubits: a control qubit and a target qubit. It flips the state of the target qubit if and only if the control qubit is in state $|1\rangle$.

For example, consider a control qubit in an arbitrary superposition $|\psi\rangle = a|0\rangle + b|1\rangle$, and a target qubit initially in state $|0\rangle$. Applying the CNOT gate results in:

$$U_{\text{CNOT}}|\psi\rangle|0\rangle = U_{\text{CNOT}}\left(a|0\rangle + b|1\rangle\right)|0\rangle = aU_{\text{CNOT}}|00\rangle + bU_{\text{CNOT}}|10\rangle = a|00\rangle + b|11\rangle.$$

Here, the gate entangles the two qubits so that the target qubit ends up matching the control qubit basis state. The matrix representation of the CNOT transform is:

$$U_{\text{CNOT}} = \begin{pmatrix} I & 0 \\ 0 & X \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix},$$

where I is the identity matrix and X is the Pauli-X (bit-flip) matrix.

The CNOT gate is one example of a quantum gate, and on its own, it is not sufficient for quantum error correction. For this, more advanced quantum codes are needed, typically involving encoding and decoding schemes that use multiple quantum gates.

2.2 Examples of Quantum Codes

Quantum codes help us protect quantum information from errors by encoding logical qubits into multiple physical qubits. This section introduces two fundamental examples of quantum codes: the three-qubit flip code and the Shor code.

The Three-Qubit Flip Code

The three-qubit flip code is used to correct single-qubit bit-flip errors. It is the quantum analog to the classical repetition code of length 3. While the no-cloning theorem does not allow cloning physical quantum states, it does not prohibit the cloning of logical states. This can be done using the copying circuit, as shown in Figure 2.

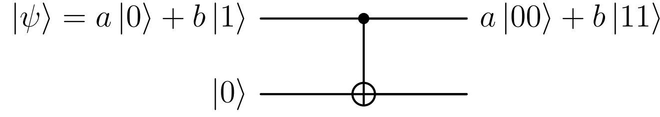

Figure 2. Copying circuit.

The copying circuit uses the CNOT gate with a fixed target qubit $|0\rangle$ to duplicate the control bit. Repeated use of this circuit allows us to create repetitions of basis states. For instance,

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \rightarrow \alpha|000\rangle + \beta|111\rangle.$$

These replicated qubits are then transmitted through a quantum channel, where some of them may flip.

The received message can be decoded using *syndrome decoding*. The decoder of the three-qubit flip code adds two additional qubits to the received message, which can be measured to find the syndrome. The measured syndrome determines if and where an error has occurred and dictates how to retrieve the original message. The actions corresponding to different measured syndromes are shown in Table 1. The complete quantum circuit for syndrome decoding can be seen in Figure 3.

Table 1. Syndrome decoding for the three-qubit flip code.

Syndrome	**Action**
00	Do nothing
10	Flip second bit
01	Flip third bit
11	Flip first bit

The computed syndrome consists of two classical bits s1 and s2. Based on the syndrome, the decoder computes an estimated error pattern (e_1, e_2, e_3). Each e_i tells us whether we need to apply a bit-flip (an X gate) to the corresponding qubit. If $e_i = 1$, the X gate is applied to fix the error. If $e_i = 0$, no correction is needed and the gate is skipped.

Let us say that the output of the quantum channel is $|101\rangle$. When computing the syndrome, two additional fixed qubits get added, resulting in $|101\rangle|00\rangle$. The state changes in the following way when moving through the four CNOT gates:

$$|101\rangle\,|00\rangle \xrightarrow{1} |101\rangle\,|10\rangle \xrightarrow{2} |101\rangle\,|10\rangle \xrightarrow{3} |101\rangle\,|11\rangle \xrightarrow{4} |101\rangle\,|10\rangle$$

After the four CNOT gates, the two added bits (the syndrome) can be measured. In this case, the syndrome is $|10\rangle$. The syndrome decoding table (see Table 1) shows that in

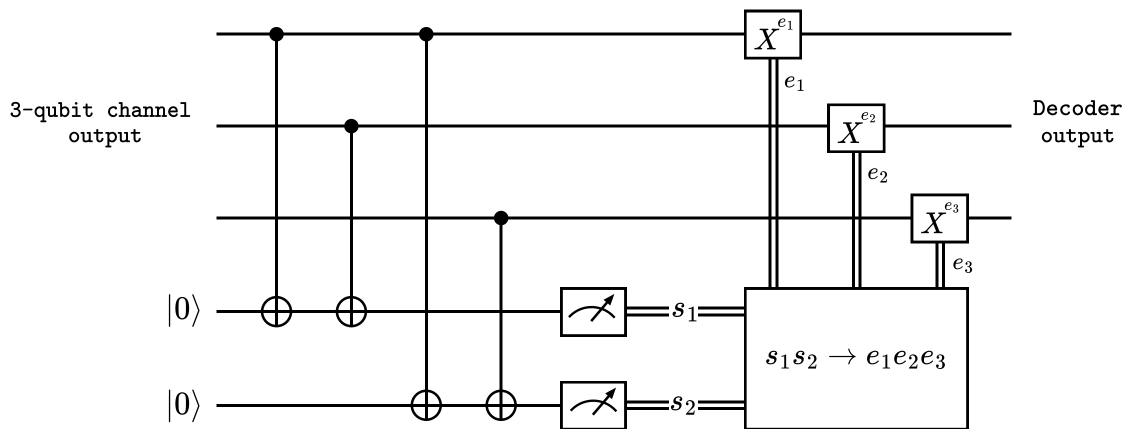

Figure 3. Quantum circuit for syndrome decoding.

order to recover the original message, the second bit of the received message must be flipped. When flipping the second bit of $|101\rangle$, we get the state $|111\rangle$, which would also be the output of the decoder. This tells us that the original message that was sent before repetition was $|1\rangle$.

The syndrome decoding in the three-qubit flip code is similar to how the classical repetition code uses a parity-check matrix to detect errors. In the classical case, the matrix checks for mismatches between pairs of bits to find the flipped one. The quantum version does the same using CNOT gates to compare qubits and store the result in syndrome bits, allowing us to locate and fix a single error.

The Shor Code

Whilst the three-qubit flip code can be used to correct bit-flip errors, it does not account for phase-flip errors Z and combined errors Y. The 9-qubit-code also know as the Shor code was introduced by P. W. Shor in 1995 [17]. The Shor code combines the three-qubit flip code with a three-qubit phase-flip code and is capable of protecting against both bit-flip and phase-flip errors. The complete encoder and decoder circuit for the Shor code can be seen in Figure 4.

