Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Sindre Byre

Short Polar Codes for Bluetooth

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Kimmo Kansanen
Co-supervisor: Daniel Ryan
July 2023

NTNU
Norwegian University of
Science and Technology

Sindre Byre

Short Polar Codes for Bluetooth

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Kimmo Kansanen
Co-supervisor: Daniel Ryan
July 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Norwegian University of
Science and Technology

Abstract

Polar codes have gained significant attention in academia and industry as the first provably capacity-achieving forward error correcting code family with provable encoding and decoding complexity. Their significance is underscored by their inclusion in the new generation wireless communication standards, 5G. While initially designed for large block lengths, the 5G standards have shed light on the possibility to use Polar codes also with short block lengths. This study aims to investigate the suitability of Polar codes for enhancing data transmission efficiency in Bluetooth by comparing their performance with convolutional codes, as used in Bluetooth, under the same conditions.

This research addresses several sub-goals, including understanding the error-correction functionality of polar codes and convolutional codes, evaluating the performance of the Successive Cancellation List (SCL) decoder and the List Viterbi Algorithm (LVA) decoder in terms of Frame-Error-Rate (FER), assessing the role of the Cyclic Redundancy Check (CRC) as a supportive element in list decoding, and conducting a comparative analysis between the CRC-aided SCL and CRC-aided LVA decoders.

Simulations were run in MATLAB with various conditions, including different list sizes L and frame lengths K, using both a 24-bit and 6-bit CRC. The Binary Phase-Shift Keying (BPSK) modulation scheme over an Additive White Gaussian Noise (AWGN) channel was chosen and different signal-to-noise ratio (SNR) values were used to plot the FER. The findings of this research indicates that polar code have better error-correction then convolution code for block lengths including and larger than $K = 64$ bits where 24 of them are dedicated to CRC-bits. Furthermore, increasing the list size improves the performance of both polar codes and convolutional codes. The use of 6-bit CRC instead of 24-bit CRC introduced false positives, making the overall code performance slightly worse, but increasing the throughput by reducing the redundant bits.

The results highlight the advantages of Polar codes over convolutional codes, particularly for larger frame sizes, and provide insights of CRC usage. While this study shows Polar codes have potential in improving data transmission efficiency in Bluetooth, it is important to consider additional factors such as complexity, varying code rates, and techniques like puncturing and shortening in future research. Overall, this investigation offers valuable insights for optimizing communication systems and showcases the potential of Polar codes in enhancing performance within short-range communication applications like Bluetooth.

Sammendrag

Denne studien undersøker anvendelsen av Polar-koder i sammenheng med Bluetooth, ved å se på bruken av Polar-koder med korte blokklengder, og sammenlikne den med konvolutionell kode under samme betingelser. Hvor bra de forskjellige dekoderene, CRC-aided List Viterbi Algorithem og CRC-aided Successive Cancellation List (SCL), presterer i å korriger overføringsfeil blir studert.

Simuleringer ble kjørt i MATLAB under varierende forhold, og resultatene indikerte at Polar-koder har bedre feilkorrigering enn konvolusjonskoder for bestemte blokklengder, og at forbedringer kan oppnås ved å øke liste-størrelsen. Studiet understreker potensialet for Polar-koder i å forbedre ytelsen innen Bluetooth, samtidig som behovet for videre forskning på flere faktorer som kompleksitet og varierende kodningsrater påpekes.

Preface

This Master's thesis, completed in summer 2023, is the result of an intensive academic journey at the Norwegian University of Science and Technology (NTNU) in Trondheim, Norway. This project has been conducted under the supervision of Professor Dr. Kimmo Kansanen from the Department of Electronic Systems and Dr. Daniel Ryan, Senior R&D Engineer at Nordic Semiconductor ASA.

This project started in January 2023, but is a continuation of the semester project carried out autumn 2022[1].

The structure of this thesis includes four main sections. The opening section introduces the broader topic of this project, aiming to give the reader a necessary context. The second section dives deeper into the topic, lying a solid groundwork of the theoretical background. Section three present the results and key findings while the final part discusses them and brings a closure to the thesis.

During this study, I received significant guidance and support from my supervisors. First and foremost I would like to thank Professor Kimmo Kansanen. He was always available, guiding me through an area of study I had not explored before, through weekly meetings and quick responses when I needed guidance.

My co-supervisor Daniel Ryan also deserves my appreciation, for attending our weekly meetings, giving insightful feedback through this journey.

As you navigate through this thesis, I hope it gives valuable insights on the topic of this study and stimulates further curiosity in this field.

Table of Contents

Abstract i

Sammendrag ii

Preface iii

Table of Contents v

List of Figures vi

1 Introduction 1

1.1 Digital Communication: Context, Application, and Challenges 1

1.2 Problem description 2

1.3 Problem definition 3

1.4 Related work 4

2 Deep Dive into Decoding and Error Correction Techniques 5

2.1 Fundamental Principles of Error Correction 5

2.1.1 Coding theory background 5

2.1.2 Classification of Error Correcting Codes 7

2.1.3 Error Types and the Role of Cyclic Redundancy Check 8

2.1.4 AWGN channel and soft decoding 10

2.2 Convolutional code 11

2.2.1 Convolutional encoding 11

2.2.2 Trellis diagram 12

2.2.3 Soft-decision Viterbi decoder 16

2.2.4 CRC-aided List Viterbi Algorithm (LVA) 18

2.3 Polar codes 21

2.3.1 Polar transform 22

2.3.2 Channel polarization 24
2.3.3 Encoding 26
2.3.4 Successive cancellation decoder 27

3 Results **29**

3.1 Experimental Setup 29
3.2 Efficacy of CRC24-aided LVA and SCL under varying block lengths and list sizes 30
3.2.1 Polar code 30
3.2.2 Convolutional code 33
3.2.3 Comparative evaluation of LVA and SCL 36
3.3 Impact of CRC usage 39

4 Discussion **40**

References **43**

Appendix **44**

A CRC-aided LVA implemented in MATLAB 44
B Convolutional code simulations 48
C CRC-aided SCL, modifications from the MATLAB function nrPolar-Decode 53
D Polar code simulations 54

List of Figures

1 Block diagram showing the context of coding, here shown in the blocks encoding and decoding. 7

2 Data frame of length K. 9

3 1/2 rate convolutional encoder, from 3.3.1 page 2736 in [10]. 12

4 Fully connected trellis with S states[]. 13

5 Section of a 4 state trellis. The label of each branch represent the corresponding input and expected output separated by '/'. 13

6 Section of the trellis diagram for the encoder in Figure 3. 14

7 Beginning of the trellis for the encoder in Figure 3. 15

8 Beginning of the trellis for the encoder in Figure 3. 16

9 At state i, the L best paths gets chosen based on the path metrics and branch metrics of the $S \cdot L$ incoming paths. 19

10 Block diagram of the process going on at state i in Figure 9. 19

11 Example of values for all metrics at time instance t for node i, for a list size of $L = 4$. The colors showcases different paths being chosen at state i, how they are being ranked, and the history that is being stored are shown in the box. 20

12 Two ways of visualizing how the 2 bit polar transform is done. 22

13 Binary tree representation of the 4 bit to 4 bit Polar transform. . . . 23

14 Expansion of the polar transform from $n = 1$ in the yellow, $n = 2$ in the blue, and $n = 3$ in the red box. 24

15 Block diagram of bit-flow from input vector $\mathbf{u}$ to received vector $\mathbf{r}$. . . 25

16 Visualization of information-flow through bit channels. 25

17 Example of a $\mathcal{P}(8, 4)$ polar code encoder. 27

18 Fundamental building block for how the message-passing is happening in the nodes on the tree. 28

19 Polar code with $K = 64$ simulated using Appendix D. 31

20 Polar code with constant frame lengths and varying list sizes L increasing from right to left. Code rate is 1/2 and the CRC24-aided SCL is used in the simulations. 32

21 Polar code with constant list size with varying frame lengths K. . . . 33

22 Convolutional code with constant frame length $K = 32$ and varying list sizes L increasing from right to left, simulated using Appendix B. 34

23 Convolutional code with $K = 64$ simulated using Appendix B. 35

24 Convolutional code with constant list size and varying frame lengths K. 36

25 Comparative performance of polar codes and convolutional codes for frame length $K = 32$ with list sizes $L = 1$ and $L = 16$. 37

26 Comparative performance of polar codes and convolutional codes for frame length $K = 64$ with list sizes $L = 1$ and $L = 16$. 38

27 Comparative performance of polar codes and convolutional codes containing 40 information bits with list sizes $L = 1$ and $L = 16$. 39

28 Study of impact of using CRC6 for $K = 32$. The dark blue plots correspond to the plot in Figure 22 using CRC24-aided LVA. The light blue uses CRC6-aided LVA. 40

29 Declaring the new vector of L codewords instead of only the best one. 53

30 Changing the CRC to the one used in Bluetooth Specifications. . . . 53

31 Making the new vector of L codewords with the best one at position 1. 53

1 Introduction

1.1 Digital Communication: Context, Application, and Challenges

Digital communication lies at the heart of our modern world. From individual handheld devices to large networks of information systems, the importance of reliable and efficient data transmission plays an important role. In this growing technological landscape, technologies like Bluetooth and the Internet of Things (IoT) stand out for their low power, short-range wireless data transfer. However, the effective use of such technologies is not without challenges. One of the primary obstacles in digital communication is the occurrence of errors during data transmission, which we describe as noisy channels. This issue underlines the importance of robust error-correcting codes (ECCs) capable of ensuring the integrity and reliability of data being transmitted.

To ensure accurate transmission of information over these networks, ECCs, such as convolutional codes, are commonly employed. These codes add redundancy to the transmitted data, enabling the receiver to detect and correct errors that might occur during transmission. However, with the rising demand for high-speed and efficient data transmission, particularly in Bluetooth and IoT contexts, the need for more efficient coding techniques has become in focus.

Despite the many advancements in the field, the introduction of Polar codes has sparked new interest in error correcting codes. Polar codes are recognized for their capability to reach Shannon's capacity limits[2], but this is typically applicable to very long block lengths. However, in practical scenarios, especially in systems like Bluetooth and IoT, short block lengths are more common. This gives rise to an intriguing question: How well do Polar codes perform when the block lengths are shorter? This question forms the focus of this study. Considering that Polar codes have been included in the 5G New Radio (NR) standards, this question gains even more significance, indicating the potential future impact of these codes on digital communication.

In summary, this study sits at the intersection of technological development and

the growing demand for reliable, efficient digital communication. The findings from this research not only contribute to the understanding of Polar codes but also have potential to impact the practical deployment of Bluetooth technologies.

1.2 Problem description

With an increase in the need for faster and more efficient communication systems, there is a constant pursuit of improving error correction codes (ECCs) to ensure reliable transmission. Among the various ECCs, Polar codes and Convolutional codes are the two types this thesis focuses on. Both are utilized in a range of applications and have shown promising capabilities in different scenarios.

Convolutional codes are well-established in the field, with their error correction capabilities proven in various applications. A common decoding method used with these codes is the Viterbi Algorithm [3]. However, an improved method called the List Viterbi Algorithm (LVA)[4] offers potentially better performance which is a topic that will be explored in this study.

On the other hand, Polar codes have become a topic of renewed interest in recent years, particularly because of their inclusion in the 5G NR standards[5]. Even though they are known to reach Shannon's capacity limits[2], it is generally at longer block lengths, which might not always be practical. So, how these codes perform at shorter block lengths becomes an essential question to investigate.

In addition, a method called Successive Cancellation List (SCL)[6] decoding, an improvement over the original Successive Cancellation (SC)[7] decoding, will be examined in the context of Polar codes. Furthermore, the role of Cyclic Redundancy Check (CRC)[8] in error detection and its effects on the overall performance of the system will also be evaluated.

The main aim is to provide a comparative analysis of the performance of the two code types at short block lengths, examining their decoding methods, including LVA and SCL, across different list sizes and block lengths, and determining whether Polar codes can be an effective improvement for Bluetooth.

1.3 Problem definition

The primary objective of this research is to investigate the performance of polar codes when used at short block lengths. The aim is to evaluate the potential of these codes to enhance data transmission efficiency in Bluetooth environments. To ensure a precise and quantifiable research goal, several sub-goals have been outlined.

The first sub-goal is to understand the error-correction functionality of polar codes and convolutional codes. Secondly, the operational performance of two decoders: the CRC-aided Successive Cancellation List (SCL) decoder and the CRC-aided List Viterbi Algorithm (LVA), will be thoroughly evaluated. These decoders will be examined under different conditions, with list sizes and block lengths of varying measures. The goal here is to study how the performance of these decoders can shift under different conditions, providing us with a clearer understanding of how they function.

The third sub-goal is centred around the use of the Cyclic Redundancy Check (CRC). Specifically, we aim to evaluate the role of CRC as an supportive element in list decoding. We will compare and assess the impact of two distinct CRCs - one comprising 24 bits, known as CRC24, and another containing 6 bits, CRC6. This comparison will help illuminate the differential effects that various CRC versions can have on decoding performance, thus providing a more comprehensive understanding of their role within the decoding process.

The fourth sub-goal involves a comparative analysis between the CRC-aided SCL and CRC-aided LVA decoders, under the same conditions. The aim is to identify which decoder proves to be more efficient at short block lengths. Here, too, we will use FER at different SNR levels as our quantifiable metric.

By undertaking these investigations, we are looking to directly evaluate how these could enhance the data transmission efficiency within Bluetooth and IoT systems. Our end goal is to minimize the error rate and thereby, improve the reliability of these communication systems. Each sub-goal contributes to the wider objective of determining if the implementation of polar codes, specifically at shorter block lengths, can significantly boost performance within Bluetooth and IoT systems. By being precise and measurable, these objectives provide a clear roadmap for this research.

1.4 Related work

Channel codes research has seen several milestone works that have greatly contributed to our understanding of error-correction codes and their efficiencies. The following research papers have significantly influenced this thesis.

In 2009, Arıkan's study [7] introduced polar codes for the first time, including a simple Successive Cancellation (SC) decoder. This work has been a major source of our understanding of polar codes. Arıkan demonstrated that channel polarization could be utilized to achieve the capacity of binary-input memoryless channels, with provable complexity, laying a solid theoretical groundwork for the development of polar codes.

I. Tal and A. Vardy (2015), in their paper [6] provided insights into the use of Successive Cancellation List (SCL) decoder for Polar codes. This method is shown to be an excellent way of decoding polar codes, and shown to outperform many of the competing ECC's. This is also the decoder standariced by 3GPP for the air interface standard in 5G[5]. The relevance of their work, for this thesis, lies in the decoder algorithm, which has been by implemented by MathWorks in their 5G Toolbox [9].

On another note, convolutional codes have also been widely studied. Viterbi's 1967 work [3] has been instrumental in this field. Viterbi introduced an efficient decoding algorithm, which is now known as the Viterbi Algorithm, for convolutional codes. His paper remains a cornerstone reference in understanding the working of convolutional codes and their decoding processes, making it particularly useful given its standard usage in Bluetooth technology [10].

Further, Seshadri and Sundberg (1994) in their paper "List Viterbi decoding algorithms with applications," [4] introduced the use of CRC for aiding their List Viterbi Algorithm (LVA). Their work contributes to this thesis by providing an in-depth understanding of how the CRC aided LVA functions, which directly parallels the CRC aided SCL.

2 Deep Dive into Decoding and Error Correction Techniques

2.1 Fundamental Principles of Error Correction

2.1.1 Coding theory background

Coding theory, a branch of information theory, plays an important role in digital communication systems. It was developed to address error correction in telecommunication systems, and has since expanded to play an important role in information transmission, storage, and data compression. Fundamentally, coding theory, and more specifically error control, involves the strategic insertion of controlled redundancy to the data, in the form of additional bits, to ensure error detection and correction during transmission. Errors, typically alterations in binary values due to noise, interference, or channel imperfections, can then be addressed effectively.

In coding theory, noise is unwanted random disturbance of useful information. A typical measure of noise is signal-to-noise ratio (SNR), that compare the level of wanted signal, to the level of noise. The SNR is calculated by dividing the signal power by the noise power and often expressed in dB, so if the if the wanted signal and level of noise is the same, we have a ratio of 1, or 0dB. If we have more signal than noise, we have a positive dB value and vice versa.

A central strategy, and one of the bigger fields in coding theory, is Forward Error Correction (FEC), a type of error control that enables the system to correct errors without the need for retransmission. Coding theory aims to maximize the data rate, referring to the volume of data transmitted over a given time, while preserving data integrity.

The synergy of information theory and coding theory lies in optimizing efficiency. Information theory establishes the channel capacity, or the maximum data rate allowing reliable communication. Coding theory, on the other hand, provides practical tools such as various types of codes to approach this limit. An good code enables communication at a rate close to channel capacity with minimal error probability.

Balancing error probability and data rate is key. As the transmission data rate

nears the channel capacity, error probability rises. However, with suitable error-correcting codes, this probability can be reduced, resulting in efficient and reliable data transmission.

Now, let us consider an ergodic channel W, such as a binary erasure channel BEC(p), with a probability p of erasing each bit sent. The message bits have a length K, and the encoded message has a length N. The ratio K/N is the code rate. If we now were to send a single bit, 1, as a message, the code rate would be 1 and the probability for an error in the received message would be $P_N^{(e)} = p$. Using code, we could reduce the probability $P_N^{(e)}$ of an error in the received message by making erasure less likely.

Repetition code, one of the simplest forms of code, where message bits are repeated, can serve as an example. If we employ repetition code $(3, 1)$, the sent message will be 111, reducing the code rate to 1/3. However, since the probability of each individual sent bit to get erased is p, the probability of the message now being lost will be $P_N^{(e)} = p^3 < p$. This increases the reliability of the transmission, but the downside is that repetition of bits increases the power cost due to additional redundant bits transmitted. In this example we used a BEC. Further on we will only consider the Additive White Gaussian Noise (AWGN) channel, discussed in a later subsection.

