Master Thesis

# Overview of Classes of MDP Convolutional Codes and Their Erasure Decoding Algorithms

**Author:**

*Yves Krähenbühl*

**Supervised by:**

*Prof. Dr. Joachim Rosenthal*

**Co-supervised by:**

*Jessica Bariffi*

Universität Zürich[UZH]

Department of Mathematics
University of Zurich
October 11, 2023

**Abstract**

Convolutional codes are a class of codes particularly well-suited for data transmission over erasure channels, commonly used in multimedia traffic over the internet. This thesis compares three erasure decoding algorithms for convolutional codes: The forward and backward decoding algorithm, the low delay decoding algorithm for modules, and the low delay decoding algorithm for linear systems. Our primary focus is on comparing these algorithms in terms of their delays, computational complexities, and erasure recovery capabilities. Additionally, we discuss various classes of maximum distance profile (MDP) convolutional codes, highlighting their advantageous properties in achieving an optimal performance with the three erasure decoding algorithms.

Through two simulations, we assess the performance of the decoding algorithms over different erasure channel models simulating practical scenarios. In a first simulation, we evaluate the recovery capability of the algorithms using their respective optimal convolutional code classes and compare them with maximum distance separable (MDS) linear block codes of the same rate. The results show that convolutional codes consistently match or even outperform MDS block codes over the different channel models.

In a second simulation, we use longer received codewords relative to the largest sliding window of a convolutional code, revealing that the recovery process is influenced not only by the quantity of erasures but also by their distribution. Notably, early erasures have a minimal influence on the remainder of the decoding process.

This thesis provides valuable insights into the selection and optimization of erasure decoding algorithms for convolutional codes, offering practical implications for multimedia data transmission over the internet.

**Acknowledgments**

First and foremost, I would like to thank my supervisor Prof. Dr. Rosenthal for giving me the opportunity to write my master thesis in the field of coding theory. I'm very thankful for the many interesting lectures he and his team held during my studies at the university.

Furthermore, I would like to thank Jessica Bariffi for her invaluable support, insightful advice, and the collaborative spirit she brought to this thesis. It was a pleasure to have her as my co-supervisor.

I also want to express my deepest gratitude to my friends and family for their unconditional support and encouragement. Without them, this thesis would not have been possible.

Thank you!

# Contents

**1 Introduction** **1**

**2 Introduction to Linear Block Codes** **5**
2.1 Definitions . . . . . 5
2.2 Distance Properties . . . . . 6

**3 Introduction to Convolutional Codes** **9**
3.1 Module Point of View . . . . . 9
3.1.1 Definitions . . . . . 9
3.1.2 Non-Catastrophic Convolutional Codes . . . . . 11
3.1.3 Distance Properties . . . . . 13
3.2 Linear System Point of View . . . . . 21

**4 Different Classes of MDP Convolutional Codes** **27**
4.1 Existence . . . . . 27
4.2 Construction . . . . . 29

**5 Erasure Decoding of Convolutional Codes** **39**
5.1 Erasure Decoding Algorithms . . . . . 39
5.1.1 Forward and Backward Decoding Algorithm . . . . . 39
5.1.2 Low Delay Decoding Algorithm for Modules . . . . . 46
5.1.3 Low Delay Decoding Algorithm for Linear Systems . . . . . 49
5.2 Comparison . . . . . 54
5.3 Simulation . . . . . 56

**6 Conclusion** **65**

**Bibliography** **67**

**A Sage Code** **71**

# Chapter 1

# Introduction

The publication of the paper "A mathematical theory of communication" by Claude E. Shannon [24] is considered to be the start of the field of coding theory. Since then, coding theory has grown rapidly and has found diverse applications in data transmission, data storage, and cryptography. It connects mathematics, computer science, engineering, and other disciplines. The central question of coding theory is as follows: Given a communication channel for data transmission or a storage device for data retention, how can we encode information to ensure efficient recovery in the event of corruption?

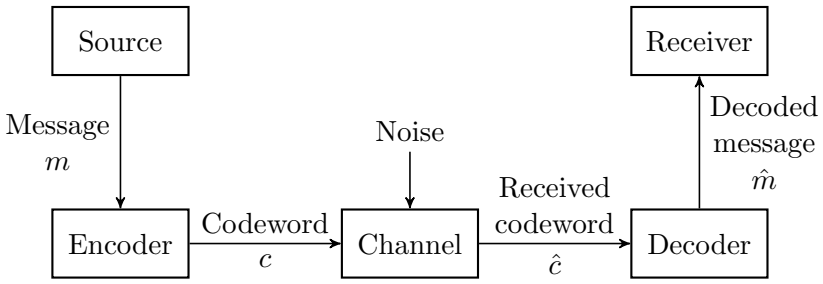


Figure 1.1: Communication model.

For data transmission, the simple communication model in Figure 1.1 demonstrates how information is transmitted from a *source* to a *receiver*. The principle is the same for data storage. At the source, a *message* is sent. It could be transmitted as it is. However, if the message becomes corrupted during transmission, it can be hard to recover it. Thus, we introduce some *redundancy* to the message, enhancing the chances of recovery even in the presence of some noise that distorts the original message. This redundancy is incorporated using an *encoder*. The encoded message is called a *codeword*, and a *code* is the set of all codewords. Subsequently, the codeword is sent through a *channel* where some *noise* may be added. Noise can appear in the form of an *error vector* which distorts the codeword, or as a scenario where each symbol either arrives correctly or does not arrive at all, referred to as an *erasure*. A channel model describes the distribution and properties through which the errors or erasures arise during the codeword transmission. The transmitted codeword with the possibly added noise is called *received codeword*. The receiver uses a *decoder*

to obtain the original message from the received codeword. The obtained message is called *decoded message*. If there is too much noise during the transmission, the decoded message might not be unique or not the same as the original message. As there is a one-to-one correspondence between the messages and codewords, the primary interest often lies in obtaining the codeword rather than the message.

There are various classes of codes that utilize different mathematical structures. One such example are linear block codes which form a linear subspace over a field with codewords all having the same fixed length. In this thesis, we will focus on another class of codes known as convolutional codes. These codes, first introduced by Peter Elias in 1955 [4], generalize linear block codes. They use polynomials to represent their codewords and can be viewed as submodules or linear systems. One notable advantage of convolutional codes lies in their effectiveness for decoding over erasure channels. They outperform linear block codes in this context by using a sliding window which gives greater flexibility in appropriately grouping erasures. As shown in [30], maximum distance profile (MDP) convolutional codes, meaning their column distances increase as rapidly as possible for as long as possible, often outperform equivalent maximum distance separable (MDS) linear block codes of the same rate and erasure correction capability. MDS linear block codes achieve the Singleton bound and therefore are optimal for erasure correction among linear block codes. Consequently, convolutional codes are particularly well-suited for transmission over erasure channels which are commonly used in multimedia traffic. The internet serves as a significant example of such a channel.

In this thesis, we look at three erasure decoding algorithms for convolutional codes. The first algorithm, known as the forward and backward decoding algorithm [30], combines forward decoding of a convolutional code with forward decoding of its reverse code. This algorithm was improved in [2], resulting in a low delay algorithm. This second algorithm is called low delay decoding algorithm for modules. The third algorithm, yet another low delay algorithm, uses a linear system representation of a convolutional code and was first introduced in [18]. Moreover, we provide an overview of different MDP convolutional code classes, including MDP, reverse MDP, and complete $j$-MDP. These classes have advantageous properties for optimizing the performance of the three decoding algorithms. A reverse MDP convolutional code is an MDP convolutional code whose reverse code is also MDP, while a complete $j$-MDP convolutional code has maximal column distances up to the $j$-th column distance.

Our main goal of this thesis is to analyse the differences between the algorithms concerning their delays, computational complexities, and erasure recovery capabilities. To gain a better understanding of their differences, we apply the algorithms in two simulations, comparing their recovery capabilities over different erasure channel models simulating practical scenarios.

In a first simulation, we choose the length of the received codewords to roughly match the size of the largest possible sliding window of the code. The simulation shows that the recovery capability achieved using the optimal class of convolutional code for each respective algorithm is as good, or even better, than that of com-

parable MDS linear block codes. Generally, the three algorithms have a similar performance. However, the low delay decoding algorithm for linear systems slightly outperforms the others, thanks to its ability to restart the decoding process once some symbols are declared as lost. Additionally, this algorithm has a lower computational complexity compared to the forward and backward decoding algorithm and the low delay decoding algorithm for modules.
In a second simulation, we use a larger codeword to emulate scenarios typical of internet streaming where codewords can be arbitrarily long. The performance in this simulation is generally worse, primarily because the largest sliding window is relatively small compared to the codeword size. We observe that the recovery capability of convolutional codes depends not only on the quantity of erasures but also on their distribution. Notably, erasures at the beginning of a received codeword, which we might declare as lost, do not substantially influence the remainder of the decoding process. As a result, a poor connection at the beginning of a transmission does not significantly affect the remainder of the transmission.

This thesis is structured as follows. In Chapter 2, we present preliminary definitions and distance properties of linear block codes. They will help to understand convolutional codes better. Chapter 3 establishes the primary definitions and distance properties for both the module and linear system point of view of convolutional codes. Additionally, we introduce classes of convolutional codes with optimal distance properties. In Chapter 4, we prove the existence of different classes of MDP convolutional codes and provide some of their constructions, which are closely tied to superregular matrices. In Chapter 5, we introduce the three decoding algorithms and show which classes of convolutional codes are optimal for each algorithm. We compare the algorithms taking into account aspects such as their delays, computational complexities, and erasure correction capabilities. Additionally, we present the results obtained from our practical simulations. Appendix A contains the Sage codes utilized for the simulations.

# Chapter 2

# Introduction to Linear Block Codes

We start with some background on a family of codes called linear block codes. This will be useful for a better understanding of convolutional codes, since they are a generalization of linear block codes. In this chapter, we provide some basic definitions and important distance properties of linear block codes, where we use [23, Chapter 1 and 2] as main reference. Additionally, throughout this thesis we let $\mathbb{F} = \mathbb{F}_q$ denote a finite field with $q$ elements.

## 2.1 Definitions

We begin with some basic definitions for linear block codes.

**Definition 2.1.1.** [23, Section 2.1] An $[n, k]$ *linear block code* $\mathcal{C}$ over a finite field $\mathbb{F}$ is a $k$-dimensional linear subspace of $\mathbb{F}^n$. The parameter $n$ is called the *code length* and $k$ is called the *code dimension*. The *code rate* $R$ is defined as

$$R = \frac{k}{n}.$$

Since a linear block code is a linear subspace of $\mathbb{F}^n$, it can be represented by a basis. Such a set of vectors forms a generator matrix.

**Definition 2.1.2.** [23, Section 2.1] Let $\mathcal{C}$ be an $[n, k]$ linear block code over $\mathbb{F}$. A *generator matrix* of $\mathcal{C}$ is a full rank $k \times n$ matrix $G$ whose rows form a basis of the code. This means,

$$\mathcal{C} = \text{rowspace}_{\mathbb{F}}(G) = \{mG \mid m \in \mathbb{F}^k\}.$$

**Definition 2.1.3.** [23, Section 2.3] Let $\mathcal{C}$ be an $[n, k]$ linear block code over $\mathbb{F}$. A *parity-check matrix* of $\mathcal{C}$ is a full rank $(n - k) \times n$ matrix $H$ such that

$$\mathcal{C} = \ker(H) = \{c \in \mathbb{F}^n \mid Hc^\top = 0\}.$$

Note that neither a generator matrix nor a parity-check matrix of a linear block code is unique. This is because there are many different bases for a subspace. Moreover, it follows that

$$HG^\top = 0 \quad \text{and} \quad GH^\top = 0.$$

As already mentioned in the introduction, we are interested in correcting a codeword with errors. First, we need to define a distance such that we can measure how much a vector $\hat{c} = c + e$, where $e$ is some error vector, differs from its original codeword $c$. There are various distances used for linear block codes such as the Hamming distance, Rank distance and Lee distance [6]. However, we will only need the most commonly used one, which is the Hamming distance. It calculates the number of differing positions between $c$ and $\hat{c}$.

**Definition 2.1.4.** [23, Section 1.3] The *Hamming weight* of a vector $c = (c_1, \ldots, c_n) \in \mathbb{F}^n$ is defined as

$$\mathrm{wt_H}(c) = |\{i \in \{1, \ldots, n\} \mid c_i \neq 0\}|.$$

The *Hamming distance* between two vectors $c_1, c_2 \in \mathbb{F}^n$ is defined as

$$\mathrm{d_H}(c_1, c_2) = \mathrm{wt_H}(c_1 - c_2). \tag{2.1}$$

It can easily be shown that the Hamming distance is indeed a distance.

**Definition 2.1.5.** [23, Section 1.3] The *minimum distance* of a linear block code $\mathcal{C}$ is defined as the minimal Hamming distance between any two distinct codewords of $\mathcal{C}$, denoted as

$$d \coloneqq d(\mathcal{C}) = \min\{\mathrm{d_H}(c_1, c_2) \mid c_1, c_2 \in \mathcal{C}, c_1 \neq c_2\}.$$

By (2.1) and since $\mathcal{C}$ is linear, we have

$$d = \min\{\mathrm{wt_H}(c) \mid c \in \mathcal{C} \setminus \{0\}\}.$$

Since the minimum distance of a code is important for its error correction capability, some literature includes the minimum distance $d$ in the code parameters. That is, an $[n, k]$ linear block code can also be denoted as an $[n, k, d]$ linear block code if its minimum distance is known.

## 2.2 Distance Properties

Now, after introducing the fundamental definitions for linear block codes, we look at some useful properties of their minimum distance. Since checking every distinct pair of codewords to get the minimum distance over a large code can be lengthy, Proposition 2.2.1 gives us a characterization of the minimum distance of a linear block code using a parity-check matrix.

**Proposition 2.2.1.** [23, Theorem 2.2] *Let $H$ be a parity-check matrix of an $[n, k]$ linear block code $\mathcal{C}$. Then, the minimum distance of the code is the largest integer $d$ such that every set of $d - 1$ columns of $H$ is linearly independent.*

*Proof.* Let $H = (h_1 \ldots h_n)$ be a parity-check matrix of $\mathcal{C}$ where $h_1, \ldots, h_n$ are the columns of $H$. Consider $c = (c_1, \ldots, c_n) \in \mathcal{C}$ with Hamming weight of $t > 0$. Let $I \subseteq \{1, \ldots, n\}$ denote the indices of the $t$ nonzero entries of $c$. Since $c$ is a codeword, we have

$$Hc^\top = \sum_{i \in I} h_i c_i = 0.$$

Thus, the columns of $H$ indexed by $I$ are linearly dependent.
Conversely, for every set of $t$ linearly dependent columns of $H$, there exists a nontrivial linear combination of these columns with at most $t$ nonzero coefficients, resulting in a nonzero codeword $c \in \mathcal{C}$.
Therefore, it follows for $d$ defined as in Proposition 2.2.1 that no nonzero codeword has Hamming weight less than $d$. Moreover, there is at least one codeword in $\mathcal{C}$ with a Hamming weight of $d$. □

We will now give the error detection, error correction and erasure correction capabilities of a linear block code which depends on the minimum distance of the code.

**Proposition 2.2.2.** [23, Proposition 1.3, 1.4 and 1.6] *Let $\mathcal{C}$ be an $[n, k, d]$ linear block code. Then the following hold:*

*(i) We can detect up to $d-1$ errors, meaning that for every $c \in \mathcal{C}$ and every $x \in \mathbb{F}^n$ with $\mathrm{d_H}(x, c) \leq d-1$ it follows that $x \notin \mathcal{C} \setminus \{c\}$.*

*(ii) We can correct up to $\left\lfloor \frac{d-1}{2} \right\rfloor$ errors, meaning that for every $x \in \mathbb{F}^n$ there exists at most one codeword $c \in \mathcal{C}$ such that $\mathrm{d_H}(x, c) \leq \left\lfloor \frac{d-1}{2} \right\rfloor$.*

*(iii) We can correct up to $d-1$ erasures.*

*Proof.* (i) Let $c \in \mathcal{C}$ and $x \in \mathbb{F}^n$ such that $\mathrm{d_H}(x, c) \leq d-1$ and $x \neq c$. By Definition 2.1.5, it follows that $x \notin \mathcal{C}$.

(ii) Let $c \in \mathcal{C}$ and $x \in \mathbb{F}^n$ such that $\mathrm{d_H}(x, c) \leq \left\lfloor \frac{d-1}{2} \right\rfloor$ and $x \neq c$. By contradiction, assume there exists $\hat{c} \in \mathcal{C} \setminus \{c\}$ such that $\mathrm{d_H}(x, \hat{c}) \leq \left\lfloor \frac{d-1}{2} \right\rfloor$. Since the Hamming distance is a distance, we have by the triangular inequality that

$$\mathrm{d_H}(c, \hat{c}) \leq \mathrm{d_H}(x, c) + \mathrm{d_H}(x, \hat{c}) \leq 2 \cdot \left\lfloor \frac{d-1}{2} \right\rfloor \leq d-1.$$

This implies that $c = \hat{c}$, which is a contradiction.

(iii) Let $c \in \mathcal{C}$ with at most $d-1$ erased symbols. Since every two distinct codewords in $\mathcal{C}$ differ in at least $d$ positions, there is exactly one codeword that agrees with $c$ on its correct positions.

□

We deduce from Proposition 2.2.2 that it is useful to have a large minimum distance for achieving a good correction performance. The Singleton bound gives an upper bound for the minimum distance depending on the code length and dimension.

**Proposition 2.2.3** (Singleton bound)**.** [25, Theorem 1] *If $\mathcal{C}$ is an $[n, k, d]$ linear block code, then*

$$d \leq n - k + 1.$$

*Proof.* Since a parity-check matrix $H$ of $\mathcal{C}$ has full rank, any $n-k+1$ columns of $H$ are linearly dependent. This implies $d \leq n-k+1$ by Proposition 2.2.1. □

Linear block codes that achieve the Singleton bound are of great interest because they represent codes with optimal correction performance. Such codes are heavily studied and are referred to as MDS codes. In the following chapters, we will explore their analogues in the field of convolutional codes.

**Definition 2.2.4.** [23, Section 4.1] An $[n, k, d]$ linear block code $\mathcal{C}$ is called *maximum distance separable (MDS)* if it achieves the Singleton bound, meaning that $d = n - k + 1$.

# Chapter 3

# Introduction to Convolutional Codes

Convolutional codes were first introduced by Peter Elias in 1955 [4]. He suggested that instead of using a constant matrix as generator matrix to use a polynomial matrix $G(z)$ whose entries belong to the polynomial ring $\mathbb{F}[z]$. Thus, they are a generalization of linear block codes. There is a multitude of definitions of convolutional codes. It has been shown in [21] that they are equivalent for all practical purposes. For this thesis, we will only focus on convolutional codes over a finite field with the module point of view and with the linear system point of view. This means that a convolutional code can be viewed as a submodule of $\mathbb{F}[z]^n$ or can be represented by a linear system. Convolutional codes are suitable for the continuous transmission over an erasure channel due to their erasure correction capabilities. They are often used for digital communication over the internet. The definitions and results of this chapter are based on [17, Chapter 1, 2 and 4].

## 3.1 Module Point of View

We start with the module point of view of convolutional codes, presenting the main definitions and results necessary for understanding this thesis.

### 3.1.1 Definitions

Recall that $\mathbb{F}[z]$ is a principal ideal domain (PID)[1]. Modules over a PID admit a basis and two different bases have the same number of elements, called the *rank* of the module.

**Definition 3.1.1.** [17, Subsection 1.2.1] A *convolutional code* $\mathcal{C}$ of *rate* $k/n$ is an $\mathbb{F}[z]$-submodule of $\mathbb{F}[z]^n$ of rank $k$. The parameter $n$ is called the *length* of the code. Then there exists a full rank matrix $G(z) \in \mathbb{F}[z]^{k\times n}$ such that

$$\mathcal{C} = \text{rowspace}_{\mathbb{F}[z]}(G(z)) = \{m(z)G(z) \mid m(z) \in \mathbb{F}[z]^k\}.$$

The polynomial matrix $G(z)$ is called a *generator matrix* of $\mathcal{C}$.

[1] A *principal ideal domain* is an integral domain in which every ideal is principal. [3, Section 8.2]

As for linear block codes, a generator matrix of a code is not unique. To show this, we introduce unimodular matrices.

**Definition 3.1.2.** [17, Subsection 1.2.1] A matrix $U(z) \in \mathbb{F}[z]^{k\times k}$ is called *unimodular* if there exists a matrix $V(z) \in \mathbb{F}[z]^{k\times k}$ such that

$$V(z)U(z) = U(z)V(z) = I_k.$$

We denote by $I_k$ the identity matrix of size $k$. It can be proven that a polynomial matrix $U(z) \in \mathbb{F}[z]^{k\times k}$ is unimodular if and only if $\det(U(z)) \in \mathbb{F} \setminus \{0\}$.

**Proposition 3.1.3.** [17, Subsection 1.2.1] *Two full rank matrices $G(z), \hat{G}(z) \in \mathbb{F}[z]^{k\times n}$ are generator matrices of the same convolutional code if and only if there exists a unimodular matrix $U(z) \in \mathbb{F}[z]^{k\times k}$ such that*

$$\hat{G}(z) = U(z)G(z). \tag{3.1}$$

If two generator matrices fulfil (3.1), we call $G(z)$ and $\hat{G}(z)$ *equivalent*. Hence, a generator matrix of a convolutional code is not unique.

Next, we show an advantage of convolutional codes compared to linear block codes. Assume we encode a message $m \in \mathbb{F}^k$ using a generator matrix $G$ of a linear block code mapping

$$m \mapsto mG = c \in \mathbb{F}^n.$$

The same can be done for convolutional codes. Here, a message $m(z) = \sum_{i=0}^{N} m_i z^i \in \mathbb{F}[z]^k$ is encoded as $c(z) = m(z)G(z) \in \mathbb{F}[z]^n$ with a generator matrix $G(z) = \sum_{i=0}^{\mu} G_i z^i$, where $\mu$ is the largest integer such that $G_\mu \neq 0$. Then $\mu$ is called the *memory* of $G(z)$. Since a generator matrix of a code is not unique, the memory is also not unique. By equating the coefficients, we have

$$\begin{aligned} c_0 &= m_0G_0, \\ c_1 &= m_0G_1 + m_1G_0, \\ c_2 &= m_0G_2 + m_1G_1 + m_2G_0, \\ &\vdots \end{aligned}$$

Note that each $c_i$ depends on $m_0, \ldots, m_i$. These additional dependencies compared to linear block codes create more protection against errors and erasures. We will use this for the decoding in Chapter 5. Moreover, it enables us to start sending a message before it is fully encoded, which is convenient for large messages. Similarly, we can already start decoding a received sequence $v(z) = \sum_{i=0}^{l} v_i z^i$ while still receiving some of the $v_i$'s.

**Definition 3.1.4.** [17, Subsection 1.2.1] Let $G(z) \in \mathbb{F}[z]^{k\times n}$ be a generator matrix of a convolutional code $\mathcal{C}$. The largest degree $\nu_i$ of any entry in the $i$-th row of $G(z)$ is called the $i$-th *row degree*. The sum of all row degrees of $G(z)$ is referred to as the *external degree*. The maximal degree of the full size minors of a generator matrix is called its *internal degree*. The internal degree of the generator matrices of $\mathcal{C}$ is called the *degree* $\delta$ of the code. A convolutional code of rate $k/n$ and degree $\delta$ is denoted as an $(n, k, \delta)$ convolutional code.

Throughout this thesis, we will refer to both the submatrix and its determinant as a minor. It is clear that $\nu_1 + \cdots + \nu_k \leq \delta$. If for some generator matrix $G(z)$ we have an equality, $G(z)$ is said to be *row-reduced*, and it is called a *minimal* generator matrix. Next, we show a characterization of minimal generator matrices.

**Proposition 3.1.5.** [17, Lemma 1.2.4] *Let $G(z) \in \mathbb{F}[z]^{k\times n}$ be a generator matrix with internal degree $\delta$ and row degrees $\nu_1, \ldots, \nu_k$. The highest row degree coefficient matrix $[G]_{hr}$ is defined as the matrix with the $i$-th row consisting of the coefficients of $z^{\nu_i}$ in the $i$-th row of $G(z)$. Then $\delta = \nu_1 + \cdots + \nu_k$ if and only if $[G]_{hr}$ has full rank.*

It can be shown that every convolutional code has a minimal generator matrix.

### 3.1.2 Non-Catastrophic Convolutional Codes

We have seen that for linear block codes parity-check matrices are useful to correct errors. The same holds for convolutional codes. However, not every convolutional code admits a parity-check matrix. In this subsection, we characterize the convolutional codes which have a parity-check matrix. These are the non-catastrophic convolutional codes. In practice, usually only non-catastrophic codes are used. We will illustrate the potential issue that can arise with a catastrophic convolutional code in Example 3.1.11.

**Definition 3.1.6.** [17, Subsection 1.2.1] Let $\mathcal{C}$ be an $(n, k, \delta)$ convolutional code. A full rank polynomial matrix $H(z) \in \mathbb{F}[z]^{(n-k)\times n}$ is called a *parity-check matrix* of $\mathcal{C}$ if

$$\mathcal{C} = \ker(H(z)) = \{c(z) \in \mathbb{F}[z]^n \mid H(z)c(z)^\top = 0\}.$$

If a convolutional code $\mathcal{C}$ has such a parity-check matrix, $\mathcal{C}$ is called *non-catastrophic*.

Again, for a generator matrix $G(z)$ and a parity-check matrix $H(z)$ of a convolutional code, we have

$$H(z)G(z)^\top = 0 \quad \text{and} \quad G(z)H(z)^\top = 0.$$

We will now introduce some key definitions before showing when a convolutional code is non-catastrophic.

**Definition 3.1.7.** [17, Definition 1.2.5] A polynomial matrix $G(z) \in \mathbb{F}[z]^{k\times n}$ with $k \leq n$ is *left-prime* if, in all factorizations $G(z) = L(z)\hat{G}(z)$, where $L(z) \in \mathbb{F}[z]^{k\times k}$ and $\hat{G}(z) \in \mathbb{F}[z]^{k\times n}$, the left factor $L(z)$ is unimodular.

**Definition 3.1.8.** [17, Definition 1.2.2] Let $G(z) \in \mathbb{F}[z]^{k\times n}$ with $k \leq n$. Then there exists a unimodular matrix $U(z) \in \mathbb{F}[z]^{n\times n}$ such that

$$G_{cH}(z) \coloneqq G(z)U(z) = \begin{pmatrix} g_{11}(z) & 0 & \ldots & 0 & 0 & \ldots & 0 \\ g_{21}(z) & g_{22}(z) & \ddots & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & 0 & \vdots & & \vdots \\ g_{k1}(z) & g_{k2}(z) & \ldots & g_{kk}(z) & 0 & \ldots & 0 \end{pmatrix},$$

where $g_{ii}(z)$ is a monic polynomial for all $1 \leq i \leq k$ such that $\deg(g_{ii}) > \deg(g_{ij})$ for $j < i$. The matrix $G_{cH}(z)$ is called the (unique) *row Hermite form* of $G(z)$.

Note that column operations change the code. Thus, $G_{cH}(z)$ often does not generate the same convolutional code as $G(z)$. Because verifying whether a polynomial matrix is left-prime using Definition 3.1.8 can be challenging, we provide simpler characterizations of a left-prime matrix.

**Theorem 3.1.9.** [14, 17] *Let $G(z) \in \mathbb{F}[z]^{k\times n}$ with $k \leq n$. Then the following are equivalent:*

*(i)* $G(z)$ *is left-prime.*

*(ii) The row Hermite form* $G_{cH}(z)$ *of* $G(z)$ *is* $\begin{pmatrix} I_k & 0 \end{pmatrix}$.

*(iii)* $G(z)$ *admits a polynomial* $n \times k$ *right inverse.*

*(iv)* $G(z)$ *can be completed to a unimodular matrix, meaning that there exists* $L(z) \in \mathbb{F}[z]^{(n-k)\times n}$ *such that* $\begin{pmatrix} G(z) \\ L(z) \end{pmatrix}$ *is unimodular.*

*(v) The ideal generated by all full size minors of* $G(z)$ *is* $\mathbb{F}[z]$.

*(vi)* $\operatorname{rk}(G(\lambda)) = k$ *for all* $\lambda \in \overline{\mathbb{F}}$, *where* $\overline{\mathbb{F}}$ *denotes the algebraic closure of* $\mathbb{F}$.

**Theorem 3.1.10.** [31, 17] *Let $\mathcal{C}$ be an $(n, k, \delta)$ convolutional code. Then $\mathcal{C}$ is non-catastrophic if and only if a generator matrix of $\mathcal{C}$ is left-prime.*

Note that if a generator matrix of a convolutional code is left-prime, then any equivalent generator matrix is left-prime. However, the same does not hold for parity-check matrices. In other words, the existence of a left-prime parity-check matrix does not necessarily imply that any other parity-check matrix of the same code is left-prime as well.

It is advantageous to work with non-catastrophic convolutional codes because they possess the property that a finite number of transmission errors can only result in a finite number of decoding errors. This, however, is not the case for catastrophic convolutional codes, as we demonstrate in Example 3.1.11.

