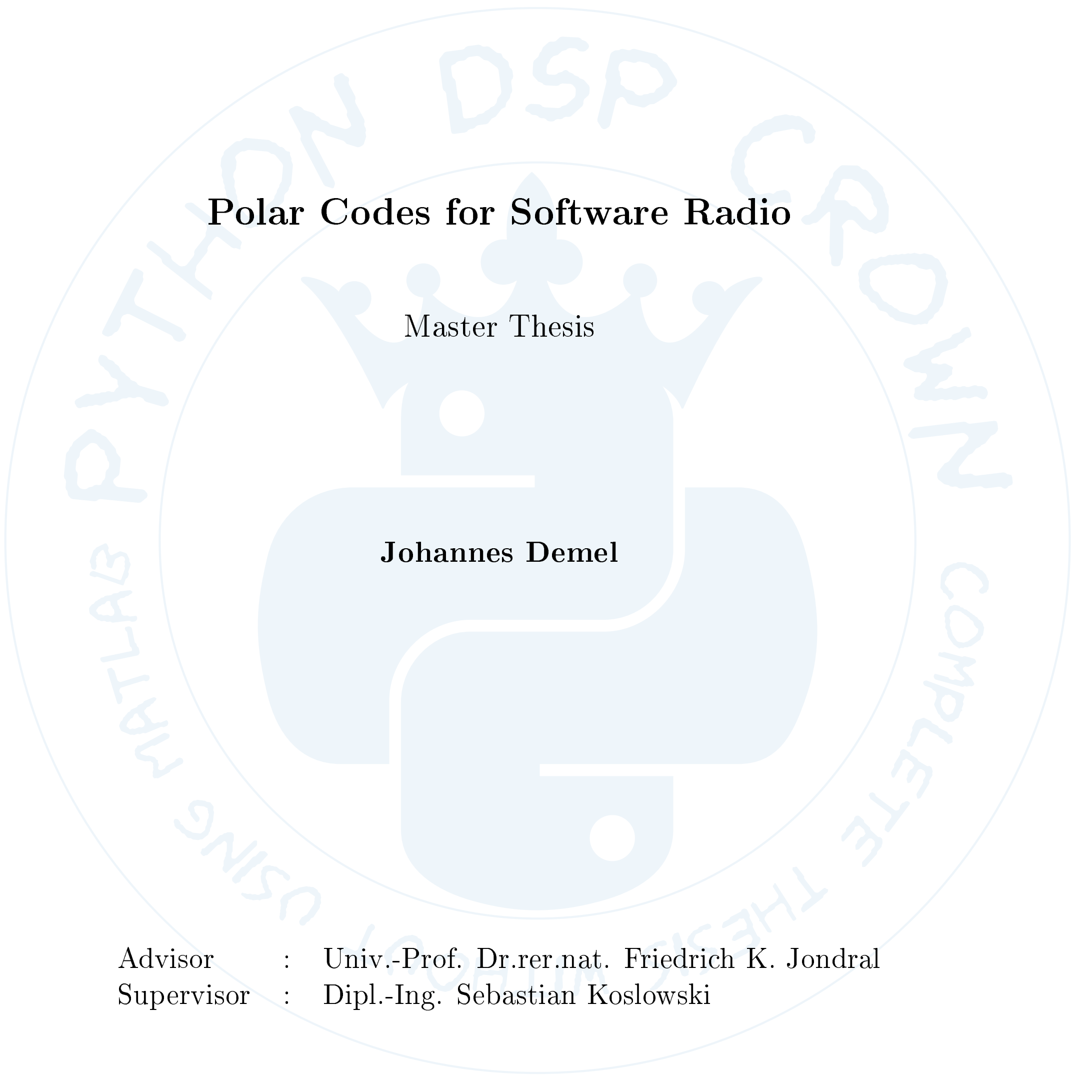
Karlsruhe Institute of Technology
Communications Engineering Lab
Univ.-Prof. Dr.rer.nat. Friedrich K. Jondral



Polar Codes for Software Radio

Master Thesis

Johannes Demel

Advisor	:	Univ.-Prof. Dr.rer.nat. Friedrich K. Jondral
Supervisor	:	Dipl.-Ing. Sebastian Koslowski

Start date	:	8th June 2015
End date	:	8th December 2015

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und unter Beachtung der Satzung des Karlsruher Instituts für Technologie (KIT) zur Sicherung guter wissenschaftlicher Praxis in der aktuellen Fassung angefertigt habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und wörtlich oder inhaltlich übernommene Stellen als solche kenntlich gemacht.

Karlsruhe, den 8.12.2015

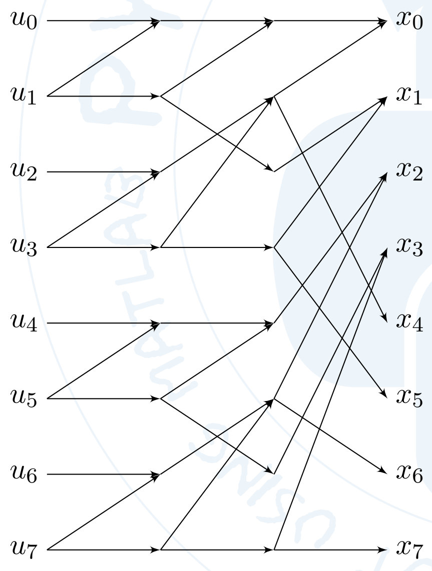
Johannes Demel

Abstract

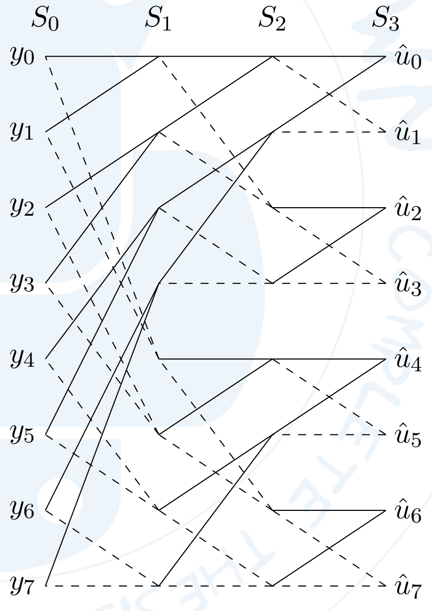
Polar codes are the first codes to asymptotically achieve channel capacity with low complexity encoders and decoders. They were first introduced by Erdal Arikan in 2009 [Ari09].

Channel coding has always been a challenging task because it draws a lot of resources, especially in software implementations. Software Radio is getting more prominent because it offers several advantages among which are higher flexibility and better maintainability. Future radio systems are aimed at being run on virtualized servers instead of dedicated hardware in base stations [RMP+15]. Polar codes may be a promising candidate for future radio systems if they can be implemented efficiently in software.

In this thesis the theory behind polar codes and a polar code implementation in GNU Radio is presented. This implementation is then evaluated regarding parameterization options and their impact on error correction performance. The evaluation includes a comparison to state-of-the-art Low-Density Parity-Check (LDPC) codes.



(a) 8 bit polar encoder



(b) 8 bit polar decoder

Figure 0.1.: Polar code encoding and decoding

Polar codes are optimized for symmetric BDMCs (symmetric-BDMCs) W. The underlying channel W is combined into a vector channel W^N which represents N uses of the underlying channel. A codeword x^N is transmitted over this vector channel W^N to a receiver which receives y^N. The most common underlying channel models are the Binary Erasure Channel (BEC), the Binary Symmetric Channel (BSC) and the Additive White Gaussian Noise (AWGN) channel.

Polar codes add redundant information by freezing certain bits in a sourceword u^N. These frozen bit positions are known to encoder and decoder along with their values. Channel construction determines the position of frozen bits by calculating the channel capacities for all synthetic channels which result from the channel polarization effect. A synthetic channel is defined by transmitting a bit in a sourceword from u_i to $\hat{u}_i$. The k synthetic channels with highest channel capacity are chosen for information bits and all others are used for frozen bits.

The polar encoder is shown in Fig. 0.1a. It has three encoding stages from left to right. The first stage combines four times two bits into a two bit vector. A node with two arrows ending in it adds bits over GF(2) otherwise the bit value is copied. This two bit process is referred to as a butterfly which constitutes polar codes of bigger block size. In the second stage this process is combined with interleaving of two bit vectors into a four bit vector. The third stage extends this process to an eight bit interleaving. Every stage doubles the size of a polar code. Generally a polar code has a block size of $N = 2^m, m \geq 0, m \in \mathbb{N}$.

The decoder employs a Successive Cancellation (SC) decoder strategy. Fig. 0.1b shows the decoder graph. The decoder uses Log Likelihood Ratios (LLRs) and thus the LLRs for the received vector y^8 are obtained. The decoding process starts with $\hat{u}_0$ on the right. The decoder recurses through the stages S along the edges from $\hat{u}_0$ to stage S_1 where it uses the LLRs from stage S_0 to update the LLRs in the upper half of the stage. Then the first two LLRs in stage S_1 are updated and then the first LLR in stage S_3. This LLR is then used to obtain a bit decision for $\hat{u}_0$. A decoder discards a bit decision for frozen bits and uses the frozen bit value instead regardless of the calculated LLR. $\hat{u}_0$ is used along with the first two LLRs in stage S_2 to obtain the next LLR in stage S_3.

Two different update rules exist which are indicated by solid and dashed lines respectively. Solid lines use the LLRs from stage S_n to update the LLRs in stage S_{n+1} with

$$f(l_a, l_b) = \operatorname{sign}(l_a) \operatorname{sign}(l_b) \min\{|l_a|, |l_b|\}. \tag{0.1}$$

The upper solid line indicates the source of l_a for a node. Dashed lines indicate that the previously estimated bits are used to update the LLRs in the next stage with

$$g(l_a, l_b, \hat{u}) = (-1)^{\hat{u}} l_a + l_b \tag{0.2}$$

The decoder in Fig. 0.1b performs a bit decision every time the LLR in stage S_3 is available.