The key question then becomes: for a message of size K, what is the minimum number of bits N, forming a codeword, needed to ensure a high probability of recovering the original message? Claude E. Shannon answered this question in 1948 by establishing the Shannon's channel capacity—the maximum rate for transmitting error-free information for given bandwidth, Gaussian noise, and signal power[2]. This capacity limit is what Arıkan proved that polar codes could achieve as block length increases[7].

In summary, coding theory offers mechanisms to maintain data integrity while striving for efficiency and reliability, laying the groundwork for comparing different coding techniques in this study: Convolutional and Polar codes.

2.1.2 Classification of Error Correcting Codes

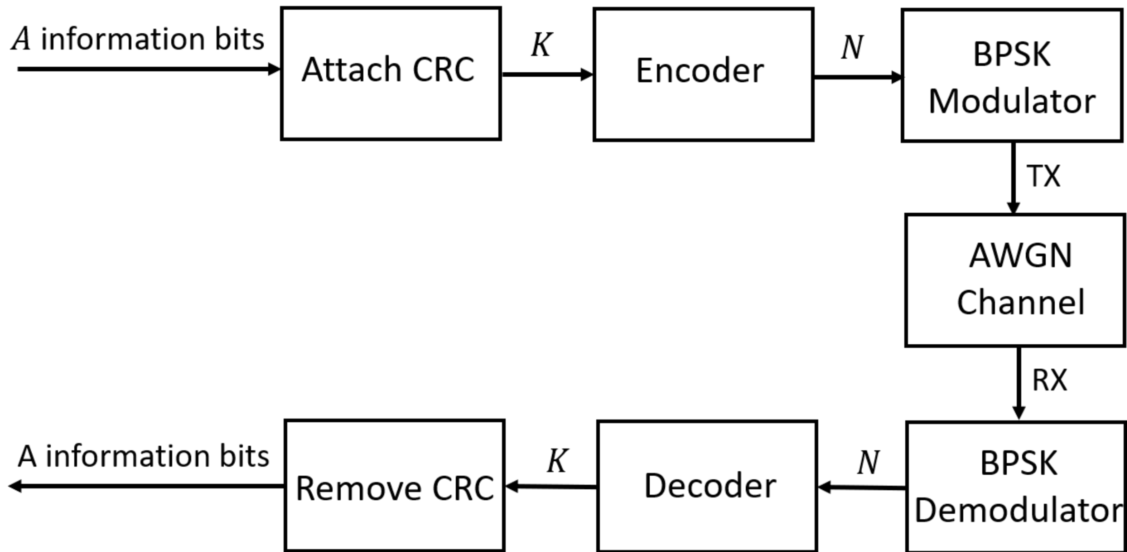


Figure 1: Block diagram showing the context of coding, here shown in the blocks encoding and decoding.

Error correcting codes (ECCs) are a cornerstone of data communication, ensuring that the information, in Figure 1 shown as A, sent from one point to another arrives accurately, even in the face of transmission errors. ECCs are widely diverse and can be classified based on several criteria such as the way they are constructed, their performance, and the complexity of their decoding algorithms.

One basic distinction is between block codes and convolutional codes. Block codes sends data in blocks of bits, treat each block independently and add redundancy to them in the form of extra bits. Some common examples of block codes are Hamming codes and Reed-Solomon codes[11]. Polar codes, which are the focus of this thesis, are a type of block codes due to their block-like structure.

Convolutional codes, in contrast, operate on serial data, treating input data as a sequence of bits that extends over time. They add redundancy to the data stream based on a process similar to mathematical convolution. Viterbi's algorithm, used for decoding convolutional codes, treats the entire received sequence as a single block and finds the most likely original data sequence.

Each type of error-correcting code has its strengths and weaknesses, and the choice of which to use depends on the specific requirements of the communication system

under consideration. Factors to consider include the characteristics of the channel (e.g., whether errors tend to occur in bursts or are randomly distributed), the data rate, and the acceptable level of errors. The purpose of this study is to compare the performance of polar codes and convolutional codes, each representing a different class within this classification, under the same conditions and evaluate their suitability for short-range communications such as Bluetooth.

2.1.3 Error Types and the Role of Cyclic Redundancy Check

A crucial aspect of evaluating and comparing different error-correcting codes (ECCs) involves understanding the metrics used to measure their performance and the different types of errors that can occur. This includes metrics like Frame Error Rate (FER) and Signal-to-Noise Ratio (SNR), as well as understanding the role of mechanisms like the Cyclic Redundancy Check (CRC) in decoding.

In this context, the CRC acts as a final layer of error detection. It uses a specific mathematical algorithm to generate a check value for each block of information bits, which is then appended to the frame before transmission[8]. Upon reception, the receiver recomputes the CRC for the received data and compares it to the received CRC. If the two values match, the frame is accepted as correctly received; otherwise, it's considered corrupted.

CRC algorithms vary primarily in their bit size, which is determined by the generator polynomial utilized. Polynomials with higher degrees will produce larger CRCs, enhancing the reliability of error[8]. Our primary focus in this thesis will be the 24-bit CRC employed in the Link Layer of the Bluetooth protocol from 3.1.1 page 2733 [10], hereby referred to as CRC24. The polynomial that defines CRC24 is $x^{24} + x^{10} + x^9 + x^6 + x^4 + x^3 + x + 1$.

In addition to CRC24, we will also examine a shorter CRC: the 6-bit CRC, which represents the minimal CRC size used in the 5G channel coding [12], hereby denoted as CRC6. The defining polynomial for CRC6 is $x^6 + x^5 + 1$.

It is important to note that while CRC6 may increase throughput, since a larger proportion of the transmitted bits are information rather than redundant CRC bits, it is likely to introduce more errors than CRC24 due to its lesser reliability. If the

message is correct, both of the CRC types will approve the message, but if there are errors, a shorter CRC like CRC6 will introduce false positives, accepting a wrong codeword as correct. Therefore, the choice between different CRC types involves a trade-off between reliability and throughput.

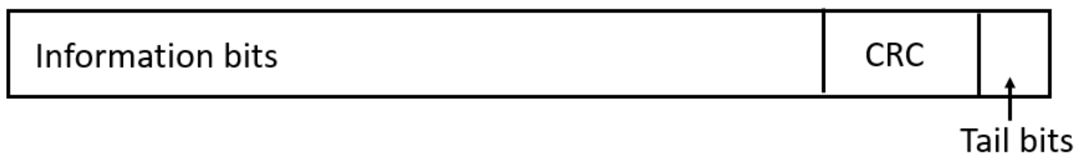


Figure 2: Data frame of length K.

The Frame Error Rate (FER) is a measure of the proportion of corrupted frames, out of the total number of transmitted frames. A data frame is considered corrupted if the CRC fails to validate the received data, as explained in the previous section. For Polar codes, a data frame of length K contains both the information bits and the appended CRC, as shown in Figure 2. Here, the CRC is attached onto the end of the data as per the procedure discussed earlier, with the specific ratio of the bits depending on the chosen frame length K and the CRC variant used.

In the case of convolutional codes, a data frame of length K is formed by the combination of information bits, CRC, and tail bits (to be discussed in a later section). The check value for the CRC gets generated based on the information bits, and the composition of the frame is shown in Figure 2, with the tail bits at the end.

FER serves as a vital metric for evaluating the efficiency of an ECC since it directly measures the successful transmission of entire data frames, which is the primary objective of data communication. A lower FER implies a higher success rate in data transmission, thus indicating a more effective error correction system.

However, the process of decoding with CRC introduces two additional types of errors: false positives and false negatives.

False positives occur when a corrupted frame is incorrectly identified as error-free

after the decoding process. This happens when the CRC by mistake validates the the received data. We hereby refer to this as CRC-error. This missclassification can result in incorrect data moving higher in the stack and can be especially harmful in critical applications where the integrity of data is paramount. On the other hand, false negatives happen when a frame is inaccurately flagged as an error despite being correctly received. This is less harmful for the application but can lead to unnecessary retransmissions and a reduction in the overall data rate.

The Signal-to-Noise Ratio (SNR) is a measure of the signal strength relative to the level of noise in a channel. Higher SNR values correspond to lower levels of noise, leading to fewer transmission errors. SNR is often used in combination with metrics like FER to evaluate the performance of different ECCs. For example, an ECC that maintains a lower FER at a given SNR level is considered superior to others that perform worse under the same conditions. The role of FER and SNR becomes particularly clear when we consider FER vs. SNR plots. These plots provide an intuitive visual representation of how an ECC's performance varies with changing channel conditions.

2.1.4 AWGN channel and soft decoding

A channel, in the context of information theory and communications, refers to a medium or pathway through which data is transmitted from one point to another. It is essentially the conduit for data transmission, playing a key role in determining the quality and reliability of the information received at the other end.

One of the fundamental models used in digital communication is the Additive White Gaussian Noise (AWGN) channel. This model assumes a signal, while traversing the communication channel, encounters noise that follows a Gaussian distribution. The term white refers to the noise having equal intensity at varying frequencies, which suggests it contains a wide range of frequency components.

In our study, we will focus specifically on the Binary Phase Shift Keying (BPSK) modulation scheme operating over an AWGN channel, as shown in Figure 1. BPSK is a digital modulation technique where the phase of the carrier signal is modified to represent binary data. When a bit is transmitted over this channel, it is affected by noise, and thus the received signal will not be a perfect representation of the

original bit but a 'soft' version of it, thus giving rise to the term 'soft decoding'.

Soft decoding leverages this additional information about the received signal's likelihood of representing a '0' or '1'. Instead of making an immediate decision about the transmitted bit (as is done in 'hard decoding'), soft decoding assigns a likelihood to each bit and uses these values in the decoding process to improve error correction performance. This strategy often provides superior error correction compared to hard decision decoding, especially over noisy channels like the AWGN channel.

2.2 Convolutional code

Convolutional codes, a type of Error Correcting Code (ECC), emerged in the late 1950s and are noted for their extensive application in digital communication systems. Unlike block codes, which work on fixed-size blocks of bits, convolutional codes work on data streams of arbitrary length. This is chosen as a focus of our study because it is the coding scheme used in Bluetooth[10].

2.2.1 Convolutional encoding

The process of convolutional encoding involves using a k/n rate encoder, which takes in k input bits at a time and produces n output bits, often reefed to as a symbol. The code rate, $R = k/n$, signifies the rate at which frames are being transmitted. For instance, if the rate is 1/2, it means that for each bit in the frame, two bits are being sent. The additional bits are redundancy added by the encoder to correct possible errors during transmission.

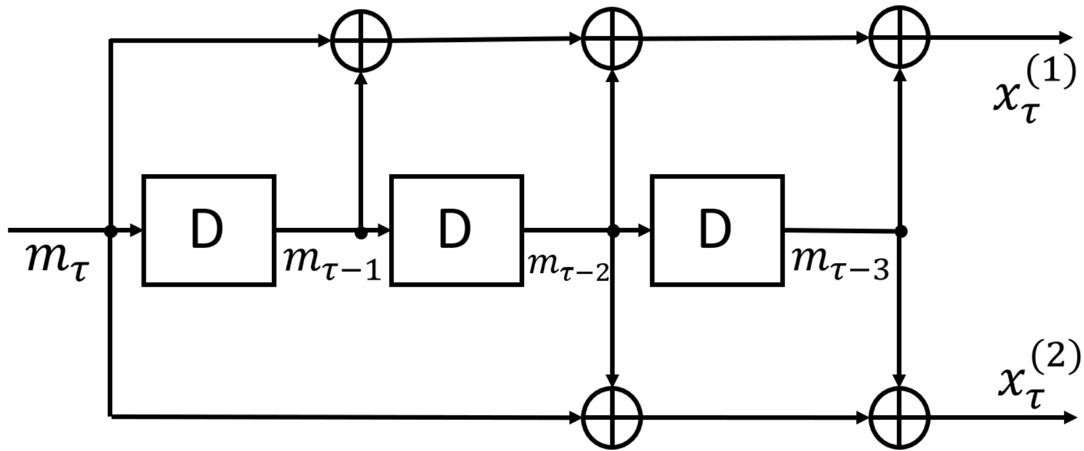


Figure 3: 1/2 rate convolutional encoder, from 3.3.1 page 2736 in [10].

A simple 1/2 rate convolutional encoder with three shift registers, D, several modulo-2 adders, $\oplus$, and input of length $k = 1$ are shown in Figure 3. The input bit, m_τ for $1 \leq \tau \leq K$, are shifted through the register, and at each stage, modulo-2 addition is performed on selected bits in the register to produce the encoded output bits, $x_\tau^{(1)}$ and $x_\tau^{(2)}$. This exact convolutional FEC encoder is the one used in the link layer of the Bluetooth protocol[10]. The initial state of the shift registers, when $\tau = 1$, is set to all zeros. We name this the zero state. We also note that an input sequence consisting of three consecutive zeros returns the encoder to its original state. This sequence is known as the termination sequence.

The size of the shift register, Q, stores the present bit, m_k, and the $Q - 1$ past bits. Q is the constraint length and for the encoder in Figure 3 $Q = 4$. Given the design of the convolutional encoder, its output is influenced by Q bits. This sets some rules for what the output can be. These limitations of the output are what we use to create a decoder.

2.2.2 Trellis diagram

A Trellis diagram is a visual representation used to describe the operation of convolutional codes, particularly during the decoding process. It displays all possible states of the encoder and the transitions between these states for every possible input bit sequence.

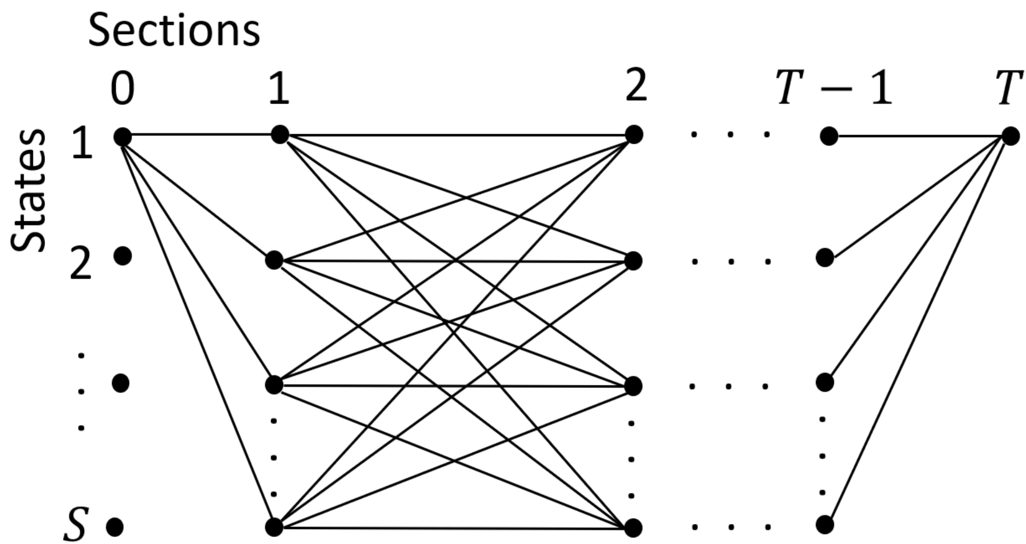


Figure 4: Fully connected trellis with S states[].

An S state fully connected trellis is shown in Figure 4. Each line in the trellis diagram represents a unit of time, going from section 0 to T. Each node at a particular section corresponds to a possible state of the encoder, at a given time, and the branches between nodes represent the transition from one state to another.

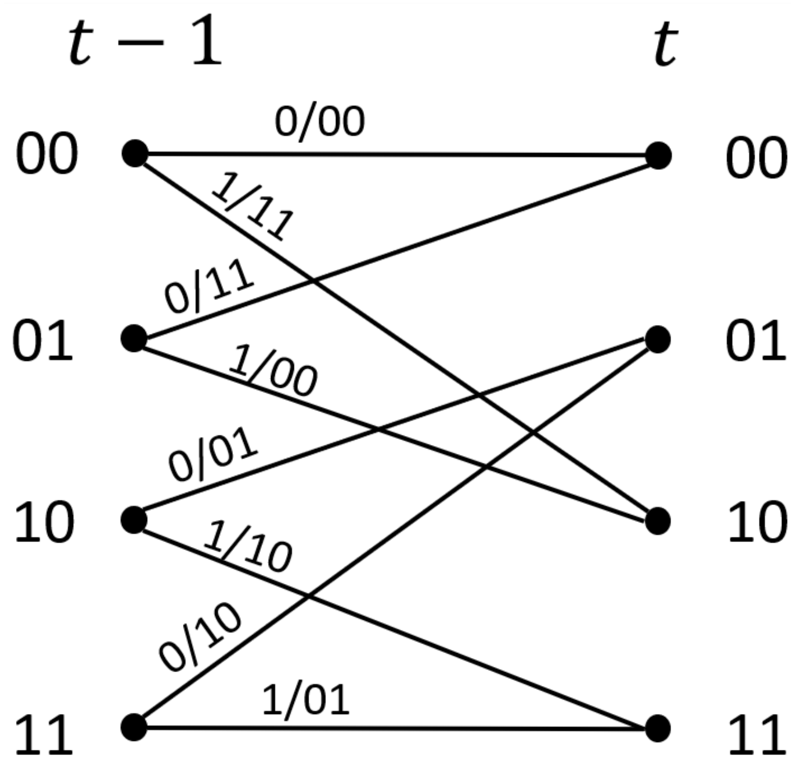


Figure 5: Section of a 4 state trellis. The label of each branch represent the corresponding input and expected output separated by '/'.

Figure 5 provides a detailed view of transitions between two specific time instances, $t-1$ and t, within a given trellis diagram. This particular diagram is of an encoder with 1/2 code rate and constraint length $Q = 3$. The constraint length define the number of states as $S = 2^{Q-1} = 4$. The states are labeled with a binary number corresponding with the bits in the shift register of the encoder. The most significant bit (msb), to the left, correspond to the incoming input, m_k in Figure 3, while the next bits correspond to the shifted inputs. Each state has a number of outgoing branches equal to the number of possible inputs for the encoder, computed as $2^k = 2$. These branches represent potential states the encoder could transition to upon receiving a certain input. This input among with the resulting output that will occur are labelled above the corresponding branch, separated by a "/". The trellis is then drawn by connecting all the outgoing branches form the states at time instance $t-1$ to all the next states the encoder can be in at time instance t.