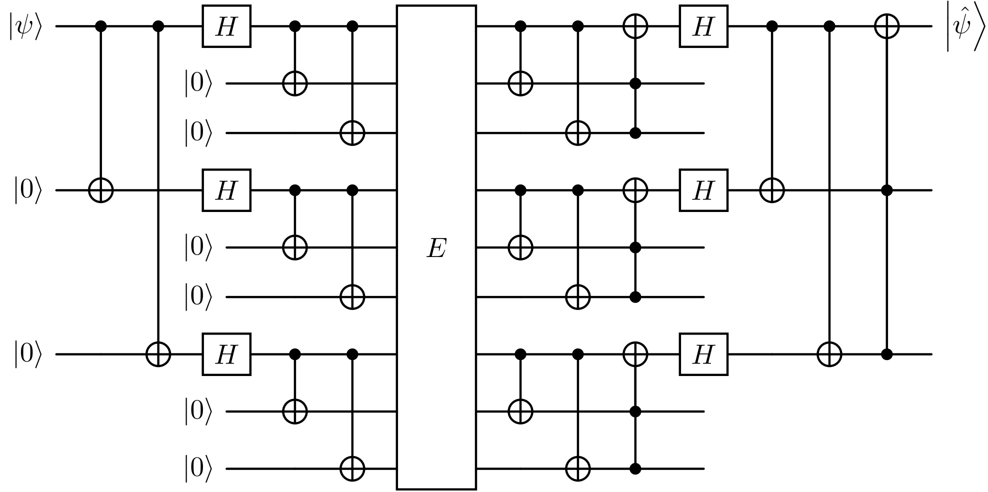

Figure 4. Quantum circuit for the Shor code.

The gate H shown in Figure 4 is the Hadamard gate, a single-qubit gate that transforms the basis states $|0\rangle$ and $|1\rangle$ into equal superpositions of $|0\rangle$ and $|1\rangle$. The matrix representation of the Hadamard gate is:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

The E gate in Figure 4 denotes the quantum channel through which quantum information is transmitted. Before a qubit is sent over the channel, it is encoded. The quantum channel can introduce both bit-flip and phase-flip errors. The decoder on the other side of the channel can detect and correct these errors.

The Shor code encoding-decoding scheme works by first encoding the input qubit $|\psi\rangle$ using the phase-flip code encoder. Next, each of the three encoded qubits is further encoded using a three-qubit bit-flip code, resulting in a total of nine qubits. The Hadamard gate H can be used to convert a bit-flip error into a phase error, and vice versa. This layered encoding scheme allows the Shor code to detect and correct both bit-flip and phase-flip errors. During decoding, the reverse process is applied: the bit-flip decoder — similar to the previously seen circuit in Figure 3 — is used first to correct potential bit-flip errors, followed by the phase-flip code decoder to correct phase errors.

The Shor code achieves error correction by encoding one logical qubit into nine physical qubits. While this redundancy can correct single errors, it comes at a cost in terms of qubit overhead. The high redundancy used by the Shor code motivates the search for more efficient quantum codes.

2.3 Stabilizer Codes

In the late 1990s, D. Gottesman introduced the stabilizer codes, which is the foundation that allows us to extend traditional linear codes to the quantum realm [18]. In this section, we start by explaining the idea of stabilizers, followed by a detailed look at stabilizer codes. After that, we introduce CSS codes as an example of this approach.

2.3.1 Stabilizers

Definition 2.1. A unitary gate U stabilizes a quantum state $|\psi\rangle$ if $U|\psi\rangle = |\psi\rangle$.

Definition 2.2. The Pauli group on n qubits, denoted $\hat{\Pi}^n$, is the set of all tensor products of n single-qubit Pauli matrices I, X, Y, and Z, each multiplied by a global factor of ± 1 or $\pm i$. Formally,

$$\hat{\Pi}^n = \{\omega P_1 \otimes P_2 \otimes \cdots \otimes P_n \mid \omega \in \{\pm 1, \pm i\},\ P_j \in \{I, X, Y, Z\}\}.$$

Definition 2.3. A stabilizer group $S = \{G\}$ is a subgroup of the Pauli group $\hat{\Pi}^n$. The vector space V stabilized by stabilizer S is the set of vectors stabilized by every group element S.

For example, the codewords for the 3-qubit bit-flip code are the following:

$$|0_L\rangle = |000\rangle, \quad |1_L\rangle = |111\rangle,$$

for logical qubits 0 and 1, respectively. The stabilizer group defining this code is

$$S = \{III,\ ZZI,\ IZZ,\ ZIZ\} \in \widehat{\Pi}^3.$$

We can confirm this by looking at which quantum states each member of the stabilizer group stabilizes. The group members, along with the states they stabilize, are shown in Table 2.

Table 2. Stabilizer sequences and corresponding quantum states.

Stabilizer sequence $G \in S$	States $v \in V$
III	All states
ZZI	$\|000\rangle, \|001\rangle, \|110\rangle, \|111\rangle$
ZIZ	$\|000\rangle, \|010\rangle, \|101\rangle, \|111\rangle$
IZZ	$\|000\rangle, \|100\rangle, \|011\rangle, \|111\rangle$

A stabilizer group can be further defined by a set of stabilizer group generators. The stabilizer group contains $n - k$ generators, where n is the number of physical qubits and

k is the number of encoded logical qubits. For our 3-qubit bit-flip code, $n = 3$ and $k = 1$, so the group can be represented by 2 generators. The set of generators for this code is

$$S = \{ZZI,\ IZZ\}.$$

Each member of the stabilizer group can be obtained as a product of the generators. The stabilizer group thus has 2^{n-k} members, since each generator can be either included or excluded from the product. Generator sets allow us to compactly represent stabilizer codes without needing to define the entire stabilizer group. The code space stabilized by these generators consists of all quantum states $|\psi\rangle$ such that $ZZI|\psi\rangle = |\psi\rangle$ and $IZZ|\psi\rangle = |\psi\rangle$.

Similar to classical error correction, where parity checks are used to verify whether a codeword is valid, stabilizers are used in quantum error correction. A codeword is valid if every stabilizer in the group leaves the quantum state unchanged. This means the entire quantum error correction code can be defined using the stabilizer group, much like a linear code can be defined by its parity-check matrix. By examining combinations of stabilizers that change the quantum state, one can determine the position of the error.

2.3.2 Stabilizer Codes

Any stabilizer can be described in terms of X and Z Pauli matrices. Let us consider a way to simplify the representation of stabilizer groups even further. For any length-n stabilizer U, the stabilizer's tensor product can be expressed using the following notation:

$$U = X_x Z_z,$$

where x and z are length-n binary indices, with ones indicating the positions of the X and Z entries in the tensor product. For example, the stabilizer $U = ZZXXI$ can be expressed as $U = X_{00110}Z_{11000}$.