**Example 3.1.11.** Consider the $(2, 1, 1)$ convolutional code $\mathcal{C}$ over $\mathbb{F}_2$ with generator matrix

$$G(z) = \begin{pmatrix} 1+z & 1+z \end{pmatrix} \in \mathbb{F}_2[z]^{1\times 2}.$$

Since $\operatorname{rk}(G(1)) = 0$, we know, by Theorem 3.1.9, that $G(z)$ is not left-prime. Therefore, $\mathcal{C}$ is a catastrophic convolutional code.
Assume we want to send the message

$$m(z) = \sum_{i=0}^{N} z^i$$

for some $N \in \mathbb{N}$. This message is encoded as

$$c(z) = m(z)G(z) = \begin{pmatrix} 1+z^{N+1} & 1+z^{N+1} \end{pmatrix}.$$

Note that the codeword $c(z)$ has four nonzero coefficients for any $N \in \mathbb{N}$, while $m(z)$ has $N + 1$. So, if during the transmission of $c(z)$ the nonzero coefficients

turned into zeros, meaning that four transmission errors occurred, we would falsely conclude that the original codeword was $c(z) = (0, 0)$. This leads to $N + 1$ decoding errors. As $N$ can be arbitrarily large, the potential impact on the decoded message is substantial.

### 3.1.3 Distance Properties

We have seen that the error and erasure correction capabilities for linear block codes depend on the minimum distance. This is also true for convolutional codes. We will again use the Hamming distance, as we did for linear block codes. The definition of the Hamming distance for polynomial vectors is analogous to its definition for constant vectors. Since the minimum distance has important characterizations with respect to its parity-check matrices, we will only consider non-catastrophic convolutional codes for this subsection.

**Definition 3.1.12.** [17, Definition 1.2.12] The *Hamming weight* of a polynomial vector $c(z) = \sum_{i=0}^{\deg(c(z))} c_i z^i \in \mathbb{F}[z]^n$ is the sum of the Hamming weights of its coefficients. That is,

$$\mathrm{wt_H}(c(z)) = \sum_{i=0}^{\deg(c(z))} \mathrm{wt_H}(c_i).$$

The *Hamming distance* between two polynomial vectors $c_1(z), c_2(z) \in \mathbb{F}[z]^n$ is defined as

$$\mathrm{d_H}(c_1(z), c_2(z)) = \mathrm{wt_H}(c_1(z) - c_2(z)).$$

**Definition 3.1.13.** [17, Definition 1.2.13] The *free distance* of a convolutional code $\mathcal{C}$ is the minimal Hamming distance between any two distinct codewords of $\mathcal{C}$, denoted as

$$d_{\mathrm{free}}(\mathcal{C}) \coloneqq \min\{\mathrm{d_H}(c_1(z), c_2(z)) \mid c_1(z), c_2(z) \in \mathcal{C}, c_1 \neq c_2\}.$$

Similarly to linear block codes, by linearity, we have

$$d_{\mathrm{free}}(\mathcal{C}) = \min\{\mathrm{wt_H}(c(z)) \mid c(z) \in \mathcal{C} \setminus \{0\}\}.$$

We can prove analogously as for linear block codes the error detection, error correction and erasure correction bound for convolutional codes.

**Proposition 3.1.14.** [17, Theorem 1.2.14] *Let $\mathcal{C}$ be a convolutional code with free distance $d$. Then the following hold:*

*(i) We can detect up to $d - 1$ errors.*

*(ii) We can correct up to $\left\lfloor \frac{d-1}{2} \right\rfloor$ errors.*

*(iii) We can correct up to $d - 1$ erasures.*

Another important concept related to distances for convolutional codes are column distances. These distances play a crucial role for the decoding properties of convolutional codes over erasure channels. This significance arises from the utilization of truncated codewords rather than complete codewords during the decoding process.

**Definition 3.1.15.** [17, Definition 1.2.17] Let $c(z) = \sum_{i=0}^{\deg(c(z))} c_i z^i \in \mathbb{F}[z]^n$. The *$j$-th truncation* of $c(z)$ is defined as $c_{[0,j]}(z) = c_0 + c_1 z + \cdots + c_j z^j$ for $j \in \mathbb{N}_0$.

Note that for $c(z) = \sum_{i=0}^{\deg(c(z))} c_i z^i \in \mathbb{F}[z]^n$ we have $c_i = 0$ if $i > \deg(c(z))$.

**Definition 3.1.16.** [17, Definition 1.2.18] For $j \in \mathbb{N}_0$, the *$j$-th column distance* of a convolutional code $\mathcal{C}$ is defined as

$$d_j^c := \min\{\mathrm{wt_H}(c_{[0,j]}(z)) \mid c_0 \neq 0\}$$

Clearly, we have $d_0^c \leq d_1^c \leq \cdots \leq d_{\mathrm{free}}(\mathcal{C})$. To calculate the column distances effectively, it is beneficial to truncate both the generator matrices and parity-check matrices.

**Definition 3.1.17.** [17, Subsection 1.2.2] Let $G(z) = \sum_{i=0}^{\mu} G_i z^i \in \mathbb{F}[z]^{k\times n}$ be a generator matrix and $H(z) = \sum_{i=0}^{\nu} H_i z^i \in \mathbb{F}[z]^{(n-k)\times n}$ a parity-check matrix of a convolutional code $\mathcal{C}$ of rate $k/n$. For $j \in \mathbb{N}_0$, we define the *sliding generator matrices* as

$$G_j^c := \begin{pmatrix} G_0 & G_1 & \dots & G_j \\ & G_0 & \dots & G_{j-1} \\ & & \ddots & \vdots \\ & & & G_0 \end{pmatrix} \in \mathbb{F}^{(j+1)k\times(j+1)n}$$

and the *sliding parity-check matrices* as

$$H_j^c := \begin{pmatrix} H_0 & & & \\ H_1 & H_0 & & \\ \vdots & \vdots & \ddots & \\ H_j & H_{j-1} & \dots & H_0 \end{pmatrix} \in \mathbb{F}^{(j+1)(n-k)\times(j+1)n},$$

where $G_i = 0$ for $i > \mu$ and $H_i = 0$ for $i > \nu$.

Using the sliding matrices we have the following properties for some codeword $c(z) = \sum_{i=0}^{\deg(c(z))} c_i z^i \in \mathcal{C}$ and some $j \in \mathbb{N}_0$:

$$(c_0, \dots, c_j) = (m_0, \dots, m_j) G_j^c \quad \text{and} \quad H_j^c (c_0, \dots, c_j)^\top = 0$$

for some $m_0, \dots, m_j \in \mathbb{F}^k$. Since $\mathcal{C}$ is non-catastrophic, $G_0$ has full rank by Theorem 3.1.9. This implies $c_0 \neq 0$ if and only if $m_0 \neq 0$. Consequently,

$$\begin{aligned} d_j^c &= \min_{m_0 \neq 0} \{\mathrm{wt_H}((m_0, \dots, m_j) G_j^c)\} \\ &= \min_{c_0 \neq 0} \{\mathrm{wt_H}(c_{[0,j]}(z)) \mid H_j^c (c_0, \dots, c_j)^\top = 0\}. \end{aligned} \tag{3.2}$$

**Proposition 3.1.18.** [9, Proposition 2.1] *Let $d \in \mathbb{N}_0$. Then the following are equivalent:*

*(i)* $d_j^c = d$.

*(ii) None of the first $n$ columns of $H_j^c$ is contained in the span of any other $d-2$ columns and one of the first $n$ columns of $H_j^c$ is contained in the span of some other $d-1$ columns of that matrix.*

*Proof.* Since we have a non-catastrophic convolutional code, we use the characterization of the column distances given in (3.2). Then, similar to linear block codes, the result follows. □

We will proceed to establish upper bounds for the free distance and the column distances. To do so, we will first introduce Lemma 3.1.19.

**Lemma 3.1.19.** [22, Lemma 2.1] *Let $G(z) \in \mathbb{F}[z]^{k\times n}$ be a row-reduced generator matrix with ordered row degrees $\nu_1 \geq \cdots \geq \nu_k$ and let $l$ be the number of indices $i$ such that $\nu_i = \nu_k$. Then*

$$d_{\text{free}}(\mathcal{C}) \leq n(\nu_k + 1) - l + 1.$$

*Proof.* Note that there exists a row-reduced generator matrix $G(z)$ with ordered row degrees, since row permutations do not change a code. As $[G]_{hr}$ has full rank, its rows are linearly independent. We transform $G(z)$ to $\hat{G}(z)$ with row operations and column permutations such that $[\hat{G}]_{hr}$ has the form

$$[\hat{G}]_{hr} = \left(\begin{array}{cc} L & \\ \hline I_l & M \end{array}\right),$$

where $L \in \mathbb{F}^{(k-l)\times n}$ and $M \in \mathbb{F}^{l\times(n-l)}$. With these operations we change the code to another code with the same free distance. Note that the last row of the new generator matrix $\hat{G}(z)$ has $l-1$ entries with degree strictly less than $\nu_k$ and the other entries have degree $\nu_k$.
The message $m(z) = (0, \ldots, 0, 1) \in \mathbb{F}[z]^k$ gives the codeword $c(z) = m(z)\hat{G}(z)$ with the new generator matrix. This codeword is the last row of $\hat{G}(z)$. Since an entry with degree $\nu_k$ has a Hamming weight of at most $\nu_k + 1$, we deduce

$$\text{wt}_{\text{H}}(c(z)) \leq (l-1)\nu_k + (n-l+1)(\nu_k+1) = n(\nu_k+1) - l + 1.$$

□

**Theorem 3.1.20.** *Let $\mathcal{C}$ be an $(n, k, \delta)$ convolutional code. Then*

*(i)* [22, Theorem 2.2] $d_{\text{free}}(\mathcal{C}) \leq (n-k)\left(\left\lfloor \frac{\delta}{k} \right\rfloor + 1\right) + \delta + 1$.

*(ii)* [9, Proposition 2.2] $d_j^c \leq (n-k)(j+1) + 1$ *for all* $j \in \mathbb{N}_0$.

*Proof.* (i) We prove the claim by maximizing the upper bound given in Lemma 3.1.19. The upper bound has the largest possible value if the smallest row degree $\nu_k$ is as large as possible and $l$ is as small as possible. Using that $\delta = \nu_1 + \cdots + \nu_k$, we note that $\nu_k$ is as large as possible if all the row degrees are as equal as possible, thus they have the values $\left\lfloor \frac{\delta}{k} \right\rfloor$ and $\left\lfloor \frac{\delta}{k} \right\rfloor + 1$. Hence,

$$\delta = l\left\lfloor \frac{\delta}{k} \right\rfloor + (k-l)\left(\left\lfloor \frac{\delta}{k} \right\rfloor + 1\right) = k\left(\left\lfloor \frac{\delta}{k} \right\rfloor + 1\right) - l,$$

which implies

$$l = k\left(\left\lfloor \frac{\delta}{k} \right\rfloor + 1\right) - \delta.$$

We conclude by substituting $\nu_k = \left\lfloor \frac{\delta}{k} \right\rfloor$ and $l = k\left(\left\lfloor \frac{\delta}{k} \right\rfloor + 1\right) - \delta$ into the bound provided by Lemma 3.1.19.

(ii) We view the subspace of vectors $c = (c_0, \ldots, c_j) \in \mathbb{F}^{(j+1)n}$ satisfying $H_j^c c^\top = 0$ as a linear block code with parity-check matrix $H_j^c \in \mathbb{F}^{(j+1)(n-k)\times(j+1)n}$. The result follows by applying the Singleton bound to this linear block code.

□

The bound given in (i) is called the *generalized Singleton bound*, since for $\delta = 0$ we have the Singleton bound for linear block codes.
We now introduce important classes of convolutional codes with optimal distance properties. In Chapter 4, we will show the existence and constructions of the different classes of MDP convolutional codes.

**Definition 3.1.21.** [17, Subsection 1.2.2] An $(n, k, \delta)$ convolutional code $\mathcal{C}$ is called *maximum distance separable (MDS)* if it achieves the generalized Singleton Bound, meaning that $d_{\text{free}}(\mathcal{C}) = (n-k)\left(\left\lfloor \frac{\delta}{k} \right\rfloor + 1\right) + \delta + 1$.
If $d_j^c$ meets the generalized Singleton bound for the smallest possible value of $j$, meaning that

$$d_M^c = (n-k)\left(\left\lfloor \frac{\delta}{k} \right\rfloor + 1\right) + \delta + 1,$$

where $M \coloneqq \left\lfloor \frac{\delta}{k} \right\rfloor + \left\lceil \frac{\delta}{n-k} \right\rceil$, then $\mathcal{C}$ is called *strongly maximum distance separable (sMDS)*.

**Lemma 3.1.22.** [9, Corollary 2.3] *Let $\mathcal{C}$ be an $(n, k, \delta)$ convolutional code. If $d_j^c = (n-k)(j+1)+1$, then $d_i^c = (n-k)(i+1)+1$ for all $i \leq j$.*

*Proof.* It suffices to prove the claim for $i = j-1$. By contradiction, assume there exists a codeword $c(z) \in \mathcal{C}$ such that $\mathrm{wt_H}(c_{[0,j-1]}(z)) \leq (n-k)j$ and $c_0 \neq 0$, i.e., $d_{j-1}^c < (n-k)j+1$. We have, for some $m_0, \ldots, m_j \in \mathbb{F}^k$, that

$$(c_0, \ldots, c_{j-1}, c_j) = (m_0 \ldots., m_{j-1}, m_j) G_j^c.$$

It follows that

$$c_j = (m_0, \ldots, m_j) \begin{pmatrix} G_j \\ G_{j-1} \\ \vdots \\ G_0 \end{pmatrix} = \sum_{i=0}^{j-1} m_i G_{j-i} + m_j G_0.$$

Then this implies

$$\mathrm{wt_H}(c_{[0,j]}(z)) = \mathrm{wt_H}(c_{[0,j-1]}(z)) + \mathrm{wt_H}(c_j) \leq (n-k)j + \mathrm{wt_H}(c_j).$$

Since $G_0$ has full rank, we can choose $m_j$ such that $c_j$ has $n-k$ nonzero entries. Thus,

$$\begin{aligned} \mathrm{wt_H}(c_{[0,j]}(z)) &\leq (n-k)j + \mathrm{wt_H}(c_j) \\ &\leq (n-k)j + (n-k) \\ &< (n-k)(j+1) + 1. \end{aligned}$$

This contradicts that $d_j^c$ meets the generalized Singleton bound. □

We are now interested in convolutional codes with column distances that increase as rapidly as possible for as long as possible. This is important for erasure decoding as we will see in Chapter 5. Note that

$$L \coloneqq \left\lfloor \frac{\delta}{k} \right\rfloor + \left\lfloor \frac{\delta}{n-k} \right\rfloor$$

is the largest possible integer such that the $j$-th column distance meets the generalized Singleton bound.

**Definition 3.1.23.** [17, Definition 1.2.23] An $(n, k, \delta)$ convolutional code $\mathcal{C}$ is called *maximum distance profile (MDP)* if for every $j \leq L$

$$d_j^c = (n-k)(j+1) + 1.$$

By Lemma 3.1.22, it is enough to show that $d_L^c = (n-k)(L+1)+1$ in order to prove that a convolutional code is MDP. We now state another result to characterize MDP convolutional codes. But first, we need the Lemma 3.1.24.

**Lemma 3.1.24.** [9, Lemma A.1] *Let $A \in \mathbb{F}^{k\times n}$ and $B \in \mathbb{F}^{n\times(n-k)}$ such that $AB = 0$, $\operatorname{rk} A = k$ and $\operatorname{rk} B = n-k$. Then the following are equivalent:*

*(i) The $k \times k$ full size minor of $A$ formed by columns with indices $1 \leq l_1 < \cdots < l_k \leq n$ is nonzero.*

*(ii) The $(n-k)\times(n-k)$ full size minor of $B$ formed by rows with indices $\{1, \ldots, n\} \setminus \{l_1, \ldots, l_k\}$ is nonzero.*

*Proof.* Without loss of generality, assume $(l_1, \ldots l_k) = (1, \ldots, k)$. We partition $A = \begin{pmatrix} A_1 & A_2 \end{pmatrix}$, where $A_1$ is the matrix formed by the columns of $A$ indexed by $1, \ldots, k$ and $A_2$ is formed by the columns indexed by $k+1, \ldots, n$. If $A_1$ is invertible, then

$$\ker(A) = \operatorname{colspace}_{\mathbb{F}} \begin{pmatrix} A_1^{-1}A_2 \\ -I_{n-k} \end{pmatrix} = \operatorname{colspace}_{\mathbb{F}}(B).$$

Hence, the last $n-k$ rows of $B$ form an invertible $(n-k)\times(n-k)$ matrix.
The other direction follows analogously. □

**Theorem 3.1.25.** [9, Theorem 2.4] *Let $\mathcal{C}$ be an $(n, k, \delta)$ non-catastrophic convolutional code. Then the following are equivalent:*

*(i) $d_j^c = (n-k)(j+1)+1$.*

*(ii) Every $(j+1)k \times (j+1)k$ full size minor of $G_j^c$ formed by columns with indices $1 \leq l_1 < \cdots < l_{(j+1)k} \leq (j+1)n$ which fulfil*

$$l_{sk+1} > sn \text{ for } s = 1, \ldots, j$$

*is nonzero.*

*(iii) Every $(j+1)(n-k) \times (j+1)(n-k)$ full size minor of $H_j^c$ formed by columns with indices $1 \leq l_1 < \cdots < l_{(j+1)(n-k)} \leq (j+1)n$ which fulfil*

$$l_{s(n-k)} \leq sn \text{ for } s = 1, \ldots, j$$

*is nonzero.*

*Proof.* We first prove the equivalence of (i) and (ii):
To show that (i) implies (ii), assume that $d_j^c = (n-k)(j+1)+1$. By contradiction, assume that there are some indices $1 \leq l_1 < \cdots < l_{(j+1)k} \leq (j+1)n$ satisfying $l_{sk+1} > sn$ for $s = 1, \ldots, j$ such that the corresponding minor of $G_j^c$ is zero. It follows that there exists some vector $m = (m_0, \ldots, m_j) \in \mathbb{F}^{(j+1)k}$ such that $mG_j^c \in \mathbb{F}^{(j+1)n}$ has zeros at positions indexed by $l_1, \ldots, l_{(j+1)k}$. We set

$$r \coloneqq \min\{i \mid m_i \neq 0\}.$$

Hence, $(m_r, \ldots, m_j)G_{j-r}^c \in \mathbb{F}^{(j-r+1)n}$ has weight at most $(j-r+1)(n-k)$ because the vector has at least $(j-r+1)k$ zero entries. Since $m_r \neq 0$, it follows $d_{j-r}^c \leq (n-k)(j-r+1)$. By Lemma 3.1.22, we obtain a contradiction since we assume that $d_j^c = (n-k)(j+1)+1$.
For the reverse direction, assume that $d_j^c \leq (n-k)(j+1)$. Let

$$r \coloneqq \min\{i \mid d_i^c \leq (n-k)(i+1)\}.$$

Therefore, there exists some vector $m = (m_0, \ldots, m_r) \in \mathbb{F}^{(r+1)k}$ with $m_0 \neq 0$ such that $mG_r^c$ has at least $k(r+1)$ zero entries. We choose the columns of $G_j^c$ corresponding to the first $k(r+1)$ position where $mG_r^c$ has a zero and augment it by the last $k(j-r)$ columns of $G_j^c$. We denote the indices of the columns by $l_1, \ldots, l_{(j+1)k}$. We claim that this $(j+1)k \times (j+1)k$ full size minor of $G_j^c$ is zero and that the columns satisfy $l_{sk+1} > sn$ for $s = 1, \ldots, j$. For the latter, we note that since $d_i^c \leq (n-k)(i+1)+1$ for $i \leq r-1$, $(m_0, \ldots, m_i)G_i^c$ has at most $k(i+1)-1$ zeros for $i \leq r-1$. This means

$$\begin{aligned} m_0 G_0^c &\text{ has at most } k-1 \text{ zeros, hence } l_k > n, \\ (m_0, m_1)G_1^c &\text{ has at most } 2k-1 \text{ zeros, hence } l_{2k} > 2n, \\ &\vdots \\ (m_0, \ldots, m_{r-1})G_{r-1}^c &\text{ has at most } rk-1 \text{ zeros, hence } l_{rk} > rn. \end{aligned}$$

This implies that $l_{sk+1} > sn$ for $1 \leq s \leq r$. For $s = 0$, we note that $l_1 > 0$ holds trivially. Moreover, the claim also holds for $s = r+1, \ldots, j$, since for each such $s$, $l_{sk+1}$ is among the remaining $k(j-r)$ columns. We are left to show that the minor is zero. For this, note that it has the form

$$\begin{pmatrix} A_1 & A_2 \\ 0 & A_3 \end{pmatrix},$$

where $A_1$ is an $(r+1)k \times (r+1)k$ submatrix of $G_r^c$ which is singular by construction. Hence, this full size minor is zero which is a contradiction.
To prove that (ii) and (iii) are equivalent, we note that $G_j^c(H_j^c)^\top = 0$ and that $G_j^c$ and $H_j^c$ have full rank. Since the minor of $H_j^c$ complementary to the minor of $G_j^c$ with the indices as in (ii) has indices as given in (iii) and vice versa, Lemma 3.1.24 completes the proof. □

In particular, if $j = L$, $\mathcal{C}$ is MDP. The full size minors described in (ii) and (iii) are called *not trivially zero* because, due to the structure of the matrix, these minors are the only ones that can possibly be nonzero. We now show how the introduced classes of convolutional codes relate to each other.

**Proposition 3.1.26.** [17, Remark 1.2.24] *Let $\mathcal{C}$ be an $(n, k, \delta)$ convolutional code. Then we have the following:*

*(i) If $\mathcal{C}$ is sMDS, then $\mathcal{C}$ is MDS.*

*(ii) If $(n-k) \mid \delta$, then $\mathcal{C}$ is sMDS if and only if $\mathcal{C}$ is MDP.*

The first property follows trivially. To prove the second property, we can use that in this case $L = M$ and compare the column distances.

We now introduce the reverse code of a convolutional code. This will be helpful for the erasure decoding in Chapter 5.

**Definition 3.1.27.** [11, Proposition 2.9] Let $\mathcal{C}$ be an $(n, k, \delta)$ convolutional code and let $G(z) = (g_{ij}(z))_{1 \le i \le k, 1 \le j \le n}$ be a row-reduced generator matrix of $\mathcal{C}$. Set $\overline{g}_{ij}(z) \coloneqq z^{\nu_i} g_{ij}(z^{-1})$, where $\nu_i$ is the $i$-th row degree of $G(z)$. Then the code $\overline{\mathcal{C}}$ with generator matrix $\overline{G}(z) = (\overline{g}_{ij}(z))_{1 \le i \le k, 1 \le j \le n}$ is also an $(n, k, \delta)$ convolutional code, called the *reverse code* of $\mathcal{C}$.

Hence, for polynomials of degree $l \in \mathbb{N}_0$, the following equivalence holds:

$$c_0 + c_1 z + \cdots + c_l z^l \in \mathcal{C}$$

if and only if

$$c_l + c_{l-1} z + \cdots + c_0 z^l \in \overline{\mathcal{C}}.$$

Note that $\mathcal{C}$ and $\overline{\mathcal{C}}$ have the same free distance due to the one-to-one correspondence between the codewords of $\mathcal{C}$ and $\overline{\mathcal{C}}$. However, the two codes may have different column distances since the truncations of $c(z) = \sum_{i=0}^{l} c_i z^i$ and $\overline{c}(z) = \sum_{i=0}^{l} c_{l-i} z^i$ involve different coefficients:

$$\begin{aligned} d_j^c(\mathcal{C}) &= \min_{c_0 \neq 0} \left\{ \sum_{i=0}^{j} \mathrm{wt_H}(c_i) \mid c(z) \in \mathcal{C} \right\}, \\ d_j^c(\overline{\mathcal{C}}) &= \min_{\overline{c}_0 \neq 0} \left\{ \mathrm{wt_H}(\overline{c}_{[0,j]}(z)) \mid \overline{c}(z) \in \overline{\mathcal{C}} \right\} \\ &= \min_{c_l \neq 0} \left\{ \sum_{i=0}^{l} \mathrm{wt_H}(c_{l-i}) \mid c(z) \in \mathcal{C} \right\}. \end{aligned}$$

**Remark 3.1.28.** Let $G(z) = \sum_{i=0}^{\mu} G_i z^i$ be a generator matrix and $H(z) = \sum_{i=0}^{\nu} H_i z^i$ be a parity-check of an $(n, k, \delta)$ convolutional code. If $(n-k) \mid \delta$, then

$$\overline{H}(z) = \sum_{i=0}^{\nu} H_{i-\nu} z^i$$

is a parity-check matrix of the reverse code. This follows, since $(n-k) \mid \delta$ implies that $H_\nu$ has full rank and thus $[H]_{hr} = H_\nu$, which is not generally the case. Similarly, if $k \mid \delta$, then $\overline{G}(z) = \sum_{i=0}^{\mu} G_{i-\mu} z^i$ is a generator matrix of the reverse code

**Definition 3.1.29.** [17, Definition 1.2.28] Let $\mathcal{C}$ be an MDP convolutional code. If $\overline{\mathcal{C}}$ is also MDP, then $\mathcal{C}$ is called *reverse MDP*.

**Definition 3.1.30.** [17, Definition 1.2.31] Let $\mathcal{C}$ be an $(n, k, \delta)$ convolutional code with a left-prime and row-reduced parity-check matrix $H(z) = \sum_{i=0}^{\nu} H_i z^i$. For $j \in \mathbb{N}_0$, we define the *partial parity-check matrices* as

$$\mathcal{H}_j := \begin{pmatrix} H_\nu & \dots & H_{\nu-j} & \dots & H_0 & & \\ & \ddots & \vdots & & \vdots & \ddots & \\ & & H_\nu & \dots & H_j & \dots & H_0 \end{pmatrix} \in \mathbb{F}^{(j+1)(n-k)\times(j+1)n},$$

where $H_i = 0$ for $i \notin \{0, \dots, \nu\}$.

**Definition 3.1.31.** [17, Definition 1.2.31] Let $(n-k) \mid \delta$ and let $\mathcal{C}$ be an $(n, k, \delta)$ convolutional code with a left-prime and row-reduced parity-check matrix $H(z) = \sum_{i=0}^{\nu} H_i z^i$, and $\nu = \frac{\delta}{n-k}$. Then $\mathcal{C}$ is called *complete $j$-MDP* if every not trivially zero full size minor of $\mathcal{H}_j$ is nonzero. A complete $L$-MDP convolutional code is usually just called *complete MDP*.

**Proposition 3.1.32.** [30, Remark 6.2] *A full size minor of $\mathcal{H}_j$ formed by columns with indices $l_1, \dots, l_{(n-k)(j+1)}$ is not trivially zero if and only if the following holds for all $s = 1, \dots, j$:*

*(i)* $l_{(n-k)s+1} > sn$,

*(ii)* $l_{(n-k)s} \leq sn + \nu n$.

*Proof.* Let us first show that if a full size minor of $\mathcal{H}_j$ is not trivially zero, then the two properties hold. By contradiction, assume that there exists some $s = 1, \dots, j$ such that one of (i) or (ii) does not hold. If (i) does not hold, then we have for some $s = 1, \dots, j$ that

$$l_{(n-k)s+1} \leq sn.$$

This means we select $s(n-k)+1$ columns from the first $s$ blocks. However, since there are at most $s(n-k)$ linearly independent columns in the first $s$ blocks, the chosen columns are linearly dependent and thus the corresponding full size minor is trivially zero, resulting in a contraction. A similar argument holds if property (ii) is not fulfilled.
Conversely, note that from (i) and (ii) it follows that we have no trivially zero minor in the triangular parts of $\mathcal{H}_j$. Therefore, the whole matrix $\mathcal{H}_j$ has no trivially zero minors. □

We require that $(n-k) \mid \delta$ in Definition 3.1.31, since $H_\nu$ has full rank if $(n-k) \mid \delta$. This implies that all $n-k$ row degrees of $H(z)$ are equal to $\nu$ and therefore, $\delta = (n-k)\nu$.

**Proposition 3.1.33.** [17, Remark 1.2.32] *Every complete MDP convolutional code is a reverse MDP and thus a MDP convolutional code.*

The result follows because the matrices $H_L^c$ and $\overline{H}_L^c$ are submatrices of the partial parity-check matrix $\mathcal{H}_L$. We note that for a complete MDP convolutional code, the not trivially zero full size minors of these submatrices are nonzero. Therefore, the code is both reverse MDP and MDP.

**Proposition 3.1.34.** [2, Lemma 3] *Let $j \in \{0, \dots, L\}$. If $\mathcal{C}$ is a complete $j$-MDP convolutional code, then $\mathcal{C}$ is also complete $i$-MDP for all $i \leq j$.*

*Proof.* It suffices to prove the claim for $i = j - 1$. Let $a \in \mathbb{F}$ be any not trivially zero full size minor of $\mathcal{H}_i$, and let $A \in \mathbb{F}^{(i+1)(n-k)\times(i+1)(n-k)}$ be the submatrix of $\mathcal{H}_i$ corresponding to the full size minor $a$, i.e., $\det(A) = a$. We denote the indices of the columns of $\mathcal{H}_i$ that form $A$ as $l_1, \dots, l_{(i+1)(n-k)}$. Let $b$ be the full size minor of $\mathcal{H}_j$ formed by the union of columns indexed by $l_1, \dots, l_{(i+1)(n-k)}$ and $n - k$ of the last $n$ columns of $\mathcal{H}_j$. Note that this minor is not trivially zero by construction. Since $\mathcal{C}$ is complete $j$-MDP, we deduce that $b$ is nonzero. We observe that $b = a \cdot det(D)$, where $D$ is an $(n - k) \times (n - k)$ submatrix of $H_0$. It follows that $a \neq 0$ and we conclude that $\mathcal{C}$ is complete $i$-MDP. □

As a short overview, Figure 3.1 shows the relations between the different characterizations of convolutional codes.