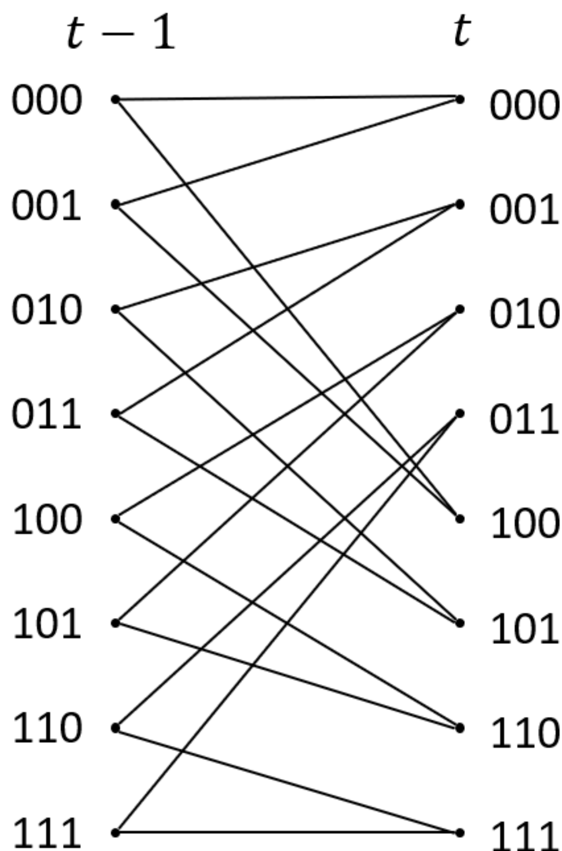


Figure 6: Section of the trellis diagram for the encoder in Figure 3.

We will now look at the trellis corresponding to the encoder addressed in Section 2.2.1. This encoder has a constraint length of 4 which results in 8 states addressed as three and three bits at each node, shown in Figure 6. We still have two outgoing

branches because of the one bit input.

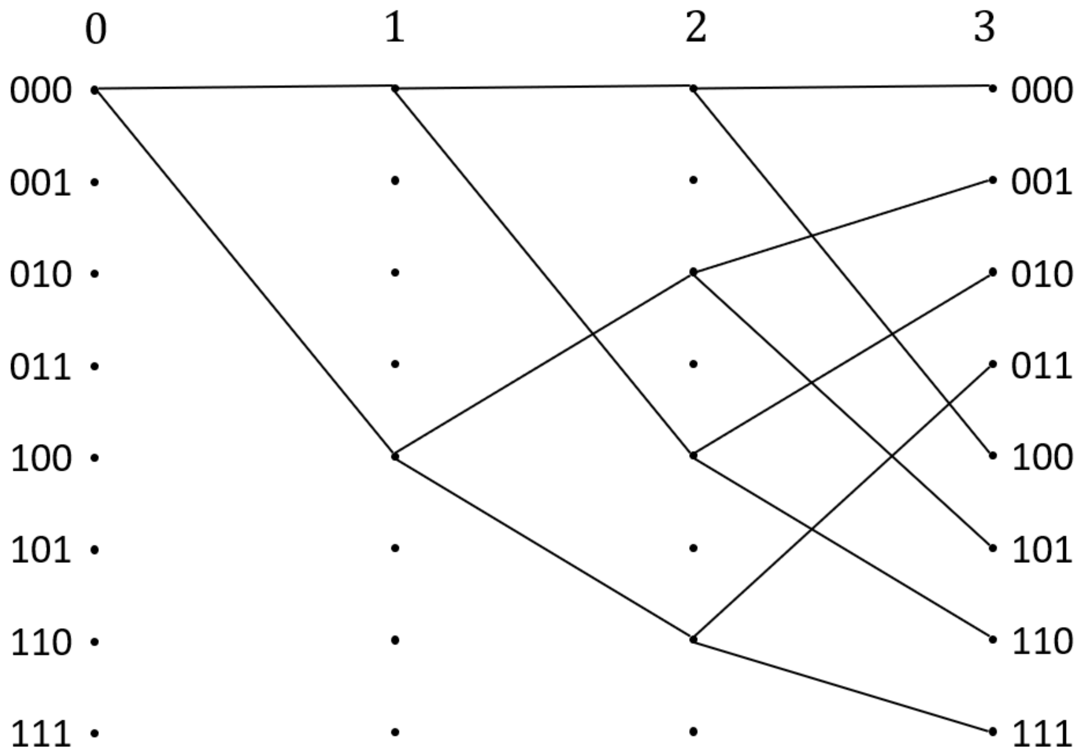


Figure 7: Beginning of the trellis for the encoder in Figure 3.

As discussed in Section 2.2.1, the encoder will start in a zero state at time instance 0. This result in branches form all other states at this instance are not valid. The beginning of the trellis is shown in Figure 7. When we now look at paths in the trellis, continuous connections from one state to another state at a later time instance, we see that the possibilities of paths is low in thee beginning. At time instance 3, there is only one possible path that reaches each of the states, starting from the zero state, 000.

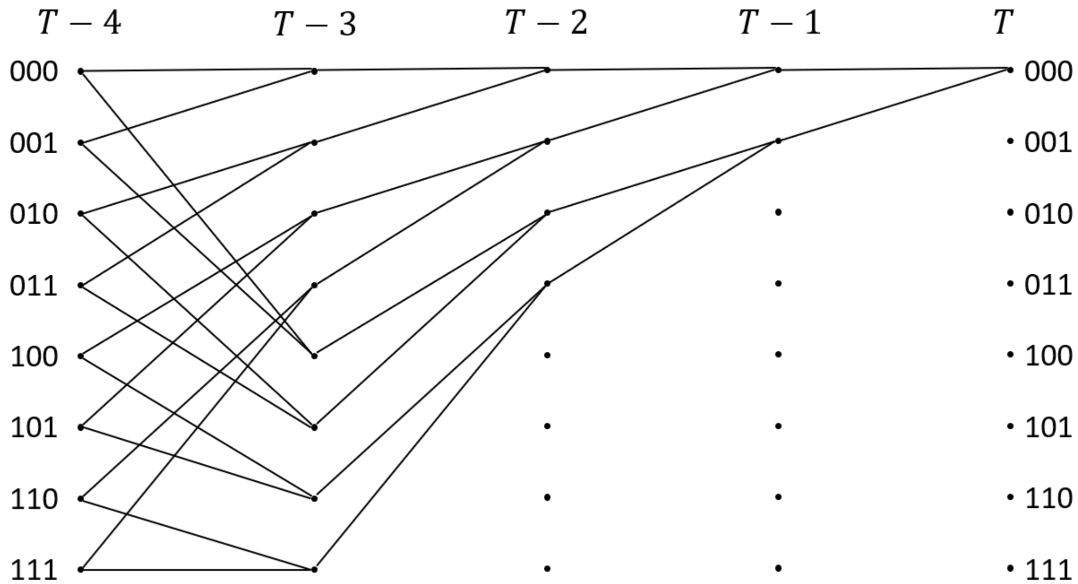


Figure 8: Beginning of the trellis for the encoder in Figure 3.

Implementing a termination sequence of length $Q-1$ as tail bits, at the end of the input frame, ensures the encoder returns back to its initial state. For the encoder discussed above, this would be 3 zeros resulting in the trellis in Figure 8. This also reduces the possible paths, due to the input only being 0 and the branches reducing to one for each node. The fact that we know the only possible state at the final instance T provides valuable insights when it comes to decoding.

The trellis diagram is especially useful when implementing the Viterbi Algorithm, as it allows for an intuitive understanding of the different possible paths through the states.

2.2.3 Soft-decision Viterbi decoder

Viterbi decoding, proposed by Andrew Viterbi in 1967, is a maximum likelihood decoding algorithm used for convolutional codes[3]. It is a dynamic programming algorithm, which means it avoids unnecessary computations by storing intermediate results for reuse in later computations.

The Viterbi decoding algorithm involves traversing the trellis of the encoder from left to right, from the start state to the end state. We assume the decoder was initialized in a zero state and that we have the termination sequence as tail bits of the frame.

When employing soft decoding in conjunction with BPSK over an AWGN channel, as discussed in Section 2.1.4, the Viterbi algorithm utilizes the square of the Euclidean distance. This distance represents the distance between two points within Euclidean space. The squared Euclidean distance is determined by calculating the difference between the received symbol and the expected symbol and then squaring the result. The expected symbols are found from the trellis, assigned to each possible branch, illustrated in the example in Figure 5. Due to the nature of BPSK, which produces real-valued symbols, and the real-valued nature of the expected symbols, the calculation remains straightforward. We name this value the branch metric, $c_t(j,i)$. This is the incremental cost that comes when moving from state j at time $t-1$ to state i at time t. $c_t(j,i)=\infty$ if j and i are not connected.

The goal of the decoder is to find the path through the trellis, from the first state to the last, that accumulates the lowest possible path metric. The path metric, denoted as $\phi_t(i,k)$, is the metric of the $k^{\mathbf{th}}$ best path beginning from the trellis's initial state (state 1 at time 0) up to state i at time instance t. It is calculated by summing up all the branch metrics of the branches the path take up to state i. The $k^{\mathbf{th}}$ best path implies the path that accumulates the $k^{\mathbf{th}}$ lowest path metric. For each time time instance in the trellis, starting at $t=1$, these cumulative metrics are stored at at each state, i for $1 \leq i \leq S$. For the Viterbi decoder, $k=1$, meaning we only store the best path metric at each state. When moving from time instance $t-1$ to t, path metric $\phi_t(i,1)$ gets calculated based on the previous path metric at time $t-1$ plus the branch metric leading to the new state i as

$$\phi_t(i,1) = \min_{1\leq j\leq S}[\phi_{t-1}(j,1) + c_t(j,i)], \qquad 1 \leq i \leq S. \tag{1}$$

This function iterates all the previous states j, resulting in S path metrics for each state i at time t, but since $k=1$, the minimum of the path metrics is chosen. Among with the branch metric calculation, the history of the previous state j in occasion of the $k^{\mathbf{th}}$ minimum path metric that is chosen, is stored in an $S \times T$ matrix, $\xi_t(i,k)$. This matrix gets filled up by iterating Equation 1 from $1 \leq t < T$.

For the initialization we set $\phi_0(i,1) = 0$ for $1 \leq i \leq S$. Since $c_t(j,i) = \infty$ if j and i are not connected, only paths starting at state 1 (at time 0) will survive the iteration of Equation 1 when we increase time and move right in the trellis. For the termination at $t=T$, we are only interested in $\xi_T(1,1)$, and therefor only need

to calculate $\phi_T(1,1)$ since we know the final state will be 1 (at time T) due to the termination sequence.

When the iteration through the trellis is done and $\xi_t(i,k)$ contains the history of all the best previous states, we can get the state sequence. The best state sequence is

$$(1, s_1, s_2..., s_{T-1}, 1), \tag{2}$$

where

$$s_t = \xi_{t+1}(s_{t+1}, 1), \qquad 1 \le t \le T-1. \tag{3}$$

This sequence will give the most likely path taken in the trellis, and we can read the corresponding inputs for this specific path.

In essence, the Viterbi Algorithm provides an efficient way to navigate the potentially exponential number of paths in a trellis diagram and identify the most likely sequence of input bits given the received output bits. It does this by systematically pruning less likely paths, finally ending with the most likely one.

2.2.4 CRC-aided List Viterbi Algorithm (LVA)

Building upon the foundation laid by the Viterbi Algorithm, we now shift our focus towards its extension - the List Viterbi Algorithm (LVA) as outlined in the seminal work of Seshadri and Sundberg [4]. The central feature of the LVA is its capacity to keep track of more than one most likely path simultaneously. This allows for a richer selection of potential decoded sequences, a feature that could significantly enhance the overall system performance. In the following section, we will delve into a more rigorous mathematical examination of the LVA. The goal is to present a thorough, yet accessible understanding of this algorithm, thereby comprehending its potential to enhance decoding performance.

The fundamental principles of the LVA are the same as for the Viterbi Algorithm. We iterate through the time instances in the trellis and calculate $\phi_t(i,k)$, the $k^{\mathbf{th}}$ lowest branch metric to reach state i at time t from the initial state 1 at time 0, same as before. The difference is that $1 \le k \le L$ where L is the list size.

The recursion will now be

$$\phi_t(i,k) = \min_{\substack{1\le j\le S\\1\le l\le L}}{}^{(k)}[\phi_{t-1}(j,l) + c_t(j,i)], \qquad 1 \le i \le S, \tag{4}$$

where $\min^{(k)}$ denotes the $k^{\mathbf{th}}$ smallest value. The main difference in the LVA from the normal Viterbi algorithm is that we in addition to iterate over the states, $1 \le j \le S$, we also for each state iterate over the lists at that state, $1 \le l \le L$.

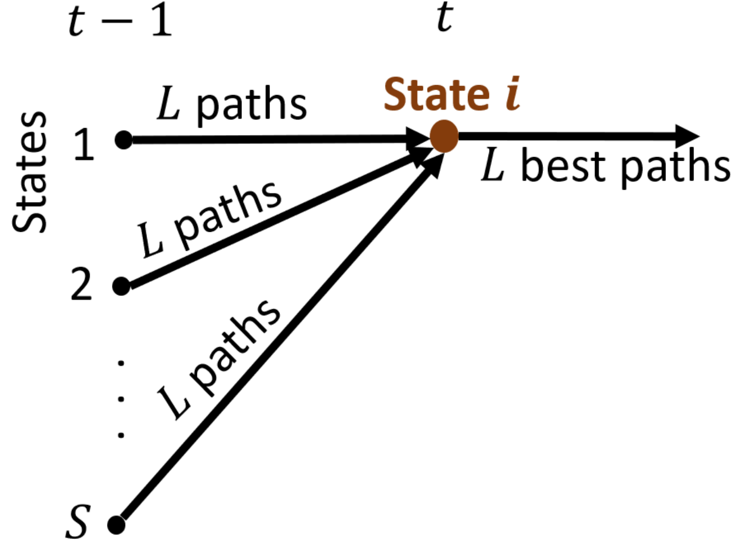


Figure 9: At state i, the L best paths gets chosen based on the path metrics and branch metrics of the $S \cdot L$ incoming paths.

This give a total of $S \cdot L$ paths to be considered for each state i, as shown in Figure 9. The state directly prior to i is referred to as state j. Each of the incoming L paths have a path metric $\phi_{t-1}(j,l)$ and branch metric $c_t(j,i)$ corresponding to them.

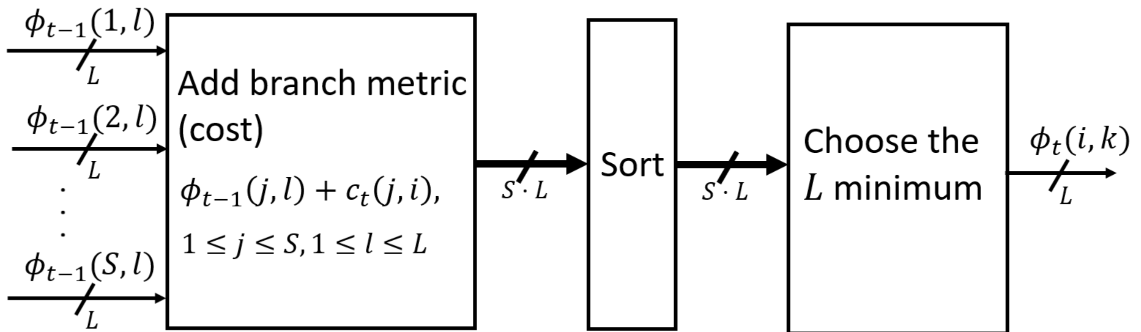


Figure 10: Block diagram of the process going on at state i in Figure 9.

Figure 10 shows a detailed block diagram of how Equation (4) operates on each

state i, marked in red at Figure 9. At the first block, $S \cdot L$ new branch metrics get calculated based on the incoming paths, before getting sorted at the next block. Then from the sorted list, the L minimum path metrics get chosen for state i. The smallest value will be $\phi_t(i, 1)$, the second smallest: $\phi_t(i, 2)$, and so on until $k = L$. Then these path metrics get sent to the next states at time $t + 1$.

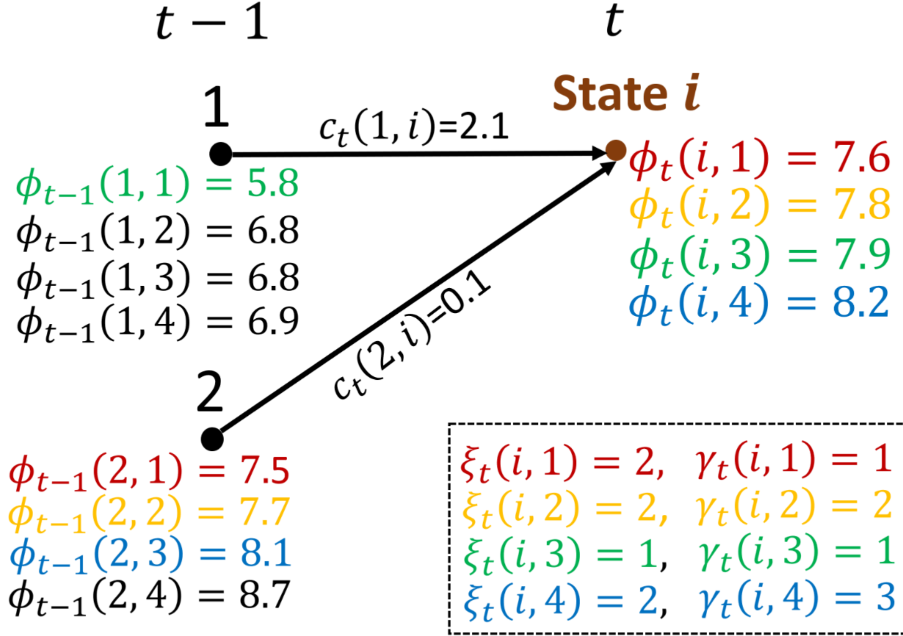


Figure 11: Example of values for all metrics at time instance t for node i, for a list size of $L = 4$. The colors showcases different paths being chosen at state i, how they are being ranked, and the history that is being stored are shown in the box.