The encoder and decoder define how channel construction is performed because of the combination of bits and their resulting transition probabilities. Channel construction can be interpreted as a recursive tree transformation shown in Fig. 0.2. The underlying channel W is combined into two synthetic channels $W_2^{(0)}$ and $W_2^{(1)}$. This process recurses through m iterations in order to construct a polar code of block size $N = 2^m$. The recursive channel construction tree shows the transformation from $W_2^{(i)}$ to $W_4^{(i)}$ to $W_8^{(i)}$. This transformation continues until the synthetic channels $W_N^{(i)}$ are calculated. Each synthetic channel is classified by its channel capacity. The capacity sum of all N channels equals the accumulated capacity of N channel uses of the underlying channel W.

The channel polarization effect is illustrated in Fig. 0.3. Fig. 0.3a shows the capacities of the synthetic channels for a polar code of block size $N = 2^9 = 512$. Most synthetic channel capacities tend to either 0 or 1. This is the channel polarization effect which grows stronger for bigger block sizes. Fig. 0.3b shows how far the synthetic channel capacities are from

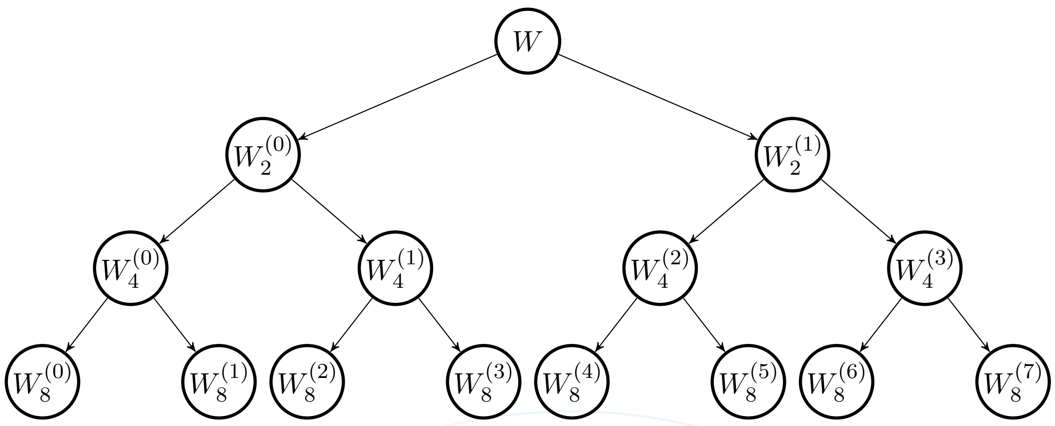
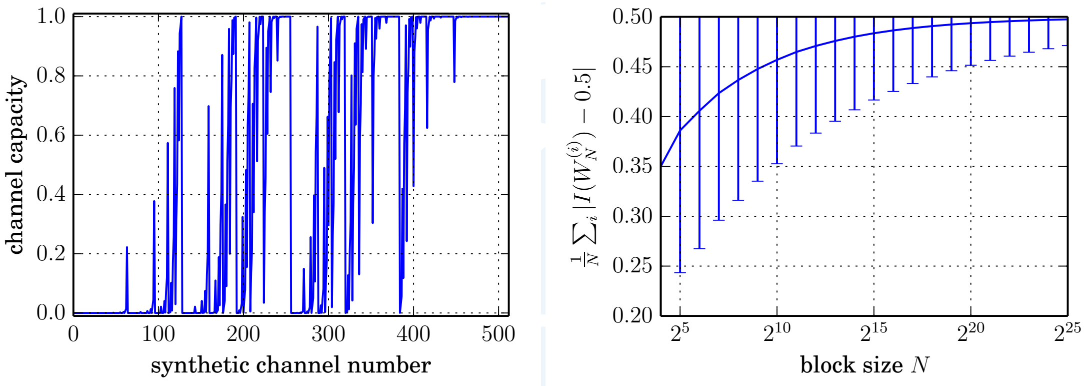


Figure 0.2.: recursive polar channel construction



(a) synthetic channel capacities for BEC, $N = 512$, dSNR $-1.59dB$

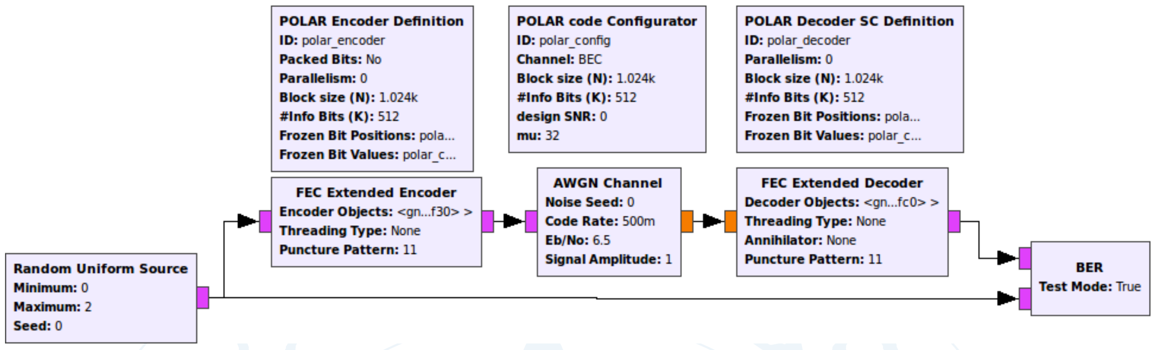
(b) average synthetic channel capacity distance from the underlying channel capacity 0.5

Figure 0.3.: Polarization effect

the capacity of the underlying channel. The chosen underlying channel has a capacity of 0.5 and the average distance tends towards 0.5 which indicates that most channels tend to be either perfect or noise-only channels. This effect motivates how the frozen bit positions are chosen. The k channels with highest capacity are chosen for information bits and the $N - k$ other channels carry frozen bits.

The polar encoder and the SC decoder are used in the proof that polar codes asymptotically achieve Shannon channel capacity. It is important to investigate how polar codes perform when the block size N does not tend towards infinity as opposed to the proof where Shannon capacity is achieved for an infinite block size. An implementation for the software radio framework GNU Radio offers the possibility to evaluate polar codes. Error correction performance can be observed for different configurations which includes different block sizes, code rates, underlying channels and different channel construction algorithms.

Software Radio systems often exhibit a bottleneck at the channel code decoder. Polar codes offer encoders and decoders which have a complexity of $\mathcal{O}(N \log N)$. Polar codes encoders and decoders can be optimized to yield high throughput on different platforms. Besides Bit-Error-Rate (BER) simulations, also throughput measurements can be conducted. GNU Radio's subproject Vector-Optimized Library of Kernels (VOLK) offers a common interface to hardware specific Single Instruction Multiple Data (SIMD) extensions. This enables the implementation of optimized polar encoders and decoders without the need to take care of the actual hardware on different machines. New VOLK kernels were implemented in order to leverage the available hardware and increase throughput via vectorization.


POLAR Encoder Definition
ID: polar_encoder
Packed Bits: No
Parallelism: 0
Block size (N): 1.024k
#Info Bits (K): 512
Frozen Bit Positions: pola...
Frozen Bit Values: polar_c...

POLAR code Configurator
ID: polar_config
Channel: BEC
Block size (N): 1.024k
#Info Bits (K): 512
design SNR: 0
mu: 32

POLAR Decoder SC Definition
ID: polar_decoder
Parallelism: 0
Block size (N): 1.024k
#Info Bits (K): 512
Frozen Bit Positions: pola...
Frozen Bit Values: polar_c...

FEC Extended Encoder
Encoder Objects: <gn...f30> >
Threading Type: None
Puncture Pattern: 11

AWGN Channel
Noise Seed: 0
Code Rate: 500m
Eb/No: 6.5
Signal Amplitude: 1

FEC Extended Decoder
Decoder Objects: <gn...fc0> >
Threading Type: None
Annihilator: None
Puncture Pattern: 11

Random Uniform Source
Minimum: 0
Maximum: 2
Seed: 0

BER
Test Mode: True

Figure 0.4.: GNU Radio simulation environment

GNU Radio offers the FECAPI which promises to separate the implementation of encoders and decoders form their integration into GNU Radio flowgraphs. The implemented encoder and decoder are shown in a stream-based flowgraph in Fig. 0.4 which is used for BER measurements. FECAPI offers encoder and decoder skeleton blocks which expect encoder and decoder variables to be dropped-in respectively. Other supported flowgraph types are tagged stream-based and message-based flowgraphs.

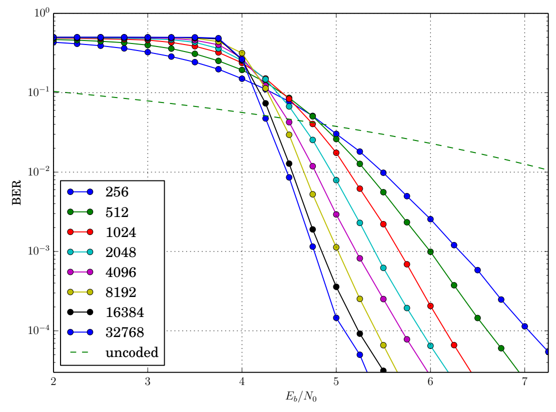
The integration of polar codes is supported by a polar code configurator which deals with channel construction and makes the results available in order to configure the encoders and decoders. The implementation is aimed at seamless integration into GNU Radio's *gr-fec* module where the FECAPI resides as well. This polar code implementation is already merged into mainline GNU Radio and readily available for anyone to use.

Simulation results for several BER simulations are presented in Fig. 0.5. The BER curves in Fig. 0.5a exhibit steeper slopes for bigger block sizes. This results from the stronger channel polarization effect. A bigger block size results in higher latency because all values of a codeword must be received before the decoder can start the decoding process. Regarding latency it is desirable to use shorter block sizes. A real system has to trade-off error correction performance with latency imposed by the need that the whole codeword needs to be received before the decoder can start a decoder pass.

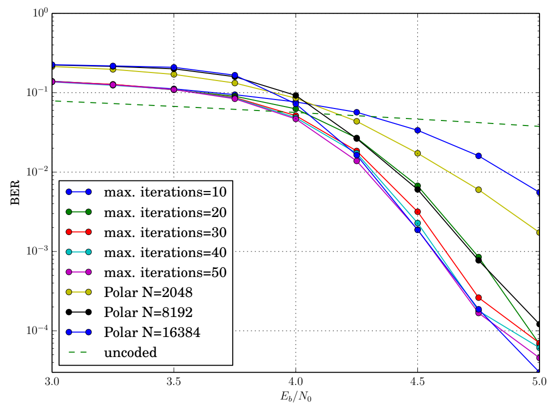
Polar codes compete with LDPC codes for application in future radio systems. This requires that error correction performance is comparable. Fig. 0.5b shows a comparison between a WiMAX standard LDPC (2304, 1152) code and polar codes of different size with code rate $R = 0.5$. LDPC codes are decoded iteratively with Belief Propagation (BP) where the maximum number of iterations determines when the decoder stops trying to correctly decode a received codeword. It can be observed that the polar code with shortest block size $N = 2048$ can compete with the LDPC code for low iterations.