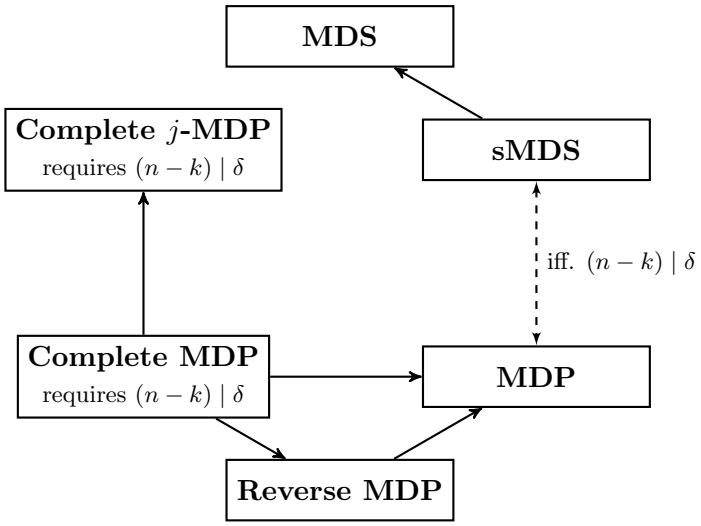


Figure 3.1: Relations between the classes of $(n, k, \delta)$ convolutional codes.

## 3.2 Linear System Point of View

In this section we give the linear system point of view of convolutional codes. The goal is to show the correspondence between convolutional codes and discrete-time linear systems.

**Definition 3.2.1.** [17, Section 1.4] For some $n, k, s \in \mathbb{N}$ and time $t \in \mathbb{N}_0$, a *(discrete-time) linear system* $\Sigma$ with *input vector* $u_t \in \mathbb{F}^k$, *state vector* $x_t \in \mathbb{F}^s$ and *output vector* $y_t \in \mathbb{F}^{n-k}$ and four *system matrices* $(A, B, C, D) \in \mathbb{F}^{s\times s} \times \mathbb{F}^{s\times k} \times \mathbb{F}^{(n-k)\times s} \times \mathbb{F}^{(n-k)\times k}$ is defined by the equations

$$\begin{aligned} x_{t+1} &= Ax_t + Bu_t, \\ y_t &= Cx_t + Du_t, \\ x_0 &= 0. \end{aligned} \tag{3.3}$$

We write $\Sigma = (A, B, C, D)$ for short. The triple of sequences

$$(\{u_t\}_{t\in\mathbb{N}_0}, \{x_t\}_{t\in\mathbb{N}_0}, \{y_t\}_{t\in\mathbb{N}_0})$$

is called the *trajectory* of the linear system $\Sigma$. If the three sequences all have finite Hamming weight, we call it a *finite-weight trajectory*.

In order to obtain the codewords of a convolutional code with respect to a linear system we will only consider finite-weight trajectories. We set for any finite-weight trajectory

$$\gamma := \max\{\text{length}\{u_i\}, \text{length}\{x_i\}, \text{length}\{y_i\}\},$$

where $\text{length}\{s_i\} := \max\{t \mid s_t \neq 0\} \in \mathbb{N}_0$. Then, going from a linear system to a convolutional code, we set

$$\begin{aligned} u(z) &:= \sum_{i=0}^{\gamma} u_{\gamma-i} z^i \in \mathbb{F}[z]^k, \\ x(z) &:= \sum_{i=0}^{\gamma} x_{\gamma-i} z^i \in \mathbb{F}[z]^s, \\ y(z) &:= \sum_{i=0}^{\gamma} y_{\gamma-i} z^i \in \mathbb{F}[z]^{n-k}. \end{aligned}$$

Thus, a trajectory fulfils the system (3.3) if and only if

$$\underbrace{\begin{pmatrix} zI_s - A & 0_{s,n-k} & -B \\ -C & I_{n-k} & -D \end{pmatrix}}_{=:E(z)} \begin{pmatrix} x(z) & y(z) & u(z) \end{pmatrix}^\top = 0, \tag{3.4}$$

where we denote by $0_{s,n-k} \in \mathbb{F}^{s\times(n-k)}$ the zero matrix.
The set of vectors $c(z) = \begin{pmatrix} y(z) \\ u(z) \end{pmatrix} \in \mathbb{F}[z]^n$ forms a submodule of $\mathbb{F}[z]^n$ of rank $k$ and, therefore, an $(n, k, \delta)$ convolutional code $\mathcal{C}$ with $s \geq \delta$ denoted by $\mathcal{C}(A, B, C, D)$. The system $\Sigma = (A, B, C, D)$ is called an *input-state-output (ISO) representation* of $\mathcal{C}$. An ISO representation with $s = \delta$ is called *minimal representation* of $\mathcal{C}$ and every code has such a representation.

Next, we show how to obtain a generator matrix from the characterization in (3.4). Note there exist polynomial matrices $L(z) \in \mathbb{F}[z]^{k\times s}$ and $G(z) \in \mathbb{F}[z]^{k\times n}$ such that

$$E(z) \big(L(z)G(z)\big)^\top = 0.$$

If $\big(L(z)G(z)\big)$ is left-prime, then $G(z)$ is a generator matrix of $\mathcal{C}$.
Conversely, we can show that for each convolutional code $\mathcal{C}$ of rate $k/n$, there exists $(A, B, C, D) \in \mathbb{F}^{s\times s} \times \mathbb{F}^{s\times k} \times \mathbb{F}^{(n-k)\times s} \times \mathbb{F}^{(n-k)\times k}$ such that $\mathcal{C} = \mathcal{C}(A, B, C, D)$. However, such an ISO representation is not unique as Proposition 3.2.2 shows.

**Proposition 3.2.2.** [17, Section 1.4] *Let $S \in \mathbb{F}^{s\times s}$ be an invertible matrix, then $\mathcal{C}(SAS^{-1}, SB, CS^{-1}, D) = \mathcal{C}(A, B, C, D)$.*

So far, we have seen how we can determine whether a convolutional code is non-catastrophic using the module point of view. However, it is also possible for the linear system point of view. We first need to define reachability and observability.

**Definition 3.2.3.** [17, Definition 1.4.1] Let $A \in \mathbb{F}^{s\times s}, B \in \mathbb{F}^{s\times k}$ and $C \in \mathbb{F}^{(n-k)\times s}$. We define the *reachability matrix* as

$$\mathcal{R}(A,B) \coloneqq \begin{pmatrix} B & AB & \dots & A^{s-1}B \end{pmatrix} \in \mathbb{F}^{s\times sk}.$$

The pair $(A,B)$ is called *reachable* if $\operatorname{rk}(\mathcal{R}(A,B)) = s$. We define the *observability matrix* as

$$\mathcal{O}(A,C) \coloneqq \begin{pmatrix} C \\ CA \\ \vdots \\ CA^{s-1} \end{pmatrix} \in \mathbb{F}^{(n-k)s\times s}$$

The pair $(A,C)$ is called *observable* if $\operatorname{rk}(\mathcal{O}(A,C)) = s$.

Now as we have introduced observability and reachability, we can use an ISO representation of a convolutional code to check whether the representation is minimal or if the code is non-catastrophic.

**Proposition 3.2.4.** [17, Theorem 1.4.3] *Let $\Sigma = (A,B,C,D)$ be an ISO representation of an $(n,k,\delta)$ convolutional code $\mathcal{C}$. Then $\Sigma$ is a minimal ISO representation of $\mathcal{C}$ if and only if $(A,B)$ is reachable. Moreover, the set of all minimal ISO representations of $\mathcal{C}$ is given by*

$$\left\{(SAS^{-1}, SB, CS^{-1}, D) \mid S \in GL_\delta(\mathbb{F})\right\}.$$

**Proposition 3.2.5.** [17, Theorem 1.4.4] *Let $\Sigma = (A,B,C,D)$ be a minimal ISO representation of an $(n,k,\delta)$ convolutional code $\mathcal{C}$, i.e., $(A,B)$ is reachable. Then $\mathcal{C}$ is non-catastrophic if and only if $(A,C)$ is observable.*

We proceed by providing an alternative characterization of MDP convolutional codes involving an ISO representation of the code. Before, we define $\mathcal{F}_0 \coloneqq D$ and for $j \geq 1$

$$\mathcal{F}_j \coloneqq \begin{pmatrix} D & & & \\ CB & \ddots & & \\ \vdots & \ddots & \ddots & \\ CA^{j-1}B & \dots & CB & D \end{pmatrix} \in \mathbb{F}^{(j+1)(n-k)\times(j+1)k}.$$

Note that we can rewrite the linear system equations given in (3.3) to obtain that $(y_0, u_0, \dots, y_j, u_j)$ represents the $j$-th truncation of a codeword if and only if

$$\left(-I_{(j+1)(n-k)} \;\middle|\; \mathcal{F}_j\right) \left(\begin{array}{c} y_0 \\ \vdots \\ y_j \\ \hline u_0 \\ \vdots \\ u_j \end{array}\right) = 0. \tag{3.5}$$

In [12, Lemma 2.3], an algebraic criterion for a trivially zero minor of $\mathcal{F}_j$ has been given. For this, assume that all entries of the nonzero blocks of $\mathcal{F}_j$ are nonzero. Then an $r \times r$ minor of $\mathcal{F}_j$ formed by rows with indices $1 \leq i_1 < \ldots < i_r \leq (j+1)(n-k)$ and by columns with indices $1 \leq j_1 < \ldots < j_r \leq (j+1)k$ is not trivially zero if and only if the indices satisfy

$$j_t \leq \left\lceil \frac{i_t}{n-k} \right\rceil \quad \text{for } t = 1, \ldots, r.$$

**Theorem 3.2.6.** [12, Theorem 2.4] *Let $\mathcal{C}$ be an $(n, k, \delta)$ convolutional code with ISO representation $(A, B, C, D)$. Then $\mathcal{C}$ has $j$-th column distance $d_j^c = (n-k)(j+1)+1$ if and only if every not trivially zero minor of $\mathcal{F}_j$ is nonzero.*

*Proof.* Assume that every not trivially zero minor of $\mathcal{F}_j$ is nonzero. We want to show that $d_j^c = (n-k)(j+1)+1$. Let

$$(y_0, u_0, \ldots, y_j, u_j)$$

be the $j$-th truncation of a codeword in $\mathcal{C}$ with $u_0 \neq 0$. Suppose $(u_0, \ldots, u_j)$ has Hamming weight $r$. This implies that at most $r-1$ rows of $\mathcal{F}_j$ lie in the left kernel of $(u_0, \ldots, u_j)^\top$, since every not trivially zero minor of $\mathcal{F}_j$ is nonzero. Consequently, by Equation (3.5), the weight of this truncated codeword is at least

$$r + (j+1)(n-k) - (r-1) = (j+1)(n-k) + 1.$$

Thus, we have $d_j^c \geq (j+1)(n-k)+1$. By Theorem 3.1.20, we conclude that $d_j^c = (j+1)(n-k)+1$.
For the reverse direction, we want to show that if there exists a not trivially zero minor of $\mathcal{F}_j$ that is zero, then $d_j^c < (n-k)(j+1)+1$. Without loss of generality, we can assume that the nonzero blocks of $\mathcal{F}_j$ do not have any zero entries. Otherwise, the result follows trivially. Suppose $\mathcal{F}_j$ has an $r \times r$ minor that is zero but not trivially zero, where $r \geq 2$.
If the $r$ rows of this minor belong to the left kernel of a vector of weight $r$, then we can form a nonzero vector

$$(u_0, \ldots, u_j)$$

of weight $r$ with $u_0 \neq 0$. As a result, the vector

$$(y_0, u_0, \ldots, y_j, u_j)$$

has a weight of at most $r + (j+1)(n-k) - r = (j+1)(n-k)$ by Equation (3.5).
Alternatively, if the $r$ rows of the minor belong to the left kernel of a nonzero vector of weight $r' < r$, the columns of the minor corresponding to the nonzero entries of the vector form an $r \times r'$ submatrix. To show that this submatrix contains a not trivially zero full size minor that is zero, we choose the last $r'$ rows of the submatrix. It follows that this minor is zero. Since we have assumed that the entries of the nonzero blocks of $\mathcal{F}_j$ are all nonzero, the algebraic criterion for the trivially zero minors of $\mathcal{F}_j$ implies that a minor is not trivially zero if and only if the last $r-i+1$ entries of its $i$-th column are nonzero. As the original $r \times r$ minor fulfils this, the constructed $r' \times r'$ minor also has this property. Therefore, we can construct a nonzero vector

$$(u_0, \ldots, u_j)$$

of weight $r'$ with $u_0 \neq 0$, leading to a vector

$$(y_0, u_0, \ldots, y_j, u_j)$$

of weight at most $r' + (j+1)(n-k) - r' = (j+1)(n-k)$ by Equation (3.5). □

Thus, for $j = L$ we have a characterization of an MDP convolutional code.

We will use the linear system point of view of convolutional codes to prove the existences and provide constructions of MDP convolutional codes in Chapter 4 as well as for the erasure decoding in Subsection 5.1.3.

# Chapter 4

# Different Classes of MDP Convolutional Codes

In this chapter, we provide the existence proofs for the different classes of MDP convolutional codes introduced in Chapter 3 and present some of their constructions. Our focus will be on MDP, reverse MDP and complete $j$-MDP convolutional codes due to their advantageous properties for erasure decoding, which we will explore further in Chapter 5. Additionally, we introduce superregular matrices, as they are closely linked to these constructions.

## 4.1 Existence

We start with the proofs of existence of the different MDP convolutional code classes. Since the focus of this thesis does not lie on MDS and sMDS convolutional codes, we will not prove their existences. The proof of the existence of MDS convolutional codes can be found in [22] and for sMDS convolutional codes in [11]. To establish the existence of the other classes, we will use the fact that the set of all convolutional codes forms a quasi-projective variety[2], which can be viewed as a Zarisky open subset[3] of the projective variety described in [20] and [22]. Our goal is to show that all codes within each class form a nonempty Zariski open set of this quasi-projective variety of convolutional codes, from which the existence follows. In Section 4.2, we will use the constructions to obtain a lower bound of the required field size for the existences of the classes.

**Theorem 4.1.1.** [12, Theorem 2.10] *For any positive integers $n, k$ and $\delta$ where $k < n$, there exists an $(n, k, \delta)$ MDP convolutional code over a sufficiently large field.*

*Proof.* Let $\overline{\mathbb{F}}$ be the algebraic closure of $\mathbb{F}$. By Theorem 3.2.6, the set of all $(n, k, \delta)$ MDP convolutional codes can be represented by the set of 4-tuples of matrices $(A, B, C, D) \in \mathbb{F}^{\delta\times\delta} \times \mathbb{F}^{\delta\times k} \times \mathbb{F}^{(n-k)\times\delta} \times \mathbb{F}^{(n-k)\times k}$ such that the not trivially zero

[2]Let $\mathbb{P}^n$ be the $n$-dimensional projective space over $\mathbb{F}$. A *projective variety* is an irreducible algebraic set in $\mathbb{P}^n$ with the induced topology. An open subset of a projective variety is a *quasi-projective variety.* [10, Section 1.1]

[3]The *Zariski topology* is a topology defined on $\mathbb{F}^n$, where its open subsets are the complements of the algebraic sets. [10, Section 1.1]

minors of $\mathcal{F}_L$ are nonzero. Based on the minors of $\mathcal{F}_L$ that must be nonzero, we have a finite set of polynomial equations. The zeroes of each of these polynomials define a proper algebraic subset of $\overline{\mathbb{F}}^{(L+1)(n-k)k}$. Thus, the complements of these subsets are nonempty Zariski open sets in $\overline{\mathbb{F}}^{(L+1)(n-k)k}$. The finite intersection of these complements is again a nonempty Zariski open set. Therefore, the set of all $(n, k, \delta)$ MDP convolutional codes forms a nonempty Zariski open set of the quasi-projective variety of convolutional codes. Thus, it follows that there exists an $(n, k, \delta)$ MDP convolutional code over a sufficiently large field. □

To prove the existence of reverse MDP convolutional codes we use similar arguments as for the proof of the existence of MDP convolutional codes. Recall that every reverse MDP convolutional code is MDP.

**Theorem 4.1.2.** [30, Theorem 4.6] *For any positive integers $n, k$ and $\delta$ where $k < n$, there exists an $(n, k, \delta)$ reverse MDP convolutional code over a sufficiently large field.*

*Proof.* Let $\overline{\mathbb{F}}$ be the algebraic closure of $\mathbb{F}$. As demonstrated in the proof of Theorem 4.1.1, the set of all $(n, k, \delta)$ MDP convolutional codes forms a nonempty Zariski open subset in $\overline{\mathbb{F}}^{(L+1)(n-k)k}$. We used that we can write this subset as a finite intersection of nonempty Zariski open sets, which we denote as $\{W_i\}_{i=0}^{l}$ for some $l \in \mathbb{N}$.
Similarly, we can characterize the convolutional codes whose reverse codes are MDP. It follows by the same arguments as in the proof of Theorem 4.1.1 that the set of all $(n, k, \delta)$ convolutional codes whose reverse code is MDP forms a nonempty Zariski open subset in $\overline{\mathbb{F}}^{(L+1)(n-k)k}$. This subset is a finite intersection of Zariski open subsets, which we denote as $\{U_i\}_{i=1}^{m}$ for some $m \in \mathbb{N}$.
Let $V$ denote the intersection of the subsets that define MDP convolutional codes and those that characterize convolutional codes whose reverse codes are MDP:

$$V \coloneqq (\cap_{i=0}^{l} W_i) \bigcap (\cap_{i=0}^{m} U_i) .$$

As the intersection of finitely many Zariski open sets is a Zariski open set, $V$ is a nonempty Zariski open set. We note that $V$ describes the set of all $(n, k, \delta)$ reverse MDP convolutional codes. Consequently, an $(n, k, \delta)$ reverse MDP convolutional code exists over a sufficiently large field extension of $\mathbb{F}$. □

To prove the existence of complete MDP convolutional code over a sufficiently large field, we again use similar arguments as in the proofs of Theorem 4.1.1 and Theorem 4.1.2.

**Theorem 4.1.3.** [15, Theorem 3.5] *For any positive integers $n, k$ and $\delta$ where $k < n$ and $(n - k) \mid \delta$, there exists an $(n, k, \delta)$ complete MDP convolutional code over a sufficiently large field.*

*Proof.* Let $\overline{\mathbb{F}}$ be the algebraic closure of $\mathbb{F}$. We know that the set of non-catastrophic $(n, k, \delta)$ convolutional codes with a parity-check matrix $H(z)$ whose row degrees are all equal to $\nu = \frac{\delta}{n-k}$ forms a Zariski open subset. Moreover, consider the subset of all polynomial matrices $H(z) \in \overline{\mathbb{F}}[z]^{(n-k)\times n}$ with $\deg(H(z)) \leq \nu$ such that they belong to a complete MDP convolutional code. In order words, the subset of all convolutional codes such that the not trivially zero full size minors of their

partial parity-check matrices $\mathcal{H}_L$ are nonzero and $\deg(H(z)) \leq \nu$. With the same argumentation as for MDP and reverse MDP convolutional codes, we know that the set of such parity-check matrices $H(z)$ forms a nonempty Zariski open set in $\overline{\mathbb{F}}^{(\nu+1)(n-k)n}$. Therefore, there exists a sufficiently large field $\mathbb{F}$ for which an $(n, k, \delta)$ complete MDP convolutional code exists. □

Since every complete MDP convolutional code is complete $j$-MDP for $j \leq L$, the existence of complete $j$-MDP convolutional codes follows from the existence of complete MDP convolutional codes.

**Corollary 4.1.4.** *Let $n, k$ and $\delta$ be positive integers such that $k < n$ and $(n-k) \mid \delta$. Let $j \leq L$. Then there exists an $(n, k, \delta)$ complete $j$-MDP convolutional code over a sufficiently large field.*

## 4.2 Construction

We now look at the construction of the different classes of MDP convolutional codes. To start, we introduce lower triangular superregular matrices which play a pivotal role as their structure can be used to construct the different MDP convolutional codes.

**Definition 4.2.1.** [17, Definition 1.3.4] Let $A \in \mathbb{F}^{l \times l}$ be a lower triangular Toeplitz matrix

$$A = \begin{pmatrix} a_1 & & & \\ a_2 & a_1 & & \\ \vdots & \ddots & \ddots & \\ a_l & \dots & a_2 & a_1 \end{pmatrix}.$$

The matrix $A$ is called a *(lower triangular) superregular* matrix if all its not trivially zero minors are nonzero.

The not trivially zero minors of a lower triangular matrix are defined as follows: For some $r = 1, \dots, l$, let

$$1 \leq i_1 < \cdots < i_r \leq l \quad \text{and} \quad 1 \leq j_1 < \cdots < j_r \leq l.$$

We denote $A^{i_1,\dots,i_r}_{j_1,\dots,j_r}$ as the submatrix of $A$ obtained by intersecting the rows indexed by $i_1, \dots, i_r$ and the columns indexed by $j_1, \dots, j_r$. If, for all $t = 1, \dots, r$, the inequality $j_t \leq i_t$ holds, the determinant of the submatrix is not trivially zero.

We will now present a construction method for superregular matrices. This method will later be useful to construct complete MDP convolutional codes.

**Lemma 4.2.2.** [9, Theorem B.1] *For $a, b \in \mathbb{N}$ with $b < a$, define*

$$X \coloneqq \begin{pmatrix} 1 & & & \\ 1 & 1 & & \\ & \ddots & \ddots & \\ & & 1 & 1 \end{pmatrix} \in \mathbb{Z}^{a \times a}$$

*and thus*

$$X^b = \begin{pmatrix} 1 & & & & & & \\ \binom{b}{1} & \ddots & & & & & \\ \vdots & & \ddots & & & & \\ \binom{b}{b-1} & & & \ddots & & & \\ 1 & \ddots & & & \ddots & & \\ & \ddots & 1 & \binom{b}{b-1} & \cdots & \binom{b}{1} & 1 \end{pmatrix}.$$

*Define* $\hat{X} \coloneqq (X^b)^{i_1,\dots,i_r}_{j_1,\dots,j_r}$ *for some* $r \in \{1,\dots,a\}$ *where* $i_1 < \cdots < i_r$ *and* $j_1 < \cdots < j_r$. *Then,*

*(i)* $\det(\hat{X}) \geq 0$.

*(ii)* $\det(\hat{X}) > 0$ *if and only if* $j_l \in \{i_s - b, \dots, i_s\}$ *for all* $s = 1, \dots, r$.

By setting $b = a - 1$, we have that all not trivially zero minors are nonzero. Consequently, when considering the matrix $X^{a-1}$ over a finite field with sufficiently large characteristic, it is a superregular matrix. We can use the bound provided in Theorem 4.2.4, which we will introduce later to determine the necessary field characteristic.

**Example 4.2.3.** We present some examples of superregular matrices over finite fields using Lemma 4.2.2:

$$\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 2 & 1 \end{pmatrix} \in \mathbb{F}_5^{3\times 3}, \quad \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 5 & 1 & 0 & 0 & 0 & 0 \\ 10 & 5 & 1 & 0 & 0 & 0 \\ 10 & 10 & 5 & 1 & 0 & 0 \\ 5 & 10 & 10 & 5 & 1 & 0 \\ 1 & 5 & 10 & 10 & 5 & 1 \end{pmatrix} \in \mathbb{F}_{23}^{6\times 6}.$$

The construction based on Lemma 4.2.2 requires the matrix entries to belong to a finite field with a large characteristic. Generally, the challenge of constructing superregular matrices is that the required field size can significantly exceed the minimum field size for the existence of a superregular matrix. In [13], the authors established a lower bound for the necessary field size for the existence of a superregular matrix.

**Theorem 4.2.4.** [13, Theorem 3.3] *Let* $\mathbb{F}$ *be a finite field such that*

$$|\mathbb{F}| > \frac{1}{2}\left(\frac{1}{l}\binom{2(l-1)}{l-1} + \binom{l-1}{\lfloor \frac{l-1}{2} \rfloor}\right) =: B_l.$$

*Then there exists an* $l \times l$ *superregular matrix over* $\mathbb{F}$.

However, since the given lower bound $B_l$ grows faster than the minimum required field size, the bound is not tight for larger superregular matrices.

We now introduce reverse-superregular matrices, which will be instrumental for the construction of reverse MDP convolutional codes.

**Definition 4.2.5.** [30, Definition 5.3] A lower triangular superregular matrix $A \in \mathbb{F}^{l\times l}$ is called *reverse-superregular* if the matrix

$$A_{\text{rev}} = \begin{pmatrix} a_l & & & \\ a_{l-1} & a_l & & \\ \vdots & \ddots & \ddots & \\ a_1 & \dots & a_{l-1} & a_l \end{pmatrix}$$

is superregular.

Clearly, not every superregular matrix is reverse-superregular. This property holds only for superregular matrices that are symmetric with respect to the lower diagonal, as in Example 4.2.3. To illustrate this further, we present an example where we do not have symmetry with respect to the lower diagonal.

**Example 4.2.6.** Let $\alpha$ be a primitive element of $\mathbb{F}_{2^3}$. Then the matrices

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \alpha^4 & 1 & 0 & 0 \\ \alpha^6 & \alpha^4 & 1 & 0 \\ \alpha^3 & \alpha^6 & \alpha^4 & 1 \end{pmatrix}, \quad A_{\text{rev}} = \begin{pmatrix} \alpha^3 & 0 & 0 & 0 \\ \alpha^6 & \alpha^3 & 0 & 0 \\ \alpha^4 & \alpha^6 & \alpha^3 & 0 \\ 1 & \alpha^4 & \alpha^6 & \alpha^3 \end{pmatrix}$$

are superregular. Thus, both $A$ and $A_{\text{rev}}$ are reverse-superregular matrices.

In [30], a method to obtain reverse-superregular matrices over a finite field using computer search has been presented. The authors did not specify a minimum required field size. However, it is clearly larger than the one for superregular matrices given in Theorem 4.2.4 as reverse-superregular matrices are superregular.

### MDP Convolutional Code

We look at the construction of MDP convolutional codes given in [13]. The approach for this construction is derived from the constructive proof of the existence of sMDS convolutional codes for $(n-k) \mid \delta$, as presented in [9]. However, the arguments can be extended to the case when $(n-k) \nmid \delta$ and therefore this construction works for all parameters.
In [9], the existence of sMDS convolutional codes is divided into three steps. Firstly, the authors show that we can construct a systematic sliding parity-check matrix $\hat{H}$ such that the corresponding code is sMDS. Such a matrix can be constructed using a sufficiently large superregular matrix. The second step involves proving the existence of a convolutional code for which $\hat{H}$ is a systematic sliding parity-check matrix. Lastly, it is checked that the constructed convolutional code indeed has degree $\delta$.

Similar to the first step, we construct an ISO representation of an $(n, k, \delta)$ MDP convolutional code with the help of a superregular matrix.

**Theorem 4.2.7.** [13, Section 2] *Let $r$ be the remainder of $\delta$ divided by $n-k$. Let $\mathcal{F}$ be a superregular matrix with dimension*

$$\begin{cases} (L+1)(n-1) \times (L+1)(n-1) & \text{if } (n-k) \mid \delta, \\ ((L+1)(n-1)+k+r-1) \times ((L+1)(n-1)+k+r-1) & \text{else.} \end{cases}$$

*For $j \in \{0, \ldots, L\}$, define the $(n-k) \times k$ matrices*

$$F_j := \mathcal{F}_{1,\ldots,k}^{j(n-1)+k,\ldots,(j+1)(n-1)}.$$

*If $r = 0$, the sequence of matrices $\{F_1, \ldots, F_L\}$ has a minimal partial realization $(A, B, C) \in \mathbb{F}^{\delta\times\delta} \times \mathbb{F}^{\delta\times k} \times \mathbb{F}^{(n-k)\times\delta}$. This means that $CA^{i-1}B = F_i$ for all $i = 1, \ldots, L$. Otherwise, additionally define*

$$F_{L+1} := \left( \begin{array}{c} \mathcal{F}_{1,\ldots,k}^{(L+1)(n-1)+k,\ldots,(L+1)(n-1)+k+r-1} \\ \hline * \end{array} \right),$$

*where the bottom $n-k-r$ rows, denoted by $*$, are to be determined so that the sequence of matrices $\{F_1, \ldots, F_{L+1}\}$ has a minimal partial realization $(A, B, C) \in \mathbb{F}^{\delta\times\delta} \times \mathbb{F}^{\delta\times k} \times \mathbb{F}^{(n-k)\times\delta}$. In both cases, set $D = F_0$ to obtain a minimal ISO representation $(A, B, C, D)$ of an $(n, k, \delta)$ MDP convolutional code.*

The constructed convolutional code is MDP, since it fulfils Theorem 3.2.6 by construction. As the construction outlined in Theorem 4.2.7 requires a superregular matrix, we obtain a lower bound for the required field size for MDP convolutional codes by the lower bound for the existence of superregular matrices in Theorem 4.2.4.

**Corollary 4.2.8.** [13, Corollary 3.4] *Let $r$ be the remainder of $\delta$ divided by $n-k$. Let $\mathbb{F}$ be a finite field satisfying*

$$|\mathbb{F}| > \begin{cases} B_{(L+1)(n-1)} & \text{if } (n-k) \mid \delta, \\ B_{(L+1)(n-1)+k+r-1} & \text{else.} \end{cases}$$

*Then, an $(n, k, \delta)$ MDP convolutional code exists over $\mathbb{F}$.*

Let us present an example how Theorem 4.2.7 is used to construct an $(n, k, \delta)$ MDP convolutional code.