In addition to storing the history of the previous state j in occasion of the $k^{\textbf{th}}$ minimum path metric that was chosen, we also store the corresponding ranking, denoted $\gamma_t(i, k)$. $\xi_t(i, k)$ will now be a three dimensional matrix, $S \times T \times L$, storing lists of the state at time $t - 1$ of the $k^{\textbf{th}}$ best path that passes through i at time t. $\gamma_t(i, k)$ will also be a $S \times T \times L$ matrix storing the corresponding ranking of the path discussed. This can be seen from Figure 11, where the box shows the stored data about the history of the paths of interest. That way it is easy to backtrack the full path, shown by colors. The k paths of interest are picked at time t, but the state $\xi_t(i, k)$ and rank $\gamma_t(i, k)$ stored are of the previous step this path was on. These matrix gets filled up by iterating Equation 4 from $1 \leq t < T$.

We use the same reasoning behind the beginning and end of the LVA as in the normal Viterbi algorithm. For the initialization we set $\phi_0(i, k) = 0$, $1 \leq i \leq S$, $1 \leq k \leq L$. Since $c_t(j, i) = \infty$ if j and i are not connected, only paths starting at

state 1 (at time 0) will survive the iteration of Equation 4 when we increase time and move right in the trellis. For the termination at $t = T$, we are only interested in $\xi_T(1, k)$ and $\gamma_T(1, k)$, and therefor only need to calculate $\phi_T(1, k)$ since we know the final state will be 1 (at time T) due to the termination sequence.

After the iteration over the all time instances, the path backtracking remains. The $k^{\mathbf{th}}$ best state sequence will be

$$(1, j_1, j_2..., j_{T-1}, 1), \tag{5}$$

where

$$\begin{aligned} j_t &= \xi_{t+1}(j_{t+1}, l_{t+1}), \\ l_t &= \gamma_{t+1}(j_{t+1}, l_{t+1}), \\ j_{T+1} &= \xi_T(1, k), \\ l_{T+1} &= \gamma_T(1, k). \end{aligned} \tag{6}$$

These sequences will give the $k^{\mathbf{th}}$ most likely path taken in the trellis, and we can read the corresponding inputs for these specific paths. After we have a list of the $k^{\mathbf{th}}$ most likely input frames, CRC is performed, starting from the top of the list. If the CRC passes, the frame is accepted as correctly received. If not, we iterate the list until a frame gets accepted. If this does not happen, we declare a frame error.

2.3 Polar codes

In his pioneering work, Arıkan introduced Polar codes, defining them as capacity-achieving codes for a specific class of channels, known as symmetric binary-input memoryless channels[7]. Central to the operation of Polar codes is the principle of channel polarization, a concept Arıkan extensively explored in his seminal paper. In this section, the fundamentals of Polar codes will be examined in detail. This includes an in-depth look into channel polarization and a look at the procedures involved in the encoding and decoding of Polar codes. Most of this section is based on previous work of this author[1].

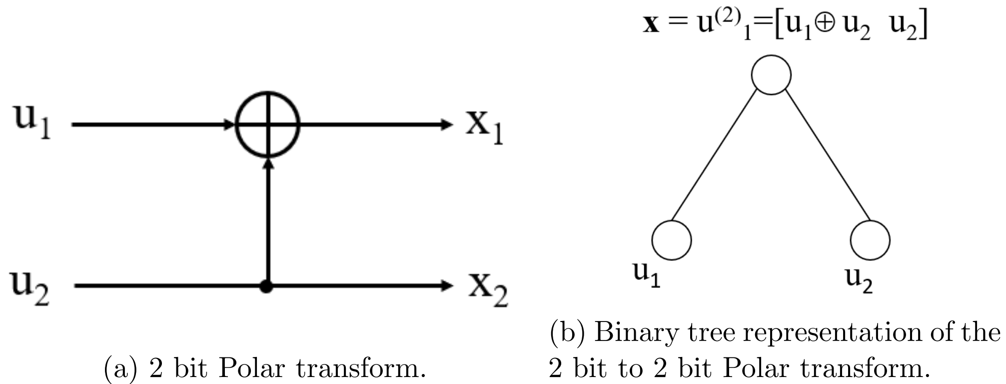
2.3.1 Polar transform

The Polar transform, also known as the Arıkan transform, is a fundamental construct in Polar codes. Fundamentally, the Polar transform processes an input of 2^n bits and generates an output of the same size, where n is any positive integer. Mathematically, it is defined as

$$\mathbf{x} = \mathbf{u} \cdot \mathbf{G}^{\otimes n}, \tag{7}$$

where $\mathbf{x}$ symbolizes the output-vector, $\mathbf{u}$ signifies the input-vector, and $\mathbf{G}^{\otimes n}$ is the generator matrix. This matrix is computed by applying the n-th Kronecker product to the polarization matrix, $\mathbf{G} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.

Consider the simplest instance where $n = 1$, and we have $\mathbf{u} = \begin{bmatrix} \mathrm{u}_1 & \mathrm{u}_2 \end{bmatrix}$ as the input-bits. The generator matrix is equivalent to the polarization matrix, resulting in the output-bits being $\begin{bmatrix} \mathrm{x}_1 & \mathrm{x}_2 \end{bmatrix} = \begin{bmatrix} \mathrm{u}_1 & \mathrm{u}_2 \end{bmatrix} \cdot \mathbf{G} = \begin{bmatrix} \mathrm{u}_1 \oplus \mathrm{u}_2 & \mathrm{u}_2 \end{bmatrix}$.



(a) 2 bit Polar transform.

(b) Binary tree representation of the 2 bit to 2 bit Polar transform.

Figure 12: Two ways of visualizing how the 2 bit polar transform is done.

The polar transform of this simple instance where $n = 1$ is shown in Figure 12a. The transformation process can also be visualized using a binary tree, as seen in Figure 12b. The input bits are positioned at the bottom, referred to as leaf nodes, while a parent node resides above. The parent node is a derivative of the two nodes beneath it: the first part is the XOR of the two nodes, and the second part is a direct copy of the right node. The superscript (2) tells how many bits the node

contain and the subscript is to separate the nodes at the same depth. The output of the transformation is represented by the uppermost node.

Moving onto the case where $n = 2$, we encounter a 4 bit to 4 bit transformation. The generator matrix now becomes the Kronecker product

$$\mathbf{G}^{\otimes 2} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}. \quad (8)$$

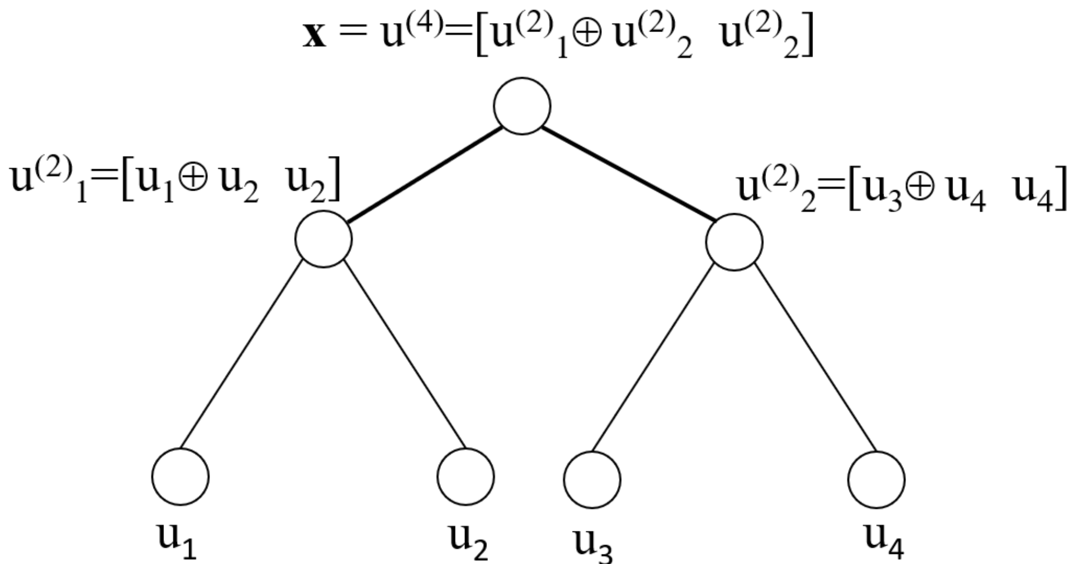


Figure 13: Binary tree representation of the 4 bit to 4 bit Polar transform.

Figure 13 illustrates the binary tree representation for this 4 bit to 4 bit transformation. Here, four leaf nodes ascend to parent nodes in pairs. An additional layer is added where the parent nodes again combine to form the output, $\mathbf{x} = \mathrm{u}^{(4)}$, a vector of length 4. In this representation, the XOR operation and copy of the right node still hold. The top node, referred to as the root, is the output of the transform. The nodes are defined by their depth, with the root being at depth 0. The nodes one step below are at depth 1 and so on. For any n, the tree extends to depth n. Nodes are also uniquely identified by numbering them starting from 0 on the left at each depth and increasing towards the right. This numbering scheme allows us

to refer to a node uniquely using its coordinates (node, depth). For instance, (3, 2) will identify the bottom rightmost node in Figure 13, while (0, 0) denotes the root.

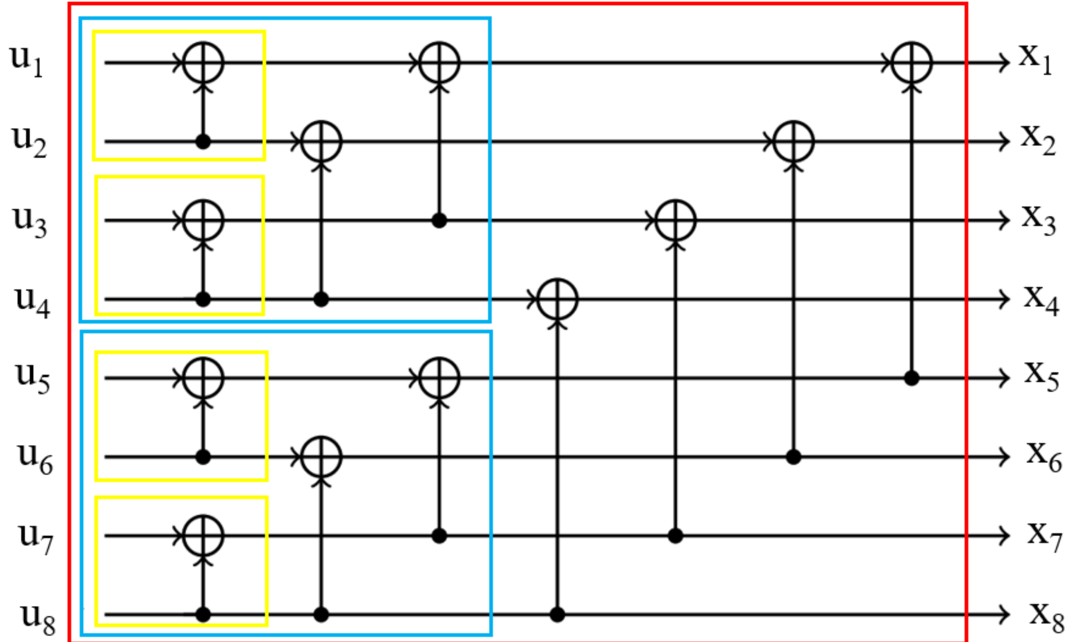


Figure 14: Expansion of the polar transform from $n = 1$ in the yellow, $n = 2$ in the blue, and $n = 3$ in the red box.

Figure 14 show a diagram of an 8-bit transformation, showcasing how the transformation expands with increasing size. In the smallest box, colored yellow, we have the same structure as in Figure 12a. If we exclusively look at the blue medium box, we have a 4-bit Polar transform, the same as in Figure 13. It is clear to see a pattern here, and this is how the bit operations will be done when the transform expands.

2.3.2 Channel polarization

Polar codes operate based on a key principle known as channel polarization. First introduced in Arıkan's paper [7], channel polarization is a transformative process that generates N synthetic bit-channels from N individual copies of a binary-input discrete memoryless channel (B-DMC). Each of these synthetic channels is uniquely characterized by its reliability, its likelihood of successfully decoding a single bit. With a sufficiently large N, the mutual information of the synthetic channels tends towards extremes, either close to 0 for highly noisy channels or near 1 for virtually noiseless channels. This polarization effect enables the creation of channels with

remarkable performance characteristics.

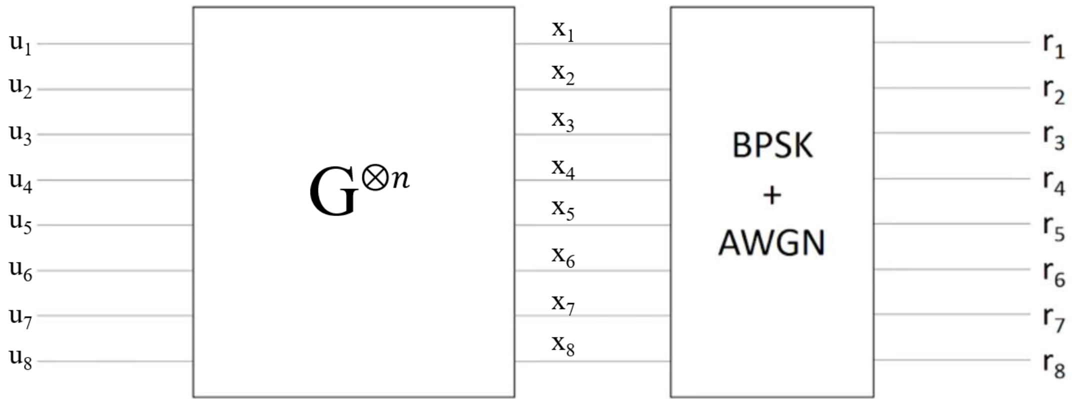


Figure 15: Block diagram of bit-flow from input vector **u** to received vector **r**.

Channel polarization can be visualized by imagining separate bit-channels for each bit we transmit, each with unique properties. The example depicted in Figure 15 illustrates an input vector **u** comprising bits u_1 to u_8 at the input to the left. This input undergoes the Polar transform (as detailed in Section 2.3.1) via the $G^{\oplus n}$ matrix, before being transmitted using BPSK modulation in an AWGN channel. The received bits r_1 to r_8 form the vector $r^{(8)}$.

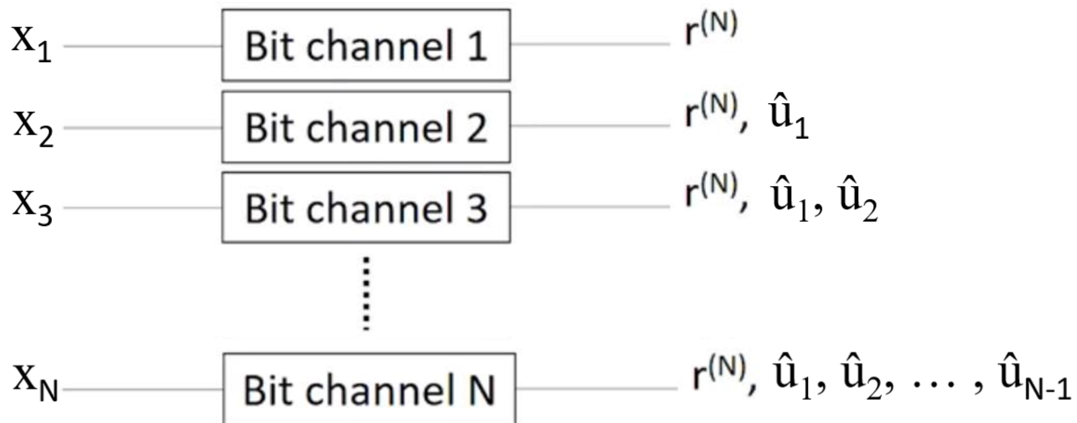


Figure 16: Visualization of information-flow through bit channels.

The distinctive properties of the bit-channels are highlighted in Figure 16. An input of N bits (where $N = 2^n$ is the block length) is fed into the channels. The bit-channel outputs, shown on the right, include the entire received vector $\mathbf{r}^{(N)}$ of length N and data about previous values. The values $\hat{u}_i$ are approximations of the message bits in **u** rather than true outputs of the channels. Consequently, bit-channels further along in the sequence (with higher numbers) deliver more information and provide

better insight into the input. Arıkan demonstrated that, under these conditions, the quality of the bit channels polarizes - some channels become highly reliable (good), and others become largely unreliable (bad). Furthermore, as N increases, all bit channels approach either high reliability (low error probability) or low reliability (high error probability) [7]. This was a groundbreaking result and a key concept for understanding how polar codes work.

Simulations and estimates enable us to examine this polarization process and identify the good and bad bit channels[13]. The ordering of these channels, arranged from worst to best, forms the reliability sequence, are an active area of research aiming to find the most precise sequence. 3GPP has suggested one such reliability sequence in the 5G standard for up to $N = 1024$, which can be used to generate smaller sequences [12]. This sequence offers practical insights into the polarization properties of polar codes, as detailed by Arıkan.