Throughput measurements for a polar code of block size $N = 2^{11} = 2048$ were facilitated. The fastest optimized polar encoder sustained 550.4 Mbit/s throughput with encoded bits.

While the fastest polar decoder achieves 7.18 Mbit/s throughput for encoded bits. All encoders and decoders run on a single core on a system with Ubuntu 14.04.3 64bit with an Intel Core i7-3630QM and 16 GB RAM. The fact that encoder and decoder run on a single core frees up resources for other operations needed in a radio system.



(a) polar code block size comparison



(b) Polar code and LDPC code comparison

Figure 0.5.: BER simulation results

Polar codes use the channel polarization effect to obtain codes which asymptotically achieve Shannon channel capacity. The asymptotically capacity achieving property yields excellent error correction performance in feasible real systems which may compete with state-of-the-art LDPC codes. The low complexity encoders and decoders enable the use of polar codes in the software radio context and ease the channel coding bottleneck which many current software radio systems experience. An open source implementation of polar codes is available in GNU Radio for experiments and usage in software radio systems. Newly designed radio systems thus have an extra option for channel coding to choose from.

Contents

1. Introduction

Modern communication systems require ever higher data rates to serve all potential users while resources to do so are scarce. Even more efficient resource usage is a key to achieve higher data rates. All components of a radio are subject to continuous optimizations in order to deliver the data rates that are required for current and future applications.

Software Radio is a technique which offers multiple advantages. Among the advantages are higher flexibility and better maintainability compared to radio systems which are implemented in hardware. During development of a communication system developers are enabled to rapidly test new algorithms and concepts instead of rebuilding a whole hardware system. Components of a software radio system are easily exchangeable and may be reused in different systems which improves maintainability and cuts down on development time.

Hardware resources can be allocated dynamically in order to use them more efficiently. Development for future radio systems aims at virtualization of ever more components in order to use resources more efficiently. This implies that more functionality is moved from dedicated hardware into software which runs on General Purpose Processors (GPPs), [RMP$^+$15]. Furthermore, system deployment is simplified and improvements are shipped more easily without the need to install new hardware. Again, this allows bugs to be fixed faster and further improves maintainability.

Data transmissions over noisy channels are error-prone. Those errors should be detected and corrected if possible in order to improve reliability of data transmissions. The objective of channel coding is to improve reliability of such a transmission while still transmitting as much data as possible. Shannon introduced the channel capacity theorem in 1949 which provides an upper limit on transmissions over noisy channels, [Sha49]. While it is theoretically known how to achieve this limit, practical systems fall short. The proposed optimal Maximum Likelihood (ML) decoder for a capacity achieving code is NP-complete and thus not usable in a real system. Thus, research is focused on finding codes that achieve this limit while using lower complexity decoders.

Polar codes are the first asymptotically capacity achieving codes with low complexity encoders and decoders. They exploit the channel polarization effect in order to efficiently transmit data. Their theoretical properties need to be tested for practical usage in order to evaluate them for future use in communication systems. Several improvements to the original polar code were proposed which improve on several parts of the code.

A fast polar code implementation for software radio enables real-world experiments in order to verify their capabilities. A software implementation enables tests in different systems. This implementation should be optimized for speed in order to test polar codes in state-of-the-art communication systems. GNU Radio is chosen for implementation because it is a widely used system which provides the infrastructure to use polar codes in a radio system. GNU Radio users benefit from a polar code implementation because they have another option for channel coding and also feedback from users enables incremental improvements to the implementation. A high throughput polar code implementation may be achieved by implementing core functions of the polar encoder and decoders algorithms in

GNU Radio's VOLK subproject. VOLK can switch between generic and hardware specific optimized implementations of a common function transparently.

Polar codes promise to asymptotically achieve channel capacity. It is, however, of major interest to evaluate their performance under practical conditions. Design parameters for polar codes need to be chosen carefully in order to achieve good performance in terms of BER. The polar code decoder promises low complexity. It is thus interesting to evaluate its throughput depending on chosen parameters and in comparison to available channel code implementations.

This thesis is structured as follows. In Chapter 2 an introduction to notation and the system model is given. Chapter 3 introduces polar codes, their properties and how they are constructed. The implementation in GNU Radio is presented in Chapter 4. A performance evaluation is discussed in Chapter 5. This thesis is concluded with Chapter 6 where results are discussed and a future outlook is given.

2. System model

Polar codes are defined for a specific system model. The objective of this chapter is to introduce the key concepts. Notations are introduced and important terms are revisited in order to refer to them.

2.1. Key channel coding concepts

The system model used throughout this thesis follows the remarks in [RU08] and [Ari09]. It is intended to define the domain for which polar codes are developed.

The objective of channel coding is to transmit information from a source to a sink over a point-to-point connection with as few errors as possible. In Fig. 2.1 a source wants to transmit binary data $u \in \mathcal{U} = \{0, 1\}$ to a sink where u represents one draw of a binary uniformly distributed random variable. The source symbols are encoded, transmitted over a channel and decoded afterwards in order to pass an estimate $\hat{u}$ to a sink.

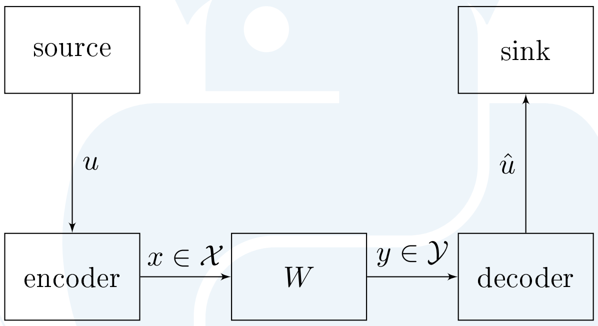


Figure 2.1.: system model for channel coding

This thesis uses a common notation for vectors which is introduced here shortly. A variable x may assume any value in an alphabet $x \in \mathcal{X}$. Multiple variables are combined into a vector $x^N = (x_0, \ldots, x_{N-1})$ of size N with its alphabet $x^N \in \mathcal{X}^N$. A subvector of x^N is denoted $x_i^j = (x_i, \ldots, x_{j-1})$ where $0 \leq i \leq j \leq N$. A vector where $i = j$ is an empty vector. A vector x^N may be split into even and odd subvectors which are denoted $x_{0,e}^{2n} = (x_0, x_2, \ldots, x_{2n-2})$, $x_{0,o}^{2n} = (x_1, x_3, \ldots, x_{2n-1})$. This numbering convention is in accordance with [Dij08], where the author makes a strong point for this exact notation and some papers on polar codes follow it too, e.g. [TV13].

2.1.1. Encoder

The encoder takes a frame u^k and maps it to a binary codeword x^N, where k and N denote the vector sizes of a frame and a codeword respectively with $k \leq N$. An ensemble of all valid codewords for an encoder is a code $\mathcal{C}$. It should be noted that $|\mathcal{C}| = |\mathcal{X}^N|$ must hold in order for the code to be able to represent every possible frame.

Not all possible symbols from $\mathcal{X}^N$ are used for transmission. The difference between all possible codewords 2^N and used codewords 2^k is called redundancy. With those two values, the code rate is defined as $R = \frac{k}{N}$. It is a measure of efficient channel usage.

The encoder is assumed to be linear and to perform a one-to-one mapping of frames to codewords. A code is linear if $\alpha x + \alpha' x' \in \mathcal{C}$ for $\forall x, x' \in \mathcal{C}$ and $\forall \alpha, \alpha' \in \mathbb{F}$ hold. It should be noted that all operations are done over the Galois field GF(2) or $\mathbb{F} = \{0, 1\}$ if not stated otherwise. Then the expression can be simplified to

$$x + x' \in \mathcal{C} \quad \text{for} \quad \forall x, x' \in \mathcal{C}. \tag{2.1}$$

A linear combination of two codewords must yield a codeword again.

For linear codes it is possible to find a generator matrix $G \in \mathbb{F}^{k \times N}$ and obtain a codeword from a frame with $x^N = u^k G^{k \times N}$. All linear codes can be transformed into systematic form $G = I_k P$. I_k is a $k \times k$ dimensional identity matrix. If G is systematic, all elements of a frame u^k are also elements of the codeword x^N. Also, a parity check matrix $H = -P^T I_{N-k}$ with dimensions $(N - k) \times N$ can be calculated from G. A parity check matrix satisfies $\forall x \in \mathcal{C} : Hx^T = 0^T$. Thus, a parity check matrix can be used to verify correct codeword reception and furthermore error correction may be performed. Error correction with H may be done, e.g. syndrome decoding.

A code can be characterized by the minimum distance between any two codewords. In order to obtain this value we use the Hamming distance. This distance $d(v^N, x^N)$ equals the number of positions in v^N that differ from x^N. Minimum distance of a code is than defined by $d(\mathcal{C}) = \min\{d(x, v) : x, v \in \mathcal{C}, x \neq v\}$. For linear codes this can be simplified to comparing all codewords to the zero codeword $d(\mathcal{C}) = \min\{d(x, 0) : x \in \mathcal{C}, x \neq 0\}$ which is called Hamming weight.

2.1.2. Channel model

In Fig. 2.2 a generic channel model W is shown. Its input is $x \in \mathcal{X}$ and its distorted output is $y \in \mathcal{Y}$. A channel is denoted $W : \mathcal{X} \to \mathcal{Y}$ along with its transition probability $W(y|x), x \in \mathcal{X}, y \in \mathcal{Y}$. A Discrete Memoryless Channel (DMC) does not have memory, thus every symbol transmission is independent from any other. Combined with a binary input alphabet it is called a binary DMC (BDMC). For a symmetric channel model, $P(y|1) = P(-y|-1)$ must hold for an output alphabet $y \in \mathcal{Y}, \mathcal{Y} \subset \mathbb{R}$ [RU08]. Assuming symmetry for a BDMC leads to a symmetric BDMC (symmetric-BDMC). In Sec. 2.2 several examples of such channels are discussed.