**Example 4.2.9.** To construct a $(3, 2, 2)$ MDP convolutional code over $\mathbb{F}_{2^6}$ using Theorem 4.2.7, consider the superregular matrix

$$\mathcal{F} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \alpha & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \alpha^9 & \alpha & 1 & 0 & 0 & 0 & 0 & 0 \\ \alpha^{33} & \alpha^9 & \alpha & 1 & 0 & 0 & 0 & 0 \\ \alpha^{33} & \alpha^{33} & \alpha^9 & \alpha & 1 & 0 & 0 & 0 \\ \alpha^9 & \alpha^{33} & \alpha^{33} & \alpha^9 & \alpha & 1 & 0 & 0 \\ \alpha & \alpha^9 & \alpha^{33} & \alpha^{33} & \alpha^9 & \alpha & 1 & 0 \\ 1 & \alpha & \alpha^9 & \alpha^{33} & \alpha^{33} & \alpha^9 & \alpha & 1 \end{pmatrix} \in \mathbb{F}_{2^6}^{8\times 8},$$

where $\alpha$ is a root of the primitive polynomial $x^6 + x + 1 \in \mathbb{F}_2[z]$. As $(n-k) \mid \delta$, we have $F_j = \mathcal{F}_{1,2}^{2j+2}$ for $j \in \{0, \ldots, 3\}$ and therefore obtain

$$F_0 = \begin{pmatrix} \alpha & 1 \end{pmatrix}, \quad F_1 = \begin{pmatrix} \alpha^{33} & \alpha^9 \end{pmatrix}, \quad F_2 = \begin{pmatrix} \alpha^9 & \alpha^{33} \end{pmatrix} \quad \text{and} \quad F_3 = \begin{pmatrix} 1 & \alpha \end{pmatrix}.$$

We know that $F_0 = D$. To compute a minimal partial realization, we set $B$ as the identiy matrix of size two. Using that $CA^{i-1} = F_i$ for all $i = 1, \dots, 3$, we get a $(3, 2, 2)$ MDP convolutional code with minimal ISO representation $(A, B, C, D)$, where

$$A = \begin{pmatrix} \alpha^{62} & \alpha^3 \\ \alpha^{15} & \alpha^{56} \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad C = \begin{pmatrix} \alpha^{33} & \alpha^9 \end{pmatrix} \quad \text{and} \quad D = \begin{pmatrix} \alpha & 1 \end{pmatrix}.$$

### Reverse MDP Convolutional Code

We introduce the constructions of reverse MDP convolutional codes given in [30]. There are no general constructions of reverse MDP convolutional codes. This is because we do not know the relation between the matrices $H_i$ and $\overline{H}_i$, for $i \in \{0, \dots, \nu\}$, and between $G_i$ and $\overline{G}_i$, for $i \in \{0, \dots, \mu\}$, across all code parameters. However, if $(n-k) \mid \delta$ and $k > \delta$ or if $k \mid \delta$ and $(n-k) > \delta$, we can use Remark 3.1.28 to construct $(n, k, \delta)$ reverse MDP convolutional codes.

**Theorem 4.2.10.** [30, Theorem 5.12] *If $(n-k) \mid \delta$ and $k > \delta$, let*

$$A \in \mathbb{F}^{(L+1)(2n-k-1)\times(L+1)(2n-k-1)}$$

*be a reverse-superregular matrix. For $j = 0, \dots, L$ we define*

$$I_j \coloneqq \{(j+1)n + j(n-k-1), \dots, (j+1)(2n-k-1)\},$$
$$J_j \coloneqq \{jn + j(n-k-1) + 1, \dots, (j+1)n + j(n-k-1)\},$$

*and let*

$$I \coloneqq \bigcup_{j=0}^{L} I_j \quad \text{and} \quad J \coloneqq \bigcup_{j=0}^{L} J_j.$$

*Set $\hat{A} \coloneqq A_J^I \in \mathbb{F}^{(L+1)(n-k)\times(L+1)n}$. Every not trivially zero full size minor of $\hat{A}$ and $\hat{A}_{\text{rev}}$ is nonzero. They are formed by the columns indexed by $1 \leq i_1 < \cdots < i_{(L+1)(n-k)}$, where $i_{s(n-k)} \leq sn$ for $s = 1, \dots, L$. Let $\mathcal{C}$ be an $(n, k, \delta)$ convolutional code with its $L$-th sliding parity-check matrix $H_L^c = \hat{A}$. Then $\mathcal{C}$ is a reverse MDP convolutional code.*

It follows by construction that the not trivially zero minors of $\hat{A}$ and $\hat{A}_{\text{rev}}$ are nonzero. Recall that $H_L^c$ consists of the matrices $H_i$, where $0 \leq i \leq L$, and that $H(z) = \sum_{i=0}^{\nu} H_i z^i$. Since $(n-k) \mid \delta$ and $k > \delta$, we have $\nu = L = \frac{\delta}{n-k}$. Thus, $H_L = H_\nu$. Therefore, all matrices in the sum $H(z) = \sum_{i=0}^{\nu} H_i z^i$ appear in $H_L^c$ and therefore we know the parity-check matrix $H(z)$. By Theorem 3.1.25, we conclude that an $(n, k, \delta)$ convolutional code with $H(z)$ as its parity-check matrix is MDP.
Furthermore, since $(n-k) \mid \delta$, we know that $\overline{H}(z) = \sum_{i=0}^{\nu} H_{\nu-i} z^i$ is a parity-check matrix of the reverse code. Thus, $\hat{A}_{\text{rev}}$ is a sliding parity-check matrix of the reverse code. Since all not trivially zero minors of $\hat{A}_{\text{rev}}$ are nonzero, the reverse code is MDP and therefore, the constructed code is reverse MDP.

**Example 4.2.11.** We will now construct a $(3, 2, 1)$ reverse MDP convolutional code using Theorem 4.2.10. For this, we use the $6 \times 6$ reverse-superregular matrix over $\mathbb{F}_{23}$ presented in Example 4.2.3. We have that $I = \{3, 6\}$ and $J = \{1, \dots, 6\}$. By

intersecting the rows indexed by $I$ and the columns indexed by $J$ of the superregular matrix, we obtain

$$H_1^c = \begin{pmatrix} H_0 & 0 \\ H_1 & H_0 \end{pmatrix} = \begin{pmatrix} 10 & 5 & 1 & 0 & 0 & 0 \\ 1 & 5 & 10 & 10 & 5 & 1 \end{pmatrix}.$$

This results in a $(3, 2, 1)$ reverse MDP convolutional code with parity-check matrix

$$H(z) = \begin{pmatrix} 10 + z & 5 + 5z & 1 + 10z \end{pmatrix}.$$

The construction of a reverse MDP convolutional code is similar if $k \mid \delta$ and $(n-k) > \delta$. However, we construct a generator matrix instead of a parity-check matrix.

**Theorem 4.2.12.** [30, Theorem 5.15] *If $k \mid \delta$ and $(n-k) > \delta$, let*

$$B \in \mathbb{F}^{(L+1)(n+k-1)\times(L+1)(n+k-1)}$$

*be the transpose of a reverse-superregular matrix. For $j = 0, \ldots, L$, set*

$$I_j \coloneqq \{jn + j(k-1) + 1, \ldots, (j+1)n + j(k-1)\},$$
$$J_j \coloneqq \{(j+1)n + j(k-1), \ldots, (j+1)(n+k-1)\},$$

*and let*

$$I \coloneqq \bigcup_{j=0}^{L} I_j \quad \text{and} \quad J \coloneqq \bigcup_{j=0}^{L} J_j.$$

*Set $\hat{B} \coloneqq B_J^I \in \mathbb{F}^{(L+1)n\times(L+1)k}$. Every not trivially zero full size minor of $\hat{B}$ and $\hat{B}_{\text{rev}}$ is nonzero. They are formed by the columns indexed by $1 \leq i_1 < \cdots < i_{(L+1)k}$ such that $i_{sk+1} > sn$ for $s = 1. \ldots, L$. Let $\mathcal{C}$ be an $(n, k, \delta)$ convolutional code with the generator matrix $G(z) = \sum_{i=0}^{L} G_i z^i$, where*

$$\hat{B} = \begin{pmatrix} G_0^\top & G_1^\top & \ldots & G_L^\top \\ & G_0^\top & \ldots & G_{L-1}^\top \\ & & \ddots & \vdots \\ & & & G_0^\top \end{pmatrix}.$$

*Then $\mathcal{C}$ is a reverse MDP convolutional code.*

Since $L = \mu = \frac{\delta}{k}$, all matrices $G_i$ appear in $\hat{B}$ enabling us to construct a generator matrix $G(z) = \sum_{i=0}^{\mu} G_i z^i$ of $\mathcal{C}$. Note there is a one-to-one correspondence between the not trivially zero full size minors of $\hat{B}$ and $G_L^c$. Thus, by Theorem 3.1.25, we conclude that a convolutional code with generator matrix $G(z)$, constructed as outlined in Theorem 4.2.12, is MDP. Similarly, a generator matrix of the reverse code, which is MDP by construction, can be obtained from $A_{\text{rev}}$. Consequently, we constructed a reverse MDP convolutional code.

**Example 4.2.13.** We will now construct a $(3, 1, 1)$ reverse MDP convolutional code over $\mathbb{F}_{23}$ using Theorem 4.2.12. We will work with the transpose of the $6 \times 6$ reverse-superregular matrix over $\mathbb{F}_{23}$ from Example 4.2.3. Therefore, we have

$$G_1^c = \begin{pmatrix} G_0^\top & G_1^\top \\ 0 & G_0^\top \end{pmatrix} = \begin{pmatrix} 10 & 1 \\ 5 & 5 \\ 1 & 10 \\ 0 & 10 \\ 0 & 5 \\ 0 & 1 \end{pmatrix}.$$

Therefore, we obtain a reverse MDP convolutional code with generator matrix

$$G(z) = \begin{pmatrix} z+10 & 5z+5 & 10z+1 \end{pmatrix}.$$

### Complete MDP Convolutional Code

The authors of [30] conjectured that the reverse MDP convolutional codes constructed using Theorem 4.2.10 are complete MDP. However, the conjecture has not been proven yet. Therefore, we show two other constructions of complete MDP convolutional codes given in [15].

**Theorem 4.2.14.** [15, Theorem 3.8] *Let $n, k, \delta \in \mathbb{N}$ such that $k < n$ and $(n-k) \mid \delta$, and let $\nu = \frac{\delta}{n-k}$. Then, $H(z) = \sum_{i=0}^{\nu} H_i z^i \in \mathbb{F}[z]^{(n-k)\times n}$ with*

$$H_0 = \begin{pmatrix} \binom{\nu n+k}{k} & \dots & 1 & \\ \vdots & & & \ddots \\ \binom{\nu n+k}{n-1} & & \dots & 1 \end{pmatrix},$$

$$H_i = \begin{pmatrix} \binom{\nu n+k}{in+k} & \dots & \binom{\nu n+k}{(i-1)n+k+1} \\ \vdots & & \vdots \\ \binom{\nu n+k}{(i+1)n-1} & \dots & \binom{\nu n+k}{in} \end{pmatrix} \quad \text{for } i = 1, \dots, \nu-1,$$

$$H_\nu = \begin{pmatrix} 1 & \dots & \binom{\nu n+k}{n-1} \\ & \ddots & \vdots \\ & 1 & \binom{\nu n+k}{k} \end{pmatrix},$$

*is a parity-check matrix of an $(n, k, \delta)$ complete MDP convolutional code if the characteristic of the field satisfies*

$$\operatorname{char}(\mathbb{F}) > \binom{\nu n+k}{\lfloor 1/2(\nu n+k) \rfloor}^{(n-k)(L+1)} \cdot \big((n-k)(L+1)\big)^{(1/2(n-k)(L+1))}.$$

Consider $X \in \mathbb{F}^{(\nu+L+1)n \times (\nu+L+1)n}$ as defined in Lemma 4.2.2. For $j = 0, \dots, L$ we set

$$I_j \coloneqq \{(\nu+j)n+k+1, \dots, (\nu+j+1)n\} \quad \text{and} \quad I \coloneqq \bigcup_{j=0}^{L} I_j.$$

Then, by construction, the not trivially zero minors of the submatrix formed by the rows of $X^{\nu n+k}$ indexed by $I$ are nonzero. This submatrix can be viewed as a partial parity-check matrix of an $(n, k, \delta)$ convolutional code. Consequently, this convolutional code is complete MDP. With the help of this partial parity-check matrix, we can obtain a parity-check matrix of the code.

**Example 4.2.15.** Using Theorem 4.2.14, we construct a $(3, 1, 4)$ complete MDP convolutional code. As outlined in the theorem, we obtain the matrices

$$H_0 = \begin{pmatrix} 7 & 1 & 0 \\ 21 & 7 & 1 \end{pmatrix}, \quad H_1 = \begin{pmatrix} 35 & 35 & 21 \\ 21 & 35 & 35 \end{pmatrix} \quad \text{and} \quad H_2 = \begin{pmatrix} 1 & 7 & 21 \\ 0 & 1 & 7 \end{pmatrix}.$$

Therefore, a convolutional code with $H(z) = H_0 + H_1 z + H_2 z^2 \in \mathbb{F}[z]^{2\times 3}$ as its parity-check matrix is complete MDP if $\operatorname{char}(\mathbb{F}) > 35^{14} \cdot 14^7$. It is notable that even for small parameters, a large characteristic is already required.

For the second construction, we require the following criterion which describes when the not trivially zero minors of a matrix are nonzero.

**Lemma 4.2.16.** [1, Theorem 3.3] *Let $\alpha$ be a primitive element of the finite field $\mathbb{F} = \mathbb{F}_{p^N}$ and $B = (b_{ij}) \in \mathbb{F}^{l \times m}$ be a matrix with the following properties:*

*(i) If $b_{ij} \neq 0$, then $b_{ij} = \alpha^{\beta_{ij}}$ for a positive integer $\beta_{ij}$.*

*(ii) If $b_{ij} = 0$, then $b_{i'j} = 0$ for any $i' > i$ or $b_{ij'} = 0$ for any $j' < j$.*

*(iii) If $j < j'$, $b_{ij} \neq 0$ and $b_{ij'} \neq 0$, then $2\beta_{ij} \leq \beta_{ij'}$.*

*(iv) If $i < i'$, $b_{ij} \neq 0$ and $b_{i'j} \neq 0$, then $2\beta_{ij} \leq \beta_{i'j}$.*

*Suppose $N$ is larger than any exponent of $\alpha$ appearing as a nontrivial term of any minor of $B$. Then every not trivially zero minor of $B$ is nonzero.*

With the help of Lemma 4.2.16, we can now construct a complete MDP convolutional code. The method has the advantage that it works for any arbitrary characteristic of the underlying field.

**Theorem 4.2.17.** [15, Theorem 3.14] *Let $n, k, \delta \in \mathbb{N}$ such that $k < n$ and $(n-k) \mid \delta$, and let $\nu = \frac{\delta}{n-k}$. Let $\alpha$ be a primitive element of a finite field $\mathbb{F} = \mathbb{F}_{p^N}$ such that $N > (L+1) \cdot 2^{(\nu+2)n-k-1}$. For every $i = 0, \ldots, \nu = \frac{\delta}{n-k}$, let*

$$H_i = \begin{pmatrix} \alpha^{2^{in}} & \cdots & \alpha^{2^{(i+1)n-1}} \\ \vdots & & \vdots \\ \alpha^{2^{(i+1)n-k-1}} & \cdots & \alpha^{2^{(i+2)n-k-2}} \end{pmatrix}.$$

*Then $H(z) = \sum_{i=0}^{\nu} H_i z^i$ is a parity-check matrix of an $(n, k, \delta)$ complete MDP convolutional code.*

Let $\hat{\mathcal{H}}$ be the matrix obtained by reversing the order of the column blocks of the partial parity-check matrix $\mathcal{H}$ with the $H_i$'s constructed as outlined in Theorem 4.2.17. Due to the construction, $\hat{\mathcal{H}}$ satisfies the four properties given in Lemma 4.2.16. By choosing a sufficiently large $N$, we deduce that the resulting convolutional code is indeed complete MDP.

**Example 4.2.18.** We provide an example of a $(3, 1, 4)$ complete MDP convolutional code utilizing this second construction method. We obtain a parity-check matrix $H(z) = H_0 + H_1 z + H_2 z^2 \in \mathbb{F}_{2^N}[z]^{2 \times 3}$ with

$$H_0 = \begin{pmatrix} \alpha & \alpha^2 & \alpha^4 \\ \alpha^2 & \alpha^4 & \alpha^8 \end{pmatrix}, \quad H_1 = \begin{pmatrix} \alpha^8 & \alpha^{16} & \alpha^{32} \\ \alpha^{16} & \alpha^{32} & \alpha^{64} \end{pmatrix} \quad \text{and} \quad H_2 = \begin{pmatrix} \alpha^{64} & \alpha^{128} & \alpha^{256} \\ \alpha^{128} & \alpha^{256} & \alpha^{512} \end{pmatrix},$$

where $\alpha$ is a primitive element of the finite field $\mathbb{F}_{2^N}$ with $N > 7 \cdot 2^{10} = 7168$.

There are no general constructions of complete $j$-MDP convolutional codes if $j < L$. However, the constructed complete MDP convolutional codes are clearly complete $j$-MDP as well. There is a construction for complete $j$-MDP convolutional codes with small parameters in [2] where the required finite field is relatively small.

In this chapter, we have seen the connection between the construction of the different classes of MDP convolutional codes and superregular matrices. Consequently, the challenge posed by the necessity of a large field for the construction of superregular matrices applies to the construction of convolutional codes as well. In [16], the already existing lower bounds for the existence of MDP and complete MDP convolutional codes have been improved. Nevertheless, these bounds remain suboptimal and do not lead to concrete constructions of these classes of codes. The problem of determining the exact minimum field size required for the existence of MDP, reverse MDP and complete $j$-MDP convolutional codes and of providing constructions of these codes is still an open problem.

# Chapter 5

# Erasure Decoding of Convolutional Codes

In this chapter, we introduce three erasure decoding algorithms for convolutional codes, namely the forward and backward decoding algorithm, the low delay decoding algorithm for modules, and the low delay decoding algorithm for linear systems. The decoding of convolutional codes over erasure channels has the advantage, compared to linear block codes, that we can decode through a sliding window, whereas for linear block codes we have fixed blocks. This gives us more flexibility as we can group the erasures appropriately to adapt to the received pattern of erasures. It has been shown in [30] that there are examples of erasure patterns that already MDP convolutional codes can recover, but MDS linear block codes of the same rate cannot recover them. We show which classes of convolutional codes are optimal for each algorithm and how the algorithms differ. Lastly, we make two simulations over different erasure channels to see the advantages and disadvantages of each algorithm when we have erasure patterns that occur in practice.

## 5.1 Erasure Decoding Algorithms

We will introduce the three decoding algorithms and discuss the classes of convolutional codes for which each algorithm can recover the most erasures. Throughout this chapter, we consider a non-catastrophic $(n, k, \delta)$ convolutional code $\mathcal{C}$.

### 5.1.1 Forward and Backward Decoding Algorithm

In the two conference papers [28] and [29], it was shown that MDP and reverse MDP convolutional codes can recover erasures, where in certain situations, MDS linear block codes may fail to. This capability was incorporated into the forward and backward decoding algorithm introduced in [30], which is an extension of the results in [27].

We now introduce the forward and backward decoding algorithm and use some results of Chapter 3 to prove which classes of convolutional codes are optimal for the algorithm. For forward decoding, let $H(z) = \sum_{i=0}^{\nu} H_i z^i$ be a left-prime parity-

check matrix of $\mathcal{C}$. Consider a codeword $c(z) = \sum_{i=0}^{l} c_i z^i \in \mathcal{C}$, then we have

$$H(z)c(z)^\top = 0.$$

By equating the coefficients, the characterization can be written as

$$\begin{pmatrix} H_0 & & & & & & \\ \vdots & \ddots & & & & & \\ H_\nu & \dots & H_0 & & & & \\ & \ddots & & \ddots & & & \\ & & H_\nu & \dots & H_0 & & \\ & & & \ddots & \vdots & & \\ & & & & H_\nu & & \end{pmatrix} \begin{pmatrix} c_0^\top \\ \vdots \\ c_l^\top \end{pmatrix} = 0, \tag{5.1}$$

where the left matrix is an $(\nu + l + 1)(n-k) \times (\nu+1)n$ matrix.

Assume we receive $v(z) = \sum_{i=0}^{l} v_i z^i$. Moreover, we need the following assumptions in order to forward decode for some $i, j \in \mathbb{N}_0$:

(i) $v_{i-\nu}, \dots, v_{i-1} \in \mathbb{F}^n$ are known.

(ii) $v_i$ contains at least one erasure.

(iii) We consider the window with delay $j$ (i.e., $v_i, \dots, v_{i+j}$).

Recall that $v_i = 0$ for $i \notin \{0, \dots, l\}$.

**Definition 5.1.1.** [7, Section 6.1] Under the assumptions above, $v_{i-\nu}, \dots, v_{i-1}$ is called *guard space*. [27, Section 3.1] The window consisting of the $(j+1)n$ symbols of $v_i, \dots, v_{i+j}$ is called *sliding window*.

Furthermore, we use the following notation:

- The *erased* components of $v_i, \dots, v_{i+j}$ are denoted by $v_i^{(e)}, \dots, v_{i+j}^{(e)}$.
- $(H_j^c)^{(e)}$ denotes the columns of $H_j^c$ that correspond to the erased components of $v_i, \dots, v_{i+j}$.
- The *received* components of $v_{i-\nu}, \dots, v_{i+j}$ are denoted by $v_{i-\nu}^{(r)}, \dots, v_{i+j}^{(r)}$.
- $\mathcal{H}_j^{(r)}$ denotes the columns of $\mathcal{H}_j$ that correspond to the received components of $v_{i-\nu}, \dots, v_{i+j}$. Similarly, $\mathcal{H}_j^{(e)}$ denotes the columns of $\mathcal{H}_j$ that correspond to the erased components of $v_{i-\nu}, \dots, v_{i+j}$.

Now, if we want to *forward decode* under the above assumptions, we need to solve the following system of linear equations corresponding to the rows of (5.1):

$$\mathcal{H}_j(v_{i-\nu}, \dots, v_{i+j})^\top = 0. \tag{5.2}$$

Since, by assumption, $v_{i-\nu}, \dots, v_{i-1}$ do not contain erasures, it suffices to solve

$$(H_j^c)^{(e)}(v_i^{(e)}, \dots, v_{i+j}^{(e)})^\top = -\mathcal{H}_j^{(r)}(v_{i-\nu}^{(r)}, \dots, v_{i+j}^{(r)})^\top. \tag{5.3}$$

We note that the erased symbols in $v_i, \dots, v_{i+j}$ can be recovered if and only if $(H_j^c)^{(e)}$ has full (column) rank. Due to the required guard space on the left, we start forward decoding at the beginning of a received sequence and move forward until the end. This is why forward decoding is sometimes also called *left-to-right decoding*.
Since forward decoding only requires a guard space and a sliding window, we do not need to wait until the whole codeword is received in order to forward decode. This is one main difference and advantage of convolutional codes compared to linear block codes.

We now prove how many erasures in a sliding window can be recovered and which classes of convolutional codes are optimal for forward decoding. That is, if an erasure pattern cannot be corrected with an MDP convolutional code, the erasures cannot be recovered by any other convolutional code with the same parameters.

**Proposition 5.1.2.** [30, Theorem 3.1] *Let $\mathcal{C}$ be an $(n, k, \delta)$ convolutional code with $j$-th column distance $d_j^c$. If there is a guard space of $\nu n$ symbols on the left and in any sliding window of length $(j+1)n$ at most $d_j^c - 1$ erasures occur, then we can recover the erasures in the first $n$ symbols of the sliding window by forward decoding.*

*Proof.* The claim follows from Proposition 3.1.18. Since none of the first $n$ columns of $H_j^c$, corresponding to an erasure in $v_i$, is in the span of any other $d_j^c - 2$ columns, the erasure can be recovered. Hence, we can completely recover the erasures in $v_i$. □

Thus, by taking the smallest possible delay $j$ for each forward decoding step, the highest number of erasures can be recovered. Moreover, since having the largest possible column distance for as long as possible is the best scenario for Proposition 5.1.2, MDP convolutional codes are optimal for forward decoding.

**Corollary 5.1.3.** [30, Corollary 3.2] *Let $\mathcal{C}$ be an $(n, k, \delta)$ MDP convolutional code. Assume that we always take the smallest possible sliding window size $j \leq L$. If there is a guard space of $\nu n$ symbols on the left and in any sliding window of length $(j+1)n$ at most $(j+1)(n-k)$ erasures occur, then we can completely recover the sequence by forward decoding.*

*Proof.* This follows from Proposition 5.1.2 and the fact that we can recover the maximum number of erasures by always choosing the smallest possible window size. Alternatively, it can also be proven with Theorem 3.1.25. □

Next, we show what happens if we do not take the smallest possible sliding window size. For this, consider a $(3, 2, 2)$ MDP convolutional code $\mathcal{C}$. Assume we want to forward decode the following received sequence, where $\star$ denotes an erasure and ✓ a correctly received symbol:

| $v_0$ | $v_1$ | $v_2$ | $v_3$ | |
|---|---|---|---|---|
| ✓ $\star$ ✓ | $\star$ ✓ ✓ | ✓ ✓ ✓ | $\star$ ✓ $\star$ | ... |

By Corollary 5.1.3, we can recover at most $j+1$ erasures in a sliding window of $3(j+1)$ symbols for $j \leq 3$.
If we take the smallest possible sliding window for each decoding step, we can recover the erasure in $v_0$ since there is one erasure in three symbols. Similarly, we can fully recover $v_1$. The recovery of the erasures in $v_3$ depends on the symbols after it, and it could possibly happen that we cannot recover these erasures.
If we take the largest possible window size, which consists of $v_0, v_1, v_2$ and $v_3$, it appears that the four erasures could be recoverable. However, $(H_j^c)^{(e)}$ has the following form, where $*$ denotes a nonzero entry:

$$(H_j^c)^{(e)} = \begin{pmatrix} * & 0 & 0 & 0 \\ * & * & 0 & 0 \\ * & * & 0 & 0 \\ * & * & * & * \end{pmatrix}.$$

Hence, $(H_j^c)^{(e)}$ does not have full rank. It is impossible to fully recover the erasures in this sliding window. This is why we always take the smallest possible sliding window size.

Requiring a guard space on the left of a sliding window in order to forward decode can be restrictive. However, since all the results so far in this subsection are analogous for reverse codes, we can also forward decode using the reverse code. This procedure is called *backward decoding* or *decoding right-to-left*. For this, we need to solve the following system of linear equations

$$\overline{\mathcal{H}}_j(v_{i+\nu}, \ldots, v_{i-j})^\top = 0, \tag{5.4}$$

where $\overline{\mathcal{H}}_j$ is the $j$-th sliding parity-check matrix of the reverse code $\overline{\mathcal{C}}$.

**Remark 5.1.4.** Let $\mathcal{C}$ be an $(n,k,\delta)$ convolutional code. If $(n-k) \mid \delta$, we know by Remark 3.1.28 that $\overline{H}(z) = \sum_{i=0}^{\nu} H_{\nu-i} z^i$ is a parity-check matrix of the reverse code $\overline{\mathcal{C}}$. Hence, we can rewrite (5.4) as

$$\mathcal{H}_j(v_{i-j}, \ldots, v_{i+\nu})^\top = 0.$$

Focussing on $\mathcal{C}$, the guard space for backward decoding is on the right of the sliding window. We note that by moving from the end of a received codeword to the beginning, we could recover the most erased symbols using backward decoding. However, we will not consider this situation, since we need to wait until the whole codeword is received for decoding from right-to-left and that could possibly not give better results than the retransmission of the lost packets. Thus, it makes sense to combine forward and backward decoding and move from left to right. Since an MDP convolutional code is optimal for forward decoding and a convolutional code whose reverse code is MDP is optimal for backward decoding, a reverse MDP convolutional code is optimal for when we combine the two decoding processes.

**Corollary 5.1.5.** [30, Theorem 4.5] *Let $\mathcal{C}$ be an $(n,k,\delta)$ reverse MDP convolutional code. Assume that we always take the smallest possible sliding window size $j \leq L$. If in any sliding window of length $(j+1)n$ at most $(j+1)(n-k)$ erasures occur and there is a guard space of $\nu n$ symbols on the left or on the right, then we can completely recover the sequence using forward and backward decoding.*

The combination of forward decoding and backward decoding was proposed as an algorithm in [30], which we call the *forward and backward decoding (FBD) algorithm.* Its pseudocode is provided in Algorithm 1. For this, we use the notation that $v_i = 0$ if and only if $v_i$ contains at least one erasure. Similarly, $v_i = 1$ means that $v_i$ does not contain erasures. We define *findzeros*$(v_i, \ldots, v_{i+j})$ as the function which returns a vector with the positions of erasures in $(v_i, \ldots, v_{i+j})$.

For the FBD algorithm, we start at the beginning of a received sequence

$$v(z) = v_0 + v_1 z + \cdots + v_l z^l$$

and search for the first $v_i$ which contains at least one erasure. If there is a guard space on the left, we attempt to forward decode with the largest possible sliding window. In this scenario, starting with the largest sliding window is feasible, since we always verify the presence of a unique solution. If there exists a unique solution for some delay, we can recover all the erasures in the window and start searching for the next erasures. Otherwise, we try to backward decode. For this, we determine the index $K$ indicating the earliest possible beginning of a guard space to be able to recover the erasures in $v_i$. Once a guard space is found, we attempt backward decoding using the largest possible sliding window. If we can backward decode for some delay, we recover the erasures and start trying to forward decode the next erasures. In case backward decoding is unsuccessful, we declare the erasures in $v_i$ as lost and try to forward decode the next erasures.

Let us now give a small example of the FBD algorithm.