As we know, stabilizer groups can be represented as sets of their generators. Additionally, we require stabilizer members to commute. It can be shown that operators $U = X_x Z_z$ and $V = X_{x'} Z_{z'}$ commute if and only if

$$(x, z') + (z, x') = \mathbf{0},$$

where (a, b) denotes the elementwise scalar product of a and b over GF(2).

We can represent full stabilizer groups, and thus also stabilizer codes, by expressing them as two $(n - k) \times n$ binary matrices. The first matrix in the pair stores the binary index vectors for the X-part of each stabilizer group generator, and the second matrix holds

the index vectors for the Z-part. These matrices are denoted as H_x and H_z, and the full stabilizer code can be represented by the $(n-k) \times 2n$ matrix

$$H = (H_x \mid H_z),$$

where H_x and H_z satisfy the following condition:

$$H_x H_z^T + H_z H_x^T = 0. \tag{1}$$

The analog to the classic linear code generator matrix is

$$G = (G_x | G_x),$$

where

$$H_x G_z^T + H_z G_x^T = 0.$$

For example, the binary matrix H storing the indices of the X and Z entries for each stabilizer group generator $S = \{ZZI,\ IZZ\}$ for the 3-qubit bit-flip code is shown below:

$$H_x = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad H_z = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}, \quad H = (H_x \mid H_z) = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

This representation shows that both stabilizers consist only of Z operators, hence the all-zero rows in H_x. The non-zero entries in H_z reflect the positions of the Z operators in each generator.

2.3.3 CSS Codes

Calderbank-Shor-Steane (CSS) codes are a class of stabilizer codes that allow the construction of quantum codes using classical linear codes. CSS codes were first introduced in 1996 by R. Calderbank and P. Shor [19], and independently by A. Steane [20].

CSS codes can be constructed from a pair of classical linear codes, C_1 and C_2, with parity-check matrices H_1 and H_2. These matrices must satisfy the orthogonality condition:

$$H_1 H_2^T = 0.$$

This ensures that all stabilizer generators commute, as required by Condition (1). A sufficient condition for this to hold is that one code is contained in the dual of the other, such as $C_2^\perp \subseteq C_1$ or $C_1^\perp \subseteq C_2$.

A CSS code can be constructed using the following block matrices:

$$H = \begin{pmatrix} H_2 & 0 \\ 0 & H_1 \end{pmatrix}, \quad G = \begin{pmatrix} G_1 & 0 \\ 0 & G_2 \end{pmatrix}, \tag{2}$$

where H_1 and H_2 are parity-check matrices of the classical codes C_1 and C_2, respectively, and G_1 and G_2 are their corresponding generator matrices.

Theorem 2.4. *Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be $[n, k_i, d_i]$ binary codes for $i = 1, 2$, and suppose that $\mathcal{C}_2^\perp \subset \mathcal{C}_1$. Then the matrices defined in equation* (2) *serve as parity-check and generator matrices for a $[[n,\ k_1 + k_2 - n,\ \min\{d_1, d_2\}]]$ quantum code $\mathcal{Q}$.*

A CSS code can correct both bit-flip and phase-flip errors, and it does so independently. Encoding and syndrome decoding are performed separately for each type of error. The syndrome of a quantum error $U = X_x Z_z$ is computed as follows:

$$s = H \begin{pmatrix} x^T \\ z^T \end{pmatrix} = \begin{pmatrix} H_2 & 0 \\ 0 & H_1 \end{pmatrix} \begin{pmatrix} x^T \\ z^T \end{pmatrix} = \begin{pmatrix} H_2 x^T \\ H_1 z^T \end{pmatrix} = \begin{pmatrix} s_X^T \\ s_Z^T \end{pmatrix}.$$

To conclude, CSS codes allow us to take pairs of linear codes and construct quantum codes that are capable of correcting phase and bit-flip errors independently. This opens the possibility of leveraging families of classical codes to build efficient quantum error-correcting codes. Among these, an interesting direction is the construction of quantum codes from pairs of classical QC-LDPC codes.

3 Construction of QQC-LDPC Codes

Quantum QC-LDPC (QQC-LDPC) codes are a class of CSS codes, where the CSS code is constructed from a pair of classical QC-LDPC codes. Thanks to their sparse structure, QC-LDPC codes support efficient decoding, while their quasi-cyclic form allows for compact representation of long codes. These properties make QC-LDPC codes especially attractive candidates for constructing quantum codes.

This chapter of the thesis focuses on the search for QQC-LDPC codes. First, we discuss which parameters should be taken into account when constructing QQC-LDPC codes. Next, we try to construct QQC-LDPC codes with good metrics ourselves. For this, we present two approaches. Lastly, we present some example codes we obtained.

3.1 Problem Statement

For a pair of classical linear codes C and D to be able to construct a CSS code, they must satisfy the following condition:

$$C^{\perp} \subset D \quad \text{or} \quad D^{\perp} \subset C.$$

This condition states that one of the codes, C or D, must include the dual of the other. It is known that quantum LDPC codes with length-4-cycles in their Tanner graphs result in poor decoding performance when using iterative decoding algorithms [21]. Therefore, we aim to construct codes with girth at least 6.

Taking this into account, when searching for good codes, we aim to construct QQC-LDPC codes from two labeled QC-LDPC matrices C and D such that the following two conditions are satisfied:

1. The codes C and D each have girth at least 6.
2. The codes C and D satisfy the *twisted condition*: either $D^{\perp} \subset C$ or $C^{\perp} \subset D$.

Other important properties to consider are the dimension and rate of the constructed QQC-LDPC code. Let H_C and H_D be length-n parity-check matrices of the QC-LDPC codes C and D, respectively. Then, the dimension of the quantum code is given by

$$k = n - \text{rank}(H_C) - \text{rank}(H_D).$$

The dimension k of the quantum code represents the number of logical qubits that can be encoded. The code length n corresponds to the number of physical qubits that are transmitted. In quantum coding theory, the notation $[[n, k]]$ is commonly used to denote

a quantum code with n physical and k logical qubits. The rate of a QQC-LDPC code, defined as the ratio of logical to physical qubits, can be calculated as

$$R = \frac{k}{n} = \frac{n - \text{rank}(H_C) - \text{rank}(H_D)}{n} = 1 - \frac{\text{rank}(H_C) + \text{rank}(H_D)}{n}.$$

A higher rate means that more actual quantum data, compared to redundant information, can be transmitted per codeword. Additionally, we should also consider the value of the lifting factor M. As the value of M increases, the length of the QC-LDPC codes grows, and thus the length of the resulting QQC-LDPC code also increases.

Amirzadea et al. [22] have shown that constructing QQC-LDPC codes from (J, L)-regular QC-LDPC codes with $J \geq 3$ results in Tanner graphs whose girths do not exceed 6. Therefore, when searching for (J, L)-regular QC-LDPC codes suitable for QQC-LDPC construction, we should focus on codes whose parity-check matrices contain only two ones per row (i.e., $J = 2$).