2.3.3 Encoding

The coding scheme of polar code is uniquely defined by three parameters, code rate $R = K/N$ as defined before, block length $N = 2^n$ and an *information set* $\mathcal{I} \subset [N]$ of cardinality K. As we see, the block length can only be of power 2, but on the other hand, the length of the message bits K can take any arbitrary value. When we are to make a code design of a (N, K) polar code, the K message bits are assigned to the bit channels with the best quality, providing the best reliability as discussed in Section 2.3.2. The remaining $N - K$ bits in the block N make up the *frozen set* and is defined as the compliment of the information set, $\mathcal{F} = \mathcal{I}^C$. These bits do not carry any information and can be set to zero.

We can now define the $\mathcal{P}(N, K)$ polar code. We start with a message vector $\mathbf{m}$ of length K bits and an input vector $\mathbf{u}$ of length N bits. This input vector $\mathbf{u} = \begin{bmatrix} \mathrm{u}_1 & \mathrm{u}_2 & \dots & \mathrm{u}_{N-1} \end{bmatrix}$ is generated by first setting $\mathrm{u}_i = 0$ if $i \in \mathcal{F}$, which are called the frozen positions. After that, the remaining K bits of $\mathbf{u}$ get filled with the message bits in $\mathbf{m}$. The codeword $\mathbf{x}$ is now simply the polar transform of $\mathbf{u}$, exactly as in the Equation 7. This results in a complexity of $\mathcal{O}(log_2(N))$. When the codeword x now is being transmitted and decoded sequentially, we obtain the polarization effect and the message bits have the positions at the best, most reliable bit channels, and the

worst bit channels are not being used to send information.

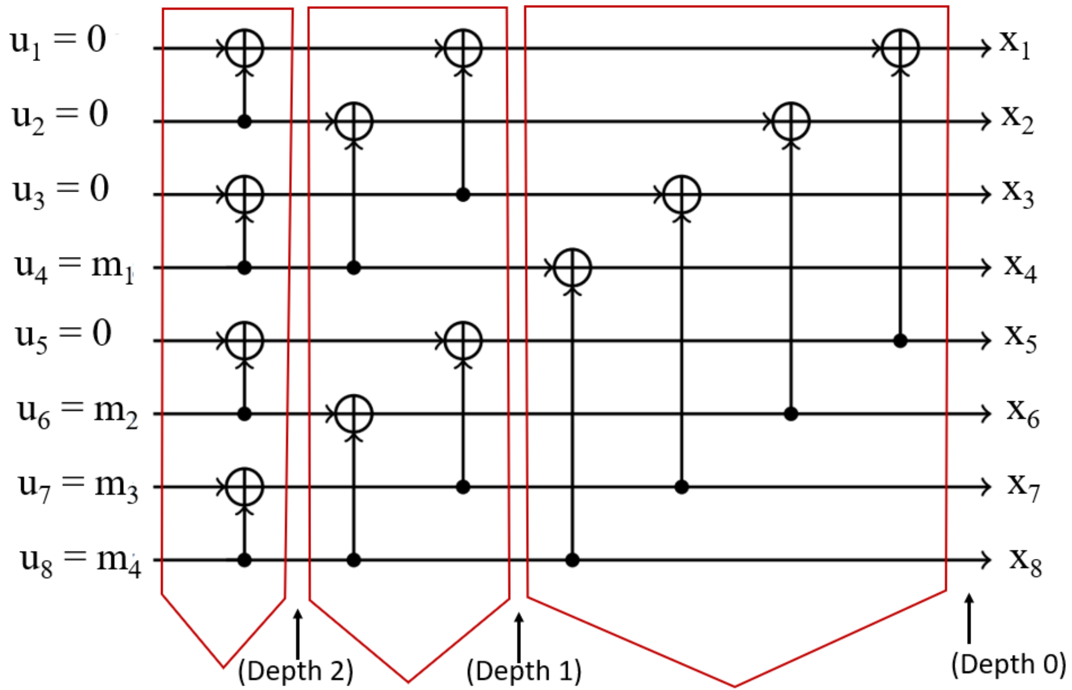


Figure 17: Example of a $\mathcal{P}(8, 4)$ polar code encoder.

The polar encoding scheme for a $\mathcal{P}(8, 4)$ can be represented as in Figure 17. Here, a reliability sequence has been used to place the K message bits at the best spots in the input. The polar transform described in Section 2.3.1 is now visualized in a new way with three steps, marked with red boxes. The link between the polar transform in Figure 17 and Figure 13 can be understood by looking at the marked depth to realize the same operations are being done. By looking at the way bits flow in Figure 17, we can also get a sense of polarization. The lower bits in the input have already given information to the bits higher up in the output.

2.3.4 Successive cancellation decoder

The polar codes are decoded using a sequential decoder, decoding them bit for bit. Arıkan suggested in his paper a successive cancelation (SC) as a sub-optimal decoder[7]. It can be visualized as a depth-first binary tree search, with the root node receiving soft information about the received code bits. The leaf nodes represent the N bits to be estimated.

There are simply two basic building blocks used in the binary tree search. One is a

two-bit single parity check code that we denote as $f(\mathrm{r}_1, \mathrm{r}_2)$, which is just a minsum with two inputs, r_1 and r_2. The other is two bits repetition, we denote as $g(\mathrm{r}_1, \mathrm{r}_2, \mathrm{b})$. The minsum function can also be written as $f(\mathrm{r}_1, \mathrm{r}_2) = \mathrm{sgn}(\mathrm{r}_1)\mathrm{sgn}(\mathrm{r}_2)\min(|\mathrm{r}_1|, |\mathrm{r}_2|)$, which can give hard decision 1 if positive and 0 if negative. The repetition function works as follows, $\mathrm{r}_2 \vee (\mathrm{r}_1 \oplus \mathrm{b})$, and can be written as the function $g(\mathrm{r}_1, \mathrm{r}_2, \mathrm{b}) = \mathrm{r}_2 + (1 - 2\mathrm{b})\mathrm{r}_1$ [5]. This uses a previously estimated bit b to make the decision.

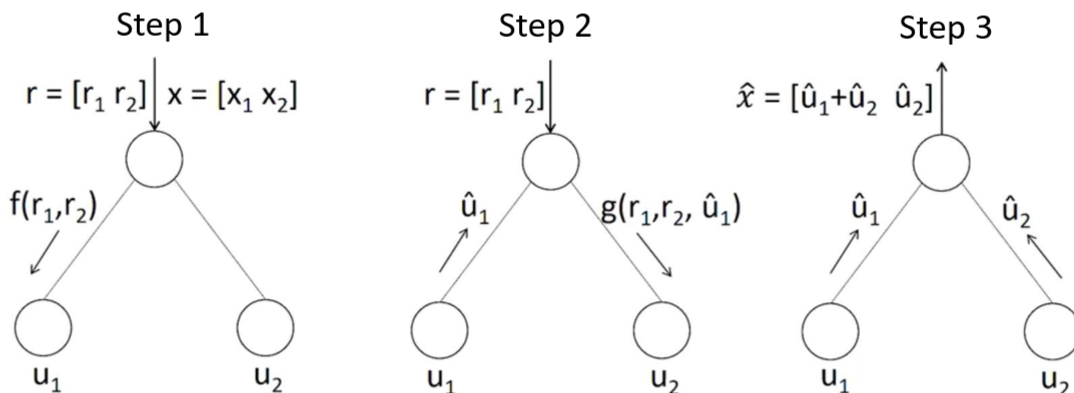


Figure 18: Fundamental building block for how the message-passing is happening in the nodes on the tree.

The depth-first binary tree search will will be done starting from the left node shown. In Figure 18 we see what happens at a specific node in the tree in three steps. First, the beliefs are coming down to the node, which is sent to the left child. If it is a leaf, the estimate $\hat{\mathrm{u}}_1$ is being sent back up. If there are more children, the same process in the figure continues until a vector of estimates gets sent back up from the left child. On step 2, we do the same to the right child, except we use the repetition function instead of the minsum, with the estimate we got from the left. On step 3, both results get combined as shown and sent up. This results in a complexity of $\mathcal{O}(N \log_2(N))$.

The idea is to let a group of SC decoders work in parallel, maintaining different codeword candidates, or paths, at the same time. Every time a leaf node is reached, the bit is estimated as both 1 and 0, doubling the number of codeword candidates; a path metric for each candidate is then calculated to discard less likely candidates, hence limiting the number of paths.

3 Results

This section presents the findings of our study, comparing polar codes and convolutional codes, and evaluating the operational performance of two decoders: the Successive Cancellation List (SCL) decoder and the List Viterbi Algorithm (LVA) decoder. Additionally, we assess the impact of two variants of Cyclic Redundancy Check (CRC) on the decoding process. The comparisons are done in terms of frame error rate (FER) for different frame lengths, K.

3.1 Experimental Setup

For the purpose of this study, we have implemented two key decoders: the CRC-aided List Viterbi Algorithm (LVA) and the CRC-aided Successive Cancellation List (SCL). We implement the CRC-aided LVA in MATLAB, shown in Appendix A, based on the approach proposed by Sundberg and Seshadri[4]. The CRC-aided SCL were implemented using the 'nrPolarDecode' MATLAB-function from the NR 5G standards, with modifications, as shown in Appendix C. The modifications includes changing the CRC from the one used in the 5G standards, to the one used in Bluetooth described in Section 2.1.3 and outputting the whole list of codewords instead of one chosen by the decoder, so the two decoders can be compared under the same conditions. Detailed elaborations of the decoders are discussed in Section 2.2.4 and 2.3.4.

Simulations of polar codes and convolutional codes in a communication system as shown in Figure 1, were implemented in MATLAB to study the decoders behavior. BPSK modulation over an AWGN channel, as discussed in Section 2.1.4, were chosen. The simulations were run under various conditions, including different list sizes L and frame lengths K, using both CRC24 and CRC6. K is the total number of bits in the frame made up of information bits, CRC bits and tail bits for convolutional codes, as shown in Figure 2. The CRC24 and CRC6 are defined in Section 2.1.3. Each simulation was run up to 200000 iterations for a given SNR or until a sufficient number of errors were detected to give a statistically reliable frame error rate (FER) with an acceptable confidence interval. For a reliable confidence level, a minimum of 100 errors was deemed necessary.

One crucial aspect of this research is the notion of a frame error. A frame error occurs when none of the decoded messages pass the Cyclic Redundancy Check (CRC). This means that the decoder, regardless of its type, could not accurately reconstruct the transmitted message, thus leading to an error on the frame level. Throughout our research, the FER is adopted as a key performance indicator to assess and compare the efficiencies of the polar and convolutional codes under various conditions.

Our research findings indicate a performance edge for polar codes over convolutional codes for frame lengths $K = 64$ and above, provided a code rate of 1/2 is maintained and CRC24. As per the results derived from the frame error rate (FER) at varying signal-to-noise ratios (SNR), polar codes demonstrated a more reliable error-correction functionality under these conditions. This indicates that for specific scenarios, particularly those with larger frame sizes than 64 bits, polar codes can effectively boost data transmission efficiency.

3.2 Efficacy of CRC24-aided LVA and SCL under varying block lengths and list sizes

To evaluate the decoders performance, all simulations uses code rate of 1/2, as specified in the Bluetooth standard. This ensures consistency in the evaluation process.

3.2.1 Polar code

We start by looking at the effects of the list size, and if the performance at some point converges.

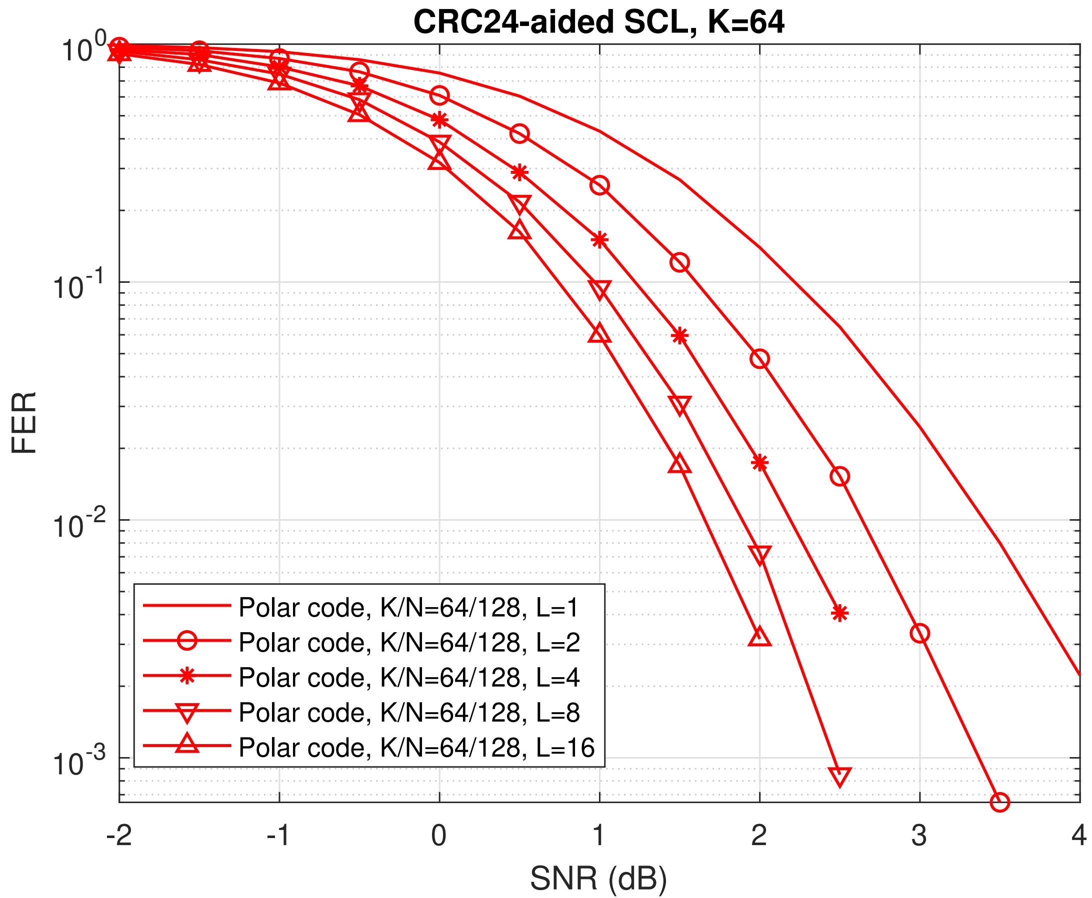
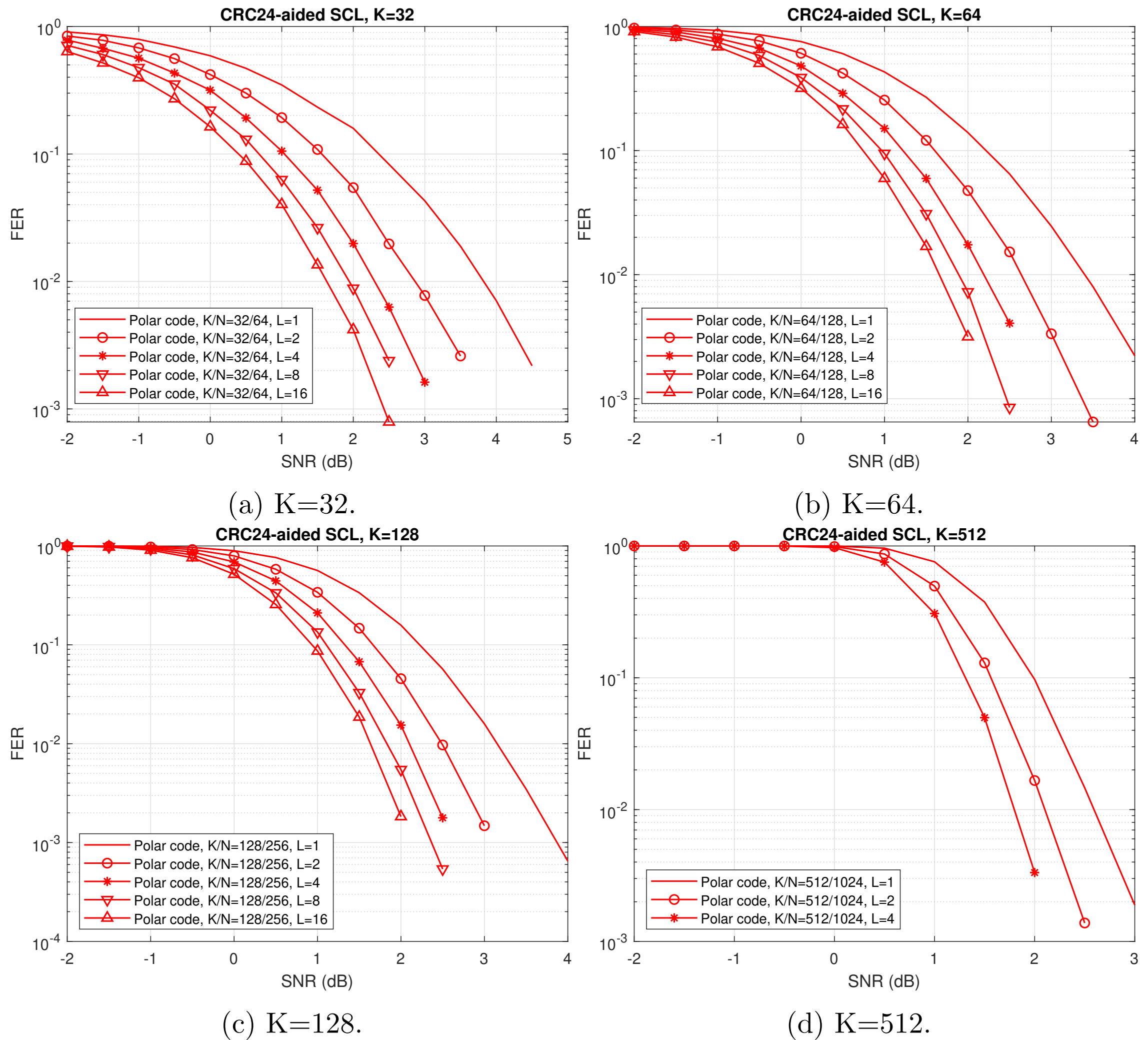


Figure 19: Polar code with $K = 64$ simulated using Appendix D.