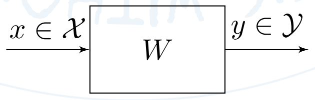


Figure 2.2.: A generic channel

This channel concept may be extended to vector channels. A vector channel W^N corresponds to N independent uses of a channel W which is denoted as $W^N : \mathcal{X}^N \to \mathcal{Y}^N$. Also, vector transition probabilities are denoted $W^N(y^N|x^N) = \prod_{i=0}^{N-1} W(y_i|x_i)$.

2.1.3. Decoder

A decoder receives a possibly erroneous codeword y and checks its validity by asserting $Hy^T = 0^T$, thus performing error detection. A more sophisticated decoder tries to correct errors by using redundant information transmitted in a codeword. An optimal decoder strategy is to maximize the a-posteriori probability. Given the probability of each codeword $P(x)$ and the channel transition probability $P(y|x)$, the task at hand is to find the most likely transmitted codeword x under the observation y, $P(x|y)$. This is denoted

$$\hat{x}^{MAP} = \underset{x \in \mathcal{C}}{\operatorname{argmax}}\, p(x|y) = \underset{x \in \mathcal{C}}{\operatorname{argmax}}\, p(y|x)\frac{p(x)}{p(y)} = \underset{x \in \mathcal{C}}{\operatorname{argmax}}\, p(y|x)p(x) \tag{2.2}$$

with Bayes' rule. Assume every codeword is transmitted with same probability $P(x^{(i)}) = P(x^{(j)}),\ \forall x^{(i)}, x^{(j)} \in \mathcal{C}$. This simplifies the equation and yields a Maximum Likelihood (ML) decoder

$$\hat{x} = \underset{x \in \mathcal{C}}{\operatorname{argmax}}\, p(y|x) \tag{2.3}$$

which estimates the most likely codeword to be transmitted given a received possibly erroneous codeword [RU08]. This decoding principle could be employed in conjunction with the Hamming distance and thus yield $\hat{x} = \operatorname{argmin}_{x \in \mathcal{C}} d(x, y)$. In conclusion the task at hand is to find a code which inserts redundancy intelligently, so a decoder can use this information to detect and correct transmission errors.

2.1.4. Asymptotically good codes

A repetition code is a very simple code which helps clarify certain key concepts in the channel coding domain. Assume the encoder and decoder shown in Fig. 2.1 use a repetition code. For example a repetition code with $k = 1$ and $N = 3$ has two codewords $\mathcal{C} = \{000, 111\}$. Thus in this example $R = \frac{1}{3}$. We can also obtain its generator and parity check matrices.

$$G = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}, \qquad H = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \tag{2.4}$$

H can be used to detect if a transmission error occurred by verifying if $Hx^T = 0^T$. In case an error occurred, a ML decoder does a majority decision to estimate the most likely codeword.

Repetition codes shed light on a problem common to a lot of codes. If reliability of a code needs to be improved, it comes at the expense of a lower code rate. Increasing N comes at the expense of decreasing $R = \frac{1}{N}$ because $k = 1$ for all repetition codes. Thus for a very reliable repetition code $\lim_{N \to \infty} R$ tends towards 0.

The above results leads to the definition of asymptotically good codes $\mathcal{C}(N_s, k_s, d_s)$ [Fri]. Two properties must hold for this class of codes,

$$R = \lim_{s \to \infty} \frac{k_s}{N_s} > 0 \quad \text{and} \quad \lim_{s \to \infty} \frac{d_s}{N_s} > 0. \tag{2.5}$$

The code rate must be > 0 for all codes which repetition codes do not satisfy. And the distance between codewords must grow proportionally to the code block size.

2.2. Channels

Several common channel models exist to describe the characteristics of a physical transmission. Common properties were discussed in Section 2.1.2 whereas in this Section the differences are targeted. The three most important channel models for polar codes are presented, namely the Binary Symmetric Channel (BSC), the Binary Erasure Channel (BEC) and the Additive White Gaussian Noise (AWGN) channel.

2.2.1. BSC channel

A transmission which is affected by bit-flips can be be modeled by a BSC. Fig. 2.3 shows the possible bit transitions from source to sink. Bits are either transmitted correctly or they are bit-flipped with transition probability p_e. The input and output alphabet of a BSC is $\mathcal{X} = \mathcal{Y} = \{0, 1\}$.

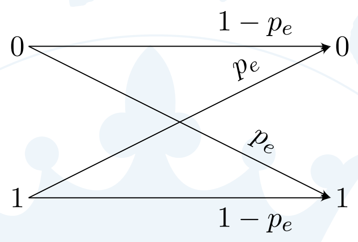


Figure 2.3.: BSC channel model

2.2.2. BEC channel

A BEC channel expects a binary input alphabet. Bits are either transmitted correctly or the information they conveyed is lost. This case is denoted by X, making the output alphabet $\mathcal{Y} = \{0, X, 1\}$. Fig. 2.4 shows the possible transitions where transition probability p_e denotes the probability that the bit information is lost during transmission.

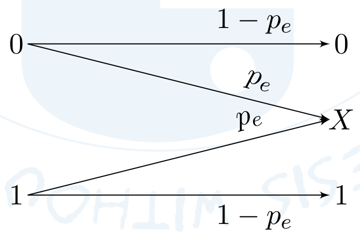


Figure 2.4.: BEC channel model

2.2.3. AWGN channel

An AWGN channel as used in this thesis has a binary input alphabet and a continuous output alphabet $\mathcal{Y} = \mathbb{R}$. Each input symbol is affected by Gaussian noise to derive an output symbol. Its average corresponds to the input symbol value and the variance can be interpreted as a measure of noise. Often the input is Non-Return-to-Zero (NRZ) encoded which turns a ML decision for a symbol into a sign decision.

2.2.4. Capacity and reliability

Channels are often characterized by two important measures, capacity and reliability. These measures are introduced in this Section. Channel capacity for symmetric-BDMC can be calculated by

$$I(W) = \frac{1}{2} \sum_{y \in \mathcal{Y}} \sum_{x \in \mathcal{X}} W(y|x) \log_2 \frac{W(y|x)}{\frac{1}{2}(W(y|0) + W(y|1))}. \tag{2.6}$$

It defines the highest rate at which a reliable transmission over a channel W can be conducted while the error probability may still tend towards 0. It is also called the Shannon capacity [Sha49] for symmetric channels. The Bhattacharyya parameter

$$Z(W) = \sum_{y \in \mathcal{Y}} \sqrt{W(y|0)W(y|1)} \tag{2.7}$$

is used to quantify a channel's reliability where a lower value for $Z(W)$ indicates higher reliability. It is also referred to as Z-parameter for obvious reasons. Also, an upper ML decision error bound is given by $Z(W)$ [Ari09].

3. Polar codes

Polar codes are the first asymptotically good codes with low complexity encoders and decoders. Erdal Arikan introduced polar codes in 2009 [Ari09]. They are designed to exploit the channel polarization effect.

The chapter is divided into several parts. Firstly, the idea behind polar codes is introduced. This includes a first glance at different parts of polar codes. It will explain the underlying channel polarization effect which is exploited for polar codes. Secondly, in-depth discussions of encoders, decoders and channel construction are presented. Enhancements to the initial approach are discussed afterwards. Additionally, several interesting properties of polar codes are discussed.

3.1. Code description

This section introduces polar codes and their parameters. A brief outline of the parameters that define polar codes is given. Then an example of a polar code is explored.

3.1.1. Polar code parameters

Polar codes are linear block codes defined by four properties, the block size N, the number of information bits k, the frozen bit positions $\mathcal{A}^\mathcal{C}$ and the frozen bit values $u_{\mathcal{A}^\mathcal{C}}^{N-k}$. A polar code's block size N defines the size of a codeword. Due to the structure of polar codes it must be a power of two, thus $N = 2^m, m \geq 0, m \in \mathbb{N}$. Then the number of information bits k in each block may vary by any integer value which holds $0 \leq k \leq N, k \in \mathbb{N}$. Given those two values, a polar code's code rate can be calculated by $R = \frac{k}{N}$.

Before a frame can be encoded, information bits must be interleaved with so-called frozen bits. Frozen bits have a known value and a known position in each codeword. Thus in order to completely specify a polar code this information must be known [Ari11].

Consider a frame $u_\mathcal{A}^k$ which holds all the information bits and a frozen bit value word $u_{\mathcal{A}^\mathcal{C}}^{N-k}$ which holds all the known frozen bits. They compose a source word $u^N = (u_\mathcal{A}^k, u_{\mathcal{A}^\mathcal{C}}^{N-k})$, where the sets $\mathcal{A} \subset \{0, \dots, N-1\}$ and $\mathcal{A}^\mathcal{C} \subset \{0, \dots, N-1\}$ with $\mathcal{A} \cap \mathcal{A}^\mathcal{C} = \emptyset$ define their respective positions. Also, $|\mathcal{A}| = k$ and $|\mathcal{A}^\mathcal{C}| = N - k$ must hold. While $u_\mathcal{A}^k$ changes in every source word, $u_{\mathcal{A}^\mathcal{C}}^{N-k}$ is known to encoder and decoder and defaults to an all-zero word. Channel construction sums up the task to find a set $\mathcal{A}^\mathcal{C}$.

3.1.2. Channel combining

The definition of polar codes may be split into channel combining and channel splitting. Those two parts explore the principle of polar codes which serves as a foundation for further discussions.

First, consider a symmetric-BDMC W with its transition probabilities $W(y|0)$, $W(y|1)$. Now N bits u^N shall be transmitted through a vector channel W^N. A channel combination

u_0	u_1	$W(0,0\|u^2)$
0	0	0.7921
0	1	0.0121
1	0	0.0979
1	1	0.0979

Table 3.1.: transition probabilities for W^2 with $y^2 = (0,0)$ and transition probability $p_e = 0.11$ for a BSC

scheme is applied which will be discussed in greater detail in Section 3.3. Here a simple example is given in order to illustrate the idea.