In addition, it would be useful to develop methods that make it possible to construct codes that are not fully connected or regular. In other words, methods that allow the base matrix to include zero entries would make it easier to explore more flexible code structures. The approaches proposed by Amirzadea et al. [22], as well as by Hagiwara and Imai [23], do not cover these cases and therefore cannot be applied in such situations.

3.2 Methodology

This section describes two approaches we tried for constructing QQC-LDPC codes. Both approaches start with a labeled matrix C whose parity-check matrix has girth ≥ 6, and attempt to construct a matrix D such that C and D satisfy Condition 2 and D also has girth ≥ 6. The first method is based on constructing matrix D from minimal weight codewords of code C. The second approach is based on two propositions highlighted by Hagiwara and Imai [23], using multiplicity-free and multiplicity-even checks to construct the pair of matrices.

3.2.1 Codeword-Based Construction

The initial idea was to see if we could construct matrix D from the minimal weight codewords of the code defined by matrix C. The steps for this approach are the following:

1. Select a labeled matrix C.
2. Find the generator matrix for C.

3. Generate all codewords of code C.
4. Select the minimum weight codewords from the generated set.
5. Find codewords whose M shifts form a parity-check matrix of the regular QC-LDPC code.
6. Construct matrix D from the codewords found in the previous step.

Using this approach, we were indeed able to find matrices that satisfy Conditions 1 and 2. Since the initial tests were successful, we decided to improve the algorithm.

The first issue we noticed was that we were storing all generated codewords along with their weights. This quickly became problematic as the number of codewords increased. To make the algorithm more memory-efficient, we modified it to keep track only of the minimum weight found so far, along with all codewords having that weight. If a codeword with a lower weight is encountered while iterating over all codewords, the list is cleared, and the minimum weight value is updated. In this way, we store in memory only the codewords with the lowest weight. For a 2×4 generator matrix with a lifting factor of $M = 5$, this meant storing only 30 codewords of length 20 in memory instead of 1024×20.

As the next step, we aimed to automate the search for good codeword pairs among all minimum weight codewords. Within the context of the algorithm, a pair of codewords is considered good if, when combined and permuted, each column of the resulting matrix has at least column weight 2. To support this, we implemented a function to check whether a given codeword is valid for labeling. A codeword of length n is considered valid if it can be divided into $\frac{n}{M}$ sections such that each section contains exactly one 1. If a codeword passes this test, it is marked as valid. For example, if $M = 5$ and the codeword length is $n = 20$ $(4M)$, the codeword must be splittable into 4 sections, each containing exactly one 1. A valid example would be: 0100 | 1000 | 0010 | 1000. While iterating over the generated codewords, we checked the validity of each one and kept only those that passed this check.

This approach worked well for 2×4 QC-LDPC codes. Next, we decided to try the algorithm with larger base matrices—for example, $2 \times L$, where $L = 6$, $L = 8$, and so on. While doing this, we quickly ran into the biggest problem with this approach: we are iterating over all possible codewords. If the base matrix and lifting factor are small (for example, 2×4 with $M = 5$), then the binary matrix would be of size 10×20, and we are iterating over $2^{10} = 1024$ codewords, which is reasonable. However, as the matrix size increases, the number of codewords to iterate over also grows exponentially. This means that searching for larger matrices is impractical with this approach and we should look for alternatives.

3.2.2 Construction via Multiplicity Constraints

The second approach for constructing a QQC-LDPC code from a pair of QC-LDPC codes that satisfy Conditions 1 and 2 is based on two propositions highlighted by Hagiwara and Imai [23]. The first proposition is stated as follows:

Proposition 3.1. *Let C be a (J, L, M)-QC LDPC code and D be a (K, L, M)-QC LDPC code. The codes C and D satisfy the twisted condition if and only if $c_j - d_k$ is multiplicity-even for any*

$$0 \leq j < J, \quad 0 \leq k < K.$$

We say that an integer sequence $x = (x_0, x_1, \ldots, x_{L-1})$ is *multiplicity-even* if every value in the sequence appears an even number of times. For example, the sequence $(0, 0, 1, 1, 2, 2)$ is multiplicity-even. Additionally, a vector is said to be *multiplicity-free* if every entry in the sequence is unique. For example, the sequence $(1, 2, 3, 4, 5)$ is multiplicity-free. The second proposition we use from Hagiwara and Imai is the following:

Proposition 3.2. *A necessary and sufficient condition for a QC-LDPC code C to have girth ≥ 6 is that $c_{j_1} - c_{j_2}$ is multiplicity-free for any*

$$0 \leq j_1 < j_2 < J.$$

To ensure that all multiplicity checks also account for rows containing zeros in the base matrix of the QC-LDPC code, that is, rows labeled with -1 in the labeled matrix, we represent these -1 labels using ∞ values. This distinction allows us to clearly differentiate between actual zeros in the base matrix and instances where a -1 arises as the difference between two row elements.

Using Propositions 3.1 and 3.2, we are able to construct a method for finding pairs of QC-LDPC codes that satisfy Conditions 1 and 2. The algorithm can largely be divided into two main steps:

1. Find a labeled matrix C for a QC-LDPC code with girth ≥ 6 (Condition 1).
2. From C, construct a labeled matrix D such that the twisted condition (Condition 2) is satisfied and the girth of code D is also ≥ 6 (Condition 1).

Step 1: Constructing matrix C

From Proposition 3.2, we know that in order for a QC-LDPC code to have girth ≥ 6, the difference between any pair of rows in the labeled matrix C must be multiplicity-free. We can use this condition to construct a suitable matrix for Step 1 of the algorithm.

We begin by defining two auxiliary functions that are essential for the algorithm: the first determines whether a vector is multiplicity-free, and the second computes the element-wise difference between two vectors.

To determine whether a vector is multiplicity-free, we count the number of unique elements it contains. One important consideration is that the input vectors are row vectors from a labeled matrix, which may include ∞ values. These special values must be handled separately. To do so, we first remove all ∞ values from the vector. Then, we count the number of unique elements among the remaining values. If the number of unique elements is equal to the number of remaining elements, the vector is considered multiplicity-free. This procedure is formalized in Algorithm 1.

Algorithm 1: Determine Whether a Vector Is Multiplicity-Free

Input: A vector $\mathbf{v} = (v_1, v_2, \ldots, v_L) \in \mathbb{R}^L \cup \{\infty\}$
Result: **true** if **v** is multiplicity-free, **false** otherwise

1 *finite* $\leftarrow \{v_i \in \mathbf{v} \mid v_i \neq \infty\}$
2 *unique* $\leftarrow \{x \in$ *finite* $\mid x$ occurs only once in *finite*$\}$
3 **if** $|$*unique*$| = |$*finite*$|$ **then**
4   **return true**
5 **else**
6   **return false**

To compute the difference between two vectors, we apply the following logic: if either element at a given position in the input vectors is ∞, then the corresponding position in the result vector is also set to ∞. The difference is only calculated when both elements at a given position are finite. This is formalized in Algorithm 2.