**Example 5.1.6.** Consider the $(3, 2, 1)$ reverse MDP convolutional code over $\mathbb{F}_{23}$ from Example 4.2.11 with parity-check matrix

$$H(z) = \begin{pmatrix} 10 + z & 5 + 5z & 1 + 10z \end{pmatrix}.$$

Note that $L = \nu = 1$. Assume we receive the sequence

$$v(z) = (\star, \star, 18) + (5, 22, 0)z + (\star, \star, \star)z^2 + (\star, 6, 1)z^3.$$

We start by trying to recover the two erasures in $v_0$, which we denote by $x_0$ and $x_1$ respectively. Using the symbols before $v_0$, which are all zero, as guard space, we try to recover $x_0$ and $x_1$ with a delay of one. This leads to the equation

$$\left(H_1^c\right)^{(e)} \begin{pmatrix} v_0^{(e)} & v_1^{(e)} \end{pmatrix}^\top = -\mathcal{H}_1^{(r)} \begin{pmatrix} v_{-1}^{(r)} & v_0^{(r)} & v_1^{(r)} \end{pmatrix}^\top$$

which corresponds to

$$\begin{pmatrix} 10 & 5 \\ 1 & 5 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = -\begin{pmatrix} 1 & 5 & 10 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10 & 10 & 5 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 18 \\ 5 \\ 22 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 5 \end{pmatrix}.$$

**Algorithm 1:** Forward and Backward Decoding Algorithm

**Data:** The received sequence $v(z) = v_0 + v_1 z + \cdots + v_l z^l$.
**Result:** The corrected sequence $v^*(z) = v_0 + v_1 z + \cdots + v_l z^l$.

```
i = 0;
while i ≤ l do
    forwardsuccess = 0;
    backwardsucccess = 0;
    if v_i = 0 then
        if (v_{i−ν}, ..., v_{i−1}) = 1 then
            j = L;
            while forwardsuccess = 0 and j ≥ 0 do
                if Eq. (5.2) has unique solution then
                    Replace erasures in v_i, ..., v_{i+j} with correct symbols using
                     Eq. (5.2);
                    forwardsuccess = 1;
                    i = i + j;
                else
                    j = j − 1;
                end
            end
        end
        if forwardsuccess ≠ 1 then
            aux = findzeros(v_i, ..., v_{i+L});
            K = i + aux[−1] + 1;
            while backwardsuccess = 0 and K ≤ min{i + L + 1, l + 1} do
                if (v_K, ..., v_{K+ν−1}) = 1 then
                    j = L;
                    while backwardsuccess = 0 and j ≥ 0 do
                        if Eq. (5.4) has unique solution then
                            Replace erasures in v_{K−j−1}, ..., v_{K−1} with correct
                             symbols using Eq. (5.4);
                            backwardsuccess = 1;
                            i = K + ν − 1;
                        else
                            j = j − 1;
                        end
                    end
                end
                K = K + 1;
            end
        end
    end
    i = i + 1;
end
```

Thus, $x_0 = 0$ and $x_1 = 1$.
Subsequently, we cannot correct the erasures in $v_2$ due to the high number of erasures. However, we can use the symbols after $v_3$ as guard space to backward decode the erasure in $v_3$, which we denote by $x_2$. This is possible with a delay of zero. Using Remark 5.1.4 we have

$$\mathcal{H}_0 \begin{pmatrix} v_3 & v_4 \end{pmatrix}^\top = 0$$

which translates to

$$x_2 = -\begin{pmatrix} 5 & 10 & 10 & 5 & 1 \end{pmatrix} \begin{pmatrix} 6 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = 6.$$

Therefore, we recovered three of the six erasures in the received sequence.

Next, we provide an example where the FBD algorithm allows for the recovery of more erasures when using a reverse MDP convolutional code compared to an MDS linear block of the same rate and erasure correction capability.

**Example 5.1.7.** Consider a $(4, 2, 24)$ reverse MDP convolutional code $\mathcal{C}$. Thus, $L = 24$. We want to decode the following received sequence of 100 symbols applying Algorithm 1:

$$\underbrace{\star \ldots \star}_{A(40)} \underbrace{\checkmark \ldots \checkmark}_{B(40)} \underbrace{\star \ldots \star}_{C(20)}$$

This means that $A$ consists of $v_0$ up to $v_9$, $B$ consists of $v_{10}$ up to $v_{19}$ and $C$ consists of $v_{20}$ up to $v_{24}$. We have in total 60 erasures in 100 symbols.
First, we try to recover the erasures in $v_0$. As the symbols before $v_0$ are all zero, we have a guard space. Since there are 40 erasures in the sliding window of the 80 symbols of $A$ and $B$, we recover all the erasures in $A$ with a delay of 19 using forward decoding by Corollary 5.1.5.
Next, we want to correct the erasures in $v_{20}$. We can use the symbols of $B$ and some symbols of $A$ as guard space to forward decode. Using the symbols of $C$ and 20 symbols after $C$, which are zero, as the sliding window, we can recover all the erasures in $C$ with a delay of 9.
To sum up, we are able to recover 60 erasures in 100 symbols. This is more than an MDS linear block code of the same rate and erasure correction capability can. In fact, a $[100, 50, 51]$ linear block code can recover 50 erasures in 100 symbols.

**Remark 5.1.8.** We have introduced several adjustments to the decoding algorithm given in [30]. These changes improve the efficiency and accuracy of the FBD algorithm.
First, we modified the calculation of the index $K$, which determines the potential starting point for a guard space to backward decode. In the original pseudocode in the paper, $K$ is computed as $K = i + aux[-1] - 1$. However, in our adapted version, we set $K = i + aux[-1] + 1$. This adjustment skips some unnecessary steps as there are still some erasures after $i + aux[-1] - 1$. To illustrate this, consider the example with $L = 3$ and the following received sequence using the same notation as for Algorithm 1:

$$v_i = 0, \quad v_{i+1} = 1, \quad v_{i+2} = 0, \quad v_{i+3} = 1.$$

Assuming there is no guard space to the left, we need to determine the index $K$. For this, we have that $aux = \{0, 2\}$. Therefore, using the algorithm in [30], we obtain $i + aux[-1] - 1 = i + 1$. However, a possible guard space can only start at $i + aux[-1] + 1 = i + 3$, as utilized in Algorithm 1.
Second, we adjusted the condition $K \leq l+1$ instead of $K \leq l$ for backward decoding. This small but significant change allows us to backward decode using the symbols after the end of a received sequence, which are zero. Consequently, we can recover more erasures with backward decoding.
Third, there is a difference between the pseudocode and the textual description in the paper. The pseudocode suggests that if forward decoding is unsuccessful, the algorithm attempts to backward decode the received sequence until this is successful. However, in the text, it is described that if forward decoding is not successful, we try to backward decode the erasures in $v_i$. If this is not possible, we restart by trying to forward decode the next erasures. We have adapted the latter method in our version of the FBD algorithm in Algorithm 1, since more erasure can be recovered as we illustrate with the next example.
Let $\mathcal{C}$ be a $(3, 2, 2)$ reverse MDP convolutional code. Suppose we receive the sequence

| $v_0$ | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ | $v_8$ | $v_9$ |
|---|---|---|---|---|---|---|---|---|---|
| ⋆⋆⋆ | ⋆⋆⋆ | ⋆⋆⋆ | ✓✓✓ | ✓✓✓ | ✓⋆✓ | ⋆✓✓ | ✓⋆✓ | ✓✓⋆ | ⋆✓✓ |

Using the algorithm given as in the pseudocode in [30], we try to forward decode the erasures in $v_0$. As this is impossible, we try backward decoding using the symbols of $v_3$ and $v_4$ as guard space. This is not possible as well. Thus, we try backward decoding until this is successful. We can use the symbols after $v_9$, which are zero, as guard space to recover the erasures in $v_6$, $v_7$, $v_8$ and $v_9$.
Conversely, using our version of the algorithm we use the guard space of $v_3$ and $v_4$ to forward decode the erasures in $v_5$ up to $v_8$. Afterwards, we can correct the erasures in $v_9$ by forward decoding.
Consequently, we were able to recover the erasures in $v_5$, whereas this was not possible using the pseudocode in the paper. Depending on the erasure pattern, the difference between the two methods may be even bigger.

### 5.1.2 Low Delay Decoding Algorithm for Modules

The next erasure decoding algorithm we look at is the low delay decoding algorithm for the module point of view of convolutional codes given in [2]. This algorithm is an improved version of the FBD algorithm. We require a guard space for both forward and backward decoding, but there are erasure patterns that can be recovered using Equation (5.2) without a guard space. This low delay decoding algorithm does not require a guard space in order to decode. Additionally, we always tried taking the largest possible delay. However, there are situations where it makes sense to recover erasures as soon as possible, for example in real-time communication or streaming applications. This is implemented in the low delay algorithm, where we can set a maximal delay within which the recovery process of an erased symbol has to be done. We always take the smallest possible delay to recover erasures. This has the advantage that it is computationally less complex and less costly to do several decoding steps with smaller windows than one decoding step with a larger window [18].

Consider the received sequence $v(z) = \sum_{i=0}^{l} v_i z^i$. We want to recover the erased symbols in this sequence using the *low delay decoding for modules (LDDM) algorithm*. Similar to the FBD algorithm, the LDDM algorithm works by decoding the received sequence from left-to-right. Before starting the algorithm, we set a maximal delay $T \leq L$, specifying the time frame within which we want to correct erasures.
Assume we want to recover the erasures in some $v_i$. For this, we use Equation (5.2) with slightly adjusted indices. The window of the sequence that we use starts from the index $t \in \{i - \nu, \ldots, i\}$, determined by the $\nu n$ symbols preceding $v_i$. If there are no erasures in this window, then $t = i - \nu$. However, if there are erasures in this window, we set $t$ such that $v_{t-1}$ contains erasures but $v_t$ up to $v_{i-1}$ do not. We take the smallest delay $j$ for which $s = i + j - t - \nu$ is nonnegative. By solving the equation

$$\mathcal{H}_s(v_t, \ldots, v_{i+j})^\top = 0, \tag{5.5}$$

we attempt to recover the erased symbols. If we cannot solve the system of linear equations, we proceed with a larger delay $j$ while ensuring that $j \leq T$. If $\mathcal{H}_j^{(e)}$ has full (column) rank, we have a unique solution. Note that when $t = i - \nu$, we are solving the same linear equations as for forward decoding.
As described in Algorithm 2, the LDDM algorithm uses this decoding step iteratively to recover the erasures in a received sequence. Moreover, we do not require that we can correct all erasures, meaning those in $v_{i+1}, \ldots, v_{i+j}$, since we want the smallest possible delay to recover the erasures in $v_i$.

**Remark 5.1.9.** In our version of the LDDM algorithm, we have introduced a slight adjustment compared to the approach in [2]. In the original paper, the algorithm sets $c = i$ when we cannot recover the erasures in $v_i$. However, this approach can result in situations where the lost declared erasures are contained within the window for the next decoding step. Consequently, it can be possible to recover all the erased symbols in the window, including those in $v_i$, but with a delay larger than the chosen maximal delay. To address this issue, we have modified our version of the LDDM algorithm. When the recovery of $v_i$ is not possible, we set $c = i + 1$. While this adjustment may lead to the recovery of fewer erasures, it aligns with the chosen maximal delay.

Next, we show which class of convolutional codes is optimal for erasure decoding using the LDDM algorithm. This depends on the sliding parity-check matrix $\mathcal{H}_s$ for $s \leq T$. Hence, for optimal erasures decoding we want that all the not trivially zero full size minors of $\mathcal{H}_s$ are nonzero. This satisfies a complete $T$-MDP convolutional code. Therefore, complete $T$-MDP convolutional codes are optimal for the LDDM algorithm with maximal delay $T$. [2, Theorem 5]

**Corollary 5.1.10.** [2, Theorem 6] *Let $\mathcal{C}$ be an $(n, k, \delta)$ complete $j$-MDP convolutional code, then it has the following decoding properties:*

*(i) If there is a guard space of $\nu n$ symbols on the left and in any sliding window of length $(j + 1)n$ at most $(j + 1)(n - k)$ erasures occur, complete recovery is possible.*

*(ii) If in a window of size of $(\nu + j + 1)n$ there are not more than $(j + 1)(n - k)$ erasures and if between position 1 and $sn$ and between position $(\nu + j + 1)n -$*

**Algorithm 2:** Low Delay Decoding Algorithm for Modules

**Data:** The received sequence $v(z) = v_0 + v_1 z + \cdots + v_l z^l$, maximal delay $T$.
**Result:** The corrected sequence $v^*(z) = v_0 + v_1 z + \cdots + v_l z^l$.

```
i = 0;
c = −(ν + 1);
while i ≤ l do
    if v_i = 0 then
        j = 0;
        success = 0;
        while j ≤ T and success = 0 do
            t = max{i − ν, c};
            s = i + j − t − ν;
            if Eq. (5.5) has unique solution for at least v_i and s ≥ 0 then
                Replace erasures in v_i (and if possible in v_{i+1}, ..., v_{i+j}) with
                 correct symbols using Eq. (5.5);
                success = 1;
            else
                if j = T then
                    c = i + 1;
                end
                j = j + 1;
            end
        end
    end
    i = i + 1;
end
```

*$sn+1$ and $(\nu+j+1)n$, for $s = 1, \ldots, j+1$, there are not more than $s(n-k)$ erasures, then full correction of all erasures in this window is possible.*

*Proof.* The first property follows by Theorem 3.1.25, since $H_s^c$ is a submatrix of $\mathcal{H}_s$ for $s \leq j$, and by Proposition 5.1.2. The second property follows by Proposition 3.1.32. □

Let us look at an example where we apply the LDDM algorithm.

**Example 5.1.11.** Let $\mathcal{C}$ be a $(3, 2, 2)$ complete 2-MDP convolutional code. We apply Algorithm 2 with a maximal delay of 2. Let us consider we receive the sequence

| $v_0$ | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ | $v_8$ | |
|---|---|---|---|---|---|---|---|---|---|
| ✓✓⋆ | ⋆✓✓ | ⋆⋆✓ | ✓✓✓ | ⋆⋆⋆ | ✓✓✓ | ✓✓⋆ | ✓⋆✓ | ✓✓✓ | ... |

As $v_0$ has one erasure in three symbols and the symbols before $v_0$ are all zero, we can recover the erased symbol with a delay of zero. Similarly, we can recover $v_1$. Afterwards, we can correct the erasures in $v_2$ with a delay of one as we have two erasures in six symbols and there is a guard space. However, we cannot recover the erasures in $v_4$ with any delay $j \leq 2$. According to the algorithm, we therefore set $c = 5$. Next, we try to recover $v_6$ which implies that $t = 5$. As $s = i+j-t-\nu = j-1$

is required to be nonnegative, we have a delay $j$ of either one or two. For $j = 1$, as there are two erasures in the window from $v_5$ to $v_7$, we cannot recover them. However, with a delay of two we are able to recover the erasures in $v_6$ and $v_7$.

Note that it is not possible to recover the erasures in $v_6$ and $v_7$ with the FBD algorithm as there is no guard space. Therefore, the LDDM algorithm is an improvement of the FBD algorithm.

### 5.1.3 Low Delay Decoding Algorithm for Linear Systems

We now introduce the low delay decoding algorithm for the linear system point of view given in [18]. Instead of using that $H(z)c(z)^\top = 0$, we use for this low delay algorithm that a codeword fulfils the linear system equations. We can set a maximal delay $T \leq L$ within which the recovery process of an erasure has to be done. The algorithm consists of a main decoding algorithm which tries to recover erasures from the lowest to the highest index of the received sequence, as well as a termination algorithm. The termination algorithm allows us, if possible, to finish the decoding process before the end of the sequence is received by recovering all the erasures.

Let $(A, B, C, D)$ be a minimal ISO representation of $\mathcal{C}$. Assume we receive some sequence

$$v(z) = \begin{pmatrix} y(z) \\ u(z) \end{pmatrix} = \sum_{i=0}^{l} \begin{pmatrix} y_{l-i} \\ u_{l-i} \end{pmatrix} z^i \in \mathbb{F}[z]^n,$$

where

$$y(z) = \sum_{i=0}^{l} y_{l-i} z^i \in \mathbb{F}[z]^{n-k},$$
$$u(z) = \sum_{i=0}^{l} u_{l-i} z^i \in \mathbb{F}[z]^{k}.$$

By the linear system equations in (3.3), we know for some $i, j \in \mathbb{N}_0$ it holds that

$$\left(\begin{array}{c|ccc} -I_{(i+j+1)(n-k)} & D & & \\ & CB & \ddots & \\ & \vdots & \ddots & D \\ & CA^{i+j+1}B & \dots & CB \end{array}\right) \begin{pmatrix} y_0 \\ \vdots \\ y_{i+j} \\ \hline u_0 \\ \vdots \\ u_{i+j} \end{pmatrix} = 0. \tag{5.6}$$

Assume that there is at least one erasure in $v_i = \begin{pmatrix} y_i \\ u_i \end{pmatrix}$ which we want to recover with a delay $j$. Therefore, we split (5.6) into the part corresponding to the symbols before $v_i$ and into the part corresponding to the symbols after $v_i$, including $v_i$. This

gives us

$$0 = \left(\begin{array}{c|ccc} -I_{i(n-k)} & \multicolumn{3}{c}{\mathcal{F}_{i-1}} \\ \hline & CA^{i-1}B & \dots & CB \\ 0_{(j+1)(n-k),i(n-k)} & \vdots & & \vdots \\ & CA^{i+j+1}B & \dots & CA^{j}B \end{array}\right) \left(\begin{array}{c} y_0 \\ \vdots \\ y_{i-1} \\ \hline u_0 \\ \vdots \\ u_{i-1} \end{array}\right)$$

$$+ \left(\begin{array}{c|c} 0_{i(n-k),(j+1)(n-k)} & 0_{i(n-k),(j+1)k} \\ \hline -I_{(j+1)(n-k)} & \mathcal{F}_j \end{array}\right) \left(\begin{array}{c} y_i \\ \vdots \\ y_{i+j} \\ \hline u_i \\ \vdots \\ u_{i+j} \end{array}\right)$$

As we are interested in recovering the erasures in $v_i$, and if possible in $v_{i+1}, \dots, v_{i+j}$, we can omit the top block row in the previous equation. Hence, we have

$$\begin{pmatrix} CA^{i-1}B & \dots & CB \\ \vdots & & \vdots \\ CA^{i+j-1}B & \dots & CA^{j}B \end{pmatrix} \begin{pmatrix} u_0 \\ \vdots \\ u_{i-1} \end{pmatrix} + \left(-I_{(j+1)(n-k)} \mid \mathcal{F}_j\right) \left(\begin{array}{c} y_i \\ \vdots \\ y_{i+j} \\ \hline u_i \\ \vdots \\ u_{i+j} \end{array}\right) = 0. \quad (5.7)$$

So, there are two cases for the main decoding algorithm:

1. We can recover the erased symbols in $v_i$ (and if possible in $v_{i+1}, \dots, v_{i+j}$) using (5.7) for some delay $j \leq T$.

2. We cannot recover the erased symbols in $v_i$ using (5.7) for any delay $j \leq T$.

**Case 1:** Let $j$ be the smallest delay for which we can recover the erasures in $v_i$. We have that $x_i = A^{i-1}Bu_0 + \dots + Bu_{i-1}$ using the linear system equations. Hence, we are able to solve

$$\left(-I_{(j+1)(n-k)} \mid \mathcal{F}_j\right) \left(\begin{array}{c} y_i \\ \vdots \\ y_{i+j} \\ \hline u_i \\ \vdots \\ u_{i+j} \end{array}\right) = - \begin{pmatrix} C \\ CA \\ \vdots \\ CA^{j} \end{pmatrix} x_i. \quad (5.8)$$

Assuming without loss of generality that $x_i$ is known, we can compute $x_{i+1}$ using the linear system equations with the recovered symbols of $v_i$. If the recovery of the erasures in $v_i$ is not possible, we attempt to recover the earliest possible $x_{i+t}$ for $i+1 \leq i+t \leq l$ as outlined in Case 2, in order to restart the decoding process.

**Case 2:** When we cannot recover the erasures in $v_i$, we declare them as lost. However, to continue the decoding process, we need to determine $x_{i+1}$ which depends on $u_i$. To achieve this, we try to recover the earliest possible $x_{i+t}$ for $i+1 \leq i+t \leq l$, by solving the following system of linear equations with the smallest possible delay $j \leq T$:

$$\begin{pmatrix} C \\ CA \\ \vdots \\ CA^j \end{pmatrix} x_{i+t} + \left(-I_{(j+1)(n-k)} \;\middle|\; \mathcal{F}_j\right) \left(\begin{array}{c} y_{i+t} \\ \vdots \\ y_{i+t+j} \\ \hline u_{i+t} \\ \vdots \\ u_{i+t+j} \end{array}\right) = 0. \tag{5.9}$$

Note that it is sufficient to recover $x_{i+t}$ to restart the decoding process at $i+t$, even if parts of $v_0, \ldots, v_{i+t-1}$ contain erased symbols.

Additionally to the main decoding algorithm, there is a termination algorithm that we run before every decoding step of the low delay decoding algorithm for linear systems. This termination algorithm enables us to possibly finish the decoding process by correcting all the erased symbols in the received sequence. For this, we make use of Corollary 5.1.12.

**Corollary 5.1.12.** [18, Proposition 1] *For $w \in \mathbb{N}_0$ and $l \in \mathbb{N}_0$, define*

$$E_w := \begin{pmatrix} CA^lB & \ldots & CB \\ \vdots & & \vdots \\ CA^{l+w}B & \ldots & CA^wB \end{pmatrix}.$$

*Then for $c(z) = \sum_{i=0}^{l} \begin{pmatrix} y_{l-i} \\ u_{l-i} \end{pmatrix} z^i \in \mathcal{C}$, we have*

$$E_w(u_0, \ldots, u_l)^\top = 0. \tag{5.10}$$

*Proof.* The result follows from Equation (5.7) with $i = l+1$ and $j = w$. Recall that $y_i = u_i = 0$ for $i > l$. □

Therefore, if we can successfully recover all the erasures in $u(z)$ using (5.10), we can recover the erasures in $y(z)$ with (5.8) when $i = 0$ and $j = l$. The termination algorithm, outlined in Algorithm 4, uses this approach. Note that the termination process works independently of the chosen maximal delay $T$ and has the ability to recover symbols even before their arrival. The combination of the main decoding algorithm and termination algorithm gives the low delay decoding for linear systems (LDDLS) algorithm, which is described in Algorithm 3. Before every decoding step using the main decoding algorithm, we run the termination algorithm.

**Remark 5.1.13.** Since we start the decoding process at $v_0$ and then go to higher indices, we backward decode in some sense due to the reversed order of the coefficient indices for the linear system point of view. Hence, the sender of the sequence either has to wait until the whole codeword is encoded before sending it or works with the reverse code.

**Algorithm 3:** Low Delay Decoding Algorithm for Linear Systems

**Data:** The received sequence $v(z) = v_l + v_{l-1}z + \cdots + v_0 z^l$, maximal delay $T$.
**Result:** The corrected sequence $v^*(z) = v_l + v_{l-1}z + \cdots + v_0 z^l$.

```
i = 0;
while i ≤ l do
    if v_i = 0 then
        Try termination algorithm (see Algorithm 4);
        j = 0;
        success = 0;
        while success = 0 and j ≤ T do
            Calculate x_i;
            if Eq. (5.8) has unique solution for at least v_i then
                Replace erasures in v_i (and if possible in v_{i+1}, ..., v_{i+j}) with
                 correct symbols using Eq. (5.8);
                success = 1;
            else
                if j < T then
                    j = j + 1;
                else
                    j = 0;
                    t = 1;
                    while success = 0 do
                        if i + t ≤ l then
                            if Eq. (5.9) has unique solution for at least x_{i+t}
                             then
                                Recover x_{i+t} and replace if possible erasures in
                                 v_{i+t}, ..., v_{i+t+j} with correct symbols using Eq.
                                 (5.9);
                                success = 1;
                                i = i + t − 1;
                            else
                                if j < T then
                                    j = j + 1;
                                else
                                    j = 0;
                                    t = t + 1;
                                end
                            end
                        else
                            Stop Algorithm;
                        end
                    end
                end
            end
        end
    end
    i = i + 1;
end
```

**Algorithm 4:** Termination Algorithm

**Data:** The received sequence $v(z) = v_l + v_{l-1}z + \cdots + v_0 z^l$.
**Result:** The (fully) corrected sequence $v^*(z) = v_l + v_{l-1}z + \cdots + v_0 z^l$.
**if** $\exists w \in \{1, \ldots, \delta - 1\}$ *such that* $E_w^{(e)}$ *has full column rank* **then**
  Recover all erasures in $u_0, \ldots, u_l$ using Eq. (5.10);
  Recover all erasures in $y_0 \ldots, y_l$ using Eq. (5.8);
  Stop Algorithm 3;
**end**

We are now interested in which classes of convolutional codes are optimal for the LDDLS algorithm. We note that if a code fulfils the following properties, it is optimal for this algorithm: [18]

1. The not trivially zero minors of $\mathcal{F}_j$ are nonzero for $j = 1, \ldots, T$ (for Case 1).
2. The not trivially zero minors of $\left(\begin{array}{c|c} C & \\ \vdots & \mathcal{F}_j \\ CA^j & \end{array}\right)$ are nonzero for $j = 1, \ldots, T$ (for Case 2).
3. For as many sets of columns of $E_w$ as possible, there exists $w = 1, \ldots, \delta - 1$ such that these columns are linearly independent (for Termination algorithm).

By Proposition 3.2.6, Property 1 means that we have a convolutional code with maximal column distance up to $T$. If $T = L$, the code is MDP. Furthermore, we note that Property 2 implies Property 1. Constructing such a matrix can be achieved using a superregular matrix. However, it is difficult to ensure the third property is fulfilled.

**Corollary 5.1.14.** *Let $\mathcal{C}$ be an $(n, k, \delta)$ convolutional code with Property 1, 2 and 3, then the following holds for $j \leq T$:*

*(i) If any sliding window of length $(j+1)n$ at most $(j+1)(n-k)$ erasures occur, complete recovery is possible (for Case 1).*

*(ii) If $x_{i+t}$ is unknown and in any sliding window of length $(j+1)n$ at most $(j+1)(n-k) - \delta$ erasures occur, complete recovery is possible, including $x_{i+t}$ (for Case 2).*

*(iii) If the erasures in $u_0, \ldots, u_l$ correspond to a set of columns of $E_w$ for some $w = 1, \ldots, \delta - 1$ that are linearly independent, the whole sequence can be recovered (for termination algorithm).*

*Proof.* We note that the not trivially zero minors of $\mathcal{F}_j$ are nonzero for $j = 1, \ldots, T$, which implies (i) by Theorem 3.2.6. Similarly, (ii) follows, but we need to recover the $\delta$ symbols of $x_{i+t}$. Moreover, we need for (iii) that the erasures in $u_0, \ldots, u_l$ correspond to a set of linearly independent columns of $E_w$ for some $w = 1, \ldots, \delta - 1$ such that Equation (5.10) has a unique solution. By the Cayley-Hamilton Theorem[4],

[4]The Cayley-Hamilton Theorem states that the minimal polynomial of a square matrix $A$ over a field divides the characteristic polynomial of $A$. [3, Proposition 12.20]

it suffices to check the linear independence of the columns of $E_w$ up to $w = \delta - 1$ since any power of $A$ greater than or equal to $\delta$ can be expressed as a linear combination of lower powers of $A$. □

Next, we illustrate on an example how the LDDLS algorithm works.

**Example 5.1.15.** Consider the $(3, 2, 2)$ MDP convolutional code $\mathcal{C}$ that we constructed in Example 4.2.9. Recall that $(A, B, C, D)$ is a minimal ISO representation of $\mathcal{C}$ with

$$A = \begin{pmatrix} \alpha^{62} & \alpha^{3} \\ \alpha^{15} & \alpha^{56} \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad C = \begin{pmatrix} \alpha^{33} & \alpha^{9} \end{pmatrix} \quad \text{and} \quad D = \begin{pmatrix} \alpha & 1 \end{pmatrix},$$

and $\alpha$ is a root of the primitive polynomial $x^6 + x + 1 \in \mathbb{F}_2[z]$. Assume we want to decode the following sequence with maximal delay 1 using Algorithm 3:

| $v_0$ | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ |
|---|---|---|---|---|---|---|---|
| $\star\star\checkmark$ | $\star\checkmark\checkmark$ | $\checkmark\checkmark\checkmark$ | $\checkmark\checkmark\checkmark$ | $\checkmark\checkmark\checkmark$ | $\checkmark\checkmark\star$ | $\checkmark\star\checkmark$ | $\star\checkmark\checkmark$ |

Since we cannot correct the erasures in $v_0$ with delay 0 or 1, we try to recover $x_1$. As the not trivially zero minors of $\left(\begin{array}{c|c} C & \\ CA & \mathcal{F}_1 \end{array}\right)$ are nonzero, we can recover some $x_{i+t}$ if there are at most $(j+1)-2$ erasures in a sliding window of length $3(j+1)$. So, it is not possible to recover $x_1$. Nevertheless, we can recover $x_2$ with a delay of one and can therefore continue the decoding process. Note that there is one erasure in three symbols for $v_5$, which we can recover. Instead of continuing the main decoding algorithm, we can now apply the termination algorithm. At this decoding step, we have

| $u_0$ | $u_1$ | $u_2$ | $u_3$ | $u_4$ | $u_5$ | $u_6$ | $u_7$ |
|---|---|---|---|---|---|---|---|
| $\star\checkmark$ | $\checkmark\checkmark$ | $\checkmark\checkmark$ | $\checkmark\checkmark$ | $\checkmark\checkmark$ | $\checkmark\checkmark$ | $\star\checkmark$ | $\checkmark\checkmark$ |

Since the first column of $\begin{pmatrix} CA^7B \\ CA^8B \end{pmatrix}$ and of $\begin{pmatrix} CAB \\ CA^2B \end{pmatrix}$ are linearly independent, we can correct the erasures in $u_0$ and $u_6$. Afterwards, we can recover the remaining erasures in the sequence. Thus, the algorithm successfully recovers the entire sequence.