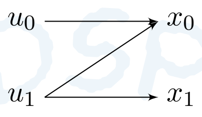


Figure 3.1.: fundamental two bit polar encoder

In Fig. 3.1 a frame u^2 is combined into a codeword x^2 which is then transmitted through a vector channel W^2. A receiver will receive y^2 and estimates $\hat{u}^2$. In the graph the vertices's on the left hold the source word u^2 and propagate them along the lines to the vertices's on the right. A vertex with to lines ending in it performs an addition with the two bits provided from the left. Thus $x_0 = u_0 + u_1$ and $x_1 = u_1$ define the encoding process. The codeword x^2 is transmitted through a vector channel W^2 and y^2 is received. Also, channel transition probabilities for y^2 may be calculated by $W^2(y^2|u^2) = W(y_0|u_0 + u_1)W(y_1|u_1)$, compare Table 3.1 as an example for $y^2 = (0,0)$. This basic example constitutes all polar codes of greater block size.

3.1.3. Channel splitting

After transmission an estimate from y^2 for u^2 is required. A Successive Cancellation (SC) decoder as described in Section 3.4 can be used to produce an estimate for the transmitted sourceword. Thus, for all $\hat{u}_i$ an estimate is calculated successively. Firstly, a vector channel W^N is split into N synthetic channels $W_N^{(i)}$, $0 \leq i < N$ with their transition probabilities

$$W_N^{(i)}(y^N, u^{i-1}|u_i) = \frac{1}{2^{N-1}} \sum_{u_{i+1}^N \in \mathcal{X}^{N-i}} W^N(y^N|u^N) \tag{3.1}$$

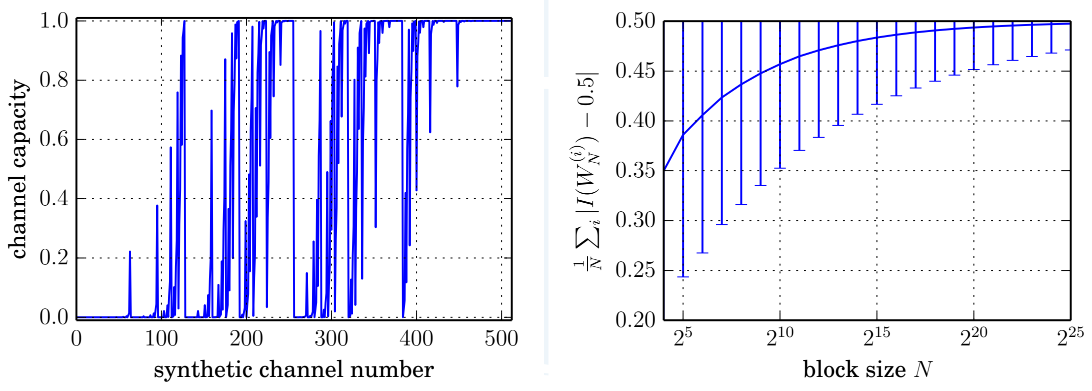
In case of W_2, the two synthetic channels are $W_2^{(0)}$ and $W_2^{(1)}$. The transition probabilities are given by $W_2^{(0)}(y^2|u_0) = \frac{1}{2}\sum_{u_1 \in \mathcal{X}} W^2(y^2|u^2)$ and $W_2^{(1)}(y^2, u_0|u_1) = \frac{1}{2}W^2(y^2|u^2)$. Most notably for $W_2^{(1)}(y^2, u_0|u_1)$, u_0 is no longer part of the input but the output. The capacity of the resulting channels compared to the capacity of the underlying channel illustrates the effect that can be observed. For a given BSC with $p_e = 0.11$, $I(W) \approx 0.500$ the resulting capacities are $I(W_2^{(0)}) \approx 0.28655$ and $I(W_2^{(1)}) \approx 0.71362$. Compared to the capacity of the underlying channel W, one capacity increased and the other decreased. This process is called polarization. Its effect increases by combining copies of W_2 and will continue to do so as N increases. It can be shown that most synthetic channel capacities either tend to 0.0 or 1.0 which is further discussed in Section 3.2.3.

A polar code of block size N with k information bits, would choose the $N-k$ least reliable synthetic channels for frozen bit positions. In other words, choose the synthetic channels with highest capacity for information bit positions $\mathcal{A}$ and all others for frozen bit positions $\mathcal{A}^{\mathcal{C}}$.

3.2. Code construction

Channel construction for polar codes is performed once for a desired code. For a given set of parameters, namely block size N and a target scenario, channel capacities $I(W_N^{(i)})$ or Bhattacharyya parameters $Z(W_N^{(i)})$ are calculated. Fig. 3.2a shows an example channel construction for a BEC channel model and an error probability which was deduced from a design-Signal-to-Noise-Ratio (SNR) (dSNR) with equation (3.25). The polarization effect strengthens with block size which is shown in Fig. 3.2b. The distance of every synthetic channel capacity from that of the underlying channel is calculated and averaged. It can be observed that the synthetic channels polarize in the sense that the average distance tends towards the capacity of the underlying channel. This indicates that most synthetic channels tend towards noise-only or noise-free channels while the standard deviation decreases.

In general, channel construction is a computationally heavy task. A recursive function to calculate a set of channel capacities $\{I(W_N^{(i)})\}$ for BEC efficiently is given in [Ari09]. Whereas channel construction for other channel models employs approximations in order to ease the computational burden. In [TV13] Tal et al. propose a technique to calculate estimates for a set with little deviation from the true value.



(a) synthetic channel capacities for BEC, $N = 512$, dSNR $-1.59dB$

(b) average synthetic channel capacity distance from the underlying channel capacity 0.5

Figure 3.2.: Polarization effect

3.2.1. Recursive channel construction

Given a channel W, a recursive channel transformation is proposed in [Ari09] to construct a set of channel capacities $\{I(W_N^{(i)})\}$. First, a transformation for two independent channels is given. The objective is to find a one-to-one mapping which satisfies

$$(W, W) \to (W', W'') \quad \text{or} \quad (W_1, W_1) \to (W_2^{(0)}, W_2^{(1)}). \tag{3.2}$$

In case of $N = 2$, equation (3.1) can be rewritten into

$$W_2^{(0)}(y^2|u_0) = \frac{1}{2}\sum_{u_1 \in \mathcal{X}} W_2(y^2|u^2) = \frac{1}{2}\sum_{u_1 \in \mathcal{X}} W(y_0|u_0 + u_1)W(y_1|u_1) \tag{3.3}$$

and

$$W_2^{(1)}(y^2, u_0|u_1) = \frac{1}{2}W_2(y^2|u^2) = \frac{1}{2}W(y_0|u_0 + u_1)W(y_1|u_1) \tag{3.4}$$

where $W' = W_2^{(0)}(y^2|u_0), W'' = W_2^{(1)}(y^2, u_0|u_1)$.

The $N = 2$ example from Section 3.1 is extended to $N = 4$ with a combination of two copies of the $N = 2$ case. The vector channel $W^4 : \mathcal{X}^4 \to \mathcal{Y}^4$ has the transition probabilities $W^4(y^4|u^4) = W^4(y^4|u^4 G_4)$ where the generator matrix G_4 is discussed in Section 3.3. The synthetic channel transition probabilities can be calculated by

$$W_4^{(i)}(y^4, u^i|u_i) = \frac{1}{4}\sum_{u_{i+1}^4 \in \mathcal{X}^{3-i}} W^4(y^4|u^4). \tag{3.5}$$

This approach requires to calculate lots of values multiple times. A recursive approach makes channel construction more efficient.

Fig. 3.3 depicts a tree which spans a recursive channel construction which is defined from a generalization of the channel transformation (3.2)

$$(W_N^{(i)}, W_N^{(i)}) \to (W_{2N}^{(2i)}, W_{2N}^{(2i+1)}). \tag{3.6}$$

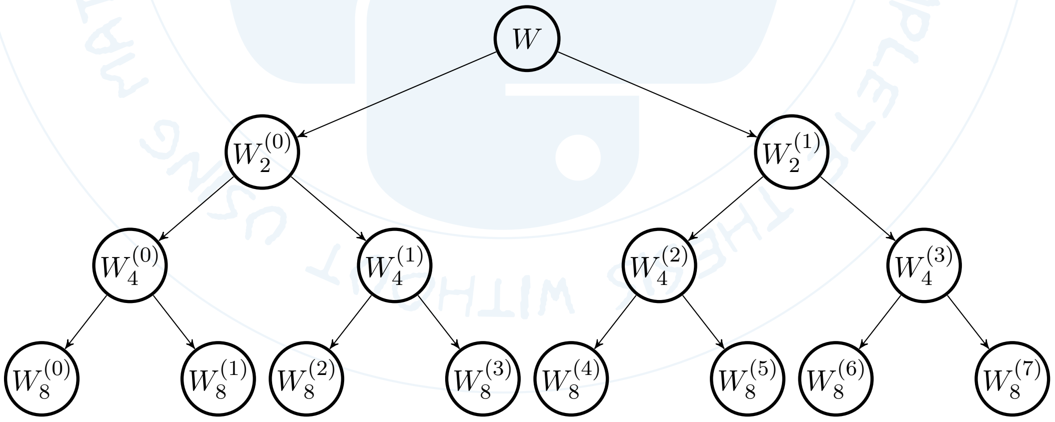


Figure 3.3.: recursive polar channel construction

The underlying channel model W_1 is transformed into $W_2^{(0)}$ and $W_2^{(1)}$ which are then transformed into $W_4^{(2i)}$. The synthetic channels $W_4^{(2i)}$ and $W_4^{(2i+1)}$ can be obtained from two copies of $W_2^{(i)}$. In other words, two copies of vector channels are transformed into a vector channel of size $2N$.

Equation (3.1) is rewritten into two equations for recursive channel transformations. In accordance with [Ari09] Proposition 3 those channel transformations are

$$W_{2N}^{(2i)}(y^{2N}, u^{2i}|u_{2i}) = \frac{1}{2} \sum_{u_{2i+1} \in \mathcal{X}} W_N^{(i)}(y_0^N, u_{0,e}^{2i} + u_{0,o}^{2i}|u_{2i} + u_{2i+1}) \cdot W_N^{(i)}(y_N^{2N}, u_{0,o}^{2i}|u_{2i+1}) \tag{3.7}$$

$$W_{2N}^{(2i+1)}(y^{2N}, u^{2i+1}|u_{2i+1}) = \frac{1}{2} W_N^{(i)}(y_0^N, u_{0,e}^{2i} + u_{0,o}^{2i}|u_{2i} + u_{2i+1}) \cdot W_N^{(i)}(y_N^{2N}, u_{0,o}^{2i}|u_{2i+1}) \tag{3.8}$$

with $n \geq 0$, $N = 2^n$, $0 \leq i < N$. The output alphabet y^{2N} for $W_{2N}^{(2i)}$ and $W_{2N}^{(2i+1)}$ is split into half for each of the vector channel copies $W_N^{(i)}$. The already decoded bits are split into $u_{0,e}^{2i}$ and $u_{0,o}^{2i}$ which form vectors of size N if combined with u_{2i} and u_{2i+1} respectively.