The implementation begins by preallocating the result vector with zeros. It then identifies all positions in the input vectors that contain ∞ and sets those positions to ∞ in the result vector. Finally, it locates the positions where both input elements are finite, computes their difference, and stores the result in the corresponding positions of the result vector.

Algorithm 2: Calculate Element-wise Difference Between Two Vectors

Input: Two vectors $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^L \cup \{\infty\}$
Result: A vector $\mathbf{r} \in \mathbb{R}^L \cup \{\infty\}$ such that $\mathbf{r}_i = \mathbf{v}_2[i] - \mathbf{v}_1[i]$, with special handling for ∞

```
1 r ← 0^L
2 infinite ← {i ∈ {1, ..., L} | v1[i] = ∞ ∨ v2[i] = ∞}
3 foreach i ∈ infinite do
4     r[i] ← ∞
5 finite ← {i ∈ {1, ..., L} | v1[i] ≠ ∞ ∧ v2[i] ≠ ∞}
6 foreach i ∈ finite do
7     r[i] ← v2[i] − v1[i]
8 return r
```

Using these algorithms, we can construct a method for generating labeled matrices whose corresponding codes have girth ≥ 6, the details of which are provided in Algorithm 3.

Algorithm 3: Generate a Multiplicity-Free Labeled Matrix with Girth ≥ 6

Input: Row count $J \in \mathbb{N}$, column count $L \in \mathbb{N}$, lifting factor $M \in \mathbb{N}$
Result: A labeled matrix $C \in (\{0, \ldots, M-1\} \cup \{\infty\} \cup \{-1\})^{J \times L}$ such that the code has girth ≥ 6

```
1 C ← −1^(J×L)
2 if zeros are permitted in the base matrix then
3     Select J distinct columns {c_1, ..., c_J} ⊆ {1, ..., L} uniformly at random;
4     for i ← 1 to J do
5         C[i, c_i] ← ∞
6 if FILL_MATRIX(C, 1, 1, J, L, M) then
7     return C
8 else
9     return ∅
```

The algorithm above calls a recursive helper function, which is responsible for populating the elements of the matrix. The recursive algorithm is presented in Algorithm 4.

The algorithm starts by constructing a $J \times L$ matrix where each element is set to -1, indicating unfilled positions. Optionally, if zeros are to be included in the base matrix of the code, then a set of ∞ values may be added to the matrix before filling begins. This

Algorithm 4: FILL_MATRIX — Recursive Backtracking Algorithm to Construct a Valid Labeled Matrix

Input: Current matrix C, current position (row, col), dimensions $J \times L$, lifting factor M

Result: **true** if a valid matrix is found, updates matrix C; otherwise **false**

```
1  while row ≤ J and C[row, col] = ∞ do
2      col ← col + 1
3      if col > L then
4          col ← 1
5          row ← row + 1
6  if row > J then
7      if C is a valid multiplicity-free matrix then
8          return true
9      else
10         return false
11 foreach val ∈ randperm(M) do
12     C[row, col] ← val
13     if IS_PARTIAL_SOLUTION_VALID(C, row) then
14         next_row ← row
15         next_col ← col + 1
16         if next_col > L then
17             next_col ← 1
18             next_row ← next_row + 1
19         if FILL_MATRIX(C, next_row, next_col, J, L, M) then
20             return true
21 C[row, col] ← −1
22 return false
```

can be done by selecting J random columns and placing one ∞ per column. Placing ∞ values first is a performance optimization we added to cut down the search space and avoid having to check if ∞ value positions are valid when recursively filling in the matrix.

The algorithm then recursively fills the rest of the matrix. It finds the next available position by moving rightward along the current row, skipping over ∞ values. If the end of a row is reached, it wraps to the beginning of the next. The base case of the recursion

is when the row index exceeds the matrix dimensions, at which point all positions have been filled. At this stage, the algorithm checks whether the matrix is multiplicity-free. If so, it returns `true`; otherwise, it returns `false` and continues the search.

To fill values, a random permutation from 0 to $M-1$ is generated. This ensures each run of the algorithm produces a different random matrix. The recursive step tries each value in the permutation, assigning it to the current position and checking whether the partial matrix remains valid. If the check passes, the function recurses to the next position.

The partial solution check is added to reduce the search space and avoid descending deep into recursion trees of invalid solutions, only to backtrack later. The check ensures that the current path could potentially lead to a valid solution and is not already invalid. This validation is performed by comparing the current row against all previously filled rows. It is only carried out if at least two positions in the current row have been assigned, as fewer values do not provide meaningful comparison. For each previous row, the algorithm computes the pairwise differences over the filled positions. If any row violates the multiplicity-free condition, the candidate value is rejected.

If no values in the permutation lead to a valid configuration for the current position, the algorithm backtracks by resetting that position to -1 and returning `false`. The preceding position then continues trying its remaining permutation values.

Step 2: Constructing matrix D from C

In the previous step, we generated a $J \times L$ labeled matrix C that satisfies Condition 1. The next step is to generate a $K \times L$ labeled matrix D such that Conditions 1 and 2 are satisfied. To keep the structure of the pairs of codes the same, we will focus on pairs where $J = K$. Once again, we can use the two propositions highlighted by Hagiwara and Imai [23]. The steps to construct a $J \times L$ labeled matrix D from C are the following:

1. Start with a $J \times L$ labeled matrix C and lifting factor M, defining a QC-LDPC code with girth ≥ 6.
2. Find all vectors that are multiplicity-even with all J row vectors of C. Formally, we are looking for vectors $x = (x_1, x_2, \ldots, x_L)$, where $x_i \in \{0, \ldots, M-1\}$. These vectors must satisfy Proposition 3.1.
3. Iterate over all such vectors and find collections of size J such that all vectors in the collection $\{d_j\}_{j=1}^{J}$ satisfy the multiplicity-free condition from Proposition 3.2. Formally, $d_{j_1} - d_{j_2}$ is multiplicity-free for any $0 \leq j_1 < j_2 < J$.
4. Construct matrix D from combinations of vectors found in the previous step. These vectors become the row vectors of the labeled matrix D. Since they are multiplicity-free with respect to each other, the girth of D is ≥ 6.

To additionally allow for zeros in the base graph of the generated matrix, we can allow the values of the labeled matrix to be selected from the range $\{-1, 0, 1, \ldots, M-1\}$ in step 2 of the algorithm, with -1s indicating the positions of the zeros in the base graph and the corresponding all-zero submatrices in the parity-check matrix of D.