## 5.2 Comparison

In this section, we will compare the three erasure decoding algorithms. First, we look at what is needed in order to successfully recover erasures in some sliding window. For the FBD algorithm, a guard space of $\nu n$ symbols is required for each decoding step. However, this requirement can be restrictive depending on the erasure pattern. In contrast, the LDDM algorithm does not necessarily need a guard space, but the erasure correction capability is weaker when we do not have a guard space. For the LDDLS algorithm, instead of relying on a guard space, we require knowing $x_i \in \mathbb{F}^{\delta}$. Notably, if $(n-k) \mid \delta$, then the $\delta$ symbols of $x_i$ are fewer than the $\nu n = \delta \cdot \frac{n}{n-k}$ symbols for a guard space.

Another difference between the algorithms is the delay. For the low delay algorithms, we want to keep the delay as small as possible for each decoding step. Clearly, it

makes sense to have a delay smaller than $L$. Additionally, for the LDDLS algorithm we want to choose a delay larger than $\left\lceil \frac{\delta}{n-k} \right\rceil - 1$ such that we can possibly recover the $\delta$ symbols of some $x_{i+t}$. Moreover, we note that the FBD algorithm has a larger delay on average, since we try taking the largest possible delay for each decoding step. For backward decoding, the delay is even larger due to the guard space on the right. However, the algorithm can be rewritten such that we always try taking the smallest possible window size, but backward decoding still requires a large delay. Another possibility for the FBD algorithm is to incorporate the restart mechanism without a guard space used for the LDDM algorithm. This would allow us to recover more erasures and does not need a guard space.

The computational complexity of each of the three algorithms is determined by the complexity of solving a linear system of $(n-k)(j+1)$ equations to recover $e \leq (n-k)(j+1)$ erasures with a delay of $j$. According to [26], solving a general system of this size has complexity $\mathcal{O}(((n-k)(j+1))^{0.8} \cdot e^2)$. Thus, both the FBD and LDDM algorithm have this complexity. As it is computationally less complex to do several small decoding steps than doing one large decoding step, the LDDM algorithm has on average a lower computational complexity than the FBD algorithm. Furthermore, for the LDDLS algorithm we can use that parts of the system of linear equations are already in echelon form. Therefore, the erasure in $(y_i, \ldots, y_{i+l})$ can be neglected for the complexity analysis as they can directly be obtained using Equation (5.8) and $(u_i, \ldots, u_{i+j})$. To recover the $\hat{e} \leq e$ erasures in $(u_i, \ldots, u_{i+j})$, we have a computation complexity of $\mathcal{O}(((n-k)(j+1))^{0.8} \cdot \hat{e}^2)$.

| Algorithm | FBD | LDDM | LDDLS |
|---|---|---|---|
| Needed | Guard space ($\nu n$ symbols) | - | $x_i \in \mathbb{F}^\delta$ |
| Delay | Largest possible ($j \leq L$) | Smallest possible ($j \leq T \leq L$) | Smallest possible ($j \leq T \leq L$) |
| Computational Complexity | $\mathcal{O}(((n-k)(j+1))^{0.8} \cdot e^2)$ | $\mathcal{O}(((n-k)(j+1))^{0.8} \cdot e^2)$ | $\mathcal{O}(((n-k)(j+1))^{0.8} \cdot \hat{e}^2)$ |
| Optimal Class | Reverse MDP | Complete $T$-MDP | MDP with Property 2 |
| Optimal Erasure Recovery | $(j+1)(n-k)$ in $(j+1)n$ | $(j+1)(n-k)$ in $(j+1)n$ or $(j+1)(n-k)$ in $(\nu+j+1)n$ | $(j+1)(n-k)$ in $(j+1)n$ or $(j+1)(n-k)-\delta$ in $(j+1)n$ |
| | | | + Termination algorithm |

Table 5.1: Differences between the erasure decoding algorithms.

In order to have the best performance of each algorithm, we need to use the optimal code for it. As we have seen in Chapter 4, the required field size using the constructions can be large. This is important when choosing which algorithm to use. For the FBD algorithm, we also need to compute the reverse code. An additional dissimilarity is that we have a termination algorithm for the LDDLS algorithm, which reduces the decoding delay. One could use that a convolutional code with free distance $d$ can correct $d-1$ erasures to have something similar for the FBD and LDDM algorithm. In Table 5.1, the main differences between the algorithms are summed up.

## 5.3 Simulation

In this section, we apply the three algorithms in two simulations to observe the recovery of erasures over different channel models simulating practical scenarios. In a first simulation, we decode codewords that are approximately the size of the largest possible sliding window using the optimal code for each algorithm. This simulation allows us to evaluate the erasure correction performance against the one of MDS linear block codes, which can recover the most erasures for linear block codes. In a second simulation, we decode codewords larger than the largest possible sliding window. This scenario emulates situations such as streaming over the internet where codewords can be arbitrarily long. Further details about the simulations and the Sage codes used are available in Appendix A.

The first erasure channel that we use is the *(discrete) memoryless uniform channel*. Here, we assume that each symbol has the same probability of being an erasure [23]. The next erasure channel model we consider is the *Gilbert-Elliott channel*, introduced in [5] and [8]. For this model, the probability that a symbol is an erasure depends on the previous symbol. We assume that it is more likely to have an erasure if the previous symbol is also an erasure. That is,

$$0 < p_{c|e} < p_{e|e} < 1,$$

where $p_{c|e}$ denotes the probability of an erasure given that the previous symbol is correct, and similarly $p_{e|e}$ represents the probability of an erasure if the previous symbol is also an erasure. Therefore, we observe that erasures occur in bursts, making it a suitable model for simulating video streams over wireless networks, see for example [19]. The Gilbert-Elliott channel model can be represented as a first order Markov chain, as illustrated in Figure 5.1.

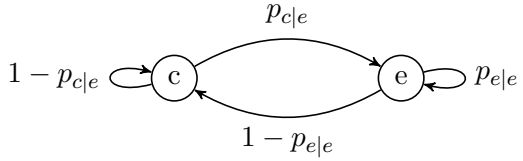


Figure 5.1: Representation of Gilbert-Elliott channel as a Markov chain.

In both simulations, we will assume for the Gilbert-Elliott channel that it is twice as likely that a symbol is an erasure given the previous symbol is an erasure than that a symbol is an erasure given the previous symbol is correct, meaning that $p_{e|e} = 2 \cdot p_{c|e}$. The third channel model, called *exponential channel*, also simulates streaming over the internet. The probability $p_i$, denoting that the $i$-th symbol is an erasure for some $i \geq 1$ and $0 < \lambda < 1$, is given by

$$p_i = e^{-i\lambda}.$$

In a streaming scenario, this intuitively means that we have a poor connection at the beginning, resulting in erasures occurring primarily at the start of a received sequence. As we progress towards the end of the sequence, the probability of an erasure decreases.

We will use the *recovery capability* to compare the results of the simulations. It is defined as

$$\Phi := \frac{\#\text{ recovered erasures}}{\#\text{ occured erasures}}.$$

In the first simulation, we use the respective optimal code for each algorithm. That is a reverse MDP convolutional code for the FBD algorithm, a complete $T$-MDP convolutional code for the LDDM algorithm with delay $T$, and an MDP convolutional code that satisfies Property 2 for the LDDLS algorithm. The parameters of the considered convolutional codes are selected such that their largest sliding window sizes are approximately 100 symbols each, while having different rates $R \in \{0.4, 0.5, 0.6, 0.75\}$. The algorithms will be applied to decode codewords of 100 symbols, ensuring comparability with the results obtained from $[100, K]$ MDS linear block codes of the same rates. The exact parameters can be found in Table 5.2.

| R | N | K | n | k | $\delta$ |
|---|---|---|---|---|---|
| 2/5 | 100 | 40 | 5 | 2 | 24 |
| 1/2 | 100 | 50 | 4 | 2 | 24 |
| 3/5 | 100 | 60 | 5 | 3 | 24 |
| 3/4 | 100 | 75 | 4 | 3 | 18 |

Table 5.2: Used parameters of $[N, K]$ MDS linear block codes and $(n, k, \delta)$ convolutional codes for the first simulation.

We start with randomly generating ten erasure patterns using each of the three previously described channel models, while keeping the number of erasures fixed. This approach ensures that the results are not overly dependent on one specific generated pattern. We then obtain the average recovery capability using the algorithms on these erasure patterns. By varying the number of erasures, we obtain the results of the simulation.

For the FBD algorithm, the results are presented in Figure 5.2. We observe that the recovery capability is higher for lower rates when the same number of erasures occur. It is noticeable that the full recovery of the received sequences is not as successful over the exponential channel compared to the other two channel models. However, if full recovery is not achievable over any channel, we are able to partially recover the most erasures over the exponential channel.
It is possible to recover more erasures with convolutional codes of lower rates, since the recovery capabilities depend on the code parameters, and lower rates allows us to correct more erasures. The recovery capability for the exponential channel is slightly lower than that for the other two channels. This is attributed to the concentration of erasures at the beginning of the sequence, which makes it harder to recover them. However, erasures occurring later in the sequence can be recovered more effectively due to their sparser distribution. Consequently, when full recovery is not attainable, we can recover a larger fraction of erasures compared to the other channels. Conversely, the erasures are more evenly distributed for the memoryless uniform and

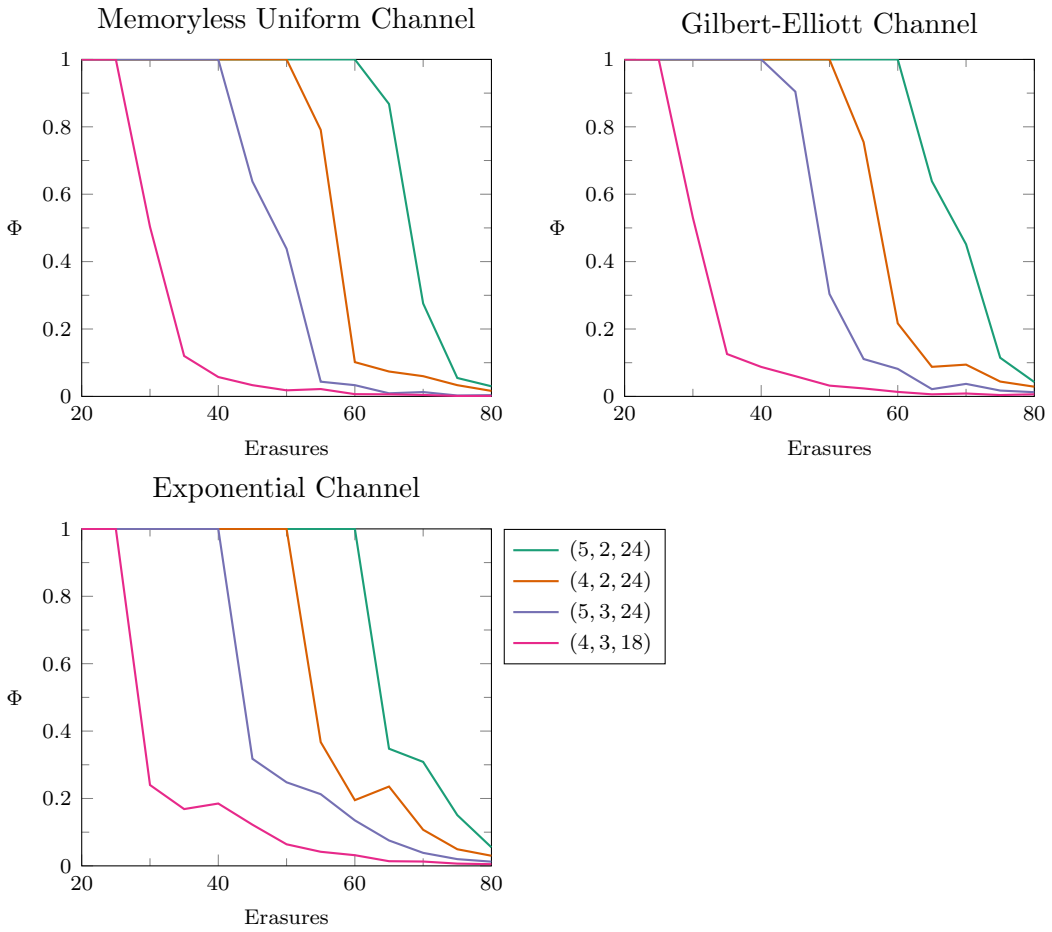


Figure 5.2: Average recovery capability of the FBD algorithm with reverse MDP convolutional codes over different erasure channels.

Gilbert-Elliott channels, which enhances the potential for fully recovering received sequences.

For the LDDM algorithm, we applied the algorithm once with delay $L/2$ and once with the largest possible delay $L$ to compare the impact of a smaller delay. The result are shown in Figure 5.3. We note, similarly as for the FBD algorithm, that the recovery capability is not as good over the exponential channel. The difference is even larger for the smaller delay $L/2$ compared to the memoryless uniform and Gilbert-Elliott channel. Although the difference between the delays $L/2$ and $L$ is not as significant for the memoryless uniform channel and the Gilbert-Elliott channel, the recovery capability changes more substantially with a smaller delay in the case of the exponential channel. Furthermore, it is worth noting that the recovery capabilities for the LDDM algorithm with delay $L$ and FBD algorithm are almost equal over all channels.

The reason for the more drastic decrease in recovery capability with a smaller delay over the exponential channel lies in the distribution of the erasures, as they are mainly located at the beginning of the sequence. In such a case, a larger delay is

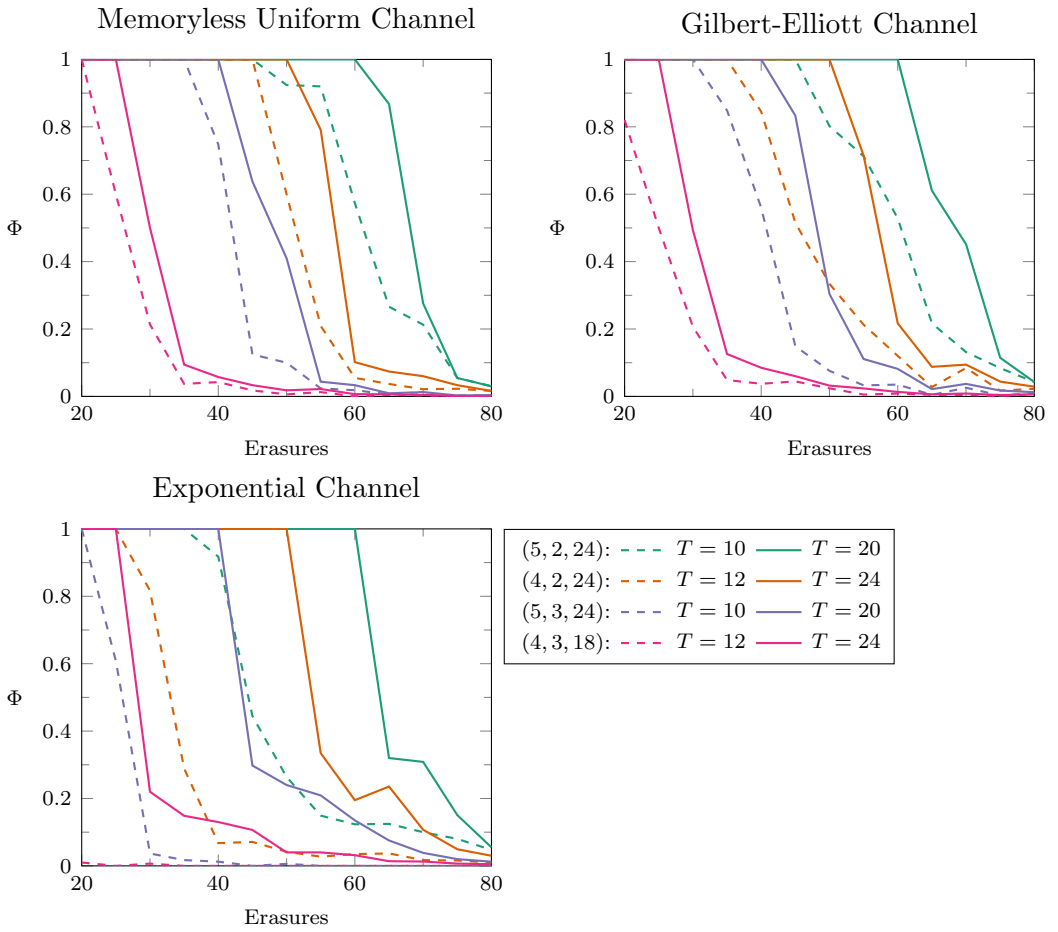


Figure 5.3: Average recovery capability of the LDDM algorithm with delay $T$ and complete $T$-MDP convolutional codes over different erasure channels.

beneficial. While the LDDM algorithm does not require a guard space, if there are only a few correctly received symbols before some $v_i$ that we want to recover (i.e., $|t-i|$ is small), a larger delay $j$ is necessary such that $s = i+j-t-\nu$ is nonnegative. Since the erasures are more evenly distributed for the memoryless uniform and Gilbert-Elliott channel, it is generally easier to recover them and therefore a smaller delay has not a big influence. The similarity with respect to the erasure correction performance between the LDDM algorithm with delay $L$ and the FBD algorithm is attributed to their shared use of forward decoding. They differ, since if forward decoding is unsuccessful, then we backward decode for the FBD algorithm, whereas we can recover erasure patterns satisfying Property (ii) for the LDDM algorithm. Since we need specific erasure patterns such that the approaches are different, the difference in the recovery capability between the two algorithms is minimal. However, the delay with the LDDM algorithm is on average smaller than with the FBD algorithm.

Lastly, we look at the simulation for the LDDLS algorithm. To recover some $x_{i+t}$, a delay of at least $\left\lfloor \frac{\delta}{n-k} \right\rfloor - 1$ is needed. However, with a delay of $\left\lfloor \frac{\delta}{n-k} \right\rfloor - 1$ we can only recover some $x_{i+t}$ if there are no erasures in the sliding window. There-

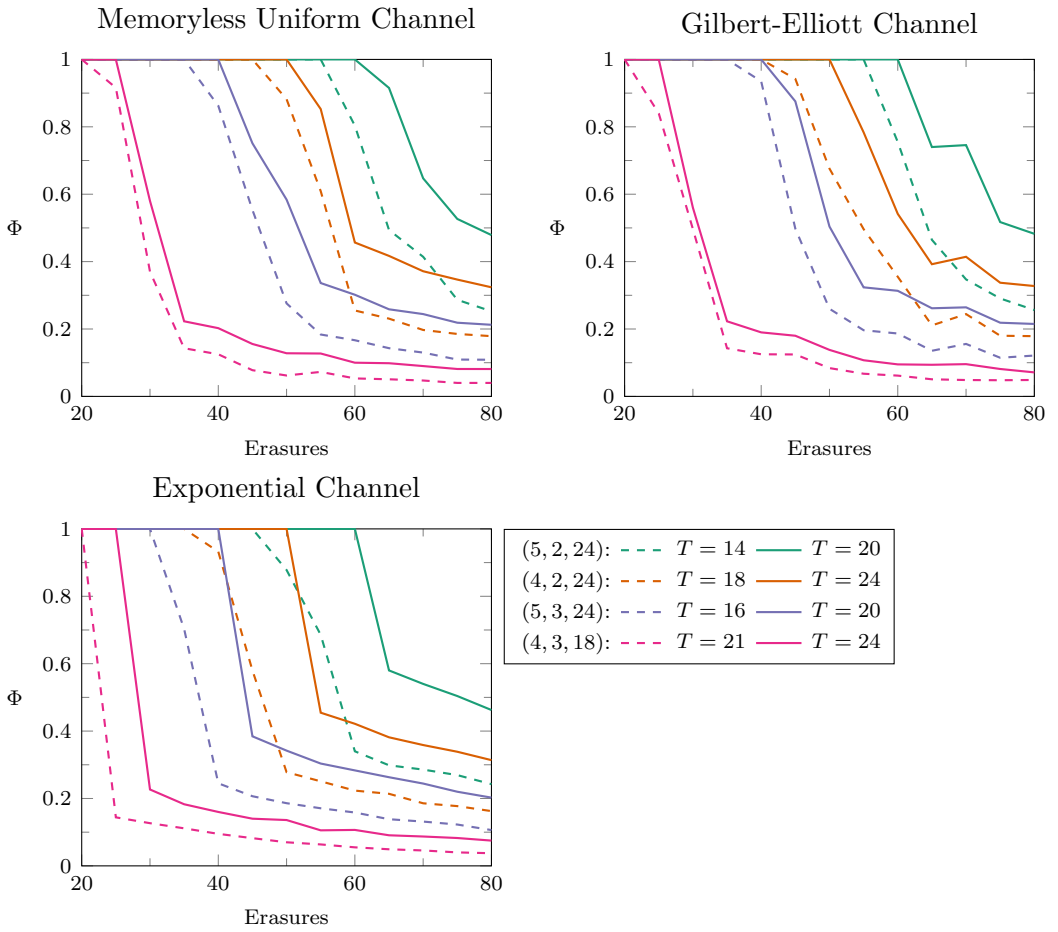


Figure 5.4: Average recovery capability of the LDDLS algorithm with delay $T$ and MDP convolutional codes fulfilling Property 2 over different erasure channels.

fore, we choose a delay that corresponds to the average of the minimal and maximal delay, meaning $T = \frac{L+\lfloor \frac{\delta}{n-k} \rfloor - 1}{2}$, along with the maximal delay $L$ for the simulation. In this simulation, we did only use the main decoding algorithm and omitted the termination algorithm, since there is no direct counterpart for the other two algorithms.

The corresponding plots are shown in Figure 5.4. The performance of this algorithm is notably better than the one of the FBD and the LDDM algorithm. This is primarily because we can always correct some of the occurring erasures, whereas for the other two algorithms we often were not able to correct any of the erasures once there were too many erasures. This outcome might be surprising initially, since the used class of code is the least restrictive. Additionally, we observe that the tradeoff between delay and performance is relatively small.

The impact of a smaller delay is not as noticeable as for the previous simulations. This is due to the smaller difference between the two chosen delays in this simulation of the algorithm. The better recovery capability of this algorithm follows, since when we declare some erasures as lost, the decoding process to recover some $x_{i+t}$ is less restrictive compared to the required guard space for the FBD algorithm and

the larger sliding window for the LDDM algorithm. Consequently, we can always recover some erasures, as restarting the decoding process is comparatively easier.

In summary, the simulation showed that the LDDLS algorithm has generally the highest recovery capability. This is due to how we can restart the decoding process once we declare some erasures as lost. A possible direction for future work could involve extending the LDDLS algorithm to incorporate the algebraic criteria for reverse MDP or complete $j$-MDP convolutional codes. The performance for the other two algorithms is very similar with a delay of $L$. The LDDM algorithm with the smaller delay achieves the lowest recovery capability. Additionally, we observe that over the exponential channel, the recovery of erasures is generally less efficient compared to the memoryless uniform channel and Gilbert-Elliott channel, due to the skewed distribution of erasures towards the beginning of the received codeword. This shows that the recovery capability of a convolutional code is influenced not only by the quantity of erasures but also by their distribution. It is worth mentioning that the results of this simulation depend on the chosen code parameters and codeword length. That is, if we choose a different $\delta$ and the codeword length such that it matches the largest sliding window, we obtain a similar result as in this simulation. In general, we saw that we are able to fully recover erasures as good as, or even better than, an MDS linear block code of the same rate and recovery rate per window. Recall that for an $[N, K]$ MDS linear block code we can fully recover at most $N - K$ erasures.

| R | n | k | $\delta$ |
|---|---|---|---|
| 1/3 | 3 | 1 | 2 |
| 1/2 | 4 | 2 | 2 |
| 3/5 | 5 | 3 | 2 |
| 2/3 | 3 | 2 | 2 |

Table 5.3: Used parameters of $(n, k, \delta)$ convolutional codes for the second simulation.

Now, we start with the second simulation. In this scenario, we again take four different convolutional codes with a similar largest possible sliding window size, while having different rates $R$. For this simulation, we use complete MDP convolutional codes constructed using the method outlined in Theorem 4.2.17 for all three algorithms. The parameters are shown in Table 5.3. The largest possible window size of these convolutional codes is around 12 symbols. However, in this simulation, we consider received codewords of 120 symbols. This simulates streaming over the internet, where the received sequences can be arbitrarily long, often exceeding the largest possible sliding window size. The rest of the simulation is done similarly as in the first one.

We start with the simulation using the FBD algorithm. The corresponding plots are shown in Figure 5.5. Since complete MDP convolutional codes are reverse MDP, we use an optimal code for this algorithm. Firstly, it is important to note that the

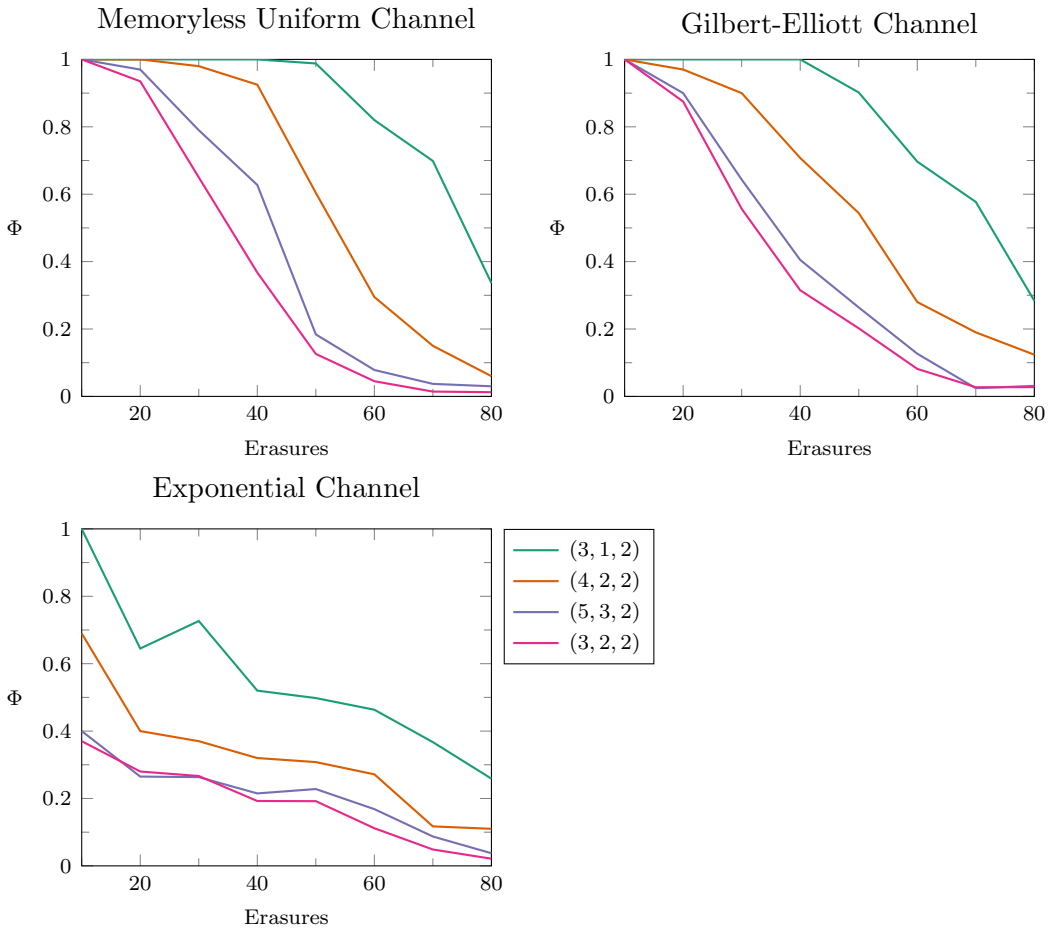


Figure 5.5: Average recovery capability of the FBD algorithm with complete MDP convolutional codes over different erasure channels.

recovery capability in this simulation is generally lower compared to the previous one. Additionally, the performance over the memoryless uniform channel is slightly better than over the Gilbert-Elliott channel. However, the recovery capability over the exponential channel is notably worse than the one of the other two channels. Achieving full recovery over the exponential channel is already difficult with only a few erasures present. Despite this, we can recover most erasures in the middle and at the end of the codewords. This is visible through the flat slope of the plots.

The decreased recovery capability in this simulation follows from the relatively small largest sliding window size. This smaller window size makes the erasure recovery more difficult, since the distribution of the erasures has a bigger influence. Even small bursts of erasures can be difficult to recover. As a result, the best performance happens over the memoryless uniform channel, followed by the Gilbert-Elliott channel. In contrast, over the exponential channel most erasures occur at the beginning of the received sequence. Due to this distribution, recovery with a small window becomes difficult. However, the distribution of erasures in the middle and at the end allows for their successful recovery. From this, we see that a poor connection at the beginning of an internet transmission does not significantly affect the remainder

of the transmission.