Figure 19 depicts a comparison of the FER performance of polar codes, decoded with CRC24-aided SCL, for frame length $K = 64$, across different SNR values. Since we use code rate of 1/2 the length of the codeword $N = 2 \cdot K = 128$. The figure illustrates results for five different list sizes, $L = 16$, $L = 8$, $L = 4$, $L = 2$, and $L = 1$, from left to right. The x-axis represents the varying SNR values, while the y-axis represents the FER on a logarithmic scale.

The results demonstrate that with increasing the list size L, the performance improve. We see that when increasing from $L = 8$ to $L = 16$, there is still a slightly but noticeable improvement. This gives us an indication of there still being room for further improvement by increasing the list size, thus resulting in a larger complexity an computation.



(a) K=32.

(b) K=64.

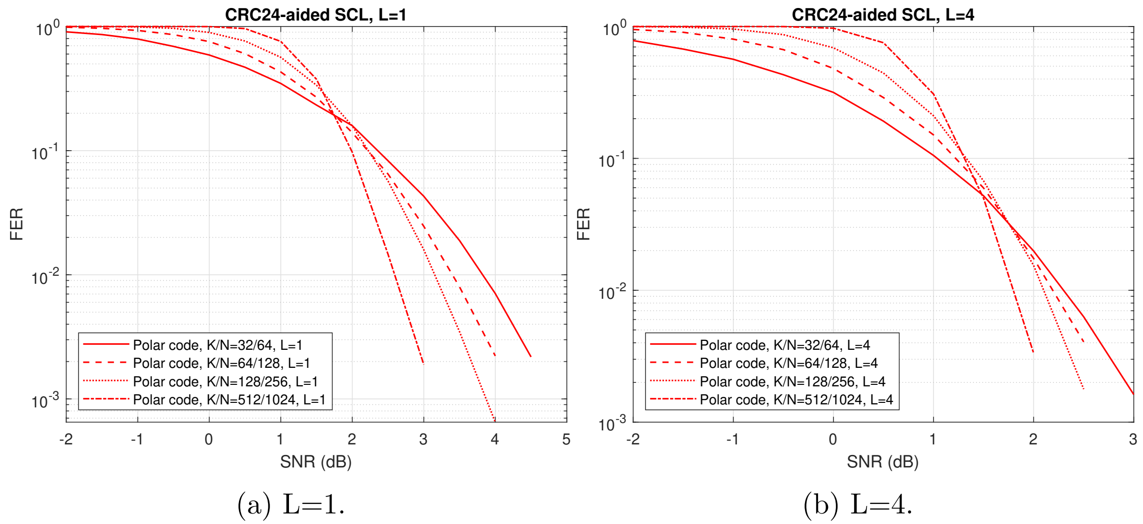
(c) K=128.

(d) K=512.

Figure 20: Polar code with constant frame lengths and varying list sizes L increasing from right to left. Code rate is 1/2 and the CRC24-aided SCL is used in the simulations.

Given the results of varying list sizes for the different block lengths $K = 32$, $K = 64$, $K = 128$, and $K = 512$ in Figure 20, we see that increasing list size improve performance, even for lists of $L = 16$. A slight tendency of convergence can be seen for the largest list sizes, but there are still improvement seen from $L = 8$ to $L = 16$, hinting even more improvement for larger list sizes, especially for the shorter block lengths. However, it is important to consider that larger list sizes correspond to higher computational requirements. Because of the large computational requirements, $L = 8$ and $L = 16$ for $K = 512$ were excluded from the simulation.

Now we will keep the list size the same, so we can see the effects block length has on code performance.



(a) L=1. (b) L=4.

Figure 21: Polar code with constant list size with varying frame lengths K.

Polar codes performed better as the frame length increases. At higher SNR values, specifically over 2.5 dB for $L = 1$ seen in Figure 21a and 2 dB for $L = 4$ in Figure 21b. For lower SNR values it is the opposite. This result comes from the polarization effect of polar codes. As discussed in Section 2.3.2 the polarization effects starts showing more for higher block lengths, moving towards capacity for infinitely large blocks.

3.2.2 Convolutional code

Now we move on to look at the effects of the list size for convolutional code, and if the performance at some point converges. The parameters of the convolutional code are the same as for Bluetooth, as described in Section 2.2.

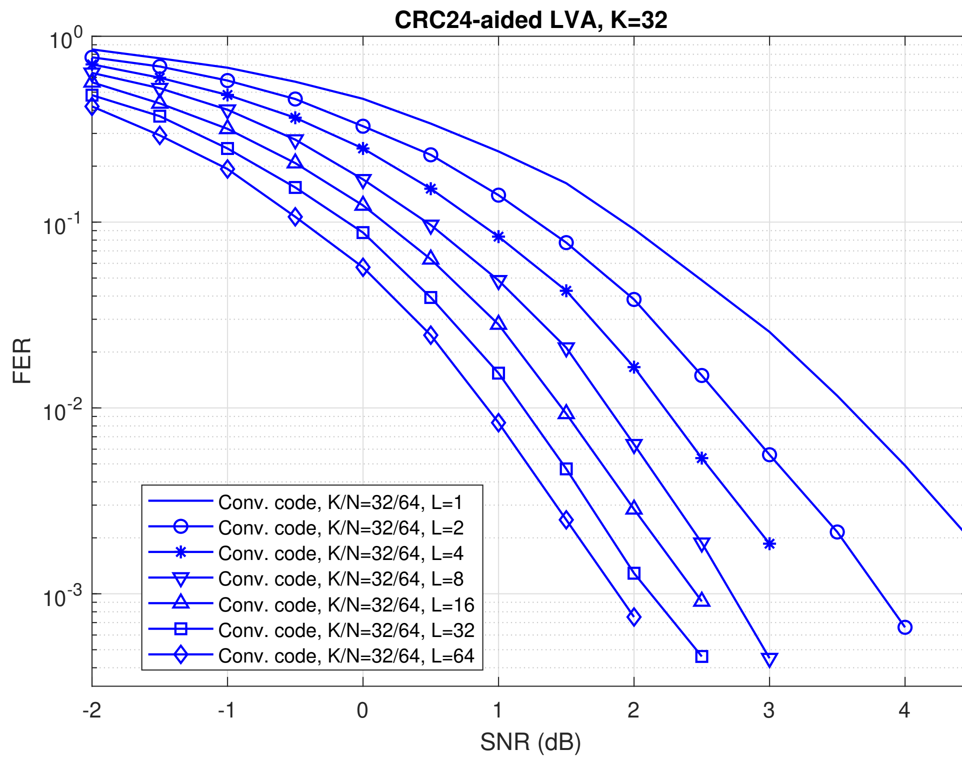


Figure 22: Convolutional code with constant frame length $K = 32$ and varying list sizes L increasing from right to left, simulated using Appendix B.

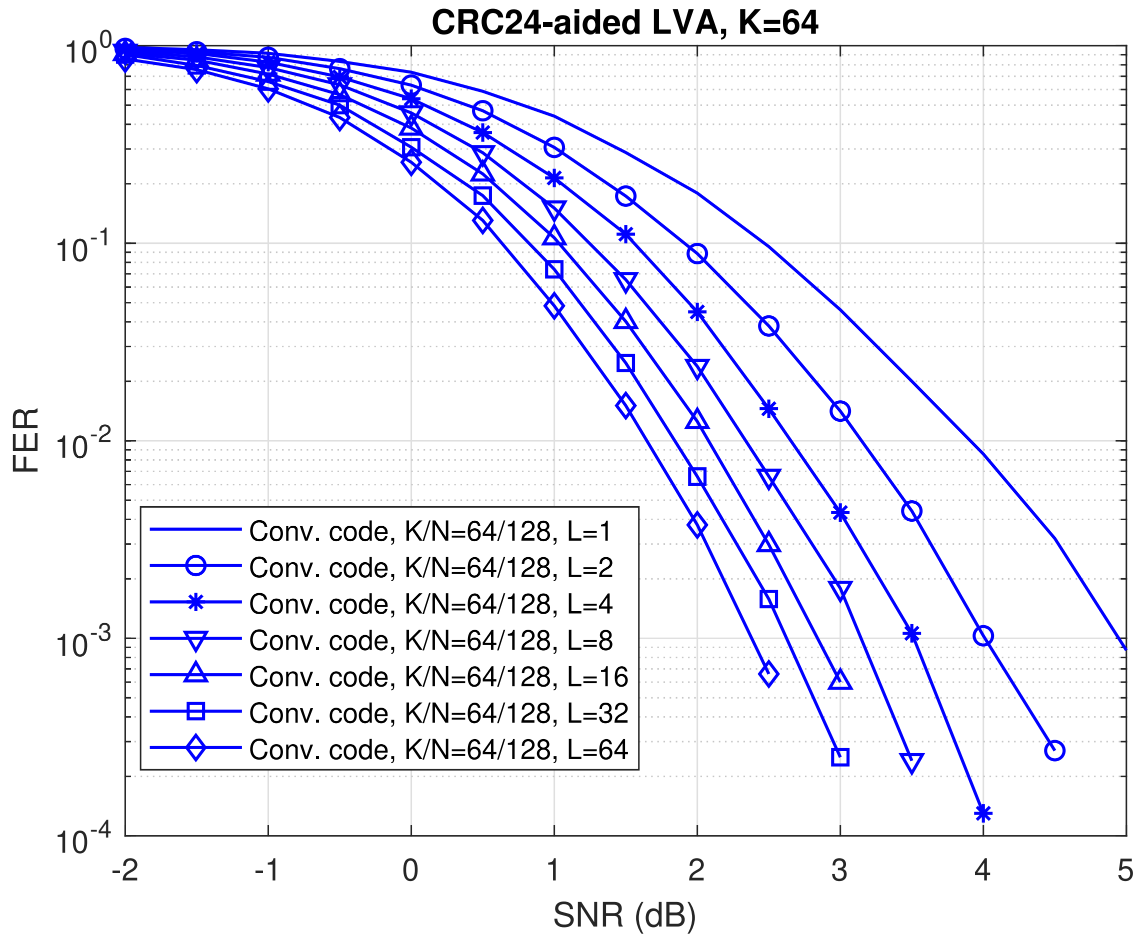
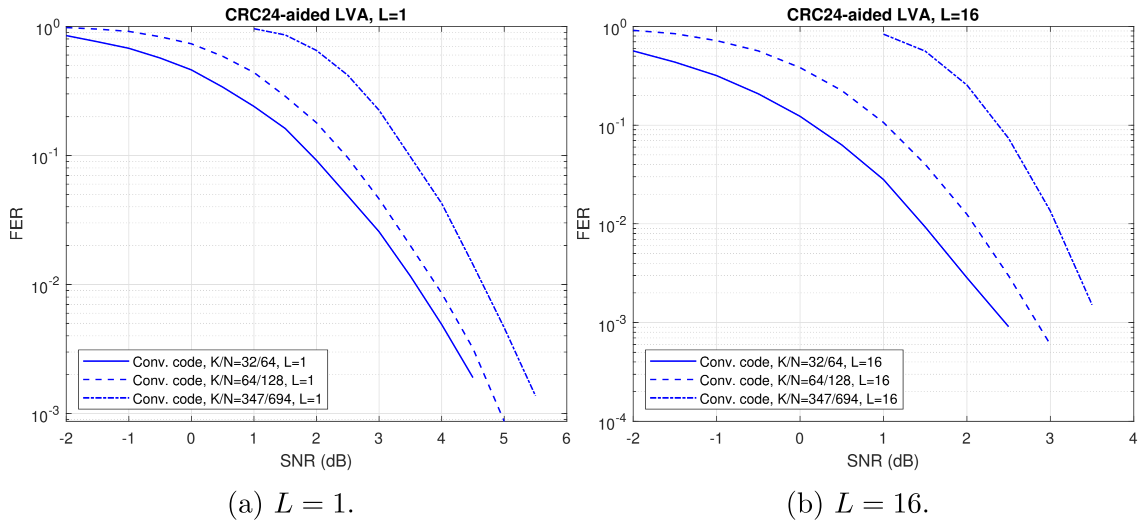


Figure 23: Convolutional code with $K = 64$ simulated using Appendix B.

Given the results of convolutional code decoded with CRC24-aided LVA, in Figure 22 and 23, we see the same tendency as for the Polar code case. We see that increasing list size improve performance, even for lists of $L = 64$. Also here, a slight tendency of convergence can be seen for the largest list sizes, but there are still improvement seen from $L = 32$ to $L = 64$.



(a) $L = 1$.

(b) $L = 16$.

Figure 24: Convolutional code with constant list size and varying frame lengths K.

When we keep the list size the same, and only vary the frame length K, we see that also here the FER curves steeper for larger size, but preform worse because it is shifted more to the right. $K/N = 347/694$ was chosen so the amount of information bits became $A = K - 24 - 3 = 320$, corresponding to 40 bytes.

An argument for why the performance generally gets worse with increase in frame length is that the encoder gets more code words to chose from, and no more information helping it choose. In addition, the polar code enhances the information about the code word when increasing the frame length.

3.2.3 Comparative evaluation of LVA and SCL

The evaluation on CRC-aided LVA and CRC-aided SCL decoders involved varying frame lengths K, and list sizes L, giving us an understanding of their operational performance under various conditions.

For frame length $K = 32$ with code rate $R = 1/2$, and testing with both list sizes $L = 1$ and $L = 16$, some noteworthy observations were made.

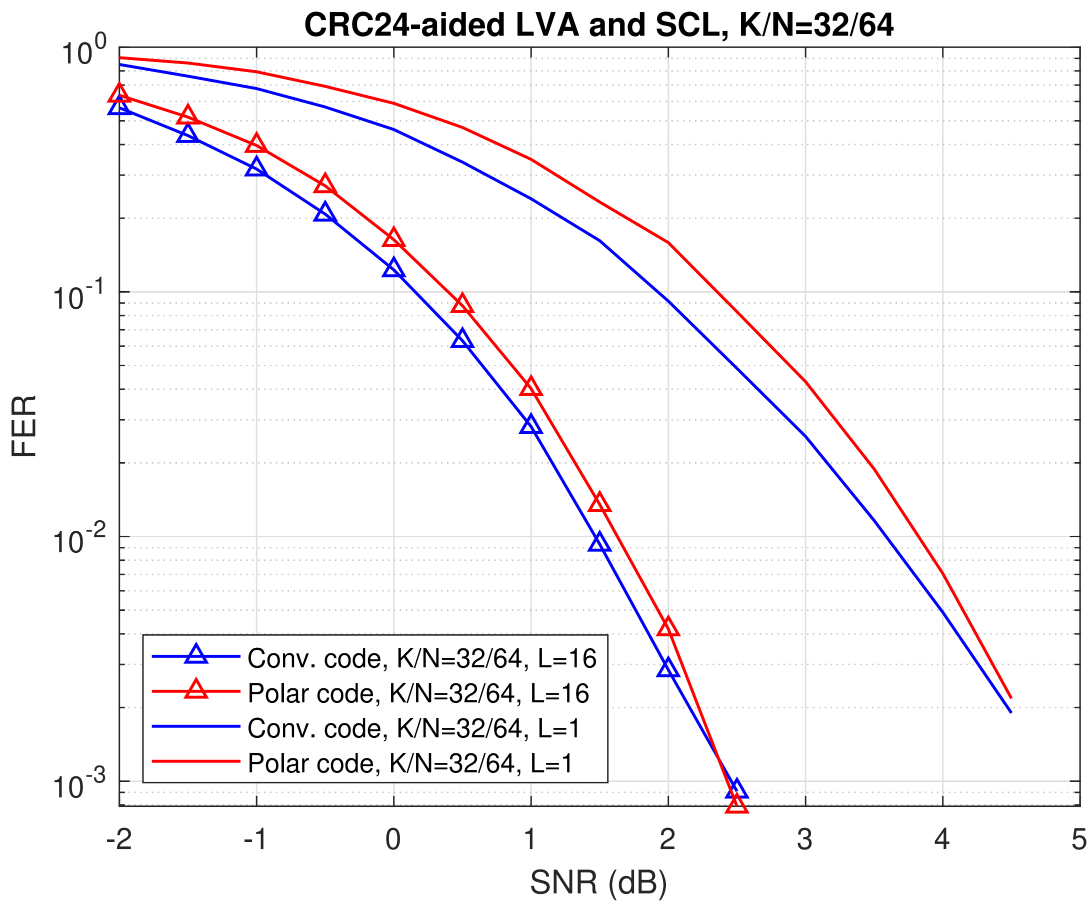


Figure 25: Comparative performance of polar codes and convolutional codes for frame length $K = 32$ with list sizes $L = 1$ and $L = 16$.

For $K = 32$ with $L = 1$ and $L = 16$, both decoders displayed a similar performance in terms of FER. For FER values down to 10^{-3} there is a slightly superior performance for LVA. At the point of FER= 10^{-3} there seems like SCL are beginning to be superior, but there is not sufficient data to tell.