With equations (3.7) and (3.8), it is possible to calculate synthetic channels for polar codes with arbitrary size and varying underlying channels. Though, this procedure is inefficient for all channels but BEC. In [TV13], a quantization algorithm is proposed to reduce complexity.

3.2.2. Capacity and reliability of polar codes

As stated before, polar codes are the first class of codes to achieve Shannon channel capacity while being asymptotically good. Also, repetition codes were introduced in Section 2.1.4 as an example of asymptotically bad codes. It can be shown that the channel transformation (3.6) preserves a channel's rate and improves its reliability. In [Ari09] several propositions are made and proved to describe these properties. Furthermore an efficient channel construction strategy can be obtained for BEC channels which may also serve as an upper bound of a synthetic channels reliability. The propositions are given the same number as in [Ari09].

A channel transformation (3.2) preserves the aggregated channel capacity of a vector channel

$$I(W') + I(W'') = 2I(W) \tag{3.9}$$

while the capacities of the individual channels polarize in the sense that their capacities move away from the average they form

$$I(W') \leq I(W'') \tag{3.10}$$

[Ari09] *Proposition 4*. Synthetic channel capacities are only equal if and only if $I(W) \in \{0, 1\}$.

In order to sketch the proof for this proposition consider the independently uniformly distributed Random Variable (RV) pair (U_0, U_1) which has a one-to-one mapping to $(X_0, X_1) = (U_0 + U_1, U_1)$. The trans-information for the two channels W' and W'' may be denoted as $I(W') = I(U_0; Y_0 Y_1)$ and $I(W'') = I(U_1; Y_0 Y_1 U_0) = I(U_1; Y_0 Y_1 | U_0)$ respectively, where (Y_0, Y_1) are RVs obtained from transmission of (X_0, X_1) over W^2. Using this result in $I(W') + I(W'')$ yields

$$I(U_0; Y_0 Y_1) + I(U_1; Y_0 Y_1 | U_0) = I(U_0 U_1; Y_0 Y_1) = I(X_0; Y_0) + I(X_1; Y_1) = 2I(W). \tag{3.11}$$

In order to prove equation (3.10) $I(W'')$ is rewritten as

$$I(W'') = I(U_1; Y_0 Y_1 U_0) = I(W) + I(U_1; Y_0 U_0 | U_1) \tag{3.12}$$

where intermediate steps are omitted. Combined with equation (3.9) the proof is completed.

Proposition 5 sets the relations between the reliabilities of the underlying and the synthetic channels.

$$Z(W'') = Z(W)^2 \tag{3.13}$$
$$Z(W') \leq 2Z(W) - Z(W)^2 \tag{3.14}$$
$$Z(W') \geq Z(W) \geq Z(W'') \tag{3.15}$$

from Proposition 5

$$Z(W') + Z(W'') \leq 2Z(W) \tag{3.16}$$

can be derived. It can be observed that channel transformation may improve overall reliability. The proofs for Proposition 5 are omitted because they mostly consist of rewriting the reliability equations. They are available in [Ari09].

The equality case for equation (3.16) deserves special attention. For a BEC the channel reliabilities are $Z(W) = p_e$ and $Z(W') = 2p_e - p_e^2$ and $Z(W'') = p_e^2$. This leads to *Proposition 6* which states if W is a BEC then W' and W'' are BECs and vice versa.

Proposition 7 generalizes Propositions 4-6 to the recursive channel transformation (3.6) $(W_N^{(i)}, W_N^{(i)}) \rightarrow (W_{2N}^{(2i)}, W_{2N}^{(2i+1)})$. It states that overall capacity is preserved while reliability may be improved.

$$I(W_{2N}^{(2i)}) + I(W_{2N}^{(2i+1)}) = 2I(W_N^{(i)}) \tag{3.17}$$
$$Z(W_{2N}^{(2i)}) + Z(W_{2N}^{(2i+1)}) \leq 2Z(W_N^{(i)}) \tag{3.18}$$

Derived from this result and recursive channel construction it follows

$$\sum_{i=0}^{N-1} I(W_N^{(i)}) = NI(W) \tag{3.19}$$
$$\sum_{i=0}^{N-1} Z(W_N^{(i)}) \leq NZ(W) \tag{3.20}$$

Bhattacharyya bounds

An upper limit on $Z(W_N^{(i)})$ for every value in a set $\{Z(W_N^{(i)})\}$ can be obtained efficiently in case W is a BEC with erasure probability p_e. Equality for equation (3.16) holds in case the underlying channel is a BEC. *Propsition 6* states that the resulting synthetic channels must be BECs as well. Together with the generalization from *Proposition 7* the Bhattacharyya parameters $Z(W_N^{(i)})$ represent an upper limit for all other BDMCs. This motivates the naming for this channel construction method to be Bhattacharyya bounds method [VVH15].

$$Z(W_N^{(2i)}) = 2Z(W_{N/2}^{(i)}) - Z(W_{N/2}^{(i)})^2 \tag{3.21}$$

$$Z(W_N^{(2i+1)}) = Z(W_{N/2}^{(i)})^2 \tag{3.22}$$

with $Z(W_1^{(0)}) = p_e$. Also, the set $\{I(W_N^{(i)})\}$ can be obtained from

$$I(W_N^{(2i)}) = I(W_{N/2}^{(i)})^2 \tag{3.23}$$

$$I(W_N^{(2i+1)}) = 2I(W_{N/2}^{(i)}) - I(W_{N/2}^{(i)})^2 \tag{3.24}$$

with $I(W_1^{(0)}) = 1 - p_e$.

p_e deduction for BEC from a dSNR $\frac{RE_b}{N_0}[dB]$ was first proposed by [VVH15] as

$$p_e = e^{-10^{\frac{RE_b}{10 \cdot N_0}}}. \tag{3.25}$$

Equation (3.25) may be used to obtain an initial p_e for the Bhattacharyya bounds method which yields an upper limit on $\{Z(W_N^{(i)})\}$. (3.21) and (3.23) then yield a limit for channel construction. In [VVH15] it is also observed that this approach is often on par with other channel construction algorithms in terms of Bit-Error-Rate (BER).

3.2.3. Polarization

Two Theorems exist for polarization which propose that the polarization effect exists and at which rate it takes effect. The proofs may be found in Arikan's paper [Ari09]. Polarization describes the process in which synthetic channels either tend to be perfect channels or tend to be noise only channels, Fig. 3.2b. Also, a minimum rate of polarization is given by an upper limit on the error probability of all information bits. This limit tends towards 0 for $N \rightarrow \infty$ or a higher block size results in stronger polarization.

Polarization theorem

A BDMC W is transformed recursively as shown in Fig. 3.3. Observing all $I(W_N^{(i)})$, it can be shown that synthetic channels polarize into two fractions as $N = 2^m \rightarrow \infty, m \in \mathbb{N}$. Those two fractions are $I(W_N^{(i)}) \in (1 - \delta, 1]$ and $I(W_N^{(i)}) \in [0, \delta)$ where a fixed $\delta \in (0, 1)$ is assumed.

The proof defines a set of Bernoulli Random Variables (RVs) which define a random walk from the root to a leaf of the tree in Fig. 3.3. A 0 indicates that the upper branch at the current vertex is chosen and 1 a lower branch where both values have equal probability. A set of m Bernoulli RVs defines a leaf in the m-th level of the channel transformation tree. It is shown with the help of Proposition 4 that such a random walk almost always ends in a leaf which is part of one of the two previously defined fractions.

Rate of polarization theorem

The aforementioned theorem proves that synthetic channels polarize. For a given underlying BDMC W with $I(W) > 0$ and a fixed $R < I(W)$ the rate at which synthetic channels polarize is proved by the rate of polarization theorem. An upper bound for reliability of the synthetic channels in an information bit position set $\mathcal{A}$ is given where $|\mathcal{A}| \geq NR$ holds. This leads to the upper bound for reliability

$$Z(W_N^{(i)}) \leq \mathcal{O}(N^{-\frac{5}{4}}), \forall i \in \mathcal{A} \tag{3.26}$$

which can be interpreted as the rate at which the reliability of information bits improves. A set $\mathcal{T}_m(\zeta)$ is assumed which contains all synthetic channel reliabilities $Z_i(\omega) \leq \zeta$ at up to the i-th level. Given the ratio $\frac{Z_{i+1}(\omega)}{Z_i(\omega)}$ for the next iteration of channel transformations, it is shown that most ratios are below a given threshold with sufficiently high probability. With this result the theorem eventually follows.

3.2.4. Efficient channel construction

It was mentioned before that there is no efficient method for channel construction for an arbitrary BDMC. Tal et al. proposed an algorithm to derive a channel representative for a given synthetic channel [TV13]. In Section 3.2.1 it was shown how to construct synthetic channels recursively. Unfortunately, the output alphabet of those channels grows exponentially. Thus, Tal et. al. propose an algorithm to reduce this output alphabet size to a maximum μ. In fact two approaches are given.

Those approaches are called upgrading and degrading quantization. Upgrading quantization refers to the fact that an upgraded channel representative is obtained with respect to capacity and vice versa. Furthermore, it was shown that upgrading and downgrading quantizations are very close in terms of deviation from their respective true value. This approach makes it feasible to obtain a set $\{Z(W_N^{(i)})\}$ for arbitrary channel models.

In most cases only the degrading channel quantization is used. Each value in a set obtained by this algorithm is compared to the value obtained by the Bhattacharyya bounds 3.2.2 and the smaller of those two values is then part of the final set. This may be done because the Bhattacharyya bounds method imposes an upper limit on synthetic channel reliability.