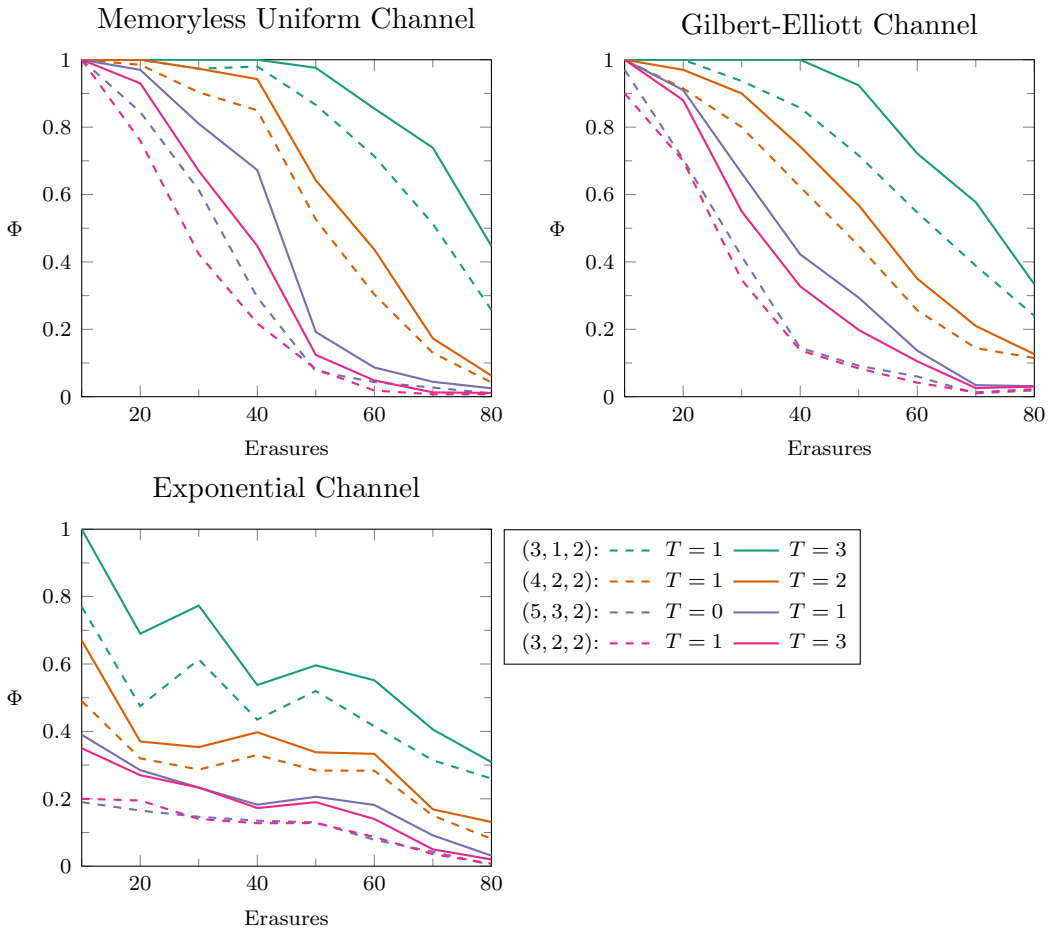


Figure 5.6: Average recovery capability of the LDDM algorithm with delay $T$ and complete MDP convolutional codes over different erasure channels.

Next, we look at the average recovery capability using the LDDM algorithm. We apply the algorithm once with a smaller delay $\lfloor \frac{L}{2} \rfloor$ and once with the largest possible delay $L$. As we are using a complete MDP convolutional code, we apply the algorithm on its optimal code with respect to erasure recovery. The results are shown in Figure 5.6. Again, the recovery capabilities using the LDDM algorithm with delay $L$ and the FBD algorithm are very similar. Furthermore, the impact of a smaller delay is not particularly significant. The reasoning for the described observation are the same as for the first simulation.

For the LDDLS algorithm, we apply the algorithm twice with the same delays as those used for the LDDM algorithm. Here, a smaller delay of $\lfloor \frac{L}{2} \rfloor$ is sufficient to restart the decoding process. Since we use complete MDP convolutional codes, they may not necessarily satisfy Property 2. As only a few of the not trivially zero minors are zero in our case, it does not significantly change the results. In this simulation, we also use the termination algorithm, and the results are shown in Figure 5.7.

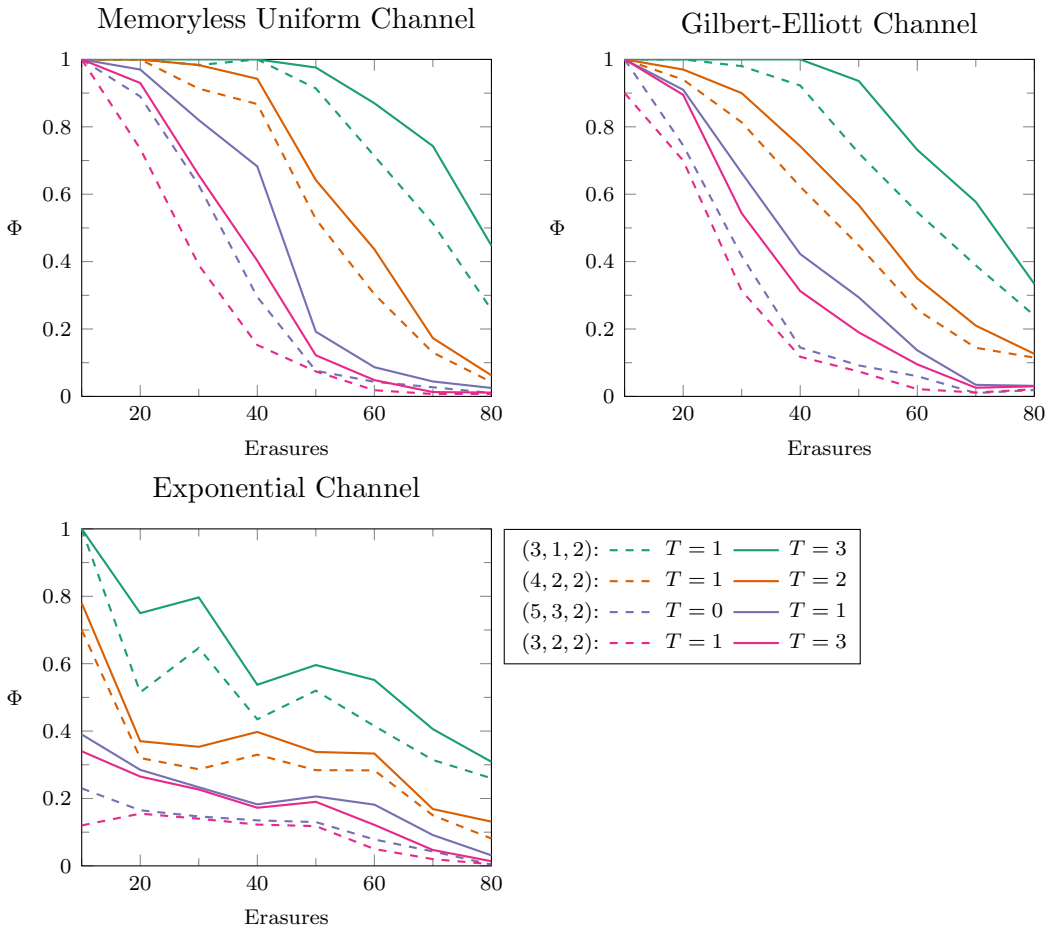


Figure 5.7: Average recovery capability of the LDDLS algorithm with delay $T$ and complete MDP convolutional codes over different erasure channels.

In contrast to the first simulation, the results for the LDDM and the LDDLS algorithm are highly similar. Thus, all three algorithms have similar average recovery capabilities. This similarity follows from the relatively small degrees $\delta$. Additionally, as the received sequences are longer relative to the largest possible sliding window, the differences between the algorithms are further evened out. In the first simulation, this was not the case, and thus the LDDLS algorithm performed better.

# Chapter 6

# Conclusion

In this thesis, we gave an overview of different classes of maximum distance profile (MDP) convolutional codes, namely MDP, reverse MDP and complete $j$-MDP convolutional codes. For all these classes of MDP convolutional codes, we outlined some of their constructions and discussed their relations between each other. The constructions automatically give us an existence statement. However, they heavily depend on the field size, and, for MDP, reverse MDP and complete $j$-MDP convolutional codes, finding the minimum required field size for the existence proof and finding good constructions remains an open problem.

Subsequently, we introduced and compared the following erasure decoding algorithms: The forward and backward decoding (FBD) algorithm, the low delay decoding for modules (LDDM) algorithm and the low delay decoding for linear systems (LDDLS) algorithm. Our analysis considered their delays, computational complexities, and erasure recovery capabilities.
The FBD algorithm combines forward decoding of a code and of its reverse code. This results in a larger average delay compared to the LDDM and LDDLS algorithm.
The LDDM algorithm is an improved version of the FBD algorithm. Instead of backward decoding when forward decoding is not possible, we try to restart the decoding process without a guard space. This leads to a smaller average delay, making it a low delay decoding algorithm. Additionally, we set a maximal delay within which the decoding process of an erased symbol must be completed.
Another low delay decoding algorithm introduced is the LDDLS algorithm, which uses a linear system representation of a convolutional code. This algorithm offers a reduced computational complexity, as parts of the codeword can be neglected for the complexity analysis. Furthermore, the algorithm includes a termination algorithm to potentially stop the decoding before the end of transmission.

In the first simulation, we assessed the recovery capabilities of the algorithms using their respective optimal convolutional code classes and compared them to MDS linear block codes of equivalent rates and erasure correction capabilities over different erasure channel models. Our results demonstrated that convolutional codes matched or even outperformed the performance of comparable MDS linear block codes. The LDDLS algorithm generally had the highest recovery capability due to its ability to restart the decoding process when some erased symbols are declared as lost. A

future work is to extend the algorithm to use the algebraic criteria for reverse MDP or complete $j$-MDP convolutional codes. We observed that the performance over the memoryless uniform and Gilbert-Elliott channel was generally similar, while the exponential channel had a weaker performance due to the skewed distribution of erasures towards the beginning of a received sequence. Thus, not only the quantity of erasures but also the erasure distribution is important for the recovery capability. In the second simulation, we applied the algorithms to longer received sequences relative to the largest possible sliding window of the used complete MDP convolutional codes. Here, the performances of the three algorithms were more similar to each other due to the specific code parameters and the longer received sequences. However, the recovery capabilities were lower compared to the first simulation, particularly over the exponential channel.

# Bibliography


[1] P. Almeida, D. Napp, and R. Pinto. Superregular matrices and applications to convolutional codes. *Linear Algebra and its Applications*, 499:1–25, 2016.

[2] P. J. Almeida and J. Lieb. Complete j-MDP convolutional codes. *IEEE Transactions on Information Theory*, 66(12):7348–7359, 2020.

[3] D. S. Dummit and R. M. Foote. *Abstract algebra*, volume 3. Wiley Hoboken, 2004.

[4] P. Elias. Coding for noisy channels. In *IRE WESCON Convention Record, 1955*, volume 2, pages 94–104, 1955.

[5] E. O. Elliott. Estimates of error rates for codes on burst-noise channels. *The Bell System Technical Journal*, 42(5):1977–1997, 1963.

[6] E. Gabidulin. A brief survey of metrics in coding theory. *Mathematics of Distances and Applications*, 66:66–84, 2012.

[7] R. G. Gallager. *Information theory and reliable communication*, volume 588. Springer, 1968.

[8] E. N. Gilbert. Capacity of a burst-noise channel. *Bell system technical journal*, 39(5):1253–1265, 1960.

[9] H. Gluesing-Luerssen, J. Rosenthal, and R. Smarandache. Strongly-MDS convolutional codes. *IEEE Transactions on Information Theory*, 52(2):584–598, 2006.

[10] R. Hartshorne. *Algebraic geometry*, volume 52. Springer Science & Business Media, 2013.

[11] R. Hutchinson. The existence of strongly MDS convolutional codes. *SIAM journal on control and optimization*, 47(6):2812–2826, 2009.

[12] R. Hutchinson, J. Rosenthal, and R. Smarandache. Convolutional codes with maximum distance profile. *Systems & control letters*, 54(1):53–63, 2005.

[13] R. Hutchinson, R. Smarandache, and J. Trumpf. On superregular matrices and MDP convolutional codes. *Linear algebra and its applications*, 428(11-12):2585–2596, 2008.

[14] T. Kailath. *Linear systems*, volume 156. Prentice-Hall Englewood Cliffs, NJ, 1980.

[15] J. Lieb. Complete MDP convolutional codes. *Journal of Algebra and Its Applications*, 18(06):1950105, 2019.

[16] J. Lieb. Necessary field size and probability for MDP and complete MDP convolutional codes. *Designs, Codes and Cryptography*, 87(12):3019–3043, 2019.

[17] J. Lieb, R. Pinto, and J. Rosenthal. Convolutional codes. In W. C. Huffman, J.-L. Kim, and P. Solé, editors, *Concise encyclopedia of coding theory*. CRC Press, 2021.

[18] J. Lieb and J. Rosenthal. Erasure decoding of convolutional codes using first-order representations. *Mathematics of Control, Signals, and Systems*, 33(3):499–513, 2021.

[19] M. M. Nasralla, C. T. Hewage, and M. G. Martini. Subjective and objective evaluation and packet loss modeling for 3D video transmission over LTE networks. In *2014 international conference on telecommunications and multimedia (temu)*, pages 254–259. IEEE, 2014.

[20] M. Ravi and J. Rosenthal. A smooth compactification of the space of transfer functions with fixed mcmillan degree. *Acta Applicandae Mathematica*, 34:329–352, 1994.

[21] J. Rosenthal. Connections between linear systems and convolutional codes. In *Codes, Systems, and Graphical Models*, pages 39–66. Springer, 2001.

[22] J. Rosenthal and R. Smarandache. Maximum distance separable convolutional codes. *Applicable Algebra in Engineering, Communication and Computing*, 10:15–32, 1999.

[23] R. Roth. *Introduction to Coding Theory*. Cambridge University Press, USA, 2006.

[24] C. E. Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[25] R. Singleton. Maximum distance q-nary codes. *IEEE Transactions on Information Theory*, 10(2):116–118, 1964.

[26] V. Strassen et al. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[27] V. Tomás. *Complete-MDP convolutional codes over the erasure channel*. PhD thesis, Departamento de Ciencia de la Computación e Inteligencia Artificial, Universidad de Alicante, 2010.

[28] V. Tomás, J. Rosenthal, and R. Smarandache. Decoding of MDP convolutional codes over the erasure channel. In *Proceedings of the 2009 IEEE International Symposium on Information Theory*, pages 556–560, 2009.

[29] V. Tomás, J. Rosenthal, and R. Smarandache. Reverse-maximum distance profile convolutional codes over the erasure channel. In *Proceedings of the 19th International Symposium on Mathematical Theory of Networks and Systems (MTNS2010)*, pages 2121–2127, 2010.

[30] V. Tomás, J. Rosenthal, and R. Smarandache. Decoding of convolutional codes over the erasure channel. *IEEE Transactions on Information Theory*, 58(1):90–108, 2012.

[31] E. Von York. *Algebraic description and construction of error-correcting codes: A linear systems point of view.* University of Notre Dame, 1997.

# Appendix A

# Sage Code

In Section 5.3, we used the following Sage codes for our simulations. We first defined the functions for generating erasure patterns, which model the channels for the simulations. In these patterns, zero denotes an erased symbol, while one denotes a correctly received symbol:

```
import random
import math


def memoryless_channel(length, p):
  """
  Function to generate an erasure pattern of a (discrete) memoryless
  ↪  uniform channel
  :param length: Number of symbols
  :param p: Probability that a symbol is an erasure (0 < p < 1)
  :return: Erasure pattern for a codeword
  """
  correct = ()
  for l in range(length):
    if random.random() <= p:
      correct = correct + (0,)
    else:
      correct = correct + (1,)

  return correct


def memoryless_channel_fixed_number(length, p, N):
  """
  Function to generate an erasure pattern of a (discrete) memoryless
  ↪  uniform channel with an arbitrary but fixed number of erasures
  :param length: Number of symbols
  :param p: Probability that a symbol is an erasure (0 < p < 1)
  :param N: Number of erasures
  :return: Erasure pattern for a codeword
  """
  correct = ()
  while correct.count(0) != N:
    correct = memoryless_channel(length, p)
```

```
  return correct


def gilbert_elliott_channel(length, p, q):
  """
  Function to generate an erasure pattern of a Gilbert-Elliott channel
  :param length: Number of symbols
  :param p: Probability that a symbol is an erasure given the previous
  ↪  symbol is also an erasure (0 < p < 1)
  :param q: Probability that a symbol is an erasure given the previous
  ↪  symbol is correct (0 < q < 1)
  :return: Erasure pattern for a codeword
  """
  correct = ()
  for l in range(length):
    if l == 0:
      if random.random() <= q:
        correct = correct + (0,)
      else:
        correct = correct + (1,)
    else:
      if correct[-1] == 0:
        if random.random() <= p:
          correct = correct + (0,)
        else:
          correct = correct + (1,)
      elif correct[-1] == 1:
        if random.random() <= q:
          correct = correct + (0,)
        else:
          correct = correct + (1,)

  return correct


def gilbert_elliott_channel_fixed_number(length, p, q, N):
  """
  Function to generate an erasure pattern of a Gilbert-Elliott channel with
  ↪  an arbitrary but fixed number of erasures
  :param length: Number of symbols
  :param p: Probability that a symbol is an erasure given the previous
  ↪  symbol is also an erasure (0 < p < 1)
  :param q: Probability that a symbol is an erasure given the previous
  ↪  symbol is correct (0 < q < 1)
  :param N: Number of erasures
  :return: Erasure pattern for a codeword
  """
  correct = ()
  while correct.count(0) != N:
    correct = gilbert_elliott_channel(length, p, q)

  return correct
```

```
def exponential_channel(length, c):
  """
  Function to generate an erasure pattern of an exponential channel
  :param length: Number of symbols
  :param c: Parameter for probability (0 < c < 1)
  :return: Erasure pattern for a codeword
  """
  correct = ()
  for l in range(1, length + 1):
    if random.random() <= math.exp(- l * c):
      correct = correct + (0,)
    else:
      correct = correct + (1,)

  return correct


def exponential_channel_fixed_number(length, c, N):
  """
  Function to generate an erasure pattern of an exponential channel with an
  ↪  arbitrary but fixed number of erasures
  :param length: Number of symbols
  :param c: Parameter for probability (0 < c < 1)
  :param N: Number of erasures
  :return: Erasure pattern for a codeword
  """
  correct = ()
  while correct.count(0) != N:
    correct = exponential_channel(length, c)

  return correct
```

To make the different channel models more comparable, we always used the same number of erasures for each respective simulation.
For implementation of the decoding algorithms, we first established the following auxiliary functions:

```
from sage.all import *


def msg_to_list(msg):
  """
  Function to get the list of symbols of a codeword
  :param msg: Codeword (i.e. vector with polynomial entries)
  :return: List with symbols of the codeword (i.e. [v_0, v_1, ...,
  ↪  v_{length}])
  """
  msg_list = [msg.coefficient_matrix(l).list() for l in range(msg.degree()
  ↪  + 1)]
  msg_list = [l for l0 in msg_list for l in l0]

  return msg_list
```

```
def list_to_msg(msg_list, n, field):
    """
    Function to reverse msg_to_list()
    :param msg_list: List with symbols of a codeword (i.e. [v_0, v_1, ...,
    ↪  v_{length}])
    :param n: Length of the code
    :param field: Field of coefficients of the codeword
    :return: Codeword (i.e. vector with polynomial entries)
    """
    F = field['z']
    z = F.gen()
    msg = matrix(F, 1, n, 0)
    for l in range(len(msg_list) // n):
        msg = msg + matrix(F, [msg_list[l0] * z ** l for l0 in range(l * n, (l
        ↪  + 1) * n)])

    return msg


def msg_to_yu(msg, n, k, length):
    """
    Function to get the symbols of y and u as lists from a codeword
    :param msg: Codeword (i.e. vector with polynomial entries (coefficients
    ↪  order is reversed here))
    :param n: Length of the code
    :param k: Dimension of the code
    :param length: Maximal degree of an entry of the codeword
    :return: List of symbols of y and u (i.e. [y_0, ..., y_{length}] & [u_0,
    ↪  ..., u_{length}])
    """
    msg_list = [msg.coefficient_matrix(l).list() for l in range(length + 1)]
    y = [msg_list[l][l0] for l in range(length, -1, -1) for l0 in range(n -
    ↪  k)]
    u = [msg_list[l][l0] for l in range(length, -1, -1) for l0 in range(n -
    ↪  k, n)]

    return y, u


def yu_to_msg(y, u, n, k, length, field):
    """
    Function to reverse msg_to_yu()
    :param y: List of symbols of y
    :param u: List of symbols of u
    :param n: Length of the code
    :param k: Dimension of the code
    :param length: Maximal degree of an entry of the codeword
    :param field: Field of the coefficients
    :return: Codeword (vector with polynomial entries (coefficients are
    ↪  reversed here))
    """
```

```
    F = field['z']
    z = F.gen()
    msg = matrix(F, 1, n, 0)
    for l in range(length, -1, -1):
        v_i = [y[l0] for l0 in range(l * (n - k), (l + 1) * (n - k))] + [u[l0]
        ↪  for l0 in range(l * k, (l + 1) * k)]
        msg = msg + matrix(F, v_i) * z ** (length - l)

    return msg


def correct_yu(correct, n, k, length):
    """
    Function to separate a correct list into parts regarding y and u with the
    ↪  same order as we decode
    :param correct: Correct list ([v_0, ..., v_{length}]_{correct})
    :param n: Length of code
    :param k: Dimension of code
    :param length: Maximal degree of an entry of the codeword
    :return: Correct lists regarding y & u (i.e. [y_{0}, ...,
    ↪  y_{length}]_{correct} & [u_0, ..., u__{length}]_{correct})
    """
    correct_y = [correct[l * n + l0] for l in range(length + 1) for l0 in
    ↪  range(n - k)]
    correct_u = [correct[l * n + l0] for l in range(length + 1) for l0 in
    ↪  range(n - k, n)]

    return correct_y, correct_u


def correct_yu_to_list(correct_y, correct_u, n, k, length):
    """
    Function to reverse correct_yu()
    :param correct_y: Correct list regarding y (i.e. [y_0, ...,
    ↪  y_{length}]_{correct})
    :param correct_u: Correct list regarding u (i.e. [u_0, ...,
    ↪  u_{length}]_{correct})
    :param n: Length of code
    :param k: Dimension of code
    :param length: Maximal degree of an entry of the codeword
    :return: Correct list (i.e. [v_0, ..., v_{length}]_{correct})
    """
    correct_list = []
    for l in range(length + 1):
        correct_list = correct_list + correct_y[l * (n - k):(l + 1) * (n - k)]
        ↪  + correct_u[l * k:(l + 1) * k]

    return correct_list


def correct_cjmdp(n, k, j, v, s, t, index_unknown):
    """
```

```
    Function to check whether the erasures in a window of (v + j + 1)n
    ↪ symbols can be recovered with a complete j-MDP convolutional code
    ↪ when there is no guard space
    :param n: Length of code
    :param k: Dimension of code
    :param j: Delay
    :param v: Maximal degree of entry in parity-check matrix
    :param s: Number of block rows the partial parity-check matrix has for
    ↪ the decoding
    :param t: First index of window we use to decode
    :param index_unknown: List of positions of erasures in window
    :return: "Not correctable" if full recovery is not possible; "None" if
    ↪ full recovery is possible
    """
    if s != 0:
      for l in range(1, j + 1):
        if len([l0 for l0 in index_unknown if l0 in range(t * n, (t + l) *
        ↪ n)]) > l * (n - k) or len([l0 for l0 in index_unknown if l0 in
        ↪ range((t + v + j + 1 - l) * n, (t + v + j + 1) * n)]) > l * (n -
        ↪ k):
          return "Not correctable"


def partial_pcm(H, j, n, k, field):
    """
    Function to construct a partial parity-check matrix
    :param H: Parity-check matrix
    :param j: Number of block rows the partial parity-check matrix should
    ↪ have
    :param n: Length of code
    :param k: Dimension of code
    :param field: Field of the coefficients
    :return: j-th partial parity-check matrix
    """
    v = H.degree()
    Zero_Mat = matrix(field, n - k, n, 0)

    # Constructing the first row
    row = H.coefficient_matrix(v)
    for l in range(v - 1, -(j + 1), -1):
      if l > -1:
        row = row.augment(H.coefficient_matrix(l))
      else:
        row = row.augment(Zero_Mat)

    Partial_pcm = row

    # Constructing the partial parity-check matrix using the previous row
    for l in range(j):
      row = row.delete_columns(list(range(row.ncols() - n, row.ncols())))
      row = Zero_Mat.augment(row)
      Partial_pcm = Partial_pcm.stack(row)
```

```
  return Partial_pcm


def systematic_mat(A, B, C, D, j, n, k, field):
  """
  Function to construct a systematic matrix (i.e. (-I | \mathcal{F}_j))
  :param A: A of an ISO representation
  :param B: B of an ISO representation
  :param C: C of an ISO representation
  :param D: D of an ISO representation
  :param j: Number of block rows the systematic matrix should have
  :param n: Length of code
  :param k: Dimension of code
  :param field: Field of the coefficients
  :return: Systematic matrix with j block rows
  """
  # Constructing the last row of \mathcal{F}_j
  Zero_Mat = matrix(field, n - k, k, 0)

  if j == 0:
    Systematic_mat = D
  else:
    row = C * A ** (j - 1) * B
    for l in range(j - 2, -1, -1):
      row = row.augment(C * A ** l * B)

    row = row.augment(D)
    Systematic_mat = row

    # Constructing \mathcal{F}_j using the previous row
    for l in range(j):
      row = row.delete_columns(list(range(k)))
      row = row.augment(Zero_Mat)
      Systematic_mat = row.stack(Systematic_mat)

  # Adding the identity matrix
  Id = - identity_matrix(field, (j + 1) * (n - k))
  Systematic_mat = Id.augment(Systematic_mat)

  return Systematic_mat


def obs_mat(A, C, j):
  """
  Function to construct an observability matrix
  :param A: A of an ISO representation
  :param C: C of an ISO representation
  :param j: Number of block rows the observability matrix should have
  :return: Observability matrix with j block rows
  """
  Obs_mat = C
  for l in range(1, j + 1):
    Obs_mat = Obs_mat.stack(C * A ** l)
```

```
  return Obs_mat


def e_w(A, B, C, w, length, k):
  """
  Function to construct E_w for the termination algorithm
  :param A: A of an ISO representation
  :param B: B of an ISO representation
  :param C: C of an ISO representation
  :param w: Number of block rows E_w should have
  :param length: Maximal degree of an entry of the codeword
  :param k: Dimension of code
  :return: E_w
  """
  # Constructing the first row of E_w
  row = C * A ** length * B
  for l in range(length - 1, -1, -1):
    row = row.augment(C * A ** l * B)

  E_w = row

  # Constructing E_w using the previous row
  for l in range(1, w + 1):
    row = row.delete_columns(list(range(row.ncols() - k, row.ncols())))
    row = (C * A ** (length + l) * B).augment(row)

    E_w = E_w.stack(row)

  return E_w


def update_x_i(x, A, B, u, k, old_index, new_index):
  """
  Function to update x_{old_index} to x_{new_index}
  :param x: x_{old_index}
  :param A: A of an ISO representation
  :param B: B of an ISO representation
  :param u: u part of codeword
  :param k: Dimension of code
  :param old_index: Index of x
  :param new_index: Index of x we want to update it to
  :return: x_{new_index}
  """
  for l in range(old_index, new_index):
    u_l = vector([u[l0] for l0 in range(l * k, (l + 1) * k)])
    x = A * x + B * u_l

  return [x, new_index]


def solvable(A, b, index_unknown, s, i, n, k, delta, mode):
  """
```

```
    Function to check whether A * x = b can be solved for x with a unique
    ↪  solution or for a part of x which is unique
    :param A: Matrix A
    :param b: Vector b
    :param index_unknown: List of indices of erasures in window
    :param s: Number of block rows of A
    :param i: Index of first v_i in window containing erasures
    :param n: Length of code
    :param k: Dimension of code
    :param delta: Degree of code
    :param mode: "yu" to check whether at least y_i & u_i is recoverable; "x"
    ↪  to check whether at least x_{t+ i} is recoverable; "v" to check
    ↪  whether at least v_i is recoverable
    :return: "Unique" if full window is recoverable; "Not unique" if at least
    ↪  parts determined with mode are recoverable
    """
    unknowns = 0
    for l in index_unknown:
      unknowns = unknowns + len(l)

    if A.augment(b).rank() == A.rank() == unknowns != 0 and s >= 0:
      # System has unique solution
      return "Unique"

    elif A.augment(b).rank() == A.rank() != 0 and mode == "yu":
      # System has multiple solutions for the LDDLS algorithm, but y_i and
      ↪  u_i are unique
      unknown_y_i = len([l for l in index_unknown[0] if l in range(i * (n -
      ↪  k), (i + 1) * (n - k))])
      unknown_u_i = len([l for l in index_unknown[1] if l in range(i * k, (i
      ↪  + 1) * k)])

      base = A.right_kernel_matrix()

      if base.matrix_from_columns(list(range(unknown_y_i))).is_zero() and
      ↪  base.matrix_from_columns(
          list(range(len(index_unknown[0]), len(index_unknown[0]) +
          ↪  unknown_u_i))).is_zero():
        return "Not unique"

    elif A.augment(b).rank() == A.rank() != 0 and mode == "x":
      # System has multiple solutions for the LDDLS algorithm, but x_{i + t}
      ↪  is unique
      base = A.right_kernel_matrix()

      if base.matrix_from_columns(list(range(base.ncols() - delta,
      ↪  base.ncols()))).is_zero():
        return "Not unique"

    elif A.augment(b).rank() == A.rank() != 0 and mode == "v" and s >= 0:
      # System has multiple solutions for the LDDM algorithm, but v_i is
      ↪  unique
      base = A.right_kernel_matrix()
```

```
    unknown_v_i = len([l for l in index_unknown[0] if l in range(i * n, (i
    ↪  + 1) * n)])

    if base.matrix_from_columns(list(range(unknown_v_i))).is_zero():
      return "Not unique"


def termination_check(E_list, index_unknown_u):
  """
  Function to check whether the termination algorithm can be used for the
  ↪  LDDLS algorithm
  :param E_list: List of E_w's
  :param index_unknown_u: Correct list regarding u
  :return: Index "w" of E_w which can be used for termination algorithm;
  ↪  "-1" if termination algorithm cannot be used
  """
  for E in E_list:
    if E.matrix_from_columns(index_unknown_u).rank() ==
    ↪  E.matrix_from_columns(index_unknown_u).ncols():
      return E_list.index(E)

  return -1


def abcd(H, delta, field):
  """
  Function to get a minimal ISO representation (A, B, C, D) via a
  ↪  parity-check matrix
  :param H: Parity-check matrix
  :param delta: Degree of code
  :param field: Field of the coefficients
  :return: A, B, C, D of an ISO representation
  """
  F = field['z']
  z = F.gen()

  G = H.right_kernel_matrix()
  n = G.ncols()
  k = G.nrows()
  G = G.reduced_form()

  row_degrees = [[G.matrix_from_rows([l]).degree(), l] for l in range(k)]
  row_degrees_sorted = sorted(row_degrees, key=lambda x: x[0],
  ↪  reverse=True)
  G = G.matrix_from_rows([row_degrees_sorted[l][1] for l in range(k)])

  X_i = []
  for l in [l0[0] for l0 in row_degrees_sorted if l0[0] != 0]:
    X_i.append(matrix(F, [z ** (l0 - 1) for l0 in range(1, l + 1)]))

  X_tilde = block_diagonal_matrix(X_i)
  X = X_tilde.stack(matrix(F, k - len(X_i), delta, 0))
```

```
  K = (z * X).augment(X).augment(G)
  K = K.dense_matrix()

  rows = []
  for l in range(k):
    for l0 in range(K.matrix_from_rows([l]).degree() + 1):
      rows.append(K.matrix_from_rows([l]).coefficient_matrix(l0).list())

  I = - matrix(field, rows).right_kernel_matrix().transpose()

  Zero_Mat = matrix(field, delta, n - k, 0)
  Id_delta = identity_matrix(field, delta)
  Id_nk = identity_matrix(field, n - k)

  X = I[2 * delta:2 * delta + (n - k), delta:delta + (n - k)]
  I = I * block_matrix([[Id_delta, Zero_Mat], [Zero_Mat.transpose(),
  ↪  -X.inverse()]])

  Y = I[2 * delta:2 * delta + (n - k), 0:delta]
  I = I * block_matrix([[Id_delta, Zero_Mat], [Y, Id_nk]])

  A = I[delta:2 * delta, 0:delta].transpose()
  B = I[2 * delta + (n - k):2 * delta + n, 0:delta].transpose()
  C = I[delta:2 * delta, delta:delta + (n - k)].transpose()
  D = I[2 * delta + (n - k):2 * delta + n, delta:delta + (n -
  ↪  k)].transpose()

  return A, B, C, D
```