To implement this algorithm, we first define a helper procedure that checks whether a vector is multiplicity-even. This procedure is presented in Algorithm 5. The algorithm begins by identifying all unique elements in the vector, excluding any occurrences of ∞. It then counts the frequency of each unique element and verifies whether each count is even. If any element appears an odd number of times, the procedure returns false; otherwise, the vector is considered multiplicity-even.

Algorithm 5: Verify Whether a Vector Is Multiplicity-Even

Input: A vector $s = (s_1, s_2, \ldots, s_L) \in \mathbb{R}^L \cup \{\infty\}$
Result: **true** if s is multiplicity-even, **false** otherwise

```
1 values ← unique(s) \ {∞}
2 foreach v ∈ values do
3     count ← #{i | s_i = v}
4     if count mod 2 ≠ 0 then
5         return false
6 return true
```

Using this helper, we can construct the algorithm for generating matrix D from C. The detailed algorithm is provided in Algorithm 6.

This algorithm first initializes an empty candidate set where potential candidate row vectors for matrix D are stored. Next, the algorithm loops over all vectors of length L and checks whether each vector is multiplicity-even with every row of matrix C. This can be verified using the helper function defined earlier in Algorithm 5. If the condition is satisfied, the vector is added to the set of potential candidate row vectors for matrix D.

The next step of the algorithm iterates over every combination of J vectors from the candidate set, constructs matrix D from these row vectors, and checks whether the rows are multiplicity-free with respect to each other. If the check is satisfied for any constructed D, a valid solution is returned. If no combination of J vectors from the candidate list yields a valid matrix D, the construction of a valid matrix is deemed not possible.

If we want to permit zeros in the base matrix of the constructed matrix D, we can generate the elements of the row vectors from the range -1 to $M-1$ instead of 0 to $M-1$. In this

Algorithm 6: Recursive Construction of a Matrix with Multiplicity Constraints

Input: A labeled matrix $C \in \mathbb{Z}^{J \times L}$, lifting factor $M \in \mathbb{N}$

Result: Matrix $D \in \mathbb{Z}^{J \times L}$ such that each row of D is multiplicity-even with all rows of C, and all rows of D are multiplicity-free with respect to each other

```
1  X ← ∅
2  foreach vector x = (x_1, ..., x_L) ∈ ({0, ..., M−1} ∪ {∞})^L do
3      if x is multiplicity-even with every row of matrix C then
4          X ← X ∪ {x}
5  foreach {x^(1), x^(2), ..., x^(J)} ∈ (X choose J) do
6      D ← [x^(1); ...; x^(J)]
7      if every pair (x^(i), x^(j)) is multiplicity-free for 1 ≤ i < j ≤ J then
8          return D
9  return ∅
```

1 $\mathcal{X} \leftarrow \emptyset$

2 **foreach** *vector* $x = (x_1, \ldots, x_L) \in (\{0, \ldots, M-1\} \cup \{\infty\})^L$ **do**

3   **if** x *is multiplicity-even with every row of matrix* C **then**

4         $\mathcal{X} \leftarrow \mathcal{X} \cup \{x\}$

5 **foreach** $\{x^{(1)}, x^{(2)}, \ldots, x^{(J)}\} \in \binom{\mathcal{X}}{J}$ **do**

6     $D \leftarrow \begin{bmatrix} x^{(1)} \\ \vdots \\ x^{(J)} \end{bmatrix}$

7   **if** *every pair* $(x^{(i)}, x^{(j)})$ *is multiplicity-free for* $1 \leq i < j \leq J$ **then**

8   **return** D

9 **return** $\emptyset$

case, the algorithm must also verify that every column and row of the constructed matrix D contains at least two non-∞ elements to ensure sufficient connectivity in the Tanner graph of the code. This additional check is optional and only applies when zero-valued entries are allowed in the base matrix.

3.3 Results

Using this approach, we were able to find new CSS codes for various choices of column count L and lifting factor M. Example results for codes with girth 8 are presented in Table 3.

Table 3. Example pairs of fully connected $2 \times L$ labeled matrices with girth $= 8$, lifting factor M, length n and dimension k.

L, M	$\mathbf{C}$	$\mathbf{D}$	$[[n, k]]$	rate
$4, 5$	$\begin{bmatrix} 0 & 1 & 4 & 2 \\ 1 & 3 & 4 & 1 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 & 1 \\ 0 & 2 & 5 & 2 \end{bmatrix}$	$[[20, 2]]$	0.10
$4, 5$	$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 3 & 3 & 0 \end{bmatrix}$	$\begin{bmatrix} 3 & 2 & 2 & 2 \\ 0 & 2 & 3 & 0 \end{bmatrix}$	$[[20, 2]]$	0.10
$6, 7$	$\begin{bmatrix} 2 & 0 & 3 & 3 & 2 & 0 \\ 6 & 0 & 2 & 6 & 0 & 2 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 2 & 1 & 5 & 0 & 4 \end{bmatrix}$	$[[42, 16]]$	0.38
$6, 7$	$\begin{bmatrix} 1 & 2 & 4 & 2 & 4 & 1 \\ 4 & 1 & 2 & 4 & 1 & 2 \end{bmatrix}$	$\begin{bmatrix} 4 & 3 & 1 & 3 & 1 & 4 \\ 3 & 1 & 4 & 4 & 3 & 1 \end{bmatrix}$	$[[42, 16]]$	0.38
$8, 9$	$\begin{bmatrix} 4 & 0 & 2 & 4 & 0 & 6 & 2 & 4 \\ 2 & 6 & 3 & 6 & 5 & 6 & 1 & 7 \end{bmatrix}$	$\begin{bmatrix} 3 & 1 & 2 & 6 & 0 & 7 & 1 & 6 \\ 0 & 3 & 2 & 2 & 3 & 2 & 0 & 4 \end{bmatrix}$	$[[72, 38]]$	0.53
$8, 9$	$\begin{bmatrix} 4 & 5 & 2 & 6 & 2 & 7 & 6 & 6 \\ 7 & 0 & 4 & 2 & 1 & 4 & 6 & 4 \end{bmatrix}$	$\begin{bmatrix} 5 & 4 & 2 & 4 & 0 & 8 & 5 & 6 \\ 7 & 1 & 5 & 4 & 1 & 6 & 4 & 2 \end{bmatrix}$	$[[72, 38]]$	0.53

As shown in Table 3, codes with girth 8 were successfully constructed for a variety of column counts $L = 4, 6, 8$ using corresponding lifting factors $M = 5, 7, 9$. For each case, multiple pairs of fully connected $2 \times L$ labeled matrices were found. The resulting codes have rates ranging from 0.10 for $L = 4$ to 0.53 for $L = 8$, showing the flexibility of the construction method for different parameter choices. It should be noted that these are just example codes, and each execution of the algorithm typically generates a new random pair of valid codes.