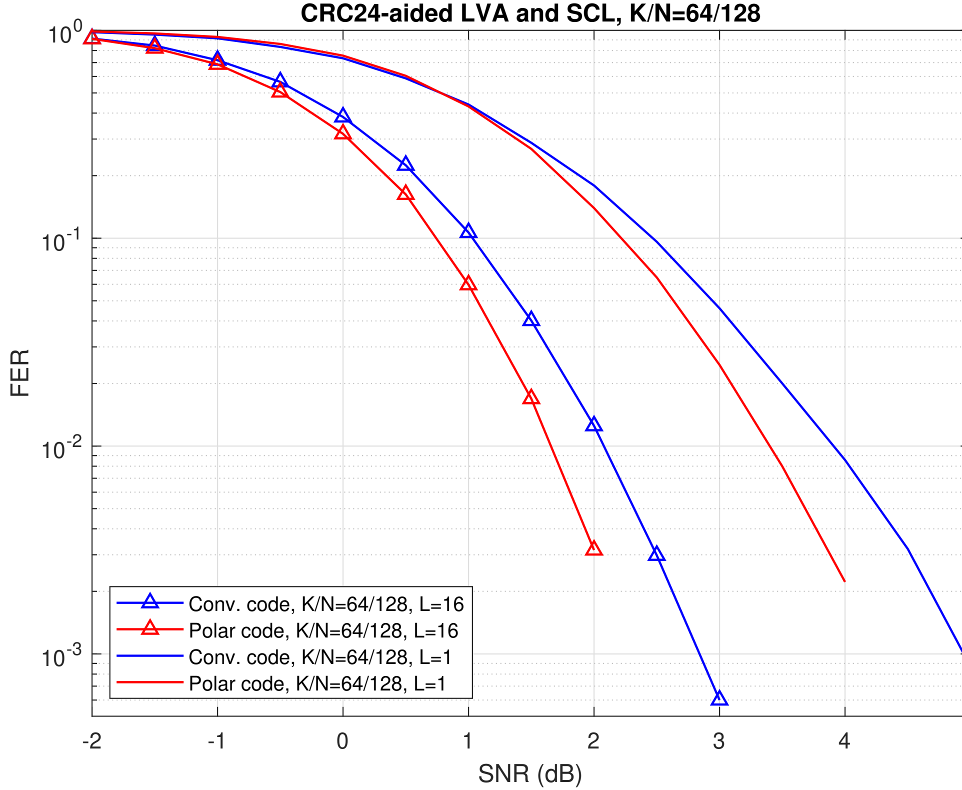


Figure 26: Comparative performance of polar codes and convolutional codes for frame length $K = 64$ with list sizes $L = 1$ and $L = 16$.

For $K = 64$ with $L = 1$ and $L = 16$, we see in Figure 26 that polar code outperforms convolutional code for almost all SNR values. With a near-identical performance in the beginning, for low SNR values, it is clear that SCL gives a steeper curve, resulting in a significant superior performance for larger SNR.

With frame length $K = 64$ and CRC24, we end up with $64 - 24 = 40$ information bits in the in the frame for polar code, and $64 - 24 - 3 = 37$ information bits in the frame for convolutional code. This makes it so the throughput, the amount of information transmitted in a given time, is lower for the convolutionel codes and the comparison is not completely fair.

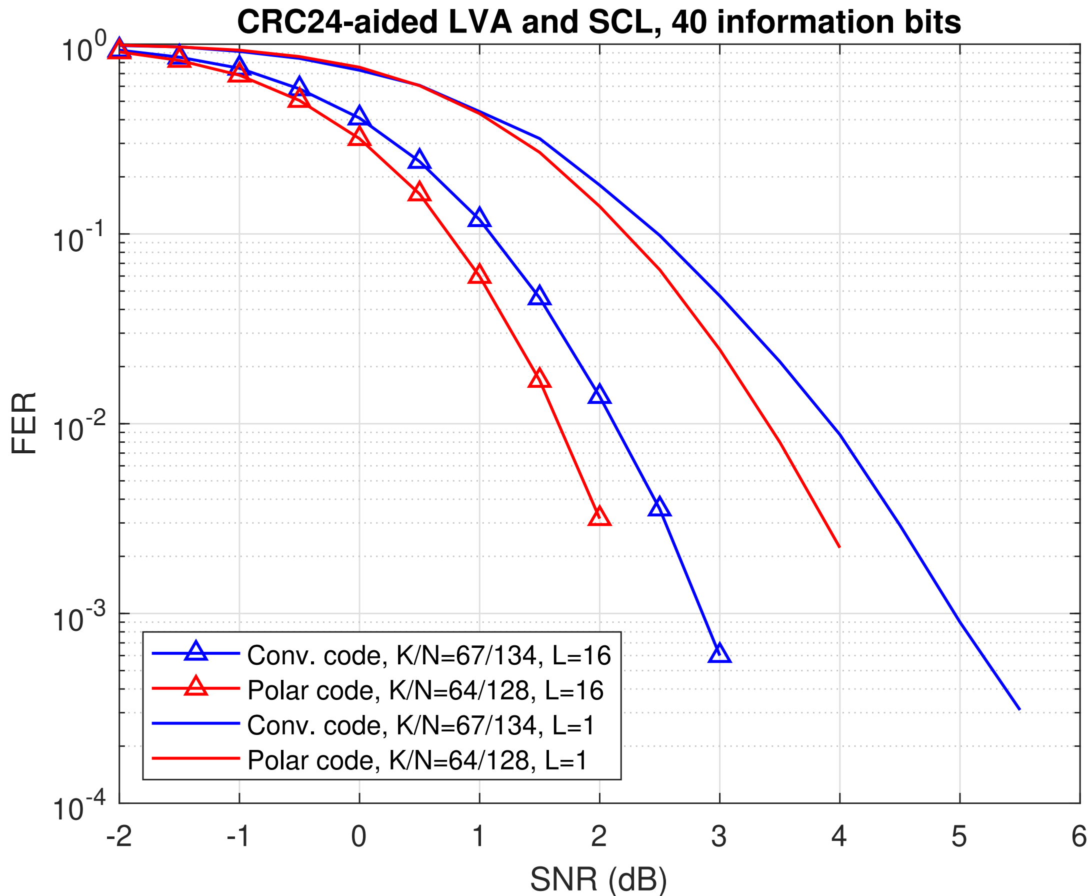


Figure 27: Comparative performance of polar codes and convolutional codes containing 40 information bits with list sizes $L = 1$ and $L = 16$.

In Figure 27 we have increased the frame length K to 67 for convolutional code, so it matches the information bits in the frames of polar codes, which is 40. The results are very similar to the the case of $K = 64$, where polar code outperforms the convolutional code for all SNR values.

3.3 Impact of CRC usage

The role of Cyclic Redundancy Check (CRC) is important in identifying whether a decoded frame is error-free or not. When we use CRC24, the frame's input contains 24 CRC-bits that must correspond with the 24 calculated check value bits at the output. If the frame is error-free, the values will match. However, if the frame contains errors, the values will most likely mismatch, but there is still a chance they match by accident. This is what we refer to as false positive. For CRC24, 1 in every 2^{24} times the outputed frame contain errors, a false positive will occure, resulting

in the frame concidered correctly decoded. For a list containing $L = 16$ decoded frames with error, the probability of CRC24 approving one of the frames as correct is $\frac{L}{2^{24}} = \frac{16}{2^{24}} = 9,5 \cdot 10^{-7}$ which can be considered negligible in our study.

For CRC6, these errors happens more frequent since it contains only 6 bits. For a list containing $L = 16$ decoded frames with error, the probability of CRC6 approving one of the frames as correct is $\frac{16}{2^6} = 0,25$. If $L = 1$ and low SNR where the output is noise, the probability of a false positive is $\frac{1}{2^6} = 0,015625$. This will have a considerable impact on the code.

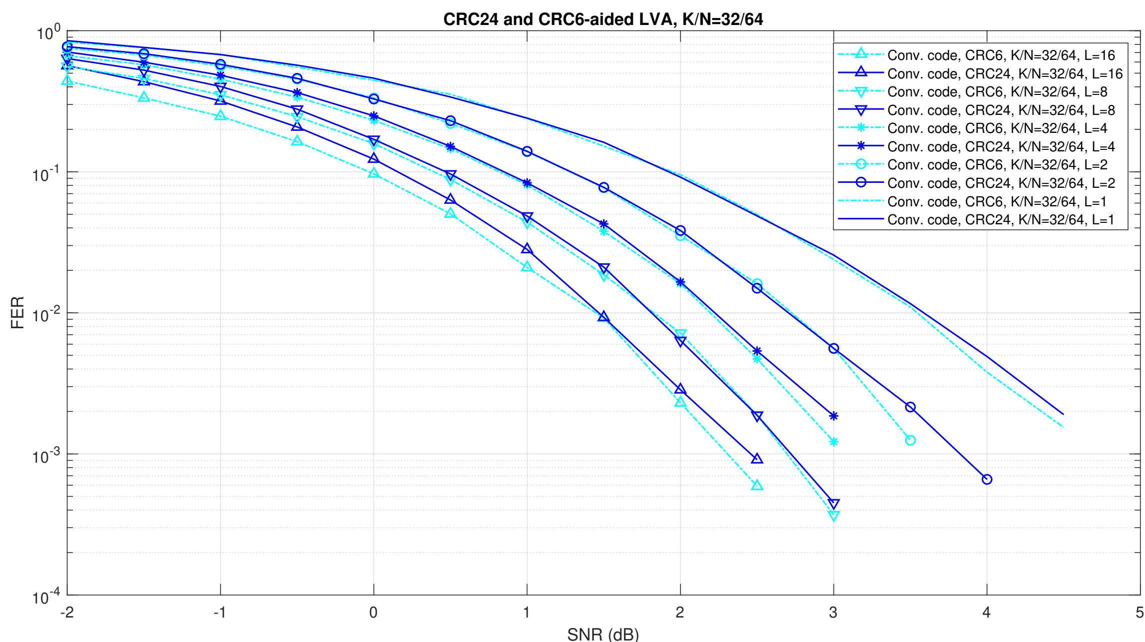


Figure 28: Study of impact of using CRC6 for $K = 32$. The dark blue plots correspond to the plot in Figure 22 using CRC24-aided LVA. The light blue uses CRC6-aided LVA.

As expected, using CRC6 results in a better FER compared to CRC24, seen in Figure 28. This comes from the fact that, similar to CRC24, CRC6 successfully detects all error-free frames, but it also detects false positives, thus giving an overall lower FER.

4 Discussion

The results from the study provide key insights on the performance and reliability of polar codes and convolutional codes, along with the comparison of their corres-

ponding decoders, the SCL and LVA. In addition, the role of CRC variants, CRC24 and CRC6, was studied.

The study showed that polar codes performed better than convolutional codes for frame lengths of $K = 64$ and higher, under the condition of a maintained code rate of 1/2 and using CRC24. This gives polar codes an advantage in scenarios where larger frame sizes than 64 are required, thus indicating that polar codes can enhance data transmission efficiency under such circumstances.

For both polar and convolutional codes, an increase in list size L showed a consistent improvement in performance. This implies that for practical applications, increasing list sizes could potentially offer more reliable data transmission. It is worth noting that although an increasing list size can potentially offer better performance, it also demands more computational resources and memory, thus leading to trade-offs in practical applications. These trade-offs must be carefully managed according to the specific requirements of the application scenario.

For the case of varying block lengths, the study showed that at higher SNR values, polar codes seemed to perform better with an increase in frame length. In contrast, convolutional codes provided better performance for larger frame lengths at a high SNR but worsened performance for low SNR values. This could be attributed to the inherent nature of these codes and their unique characteristics at different SNR levels.

Moreover, the comparison of the decoders revealed that both decoders showed similar performance for a frame length of $K = 32$. However, for a frame length of $K = 64$, the SCL decoder provided a significant advantage in terms of the frame error rate, thus highlighting its superiority for larger frame lengths.

Furthermore, the study highlighted the essential role of CRC in the decoding process. With CRC24, the probability of false positives (wrongly identifying a frame with errors as error-free) is negligible. However, with CRC6, the chance of false positives increases considerably. As such, the choice of CRC variant is a vital consideration in code design. The impact of using CRC can clarely be seen, since in the cases of $L = 1$, the CRC are not being utilized.

It is important to note that while this study provides valuable insights into the per-

formance of these codes and decoders, real-world scenarios could involve additional factors such as channel noise, delay constraints, and computational resources, which could affect their overall performance. These additional factors should be taken into account when implementing these codes in practical applications.

One significant limitation of polar codes, particularly for the Successive Cancellation List (SCL) decoding, lies in the constraint of having frame lengths (K) that are a power of 2. This is due to the nature of polar codes' encoding, which is based on Kronecker products of a 2x2 kernel, discussed in Section 2.3.1. In scenarios where a frame length that is not a power of 2 is required, complexities arise in implementing SCL decoding effectively. To address this, techniques such as puncturing, shortening, or extending are utilized[5]. In puncturing, certain bits are intentionally removed after encoding to reduce the frame length to the required size. Although this introduces the possibility of error propagation, it does allow for flexibility in the frame length. Similarly, shortening involves the allocation of some bits as fixed known bits, thus reducing the effective frame length, while in extending, additional dummy bits are added to make up the frame length. These techniques, while providing solutions, also introduce additional layers of complexity in the encoding and decoding process, possibly affecting performance, and require further research to optimize their implementations. It is also worth noting that these strategies can influence other aspects, such as code rate, error-correcting capabilities, and complexity, which must be considered in the overall performance analysis of polar codes.

In conclusion, this study has successfully provided a comprehensive comparative analysis of polar codes and convolutional codes, along with their corresponding decoders, under varying conditions. The results from the study could provide valuable insights for the design and implementation of efficient and reliable communication systems.

References

[1] *S. Byre* (2023) Polar Code with short block length,

[2] *C. E. Shannon* (1948) A mathematical theory of communication, The Bell System Technical Journal, vol. 27, no. 3, pp. 379–423.

[3] *A. J. Viterbi* (1967) Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm, IEEE Transactions on Information Theory, Vol. IT-13, pp. 260-269.

[4] *N. Seshadri and C. . -E. W. Sundberg* (1994) List Viterbi Decoding Algorithms with Applications, IEEE Transactions on Communications, vol. 42, no. 234, pp. 313-323.

[5] *V. Bioglio C. Condo, I. Land* (2020) Design of Polar Codes in 5G New Radio, IEEE Trans. Inf. Theory.

[6] *I. Tal and A. Vardy* (2015) List Decoding of Polar Codes, IEEE Transactions on Information Theory, vol. 61, no. 5, pp. 2213-2226.

[7] *Arıkan, E.* (2009) Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels, IEEE Trans. Inf. Theory 55, pp. 3051–3073.

[8] *A. K. Singh* (2017) Comprehensive study of error detection by cyclic redundancy check, 2nd International Conference for Convergence in Technology (I2CT), Mumbai, India, 2017, pp. 556-558.

[9] *5G New Radio Polar Coding*, https://www.mathworks.com/help/5g/gs/polar-coding.html#NewRadio5GPolarCodingExample-10, (used: 26.06.2023),

[10] *Core Specification Working Group* (2021) Bluetooth Core Specification,

[11] *R. H. Morelos-Zaragoza* (2002) The Art of Error Correcting Coding, Germany: Wiley.

[12] *V. Bioglio C. Condo, I. Land* (2022) 3GPP TS 38.212,

[13] *H. Vangala, E. Viterbo, Y. Hong* (2015) A Comparative Study of Polar Code Constructions for the AWGN Channel,

Appendix

A CRC-aided LVA implemented in MATLAB

```
function decodedMessages = list_viterbi_mod_3(
    encodedBitMessage, trellis, L)
    % Inputs:
    % encodedBitMessage: Convolutionally encoded bit message
        . 1 x N.
    % trellis: Trellis structure used for encoding
    % L: List size

    [numStates, numTransitions] = size(trellis.nextStates);
        %Number of stages and transitions.
    M = length(encodedBitMessage) / trellis.numInputSymbols;
         % Time instances
    numOutput = log2(trellis.numOutputSymbols); %Bits per
        output

    % Initialization
    phi = ones(L, numStates) * 1000; % Cost array. Stores L
        costs for each state
    for i = 1:numTransitions
        output = trellis.outputs(1, i);
        bin = dec2bin(output, numOutput) == '1';
        cost = sum(xor(encodedBitMessage(1:trellis.
            numInputSymbols), bin));
        nextState = trellis.nextStates(1, i);
        phi(1, nextState + 1) = cost;
    end

    Xi = zeros(L, numStates, M, 2); % Previous state indices
        . Stores the path history for each time instant. The
        4th dimention is used to store r from the paper. This
```

```
        is the ranking of the path.
22    Xi(:, :, 1, 1) = 1;
23    %Xi(:, 3, 1, 1) = 1;
24    for ell = 1:L
25        Xi(ell, :, 1, 2) = ell;
26        %Xi(ell, 3, 1, 2) = ell;
27    end
28
29    % Create stateMap, to store prevState.
30    stateMapRaw = ones(numStates* numTransitions, 3); %
        Stores 3 values for each line in the trellis: [state,
        prevState, Bt]. Bt is the bit telling if the path
        was up (0) or down(1).
31    for i = 1:numStates
32        for j = 1:numTransitions
33            nextState = trellis.nextStates(i, j) + 1;
34            stateMapRaw((i - 1) * numTransitions + j, :) = [
                nextState i j - 1]; %Bt gets right because
                the structure of trellis.nextStates. The
                index of the column tell the movment from a
                state.
35        end
36    end
37    stateMap = sortrows(stateMapRaw);
38
39    % Recursion
40    for t = 2:M
41        newPhi = ones(L, numStates) * 9999; %phi gets
            updated for every iteration over states, so
            storing the new phi here.
42        for i = 1:numStates
43            Z = zeros(L * numTransitions, 1); %Storing all
                the costs comming in to the state i.
44            X = zeros(L * numTransitions, 1); %Storing the
```

```
        previous states for each cost in Z.
45      for j = 1:numTransitions
46          prevState = stateMap((i - 1) *
              numTransitions + j, 2);
47          prevBt = stateMap((i - 1) * numTransitions +
              j, 3); %Bt is the bit telling if the
              path was up (0) or down(1)
48          prevOutput = trellis.outputs(prevState,
              prevBt + 1);
49          prevOutputBin = ones(1, numOutput) * -1;
50          for k = 1:numOutput
51              prevOutputBin(k) = bitand(bitshift(
                  prevOutput, -numOutput + k), 1);
52          end
53
54          prevOutputBin = 1 - 2 * prevOutputBin;
55
56          cost = sum((encodedBitMessage((t-1) *
              trellis.numInputSymbols +1 : (t) *
              trellis.numInputSymbols) - prevOutputBin)
              .^2);
57
58          Z((j - 1) * L + 1:j * L) = phi(:, prevState)
               + cost; %Z will contain first the L best
               (sorted) BM from transition 1, then L
              best from j=2 and so on. This means the
              modulo of the idx would be the
              corrsponding ranking because each L is
              sortet. Will be used later.
59          X((j - 1) * L + 1:j * L) = prevState; %All
              the states related to Z
60      end
61      [zSorted, idxs] = sort(reshape(Z, numTransitions
          * L, 1));
```

```
62            newPhi(:, i) = zSorted(1 : L);
63            for k = 1:L
64                Xi(k, i, t, 1) = X(idxs(k)); %indxs(k)
                     coresponds to the index from the previous
                      Z with blocks of L and L.
65                Xi(k, i, t, 2) = mod(idxs(k) - 1, L) + 1; %
                     modulo with L here because before sort,
                     idxs would be 1,2..L,L+1,L+2..2L, and
                     this is the correct ranking for the L and
                      L sorted values in Z. Finding the
                     ranking within these boxes.
66            end
67        end
68        phi = newPhi;
69    end
70
71    % Path Backtracking.
72    decodedMessages = ones(L, M);
73    for ell = 1:L
74        currentState = 1;
75        currentRank = ell;
76        for time = M:-1:2
77            pState = Xi(currentRank, currentState, time, 1);
78            pRank = Xi(currentRank, currentState, time, 2);
                 %Ranking of the previous best path. This
                  helps the algorithem to distinguish the paths
                  .
79            inputSymbol = find(trellis.nextStates(pState,:)
                 ==currentState-1)-1;
80            decodedMessages(ell, time) = inputSymbol;
81            currentState = pState;
82            currentRank = pRank;
83        end
84        pState = 1;
```

```
85            inputSymbol = find(trellis.nextStates(pState,:)==
                 currentState-1)-1; %This is for the case M=1.
86            decodedMessages(ell, 1) = inputSymbol;
87        end
88 end
```