Degrading quantization

A degrading quantization is the commonly used strategy to reduce output alphabet size. In accordance with [TV13][Section V(A)] a recursive algorithm is defined. The output alphabet $\mathcal{Y}$ of a channel $W : \mathcal{X} \to \mathcal{Y}$ is reduced by two. The result is a degraded channel $Q : \mathcal{X} \to \mathcal{Z}$. Its output alphabet size is defined by

$$\mathcal{Z} = \mathcal{Y} \setminus \{y_0, \bar{y}_0, y_1, \bar{y}_1, \} \cup \{z_{0,1}, \bar{z}_{0,1}\} \tag{3.27}$$

where $\bar{y}$ is y's conjugate. The merging operation is defined as

$$Q(z|x) = \begin{cases} W(z|x) \text{ if } z \notin \{z_{0,1}, \bar{z}_{0,1}\} \\ W(y_0|x) + W(y_1|x) \text{ if } z = z_{0,1} \\ W(\bar{y}_0|x) + W(\bar{y}_1|x) \text{ if } z = \bar{z}_{0,1} \end{cases} . \tag{3.28}$$

The objective is to find the two symbols y_0, y_1 to merge which maximize capacity for Q. In order to do so each symbol is associated with a Likelihood Ratio (LR).

$$L_W(y) = \frac{W(y|0)}{W(y|1)} = \frac{W(y|0)}{W(\bar{y}|0)} \tag{3.29}$$

In this case $L(y) = \frac{1}{L(\bar{y})}$ holds. From each symbol and its conjugate a representative LR is chosen which satisfies $L(y) \geq 1$. All those representatives are ordered $1 \leq L(y_1) \leq \cdots \leq L(y_L)$. There are L representatives in total. For each neighboring pair $L(y_i)$ and $L(y_{i+1})$ in this list a capacity loss is calculated if they were merged. The merge with the lowest loss is chosen and performed. For the sake of clarity, several shorthands are defined - specifically $a = W(y_i)$, $b = W(\bar{y}_i)$, $a' = W(y_{i+1})$, $b' = W(\bar{y}_{i+1})$. Then a capacity loss can be calculated by

$$\Delta I(\, a, b, a', b' \,) = C(a, b) + C(a', b') - C(a + a', b + b'). \tag{3.30}$$

where $C(a, b)$ is defined as

$$C(a, b) = -(a + b)\log_2(\frac{a + b}{2}) + a\log_2(a) + b\log_2(b). \tag{3.31}$$

The merged symbol z is replaces y_i and y_{i+1} in the list as a single symbol. This degrading merge process is repeated until the list contains a maximum of μ elements.

It is important to note that this is only a sketch for the full algorithm. All the details and assumptions are laid out in [TV13] and some more explanations may be found in [VVH15].

3.2.5. Code structure

Polar codes are defined by a recursive channel transformation. Each iteration doubles code size. This process can be illustrated with the help of a binary tree shown in Fig. 3.4 [GSL+15]. Thus polar codes are defined by constituent codes.

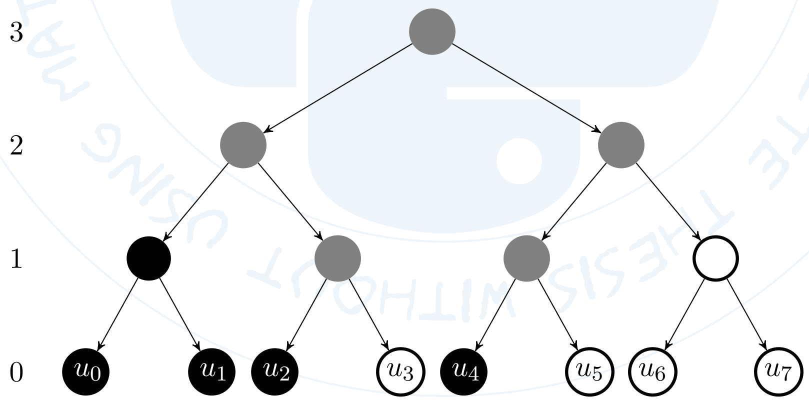


Figure 3.4.: polar code tree

First, a top-down view on Fig. 3.4 discussed. Starting at the root node at the top in the third level, two channels are constructed. Thus the tree splits into two subtrees. This process is repeated down to the zeroth level where the tree has eight leaf nodes.

The binary tree has four levels. On the zeroth level, leafs for each bit in a polar code with block size $N = 8$ and $k = 4$ are shown. Black leafs correspond to frozen bits whereas white leafs correspond to information bits. On the first level, parent nodes infer the color from their child nodes. Parent nodes which have child nodes with different colors turn gray. Several constituent codes can be observed on level 1. With (u_0, u_1) a rate-0 constituent code is formed. On the other end (u_6, u_7) form a rate-1 constituent code.

Up another level, two more constituent code types can be observed. (u_0, u_1, u_2, u_3) form a repetition code because only two different codewords exist and are chosen depending on the value of u_3. The other half forms a Single-Parity-Check (SPC) code with (u_4, u_5, u_6, u_7) where u_4 may be used as a parity bit in a decoder. The root node on the fourth level does not represent a special code anymore.

Identifying different types of constituent codes is not necessary for polar codes in general. Though, those specialized constituent codes offer significant simplifications for decoders. They were proposed in [GSL+15] where such a decoder is called Fast-Simplified SC (Fast-SSC) decoder.

3.2.6. Systematic polar codes

Most of the theory regarding polar codes describes their properties with regard to non-systematic codes. It should be mentioned that polar codes can be made systematic because any linear code can be made systematic [Ari11].

A set $\mathcal{A}$ of a systematic polar code defines the position of each information bit in a source word u^N and the corresponding codeword x^N. More specifically $u_i = x_i$ must hold. In [Ari11] systematic polar codes were first proposed and an efficient structure for polar code encoding was proposed in [STG+15]. The efficient polar encoding structure restricts the encoder to properly defined polar codes. Properly constructed polar codes abide by domination contiguity. If a specific polar code is indeed domination contiguous can be validated by

$$(E \cdot F^{\otimes m} \cdot E^T) \cdot (E \cdot F^{\otimes m} \cdot E^T) = I \tag{3.32}$$

where $F^{\otimes m}$ is defined in (3.3.1). The expanding matrix E has dimensions $k \times N$ and is defined as

$$E = (E_{i,j})_{i=0,j=0}^{k-1,N-1}, \text{ where } E_{i,j} = \begin{cases} 1 \text{ if } j = \alpha_i \\ 0 \text{ otherwise} \end{cases} \tag{3.33}$$

Lastly, α_i is defined as the i-th element of an ordered active set $\mathcal{A}$ where $0 \leq \alpha_0 < \alpha_1 < \cdots < \alpha_{k-1} \leq N - 1$. Besides the theoretical results, domination contiguity yields a simple validation tool for polar code channel construction.

In order to encode a polar code systematically, a source word u^N is encoded with $v^N = u^N F^{\otimes m}$. Then for every frozen bit position $\mathcal{A}_{\mathcal{C}}$, v^N is reset to the corresponding frozen bit value. It is then once again encoded by $x^N = v^N F^{\otimes m}$ and thus made systematic. A systematic decoder only needs an additional encoder pass at the end to recover the estimate $\hat{u}^N$. It will be shown that the additional encoder pass only draws a minor amount of resources while the BER is decreased noticeably and the Frame Error Rate (FER) does not benefit from systematic codes [Ari11].

3.3. Encoder

A polar code encoder needs the values described in 3.1. It takes in a frame $u_{\mathcal{A}}^k$, interleaves it with frozen bits $u_{\mathcal{A}^c}^{N-k}$ into a source word u^N and encodes it. In this section a matrix definition of a polar encoder and a more efficient graph-based encoder are introduced.

3.3.1. Matrix definition

Polar codes are linear codes, thus a generator matrix G_N can be formulated. A codeword is then obtained by $x^N = u^N G_N$ where $G_N = B_N F^{\otimes m}$. The matrices G_N, B_N and $F^{\otimes m}$ are square matrices. The encoder can be split into a constant $x_{\mathcal{A}^c}^N = u_{\mathcal{A}^c}^{N-k} G_{\mathcal{A}^c}$ and a dynamic part $x_{\mathcal{A}}^N = u_{\mathcal{A}}^k G_{\mathcal{A}}$ where $x^N = x_{\mathcal{A}^c}^N + x_{\mathcal{A}}^N$. A linear generator matrix with dimensions $N \times k$ is then given by $G_{\mathcal{A}}$.

The matrix $F^{\otimes m}$ is obtained by the m-th Kronecker product of a kernel matrix $F = \left(\begin{smallmatrix}1 & 0\\ 1 & 1\end{smallmatrix}\right) = F^{\otimes 1}$. An example for a polar code of block size $N = 2^2$ is

$$F^{\otimes 2} = \begin{pmatrix} F^{\otimes 1} & \begin{smallmatrix}0 & 0\\ 0 & 0\end{smallmatrix} \\ F^{\otimes 1} & F^{\otimes 1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \tag{3.34}$$

Bit reversal vs. natural order

The bit-reversal matrix B_N is a permutation matrix. It has one 1 element in every column and besides it is all-zero. Bit-reversal refers to the fact that the binary representation of a number is read out in reversed order. For a block size $N = 2^m$, m represents the number of considered bits to reverse. It is important to note that bit-reversal must be performed with zero-based numbering in order to yield correct results. In case of B_N, the columns are enumerated and each column's bit reversed number determines the column's row with a 1.