4 Discussion

In this section, we discuss the results of the two construction methods for QQC-LDPC codes. First, we briefly compare the methods, then discuss their limitations, and finally suggest directions for future improvement.

4.1 Comparison

Both proposed methods — the codeword-based and multiplicity constraint-based constructions — successfully generated QQC-LDPC code pairs with large girth (no cycles of length 4) and satisfied the CSS twisted condition (one code dual is contained in the other). This shows that multiple viable strategies exist for constructing quantum LDPC codes.

The **codeword-based construction** uses minimal-weight codewords of a QC-LDPC code C to derive matrix D. This approach naturally satisfies the twisted condition, as each parity-check in D corresponds to a codeword in C, ensuring $C^{\perp} \subseteq D$ (or vice versa). Selecting low-weight codewords helps maintain sparsity and large girth. This method worked well for small base matrices, though finding suitable codewords became harder as parameters L or M increased.

The **multiplicity constraint-based construction** applies conditions from Hagiwara and Imai [23] to systematically generate labeled matrices for C and D, ensuring both the twisted condition and absence of 4-cycles without brute-force search. Our results confirm the effectiveness of their theoretical framework.

It should be noted, however, that Hagiwara and Imai's method was limited to fully-connected, regular base matrices (every entry in the base matrix is non-zero and all rows/columns have equal weight). This limitation may exclude some valid large-girth solutions. We extended this approach to support irregular base matrices (allowing some zero entries), broadening the search space and allowing solutions previously overlooked.

Amirzade et al. [22] proposed an efficient algorithm for constructing QQC-LDPC codes from fully connected QC-LDPC codes with girth ≥ 8 and column weight 2. While their method is effective, it does not allow for searching of column weights larger than 2 and irregular structures — which is something our method is not restricted to.

Our approaches successfully found codes at small matrix sizes with girths comparable to those reported in recent literature [22, 23], which supports the correctness of our construction framework.

In summary, of the two proposed methods, the multiplicity-based construction offers a more principled and potentially scalable approach for generating QQC-LDPC codes.

4.2 Limitations

While the construction approaches proved effective for moderate-size examples, several limitations became evident. Algorithmic complexity is a primary concern for both methods. The codeword-based approach involves generating codewords of the classical code C and checking their suitability for D, which is exponential in complexity for large codes. Although we limited our search to minimal-weight codewords, even that subset becomes large as n increases, making the approach impractical for very long codes.

The multiplicity-based approach reduces the reliance on brute force by using mathematical constraints, but it in turn may require solving a complicated constraint satisfaction problem. As the size of the base matrix grows, ensuring that all difference sequences are multiplicity-free and pairs between C and D are multiplicity-even becomes combinatorially difficult. We did not rigorously analyze the time complexity of these algorithms in this thesis, but based on our observations, the run time grew quickly with L and M, indicating scalability challenges.

Additionally, our focus on fully connected base matrices meant that we explored a restricted set of structures. Although we did extend the second approach to allow zero entries in the base matrix, this was not exhaustively explored. There may be solutions with higher rate or larger girth in that irregular space, but finding them is non-trivial.

Finally, our evaluation of code quality was limited to girth, rate, and the satisfaction of the CSS condition. These are important metrics, but they do not fully guarantee excellent performance. A more complete assessment would require simulating code performance under a quantum error model, which was beyond the scope of this work.

4.3 Future Work

Based on the results of this thesis, there are several ways future research can improve the construction of QQC-LDPC codes and address current limitations:

- **Algorithmic Optimization:** Design more efficient algorithms for constructing the codes. For the codeword-based method, this could mean using smarter ways to search for low-weight codewords, such as heuristic or probabilistic techniques instead of checking all possibilities. For the multiplicity-based method, more advanced optimization or constraint-solving algorithms could help explore the search space more effectively.

- **Using Irregular and Larger Base Matrices:** Study base matrices that aren't regular — ones that have different numbers of ones in each row/column or include zeros. These irregular designs might lead to better codes. Also, testing the methods

with larger base matrices and higher lifting values M could lead to codes that are more suitable for real-world quantum systems.

- **Analyzing Complexity:** Take a closer look at how much time and memory the construction methods use. Understanding how the complexity grows with parameters like L and M can help find and fix performance issues.
- **Testing Code Performance:** Run simulations to see how well the constructed codes perform in practice. Use quantum or classical decoders to check how often they correct errors successfully and how quickly the decoding process works.

By exploring these directions, future work can possibly build on this thesis to create quantum QC-LDPC codes with large girths more efficiently and with improved error-correcting performance suitable for practical quantum systems.

Conclusion

The objective of this thesis was to develop systematic methods for constructing QQC-LDPC codes with high girth that satisfy the theoretical requirements of quantum coding. First, a theoretical foundation was established by reviewing relevant concepts from classical and quantum coding theory. Next, the properties of good QQC-LDPC codes were discussed, and the importance of girth in decoding performance was explained.

Two approaches to construct QQC-LDPC codes were proposed. In the first method, minimal-weight codewords were used to derive compatible matrix pairs. In the second method, a construction based on multiplicity conditions proposed by Hagiwara and Imai [23] was implemented and further developed to handle irregular base structures. Both approaches led to the discovery of QQC-LDPC code pairs with high girths, comparable to those found in recent literature.

In future work, the proposed methods could be enhanced by exploring algorithmic optimizations and applying them to irregular or larger base matrices. Additionally, performance testing and complexity analysis would help assess the practicality of these constructions in real-world quantum systems.

Overall, the goal set at the beginning of the thesis was achieved, and a practical contribution to the construction of QQC-LDPC codes that satisfy the necessary stabilizer and girth conditions was made.

References

[1] R. P. Feynman. Simulating Physics with Computers. *International Journal of Theoretical Physics*, 1982, Vol. 21, p. 467–488.

[2] P. W. Shor. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. USA: IEEE Computer Society, 1994.

[3] T. Monz, D. Nigg, E. A. Martinez, M. F. Brandl, P. Schindler, R. Rines, S. X. Wang, I. L. Chuang, and R. Blatt. Realization of a scalable Shor algorithm. *Science*, 2016, Vol. 351, p. 1068–1070.

[4] L. K. Grover. A fast quantum mechanical algorithm for database search. *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, p. 212–219.

[5] J. J. Goings, A. White, J. Lee, C. S. Tautermann, M. Degroote, C. Gidney, T. Shiozaki, R. Babbush, and N. C. Rubin. Reliably assessing the electronic structure of cytochrome P450 on today's classical computers and tomorrow's quantum computers. *Proceedings of the National Academy of Sciences*, 2022. https://doi.org/10.1073/pnas.2203533119 (14.05.2025).