In the first simulation, the chosen parameters would have required extremely large field sizes for the codes using the constructions in Section 4.2. Thus, we used the theoretical results for the recoverable erasures per sliding window with their respective optimal codes. Below, we provide the code we employed to recover erasures with an $(n, k, \delta)$ reverse MDP convolutional code using the FBD algorithm:

```
def correction_rmdp(correct, n, k, delta):
  """
  Function which corrects erasures of an (n, k, delta) reverse-MDP
  ↪  convolutional code using the FBD algorithm
  :param correct: Correct list (i.e. [v_0, ..., v_{length}]_{correct})
  :param n: Length of code
  :param k: Dimension of code
  :param delta: Degree of code
  :return: Updated correct list after applying the algorithm
  """
  v = delta // (n - k)
  length = len(correct) // n - 1
  i = 0

  while i <= length:
    forwardsuccess = 0
    backwardsuccess = 0

    # Forward decoding
```

```
    if [1] * n != [correct[l] for l in range(i * n, (i + 1) * n)]:
      L = delta // k + v
      if 0 not in [correct[l] for l in range((i - v) * n, i * n) if 0 <=
      ↪  l]:
        j = 0

        while forwardsuccess == 0 and j <= L:
          index_unknown = [l for l, l0 in enumerate(correct) if l0 == 0 and
          ↪  l in range((i - v) * n, (i + j + 1) * n)]

          if len(index_unknown) <= (j + 1) * (n - k):
            # We can forward decode
            for l in index_unknown:
              correct[l] = 1

            forwardsuccess = 1
            i = i + j

          else:
            # We cannot forward decode, so we try a larger window
            j += 1

      # Backward decoding
      if forwardsuccess != 1:
        aux = [l for l, l0 in enumerate(correct) if l0 == 0 and l in
        ↪  range(i * n, (i + L + 1) * n)]
        K = (aux[-1] - (aux[-1] % n)) // n + 1

        while backwardsuccess == 0 and K <= min(i + delta // k + v + 1,
        ↪  length + 1):

          if 0 not in [correct[l] for l in range(K * n, (K + v) * n) if l
          ↪  in range((length + 1) * n)]:
            if [1] * n != [correct[l] for l in range((K - 1) * n, K * n)]:
              j = 0
              while j <= L:
                index_unknown = [l for l, l0 in enumerate(correct) if l0 ==
                ↪  0 and l in range((K - j - 1) * n, (K + v) * n)]

                if len(index_unknown) <= (j + 1) * (n - k):
                  # We can backward decode
                  for l in index_unknown:
                    correct[l] = 1

                  backwardsuccess = 1
                  i = (aux[-1] - (aux[-1] % n)) // n + v

                  index_unknown = [l for l, l0 in enumerate(correct) if l0
                  ↪  == 0 and l in range((K - L - 1) * n, (K + v) * n)]

                  if len(index_unknown) != 0:
                    K_new = (index_unknown[-1] - (index_unknown[-1] % n))
                    ↪  // n + 1
```

```
                    L = L - (K - K_new)
                    K = K_new
                    j = 0

                  else:
                    j = L + 1

                else:
                  # We cannot backward decode, so we try a larger window
                  j += 1

          K += 1

    i += 1

  return correct
```

The LDDM algorithm was implemented as follows for an $(n, k, \delta)$ complete $T$-MDP convolutional code:

```
def correction_cjmdp(correct, n, k, delta, T):
  """
  Function which corrects erasures of an (n, k, delta) complete T-MDP
  ↪  convolutional code using the LDDM algorithm
  :param correct: Correct list (i.e. [v_0, ..., v_{length}]_{correct})
  :param n: Length of code
  :param k: Dimension of code
  :param delta: Degree of code (with (n - k) dividing degree)
  :param T: Maximal delay for decoding process
  :return: Updated correct list after applying the algorithm
  """
  v = delta // (n - k)
  length = len(correct) // n - 1

  c = -(v + 1)
  i = 0

  while i <= length:
    if [1] * n != [correct[l] for l in range(i * n, (i + 1) * n)]:
      j = 0
      success = 0

      while j <= T and success == 0:
        t = max(i - v, c)
        s = i + j - t - v

        index_unknown = [l for l, l0 in enumerate(correct) if l0 == 0 and l
        ↪  in range(t * n, (i + j + 1) * n)]

        if t == i - v:
          # We have a guard space

          if len(index_unknown) <= (j + 1) * (n - k):
            # We can recover the erasures in the window
```

```
            for l in index_unknown:
              correct[l] = 1

            success = 1
            i = i + j

          else:
            if j == T:
              # We cannot recover the erasures in v_i
              c = i + 1

            # We try a larger window size
            j += 1

        elif t != i - v:
          # We do not have a guard space
          x = correct_cjmdp(n, k, j, v, s, t, index_unknown)

          if len(index_unknown) <= (s + 1) * (n - k) and x != "Not
          ↪  correctable" and s >= 0:
            for l in index_unknown:
              correct[l] = 1

            success = 1
            i = i + j

          else:
            if j == T:
              # We cannot recover the erasures in v_i
              c = i + 1

              # We try a larger window size
            j += 1

    i += 1

  return correct
```

We used the following code to get the recovery capability of an $(n, k, \delta)$ convolutional code fulfilling Property 2 with the LDDLS algorithm. We did not implement the termination algorithm:

```
def correction_mdp(correct, n, k, delta, T):
  """
  Function which corrects erasures of an (n, k, delta) convolutional code
  ↪  satisfying Property 2 using the LDDLS algorithm
  :param correct: Correct list (i.e. v_{0}, ..., v_{length})_{correct},
  ↪  i.e. in the order we receive the v_i's)
  :param n: Length of code
  :param k: Dimension of code
  :param delta: Degree of code
  :param T: Maximal delay for decoding process
  :return: Updated correct list after applying the algorithm
  """
```

```
length = len(correct) // n - 1
correct_y, correct_u = correct_yu(correct, n, k, length)

i = 0

while i <= length:
  if [1] * (n - k) != [correct_y[l] for l in range(i * (n - k), (i + 1) *
  ↪  (n - k))] or [1] * k != [correct_u[l] for l in range(i * k, (i + 1)
  ↪  * k)]:
    j = 0
    success = 0

    # Main Decoding Algorithm
    while j <= T and success == 0:
      index_unknown_y = [l for l, l0 in enumerate(correct_y) if l0 == 0
      ↪  and l in range(i * (n - k), (i + j + 1) * (n - k))]
      index_unknown_u = [l for l, l0 in enumerate(correct_u) if l0 == 0
      ↪  and l in range(i * k, (i + j + 1) * k)]

      if len(index_unknown_y) + len(index_unknown_u) <= (j + 1) * (n -
      ↪  k):
        # We can recover the erasures
        for l in index_unknown_y:
          correct_y[l] = 1

        for l in index_unknown_u:
          correct_u[l] = 1

        success = 1
        i = i + j

      else:
        # We try a larger window size
        if j < T:
          j += 1

        else:
          # We cannot recover the erasures in v_i, so we try recovering
          ↪  some x_{i + t}
          t = 1
          j = 0

          while success == 0:
            if i + t <= length:
              index_unknown_y = [l for l, l0 in enumerate(correct_y) if
              ↪  l0 == 0 and l in range((i + t) * (n - k), (i + t + j +
              ↪  1) * (n - k))]
              index_unknown_u = [l for l, l0 in enumerate(correct_u) if
              ↪  l0 == 0 and l in range((i + t) * k, (i + t + j + 1) *
              ↪  k)]

              if len(index_unknown_y) + len(index_unknown_u) + delta <=
              ↪  (j + 1) * (n - k):
```

```
                    # We can recover x_{i + t} and the erasures in the window
                    for l in index_unknown_y:
                      correct_y[l] = 1

                    for l in index_unknown_u:
                      correct_u[l] = 1

                    success = 1
                    i = i + t - 1

                  else:
                    # We try a larger delay
                    if j < T:
                      j += 1

                    else:
                      # We cannot recover x_{i + t} and start now trying to
                      ↪  recover x_{i + t + 1}
                      j = 0
                      t += 1

                else:
                  # We cannot recover any x_{i + t} and stop the algorithm
                  success = 1
                  i = i + t

    i += 1

  return correct_yu_to_list(correct_y, correct_u, n, k, length)
```

In the second simulation, the choice of smaller degrees $\delta$ enabled us to construct complete MDP convolutional codes with the method using Theorem 4.2.17 that do not require very large field sizes. We implemented this construction method as follows:

```
def complete_MDP(n, k, delta, p):
  """
  Function to construct a parity-check matrix of a complete MDP
  ↪  convolutional code
  :param n: Length of the code
  :param k: Dimension of the code
  :param delta: Degree of the code (with (n - k) dividing degree)
  :param p: Characteristic of finite field of the polynomial coefficients
  :return: Parity-check matrix of complete MDP convolutional code; field of
  ↪  the coefficients; a primitive
  elements of the field
  """
  v = delta // (n - k)
  L = delta // k + v
  N = (L + 1) * 2 ** ((v + 2) * n - k - 1) + 1

  field = GF(p ** N, name='a')
  a = field.gen()
  F = field['z']
```

```
  z = F.gen()

  H = matrix(F, n - k, n, 0)
  for l in range(v + 1):
    H_l = [[a ** 2 ** (l * n + l0) for l0 in range(n)]]

    # Constructing the parity-check matrix using the previous row
    for l0 in range(1, n - k):
      row = H_l[-1]
      H_l.append([l1 ** 2 for l1 in row])

    H = H + matrix(F, H_l) * z ** l

  return H, field, a
```

Since the codes used in the first simulation only utilize the theoretical results, we adapted them such that the functions match the given pseudocodes in Section 5.1. Thus, we need a parity-check matrix of the code and a randomly chosen codeword for the transmission. We implemented the FBD algorithm as follows:

```
def FBD_alg(H, msg, correct, delta, field):
  """
  The FBD algorithm
  :param H: Parity-check matrix of code
  :param msg: Codeword to applying algorithm on (as vector with polynomial
  ↪  entries)
  :param correct: Correct list corresponding to msg
  :param delta: Degree of code (with (n - k) dividing degree)
  :param field: Field of the coefficients
  :return: Decoded message (as vector with polynomial entries); updated
  ↪  correct list after applying algorithm
  """
  n = H.ncols()
  k = n - H.nrows()
  v = H.degree()
  L = delta // k + delta // (n - k)
  msg_list = msg_to_list(msg)
  length = len(correct) // n - 1
  i = 0

  while i <= length:
    forwardsuccess = 0
    backwardsuccess = 0

    # Forward decoding
    if [1] * n != [correct[l] for l in range(i * n, (i + 1) * n)]:
      if 0 not in [correct[l] for l in range((i - v) * n, i * n) if 0 <=
      ↪  l]:
        j = L

        while forwardsuccess == 0 and j >= 0:
          index_known = [l for l, l0 in enumerate(correct) if l0 == 1 and l
          ↪  in range((i - v) * n, (i + j + 1) * n)]
```

```
        index_unknown = [l for l, l0 in enumerate(correct) if l0 == 0 and
        ↪  l in range((i - v) * n, (i + j + 1) * n)]

        Partial_pcm = partial_pcm(H, j, n, k, field)
        Ppcm_known = Partial_pcm.matrix_from_columns([l - (i - v) * n for
        ↪  l in index_known])
        Ppcm_unknown = Partial_pcm.matrix_from_columns([l - (i - v) * n
        ↪  for l in index_unknown])
        msg_known = vector([msg_list[l] for l in index_known])
        b = -(Ppcm_known * msg_known)
        b = matrix(b).transpose()

        if solvable(Ppcm_unknown, b, [index_unknown], j, i, n, k, delta,
        ↪  0) == "Unique":
          # We can forward decode
          solution = Ppcm_unknown.solve_right(b)
          counter = 0

          for l in index_unknown:
            msg_list[l] = solution[counter][0]
            correct[l] = 1
            counter += 1

          forwardsuccess = 1
          i = i + j

        else:
          # We cannot forward decode, so we try a smaller window
          j -= 1

    # Backward decoding
    if forwardsuccess != 1:
      aux = [l for l, l0 in enumerate(correct) if l0 == 0 and l in
      ↪  range(i * n, (i + L + 1) * n)]
      K = (aux[-1] - (aux[-1] % n)) // n + 1

      while backwardsuccess == 0 and K <= min(i + L + 1, length + 1):
        if 0 not in [correct[l] for l in range(K * n, (K + v) * n) if l
        ↪  in range((length + 1) * n)]:
          j = L
          while backwardsuccess == 0 and j >= 0:
            index_known = [l for l, l0 in enumerate(correct) if l0 == 1
            ↪  and l in range((K - j - 1) * n, (K + v) * n)]
            index_unknown = [l for l, l0 in enumerate(correct) if l0 == 0
            ↪  and l in range((K - j - 1) * n, (K + v) * n)]

            # Since (n - k) divides delta, we use Remark 5.1.4
            Partial_pcm = partial_pcm(H, j, n, k, field)
            Ppcm_known = Partial_pcm.matrix_from_columns([l - (K - j - 1)
            ↪  * n for l in index_known])
            Ppcm_unknown = Partial_pcm.matrix_from_columns([l - (K - j -
            ↪  1) * n for l in index_unknown])
            msg_known = vector([msg_list[l] for l in index_known])
```

```
            b = -(Ppcm_known * msg_known)
            b = matrix(b).transpose()

            if solvable(Ppcm_unknown, b, [index_unknown], j, i, n, k,
            ↪  delta, 0) == "Unique":
              # We can backward decode
              solution = Ppcm_unknown.solve_right(b)
              counter = 0

              for l in index_unknown:
                msg_list[l] = solution[counter][0]
                correct[l] = 1
                counter += 1

              backwardsuccess = 1
              i = K + v - 1

            else:
              # We cannot backward decode, so we try a smaller window
              j -= 1

        K += 1

    i += 1

  return list_to_msg(msg_list, n, field), correct
```

Similarly, we implemented the LDDM algorithm:

```
def LDDM_alg(H, msg, correct, T, field):
  """
  The LDDM algorithm
  :param H: Parity-check matrix of code
  :param msg: Codeword to applying algorithm on (as vector with polynomial
  ↪  entries)
  :param correct: Correct list corresponding to msg
  :param T: Maximal delay for decoding process
  :param field: Field of the coefficients
  :return: Decoded message (as vector with polynomial entries); updated
  ↪  correct list after applying algorithm
  """
  n = H.ncols()
  k = n - H.nrows()
  v = H.degree()
  msg_list = msg_to_list(msg)
  length = len(correct) // n - 1

  c = -(v + 1)
  i = 0

  while i <= length:
    if [1] * n != [correct[l] for l in range(i * n, (i + 1) * n)]:
      j = 0
      success = 0
```

```
while j <= T and success == 0:
  t = max(i - v, c)
  s = i + j - t - v

  index_known = [l for l, l0 in enumerate(correct) if l0 == 1 and l
  ↪  in range(t * n, (i + j + 1) * n)]
  index_unknown = [l for l, l0 in enumerate(correct) if l0 == 0 and l
  ↪  in range(t * n, (i + j + 1) * n)]

  Partial_pcm = partial_pcm(H, s, n, k, field)
  Ppcm_known = Partial_pcm.matrix_from_columns([l - t * n for l in
  ↪  index_known])
  Ppcm_unknown = Partial_pcm.matrix_from_columns([l - t * n for l in
  ↪  index_unknown])
  msg_known = vector([msg_list[l] for l in index_known])
  b = -(Ppcm_known * msg_known)
  b = matrix(b).transpose()

  if solvable(Ppcm_unknown, b, [index_unknown], s, i, n, k, 0, "v")
  ↪  == "Unique":
    # We can recover all the erasures in the window
    solution = Ppcm_unknown.solve_right(b)
    counter = 0

    for l in index_unknown:
      msg_list[l] = solution[counter][0]
      correct[l] = 1
      counter += 1

    success = 1
    i = i + j

  elif solvable(Ppcm_unknown, b, [index_unknown], s, i, n, k, 0, "v")
  ↪  == "Not unique":
    # We cannot recover all the erasures in the window, but we can
    ↪  recover at least the ones in v_i (and if possible more)
    solution = Ppcm_unknown.solve_right(b)
    base = Ppcm_unknown.right_kernel_matrix()
    unknown_v_i = len([l for l in index_unknown if l in range(i * n,
    ↪  (i + 1) * n)])
    recoverable = list(range(unknown_v_i))

    for l in range(unknown_v_i, base.ncols()):
      if base.column(l).is_zero():
        recoverable.append(l)

    counter = 0

    for l in recoverable:
      msg_list[index_unknown[l]] = solution[counter][0]
      correct[index_unknown[l]] = 1
      counter += 1
```

```
            success = 1

          else:
            # We try a larger window size
            if j == T:
              c = i + 1

            j += 1
      i += 1

  return list_to_msg(msg_list, n, field), correct
```

Lastly, the LDDLS algorithm, including the termination algorithm, was implemented as follows:

```
def LDDLS_alg(A, B, C, D, msg, correct, T, delta, field):
  """
  The LDDLS algorithm
  :param A: A of a minimal ISO representation
  :param B: B of a minimal ISO representation
  :param C: C of a minimal ISO representation
  :param D: D of a minimal ISO representation
  :param msg: Codeword to applying algorithm on (as vector with polynomial
  ↪  entries (coefficients order is reversed
  here))
  :param correct: Correct list corresponding to msg
  :param T: Maximal delay for decoding process
  :param delta: Degree of code
  :param field: Field of the coefficients
  :return: Decoded message (as vector with polynomial entries); updated
  ↪  correct list after applying algorithm
  """
  n = D.nrows() + D.ncols()
  k = D.ncols()
  length = len(correct) // n - 1
  y, u = msg_to_yu(msg, n, k, length)
  correct_y, correct_u = correct_yu(correct, n, k, length)
  E_list = [e_w(A, B, C, w, length, k) for w in range(1, delta)]
  X = [vector(field, [0] * delta), 0]
  i = 0

  while i <= length:
    if [1] * (n - k) != [correct_y[l] for l in range(i * (n - k), (i + 1) *
    ↪  (n - k))] or [1] * k != [correct_u[l] for l in range(i * k, (i + 1)
    ↪  * k)]:
      j = 0
      success = 0

      # For termination check
      index_unknown_u = [l for l, l0 in enumerate(correct_u) if l0 == 0]
      w = termination_check(E_list, index_unknown_u)

      if w >= 0:
```

```
    # We can use the termination algorithm
    E = E_list[w]
    index_known_u = [l for l, l0 in enumerate(correct_u) if l0 == 1]
    index_known_y = [l for l, l0 in enumerate(correct_y) if l0 == 1]
    index_unknown_y = [l for l, l0 in enumerate(correct_y) if l0 == 0]

    E_known = E.matrix_from_columns(index_known_u)
    E_unknown = E.matrix_from_columns(index_unknown_u)
    b = -(E_known * vector([u[l] for l in index_known_u]))
    b = matrix(b).transpose()

    solution_u = E_unknown.solve_right(b)
    counter = 0

    for l in index_unknown_u:
      u[l] = solution_u[counter][0]
      correct_u[l] = 1
      counter += 1

    Systematic_mat = systematic_mat(A, B, C, D, length, n, k, field)
    Systematic_mat_known =
    ↪  Systematic_mat.matrix_from_columns(index_known_y +
    ↪  list(range((length + 1) * (n - k), (length + 1) * n)))
    Systematic_mat_unknown =
    ↪  Systematic_mat.matrix_from_columns(index_unknown_y)
    msg_known = vector([y[l] for l in index_known_y] + u)
    b = - (Systematic_mat_known * msg_known)
    b = matrix(b).transpose()

    solution_y = Systematic_mat_unknown.solve_right(b)
    counter = 0

    for l in index_unknown_y:
      y[l] = solution_y[counter][0]
      correct_y[l] = 1
      counter += 1

    return yu_to_msg(y, u, n, k, length, field),
    ↪  correct_yu_to_list(correct_y, correct_u, n, k, length)

  # Main decoding algorithm
  while j <= T and success == 0:
    index_known_y = [l for l, l0 in enumerate(correct_y) if l0 == 1 and
    ↪  l in range(i * (n - k), (i + j + 1) * (n - k))]
    index_unknown_y = [l for l, l0 in enumerate(correct_y) if l0 == 0
    ↪  and l in range(i * (n - k), (i + j + 1) * (n - k))]
    index_known_u = [l for l, l0 in enumerate(correct_u) if l0 == 1 and
    ↪  l in range(i * k, (i + j + 1) * k)]
    index_unknown_u = [l for l, l0 in enumerate(correct_u) if l0 == 0
    ↪  and l in range(i * k, (i + j + 1) * k)]

    Systematic_mat = systematic_mat(A, B, C, D, j, n, k, field)
```

```
Systematic_mat_known = Systematic_mat.matrix_from_columns([l - i *
↪  (n - k) for l in index_known_y] + [l - i * k + (j + 1) * (n -
↪  k) for l in index_known_u])
Systematic_mat_unknown = Systematic_mat.matrix_from_columns([l - i
↪  * (n - k) for l in index_unknown_y] + [l - i * k + (j + 1) * (n
↪  - k) for l in index_unknown_u])
X = update_x_i(X[0], A, B, u, k, X[1], i)
msg_known = vector([y[l] for l in index_known_y] + [u[l] for l in
↪  index_known_u] + list(X[0]))
b = -(Systematic_mat_known.augment(obs_mat(A, C, j)) * msg_known)
b = matrix(b).transpose()

if solvable(Systematic_mat_unknown, b, [index_unknown_y,
↪  index_unknown_u], j, i, n, k, delta, "yu") == "Unique":
  # We can recover all the erasures in the window
  solution = Systematic_mat_unknown.solve_right(b)
  counter = 0

  for l in index_unknown_y:
    y[l] = solution[counter][0]
    correct_y[l] = 1
    counter += 1

  for l in index_unknown_u:
    u[l] = solution[counter][0]
    correct_u[l] = 1
    counter += 1

  success = 1
  i = i + j


elif solvable(Systematic_mat_unknown, b, [index_unknown_y,
↪  index_unknown_u], j, i, n, k, delta, "yu") == "Not unique":
  # We cannot recover all the erasures in the window, but we can
  ↪  decode at least the ones in v_i (and if possible more)
  solution = Systematic_mat_unknown.solve_right(b)
  base = Systematic_mat_unknown.right_kernel_matrix()
  unknown_y_i = len([l for l in index_unknown_y if l in range(i *
  ↪  (n - k), (i + 1) * (n - k))])
  unknown_u_i = len([l for l in index_unknown_u if l in range(i *
  ↪  k, (i + 1) * k)])
  recoverable = list(range(unknown_y_i)) +
  ↪  list(range(len(index_unknown_y), len(index_unknown_y) +
  ↪  unknown_u_i))

  for l in range(0, base.ncols()):
    if l not in recoverable:
      if base.column(l).is_zero():
        recoverable.append(l)

  for l in recoverable:
    if l in range(len(index_unknown_y)):
```

```
            y[index_unknown_y[l]] = solution[l][0]
            correct_y[index_unknown_y[l]] = 1

          else:
            u[index_unknown_u[l - len(index_unknown_y)]] = solution[l][0]
            correct_u[index_unknown_u[l - len(index_unknown_y)]] = 1

        success = 1

      else:
        # We try a larger window size
        if j < T:
          j += 1

        else:
          # We cannot recover the erasures in v_i, so we try recovering
          ↪  some x_{i + t}
          t = 1
          j = 0

          while success == 0:
            if i + t <= length:
              index_known_y = [l for l, l0 in enumerate(correct_y) if l0
              ↪  == 1 and l in range((i + t) * (n - k), (i + t + j + 1)
              ↪  * (n - k))]
              index_unknown_y = [l for l, l0 in enumerate(correct_y) if
              ↪  l0 == 0 and l in range((i + t) * (n - k), (i + t + j +
              ↪  1) * (n - k))]
              index_known_u = [l for l, l0 in enumerate(correct_u) if l0
              ↪  == 1 and l in range((i + t) * k, (i + t + j + 1) * k)]
              index_unknown_u = [l for l, l0 in enumerate(correct_u) if
              ↪  l0 == 0 and l in range((i + t) * k, (i + t + j + 1) *
              ↪  k)]

              Systematic_mat = systematic_mat(A, B, C, D, j, n, k, field)
              Systematic_mat_known = Systematic_mat.matrix_from_columns(
              ↪  [l - (i + t) * (n - k) for l in index_known_y] + [l -
              ↪  (i + t) * k + (j + 1) * (n - k) for l in
              ↪  index_known_u])
              Systematic_mat_unknown =
              ↪  Systematic_mat.matrix_from_columns([l - (i + t) * (n -
              ↪  k) for l in index_unknown_y] + [l - (i + t) * k + (j +
              ↪  1) * (n - k) for l in index_unknown_u])
              Systematic_mat_unknown =
              ↪  Systematic_mat_unknown.augment(obs_mat(A, C, j))
              msg_known = vector([y[l] for l in index_known_y] + [u[l]
              ↪  for l in index_known_u])
              b = -(Systematic_mat_known * msg_known)
              b = matrix(b).transpose()
```

```
if solvable(Systematic_mat_unknown, b, [index_unknown_y,
↪  index_unknown_u, list(range(len(index_unknown_y) +
↪  len(index_unknown_u), len(index_unknown_y) +
↪  len(index_unknown_u) + delta))], j, i, n, k, delta,
↪  "x") == "Unique":
  # We can recover x_{i + t} and the erasures in the window
  solution = Systematic_mat_unknown.solve_right(b)
  counter = 0

  for l in index_unknown_y:
    y[l] = solution[counter][0]
    correct_y[l] = 1
    counter += 1

  for l in index_unknown_u:
    u[l] = solution[counter][0]
    correct_u[l] = 1
    counter += 1

  X =
  ↪  [vector(solution.matrix_from_rows(list(range(solution.nrows()
  ↪  - delta, solution.nrows())))), i + t]
  success = 1
  i = i + t - 1

elif solvable(Systematic_mat_unknown, b, [index_unknown_y,
↪  index_unknown_u, list(range(len(index_unknown_y) +
↪  len(index_unknown_u), len(index_unknown_y) +
↪  len(index_unknown_u) + delta))], j, i, n, k, delta,
↪  "x") == "Not unique":
  # We can recover x_{i + t} (and if possible some parts of
  ↪  v_{i + t}, ..., v_{i + t + j})
  solution = Systematic_mat_unknown.solve_right(b)
  base = Systematic_mat_unknown.right_kernel_matrix()
  recoverable = list(range(base.ncols() - delta,
  ↪  base.ncols()))

  for l in range(base.ncols() - delta):
    if base.column(l).is_zero():
      recoverable.append(l)

  for l in recoverable:
    if l < len(index_unknown_y):
      y[index_unknown_y[l]] = solution[l][0]
      correct_y[index_unknown_y[l]] = 1

    elif l < len(index_unknown_y) + len(index_unknown_u):
      u[index_unknown_u[l - len(index_unknown_y)]] =
      ↪  solution[l][0]
      correct_u[index_unknown_u[l - len(index_unknown_y)]]
      ↪  = 1
```

```
                X =
                ↪  [vector(solution.matrix_from_rows(list(range(solution.nrows()
                ↪  - delta, solution.nrows())))), i + t]

                success = 1
                i = i + t - 1

              else:
                # We try a larger delay
                if j < T:
                  j += 1

                else:
                  # We cannot recover x_{i + t} and start now trying to
                  ↪  recover x_{i + t + 1}
                  j = 0
                  t += 1

            else:
              # We cannot recover any x_{i + t} and stop the algorithm
              success = 1
              i = i + t
  i += 1

return yu_to_msg(y, u, n, k, length, field),
↪  correct_yu_to_list(correct_y, correct_u, n, k, length)
```