B Convolutional code simulations

```
1 clear all
2 close all
3
4 % Simulation parameters
5 maxIterations = 100000; %Maximum number of iterations to run
      the simulation
6 maximumFER = 1000; % Maximum number of errors during
     simulation before good enough
7 min_iterations = 5000; % inimum number of errors during
     simulation before good enough
8
9 trellis = poly2trellis(4,[17 13]); %Generation of the
     trellis for the convolutional code. 4,[17 13] is the one
     used in Bluetooth
10 BlockSize_sim = [32 64]; %Number of data bits per block.
11 L_sim = [32 64]; %List size
12
13 % CRC
14 CRClength = 24;
15 crcGenerator = comm.CRCGenerator('Polynomial',[24 10 9 6 4
     3 1 0]); %Same polynomial used in Bluetooth
16 crcDetector = comm.CRCDetector('Polynomial',[24 10 9 6 4 3
     1 0]); %[24 23 21 20 17 15 13 12 8 4 2 1 0] 24c, used in
     5G standards. Not used here. Here we use: [24 10 9 6 4 3
     1 0] or [6 5 0]. CRC24 - CRC6
17
18 out=[];
```

```
19 fileData = fopen('data.txt', 'w');
20
21 for BlockSize = BlockSize_sim
22     fprintf(fileData, '\nConvolutional code\nBlockSize %f\n'
       , BlockSize);
23     for L = L_sim
24         fprintf(fileData, 'L = %f\n', L);
25
26         errStats = zeros(L, 1);
27         crc_undetected_miss = 0;
28
29         for SNRdB = -2:0.5:10
30             SNR = 10.^(SNRdB/10);
31             noiseVar = 1./SNR; %sigma^2
32             frameErrors = 0;
33             frameErrorsCRC = 0;
34             frameErrorsCRCmiss = 0;
35
36             for i = 1:maxIterations
37                 %RNG
38                 t=clock();%reset clock
39                 seed=t(6) * 1000; % Seed with the second
                       part of the clock array.
40                 rng(seed);
41
42                 %Data
43                 msg = randi([0 1], BlockSize - CRClength -
                       3, 1);
44                 dataCRC = crcGenerator(msg);
45                 data = [dataCRC;0;0;0];
46                 encodedData = convenc(data,trellis);
47
48                 %Channel
49                 modSignal = 1 - 2 * encodedData; %BPSK
```

```
        modulation
50          receivedSignal = modSignal + sqrt(noiseVar)
                * randn(length(modSignal),1);% Add White
                Gaussian noise
51          demodSignal = receivedSignal * 2 / noiseVar;
                 %Soft demodulation
52
53          %Viterbi decoding
54          decodedList_mod = list_viterbi_mod_3(
                demodSignal.', trellis, L);
55
56          %Error calculating
57          ginieFlag = false;
58          for ell = 1:L
59              [msg_decoded,err] = crcDetector(
                    decodedList_mod(ell,1:end-3).');
60              if ~err %If correct crc
61                  if ~any(msg_decoded~=msg)
62                      errStats(ell, 1) = errStats(ell,
                            1) + 1; %true positive
63                  else
64                      frameErrorsCRCmiss =
                            frameErrorsCRCmiss + 1;%
                            false positive
65                  end
66                  break
67              end
68
69              if decodedList_mod(ell,1:end-3-CRClength
                    ) == msg %If not any misstakes in the
                    msg:
70                  ginieFlag = true;
71              end
72
```

```
73                  if ell == L %If all have failed the crc.
74                      frameErrors = frameErrors + 1; %
                            negative
75                      if ginieFlag
76                          frameErrorsCRC = frameErrorsCRC
                                + 1; %false negative %This
                                does not include the
                                errorfree msg whith errors
                                only in the crc-bits
77                      end
78                  end
79              end
80
81              %Good enough if we reach high enough errors
82              if(frameErrors>maximumFER  && i>
                    min_iterations)
83                  break
84              end
85          end
86          fprintf(fileData, '%f, CRCfail: %f, false neg: %
                f, false pos: %f, i: %f\n', SNRdB,
                frameErrors/i, frameErrorsCRC/i,
                frameErrorsCRCmiss/i, i);
87          out=[out;SNRdB,frameErrors/i, frameErrorsCRC/i,
                frameErrorsCRCmiss/i]
88
89          %Simulate until frameErrors<100
90          if(frameErrors < 100)
91              break
92          end
93      end
94  end
95  end
96  fclose(fileData);
```

```
97
98 disp(errStats.');
99 errStats = errStats./sum(errStats);
100 disp(errStats.');
101
102 %Plot frame error rate (FER) Vs SNR
103 semilogy(out(:,1),out(:,2)),grid,xlabel('SNR (dB)'),ylabel('
       FER');
```

C CRC-aided SCL, modifications from the MATLAB function nrPolarDecode

```
  1 function out = nrPolarDecode_new(in,K,E,L,varargin)                      x function out = nrPolarDecode(in,K,E,L,varargin)                          1
  2 %nrPolarDecode Polar decode                                              . %nrPolarDecode Polar decode                                              2
  3 %   DECBITS = nrPolarDecode(REC,K,E,L) decodes the rate-recovered input, . %   DECBITS = nrPolarDecode(REC,K,E,L) decodes the rate-recovered input, 3
  4 %   REC, for a (N,K) Polar code, using a CRC-aided successive-cancellation . %   REC, for a (N,K) Polar code, using a CRC-aided successive-cancellation 4
                                        [174 unmodified lines hidden]
179                                                                          .                                                                          179
180     % A restriction for uplink (for PC-Polar)                            .     % A restriction for uplink (for PC-Polar)                            180
181     % length K must be greater than 17 and less than or equal to 25      .     % length K must be greater than 17 and less than or equal to 25      181
182     %coder.internal.errorIf( nMax==10 && ~iIL && crcLen==6 && ~padCRC && ... x     coder.internal.errorIf( nMax==10 && ~iIL && crcLen==6 && ~padCRC && ... 182
183     %   (K < 18 || K > 25), 'nr5g:nrPolar:InvalidKULPC');                x         (K < 18 || K > 25), 'nr5g:nrPolar:InvalidKULPC');                183
184                                                                          .                                                                          184
185     % Validate rate-matched output length which must be less than or equal .     % Validate rate-matched output length which must be less than or equal 185
186     % to 8192 and greater than K                                         .     % to 8192 and greater than K                                         186
                                        [6 unmodified lines hidden]
193                                                                          .                                                                          193
194 end                                                                      . end                                                                      194
195                                                                          .                                                                          195
196 function dec_array = lclPolarDecode(in,F,L,iIL,crcLen,padCRC,rnti,qPC)   x function dec = lclPolarDecode(in,F,L,iIL,crcLen,padCRC,rnti,qPC)         196
```

Figure 29: Declaring the new vector of L codewords instead of only the best one.

```
317             crcDetector  = comm.CRCDetector('Polynomial',[24 10 9 6 4 3 1 0]); % <                                                          -
318             %[~, errFlag] = nrCRCDecode(padCRCMsg,polyStr,rnti);               x             [~, errFlag] = nrCRCDecode(padCRCMsg,polyStr,rnti);  317
319             [~, errFlag] = crcDetector(padCRCMsg);                             <                                                          -
320             if errFlag      % ~0 => fail                                       .             if errFlag      % ~0 => fail                 318
321                 continue;   % move to next path                                .                 continue;   % move to next path          319
322             end                                                                .             end                                          320
```

Figure 30: Changing the CRC to the one used in Bluetooth Specifications.

```
345                                                                          .                                                                          343
346     % Get decoded bits                                                   .     % Get decoded bits                                                   344
347     dec_array = zeros(L, K);                                             <                                                                          -
348                                                                          <                                                                          -
349     for pathIdx = 1:L                                                    <                                                                          -
350         [sc, sttStr] = getArrayPtrC(sttStr,arrayPtrLLR,arrayPtrC,mplus1, ... x     [sc, sttStr] = getArrayPtrC(sttStr,arrayPtrLLR,arrayPtrC,mplus1, ...   345
351                         pathIdx);                                        x                         pathIdx1);                                       346
352         decCW = sttStr.savedCWs(:,sc);      % N, with frozen bits        x     decCW = sttStr.savedCWs(:,sc);      % N, with frozen bits            347
353         dec = decCW(F==0,1);                % K, info + nPC bits only    x     dec = decCW(F==0,1);                % K, info + nPC bits only        348
354         dec(piInterl+1) = dec;                                           x     dec(piInterl+1) = dec;              % Deinterleave output, K+nPC     349
355                                                                          x                                                                          350
356         dec_array(pathIdx,:) = dec;                                      <                                                                          -
357     end                                                                  x end                                                                      351
358                                                                          x                                                                          352
359     tmp = dec_array(1,:);                                                <                                                                          -
360     dec_array(1,:) = dec_array(pathIdx1,:);                              <                                                                          -
361     dec_array(pathIdx1,:) = tmp;                                         <                                                                          -
362                                                                          <                                                                          -
363 end                                                                      <                                                                          -
364                                                                          <                                                                          -
```

Figure 31: Making the new vector of L codewords with the best one at position 1.

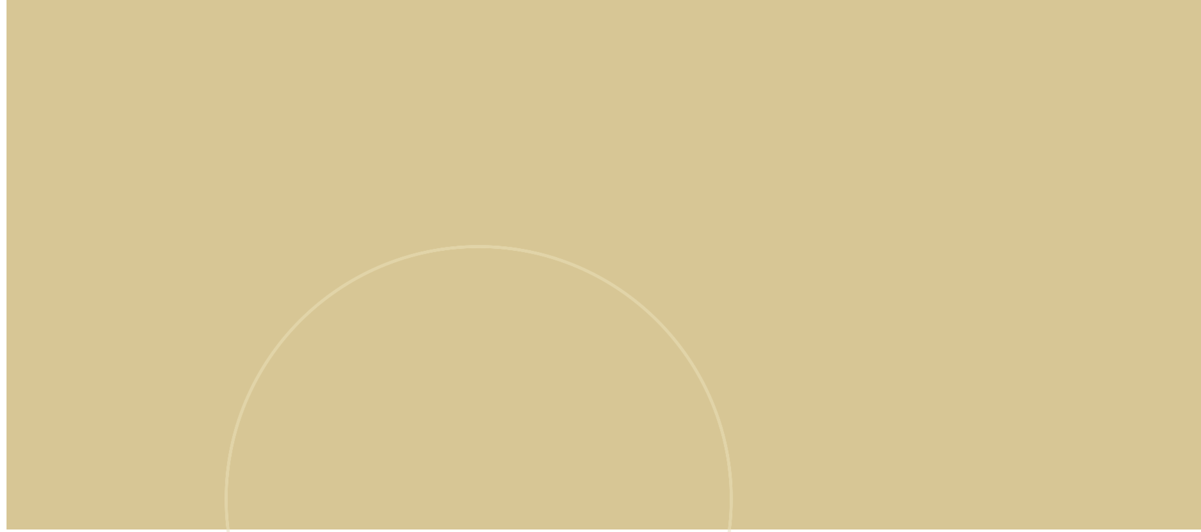
D Polar code simulations

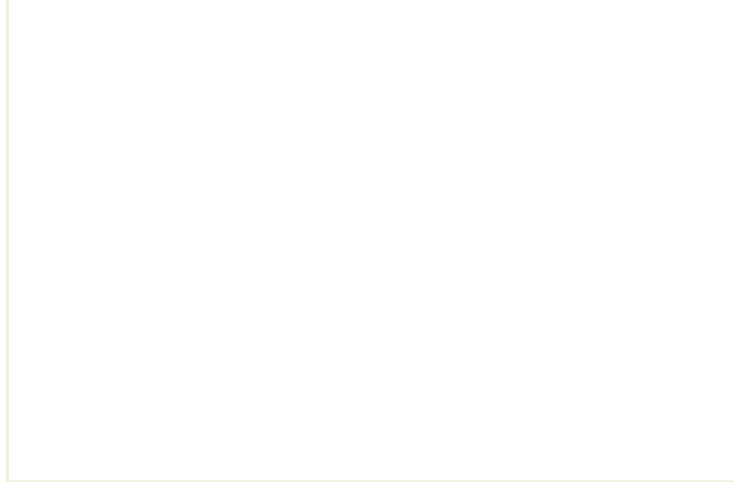
```
clear all
close all

% Simulation parameters
K_sim = [32 64]; %40 320 2040. 32 64 done
L_sim = [16 32 64];
numFrames = 10000; % Number of frames to simulate
min_error_count = 1000;
min_iterations = 5000;
s = rng(100); % Seed the RNG for repeatability

% CRC
crcLen = 24; %6;
crcGenerator  = comm.CRCGenerator('Polynomial',[24 10 9 6 4
    3 1 0]); %Same polynomial used in Bluetooth
%[24 23 21 20 17 15 13 12 8 4 2 1 0] 24c, used in 5G
    standards. Not used here
%[24 10 9 6 4 3 1 0]  - [6 5 0]

out=[];
fileData = fopen('dataPolar.txt', 'w');

for K = K_sim
    E = 2 * K;
    fprintf(fileData, '\n\n K = %f\n', K);
    for L = L_sim
        fprintf(fileData, 'L = %f\n', L);

        for SNRdB=-2:0.5:15
            SNR = 10.^(SNRdB/10);
            noiseVar = 1./SNR;
            frameErrors = 0;
            frameErrorsCRC = 0;
```

```
32          frameErrorsCRCmiss = 0;
33
34          for i = 1:numFrames
35              %RNG
36              t=clock();
37              seed=t(6) * 1000; % Seed with the second
                   part of the clock array.
38              rng(seed);
39
40              %Data
41              msg = randi([0 1],K-crcLen,1);% Generate a
                   random message
42              msgcrc = crcGenerator(msg);% Attach CRC
43
44              % Polar encode
45              encOut = nrPolarEncode(msgcrc,E,10,false); %
                   nMax, inputIL
46
47              %Channel
48              modOut = 1 - 2 * encOut; %BPSK modulation
49              rSig = modOut + sqrt(noiseVar) * randn(
                   length(modOut),1); % Add White Gaussian
                   noise
50              rxLLR = rSig * 2 / noiseVar;% Soft
                   demodulate
51
52              % Polar decode
53              decBits = nrPolarDecode_new(rxLLR,K,E,L,10,
                   false,crcLen);%false
54
55              % Compare msg and decoded bits
56              if any(decBits(1,K-crcLen+1:end).'~=msgcrc(K
                   -crcLen+1:end)) % the first on the list
                   will allways have passed the crc, if
```

```
                there existt a codeword on the list that
                have passed it.
57              frameErrors = frameErrors + 1; %Failed
                   CRC. (msg may be correct). (If CRC
                   was correct, the packet is allways
                   correct because CRC24)
58
59              ginieFlag = false;
60              for ell = 1:L
61                  if decBits(ell,1:K-crcLen).' == msg
                      %If not any misstakes in the msg:
62                      ginieFlag = true;
63                  end
64              end
65
66              if ginieFlag
67                  frameErrorsCRC = frameErrorsCRC + 1;
68              end
69          else
70              if any(decBits(1,1:K-crcLen).' ~= msg)
71                  frameErrorsCRCmiss =
                      frameErrorsCRCmiss + 1; %false
                      positive
72              end
73
74          end
75
76          if(frameErrors>min_error_count && i>
                min_iterations)
77              break
78          end
79      end
80
81      fprintf(fileData, '%f, CRCfail: %f, msgFail: %f,
```

```
                    false_pos: %f , i: %f\n', SNRdB, frameErrors
                    /i, frameErrorsCRC/i, frameErrorsCRCmiss/i, i
                    );
82              out=[out;SNRdB,frameErrors/i, frameErrorsCRC/i,
                    frameErrorsCRCmiss/i]
83              if(frameErrors < 100)
84                  break
85              end
86              rng(s);% Restore RNG
87          end
88      end
89  end
90  semilogy(out(:,1),out(:,2),'LineWidth',1.5),grid,xlabel('SNR
        (dB)'),ylabel('FER'),title('SCL')
```





Norwegian University of
Science and Technology