[6] M. T. Nguyen, Y.-L. Lee, D. Alfonso, Q. Shao, and Y. Duan. Description of reaction and vibrational energetics of CO2–NH3 interaction using quantum computing algorithms. *AVS Quantum Science*, 2023. https://doi.org/10.1116/5.0137750 (13.05.2025).

[7] G. Greene-Diniz, D. Z. Manrique, W. Sennane, Y. Magnin, E. Shishenina, P. Cordier, P. Llewellyn, M. Krompiec, M. J. Rancic, and D. M. Ramo. Modelling carbon capture on metal-organic frameworks with quantum computing. *EPJ Quantum Technology*, 2022. https://doi.org/10.1140/epjqt/s40507-022-00155-w (13.04.2025).

[8] J. Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2018. https://doi.org/10.22331/q-2018-08-06-79 (02.05.2025).

[9] R. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 1962, Vol. 8, p. 21–28.

[10] R. Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, 1981, Vol. 27, p. 533–547.

[11] S. Lin and D. J. Costello. Error Control Coding: Fundamentals and Applications. Englewood Cliffs, NJ: Prentice-Hall. 1983.

[12] C. E. Shannon. A Mathematical Theory of Communication. *Bell Labs Technical Journal*, 1948, Vol. 27, p. 379–423.

[13] D.J.C. MacKay. Good error-correcting codes based on very sparse matrices. *IEEE Transactions on Information Theory*, 1999, Vol. 45, p. 399–431.

[14] T. Richardson and R. Urbanke. Modern Coding Theory. Cambridge: Cambridge University Press. 2008.

[15] F.R. Kschischang, B.J. Frey, and H.-A. Loeliger. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, 2001, Vol. 47, p. 498–519.

[16] M. Fossorier. Quasi-Cyclic Low-Density Parity-Check Codes From Circulant Permutation Matrices. *Information Theory, IEEE Transactions on*, Vol. 50, p. 1788–1793.

[17] P. W. Shor. Scheme for reducing decoherence in quantum computer memory. *Physical Review A*, 1995, Vol. 52, p. R2493–R2496.

[18] D. E. Gottesman. Stabilizer Codes and Quantum Error Correction. California Institute of Technology Division of Physics, Mathematics and Astronomy. Dissertation (Ph.D.). 1997. https://thesis.library.caltech.edu/2900/ (23.04.2025).

[19] A. R. Calderbank and P. W. Shor. Good quantum error-correcting codes exist. *Physical Review A*, 1996, Vol. 54, p. 1098–1105.

[20] A. Steane. Multiple-particle interference and quantum error correction. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 1996, Vol. 452, p. 2551–2577.

[21] Z. Babar, P. Botsinis, D. Alanis, S. X. Ng, and L. Hanzo. Construction of Quantum LDPC Codes From Classical Row-Circulant QC-LDPCs. *IEEE Communications Letters*, 2016, Vol. 20, p. 9–12.

[22] F. Amirzade, D. Panario, and M.-R. Sadeghi. Girth Analysis of Quantum Quasi-Cyclic LDPC Codes. *Problems of Information Transmission*, 2024, Vol. 60, p. 71–89.

[23] M. Hagiwara and H. Imai. Quantum Quasi-Cyclic LDPC Codes. *2007 IEEE International Symposium on Information Theory*. IEEE, 2007, p. 806–810.

Appendix

I. Sum-Product Algorithm

This appendix provides the step-by-step pseudocode of the sum-product algorithm used for decoding LDPC codes. A description of symbols and parameters used is provided at the end of the algorithm.

Algorithm 7: Sum-Product Algorithm

```
1  Initialization: Q_ji^(0)(0) = 1 − p_j,  Q_ji^(0)(1) = p_j,  t = 0 ;
2  while t < N_iter do
3      for i = 1 to r do
4          P_ij^(t)(0) = (1 + ∏_{h=1,h≠j}^{K} (1 − 2Q_hi^(t−1)(1))) /2 ;
5          P_ij^(t)(1) = 1 − P_ij^(t)(0) ;
6      for j = 1 to n do
7          Q_ji^(t)(0) = K_ij (1 − p_j) ∏_{k=1,k≠i}^{J} P_kj^(t)(0) ;
8          Q_ji^(t)(1) = K_ij p_j ∏_{k=1,k≠i}^{J} P_kj^(t)(1) ;
9      for j = 1 to n do
10         Q_j(0) = K_j (1 − p_j) ∏_{k=1}^{J} P_kj(0) ;
11         Q_j(1) = K_j p_j ∏_{k=1}^{J} P_kj(1) ;
12         if Q_j(1) > Q_j(0) then
13             x̂_j = 1
14         else
15             x̂_j = 0
16     if s = 0 then
17         break
18     t ← t + 1
19 return x̂
```

Terminology and Symbols:

- t represents the current iteration index;
- N_{iter} indicates the maximum number of allowed iterations;
- s refers to the syndrome vector;
- $p_i = P(x_i = 1 \mid \mathbf{y})$ denotes the probability, given the received vector $\mathbf{y}$, that the bit x_i equals 1;

- $P_{ij}^{(t)}(c)$ is the probability that $x_j = c, c \in \{0, 1\}$, as estimated by the i-th check node (associated with event S_i) at iteration t;
- $Q_{ji}^{(t)}(c)$ is the probability that $x_j = c, c \in \{0, 1\}$ as used by the i-th check node at the t-th iteration in the computation of the following expression:

$$\frac{P(x_i = 0 \mid \mathbf{y}, S)}{P(x_i = 1 \mid \mathbf{y}, S)} = \frac{(1 - p_i)}{p_i} \prod_{j=1}^{J} \frac{1 + \prod_{\substack{h=1 \\ h \neq i}}^{K} (1 - 2p_{jh})}{1 - \prod_{\substack{h=1 \\ h \neq i}}^{K} (1 - 2p_{jh})}$$

- K_{ij} are normalization factors chosen so that $Q_{ji}^{(t)}(0) + Q_{ji}^{(t)}(1) = 1$.

II. Licence

Non-exclusive licence to reproduce the thesis and make the thesis public

I, **Daglas Aitsen**,

1. grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the digital archives of the University of Tartu until the expiry of the term of copyright, my thesis

   Search for Good Quantum Error-Correcting Codes,

   supervised by Irina Bocharova;

2. grant the University of Tartu a permit to make the thesis specified in point 1 available to the public via the web environment of the University of Tartu, including via the digital archives, under the Creative Commons licence CC BY NC ND 4.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright;

3. am aware of the fact that the author retains the rights specified in points 1 and 2;

4. confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Daglas Aitsen
14/05/2025