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \rightarrow B_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \Rightarrow G_4 = B_4 F^{\otimes 2} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \tag{3.35}$$

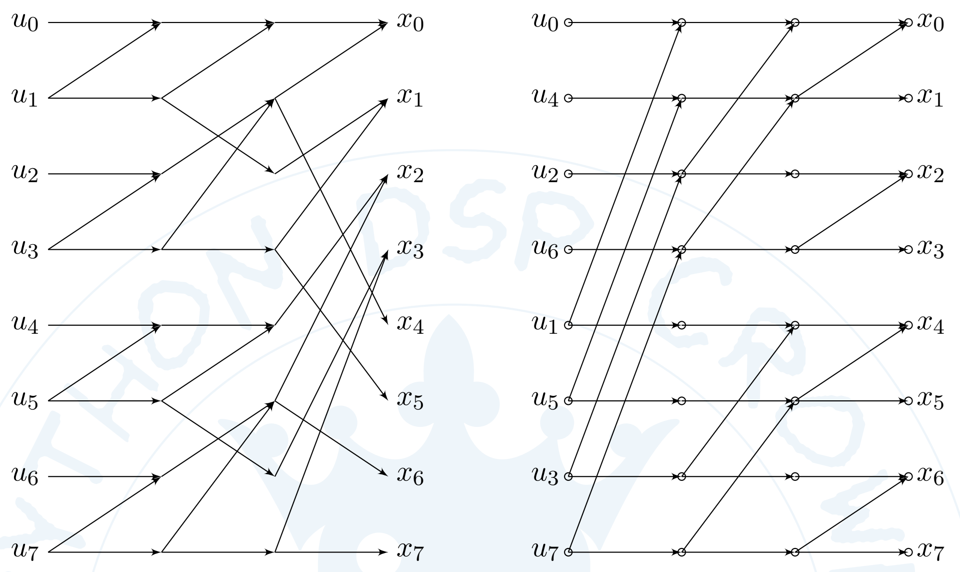
The matrices above illustrate the transformation from an identity matrix to a bit-reversal matrix along with the resulting encoder matrix.

If B_N is applied to G_N or to each source word u^N will be up to the implementation. Hardware implementations tend to apply B_N to each u^N. In the software radio domain it is preferable to apply B_N to G_N once because data elements are aligned better for a software implementation.

3.3.2. Graph representation

Matrix multiplications for encoding yield bad performance because matrix multiplications have $\mathcal{O}(N^2)$ runtime. But matrix multiplications are not required because exploiting the structure of G_N leads to a graph-based representation of the encoder which improves runtime to $\mathcal{O}(N \log N)$. A simple example for an encoder was given in Fig. 3.1.

Encoders with larger block sizes are constituted of this butterfly structure as depicted in Fig. 3.5. The results of the butterflies are interleaved in order to serve as input for the next stage of the encoding process. Two graphs are shown which represent different options to approach encoding. The natural bit-order structure, which represents $G_8 = B_8 F^{\otimes 3}$, in Fig. 3.5a seems to be more chaotic at first glance. Though, it does not require a bit-reversal operation for the whole source word before encoding.



(a) 8 bit polar encoder in natural bit order, $G_8 = B_8 F^{\otimes 3}$

(b) 8 bit polar encoder in bit-reversed order, $G_8 = F^{\otimes 3}$

Figure 3.5.: POLAR encoder comparison

The encoders shown in Fig. 3.5 have four stages. In each stage a pair (u_i, u_{i+1}) is added over GF(2) in a node with two lines ending in it. This is a combine operation. u_{i+1} is copied over into the next stage, making it a copy operation.

3.4. Decoders

Decoders for polar codes come in different flavors. Arikan proposed a Successive Cancellation (SC) decoder in [Ari09] for his proof that polar codes achieve channel capacity. A SC decoder strategy may yield $\mathcal{O}(N \log N)$ algorithmic complexity. Thus it is the baseline in terms of error-correction performance compared to other strategies which will turn out to be more resource intensive.

3.4.1. Successive Cancellation

The basic idea of SC decoding is to decode one bit at a time. The decoder will produce an estimate $\hat{u}^N$ for a source word. The information bits will be extracted from $\hat{u}^N$ and a frame estimate $\hat{u}^k$ will be passed on to the sink.

Log Likelihood Ratio's are used in a SC decoder as intermediate values. The LR is defined in equation (3.29) and a Log Likelihood Ratio (LLR) is defined with it by

$$l(y_i) = \ln(L(y_i)) = \ln\left(\frac{W(y_i|0)}{W(y_i|1)}\right) \tag{3.36}$$

A decoder bit decision will be made from those values following

$$\hat{u}_i = \begin{cases} 0, & \text{if } L(y_i) \geq 1 \\ 1, & \text{otherwise} \end{cases} = \begin{cases} 0, & \text{if } l(y_i) \geq 0 \\ 1, & \text{otherwise} \end{cases} \tag{3.37}$$

Polar decoders will mostly use LLRs because the equations involved are more numerically stable and easier to calculate [BPB14].

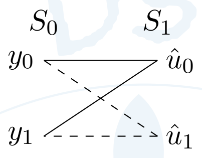


Figure 3.6.: fundamental two bit polar decoder

Similar to the encoder, a block size $N = 2^1$ explains the decoder strategy as shown in Fig. 3.6. On the left, the received codeword y^2 is shown. An initial LLR value is assigned to each value in the received codeword. If NRZ coding was used, also the channel output values may be used. y^2 is located in stage S_0 in the graph. A stage S_i denotes a column of nodes. On the left the estimated source word is denoted with $\hat{u}^2$ in stage S_1.

Equation (3.39) is applied to calculate an LLR for the upper node in stage S_1. From this LLR $\hat{u}_0$ is obtained by applying equation (3.37). $\hat{u}_0$ is then used in conjunction the two LLRs in stage S_0 with equation (3.41) to calculate a LLR for the lower node in stage S_1. Again equation (3.37) is used to obtain the estimate $\hat{u}_1$. This is the complete decoding process for a two bit polar code. It is noteworthy that this graph structure may be compared to a Fast Fourier Transform (FFT) butterfly structure.

Recursive update equations are introduced in [Ari09] for SC decoders. Those equations are derived from the recursive channel construction transformations for LRs. The update rule for solid lines in a decoder graph is then given as

$$L(L_a, L_b) = \frac{L_a \cdot L_b + 1}{L_a + L_b} \tag{3.38}$$

L_a and L_b are obtained from values in stage S_{i-1} to calculate a LLR in stage S_i. In [BPB14], this equation is approximated for LLRs with the *min-sum approximation* as

$$f(l_a, l_b) = \text{sign}(l_a)\,\text{sign}(l_b)\min\{|l_a|, |l_b|\} \tag{3.39}$$

where l_a and l_b are LLRs as well as the result. Likewise, the update rule for dashed lines is defined as

$$L(L_a, L_b, \hat{u}) = L_a^{1-2\hat{u}} \cdot L_b \tag{3.40}$$

and again can be reformulated for LLRs as

$$g(l_a, l_b, \hat{u}) = (-1)^{\hat{u}}\, l_a + l_b = \begin{cases} l_b + l_a & \text{if } \hat{u} = 0 \\ l_b - l_a & \text{otherwise} \end{cases} \tag{3.41}$$

Fig. 3.7 shows an 8 bit decoder. In this case the graph consists of four stages. In general a decoder graph has $m+1$ stages. Again, the decoding process starts by providing LLRs for the received codeword y^8. Stage S_3 holds decision nodes, while stage S_0 holds y^8's LLRs. First, $\hat{u}_0$ is decoded. Starting on the right, the decoder recurses into the graph along the edges to the left. So far no calculations were performed, thus, from stage S_3 the decoder follows the solid lines into stage 2 and then 1, in order to update the corresponding LLRs there. In stage S_1 all solid lines to the left into stage S_0 end in nodes which already hold LLRs. Four nodes in stage S_1 are updated via equation (3.39). The upper solid line into a node in stage S_1 points to l_a, while the lower line points to l_b. Back in stage S_2 the first two LLRs can be calculated afterwards. Finally in stage S_3, the LLR for $\hat{u}_0$ can be calculated. Then equation (3.37) is used to decide which bit was most likely transmitted. If the used polar code has a frozen bit at position $\hat{u}_0$, its frozen bit value is used.

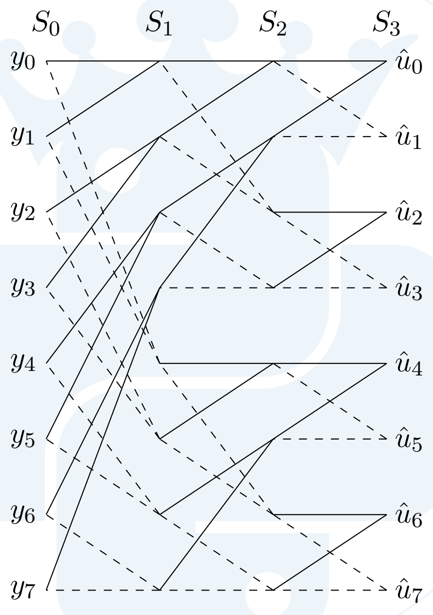


Figure 3.7.: 8 bit polar decoder

The decoding process continues with $\hat{u}_1$. The LLRs in stage S_2 for $\hat{u}_1$ are already calculated. Together with $\hat{u}_0$ and equation (3.41) the missing LLR is obtained. Then a bit decision is made or the frozen bit value at that position is used.

For $\hat{u}_2$ the decoder has to recurse into stage S_1 and use equation (3.41) to update two yet missing LLRs. But also, two $\hat{u}$ values need to be provided. The last two decoded bits $\hat{u}_0^2$ are re-encoded with an encoder of block size $N = 2$. Those re-encoded bits serve as input in order to calculate the two LLRs in stage S_2. Then an LLR for $\hat{u}_2$ can be calculated and a bit decision is made. $\hat{u}_2$ is used as input to calculate a LLR from the corresponding LLRs in stage S_2 before a bit decision yields $\hat{u}_3$.

The next bit to be decoded is $\hat{u}_4$. The decoder recurses into stage S_1 and calculates LLRs with equation (3.41). Only this time four bits need to be re-encoded. The four previously decoded bits $\hat{u}_0^4$ are re-encoded in an encoder with block size $N = 4$ and its result is used as input for the next four calculations in stage S_1. The decoding process continues just like described before with $\hat{u}_0$ to $\hat{u}_3$ but this time $\hat{u}_4$ and following are decoded and then used for the next bits to be decoded.

Larger block sizes extend a decoder with additional stages but the underlying principle with a depth first LLR calculation process is the same. The graph structure may be interpreted as being composed of butterflies, e.g. (y_0, y_1) in S_0 with there connected nodes in stage S_1. However, this may be better observed in a graph in reversed bit order. Here it shall serve as a reminder that such an interpretation exists but is mainly used in hardware implementations.

The graph structure follows natural bit order which is preferred in the software domain. Vector optimizations require data to be aligned in arrays next to each other. It can be observed in Fig. 3.7 that natural bit order fulfills this requirement. LLRs, needed for an update, are located on neighboring nodes and the results are deinterleaved onto neighboring nodes as well.

3.4.2. SC List decoder

Error correction performance for polar codes increases with block size. Though, that comes at the expense of latency. A sufficiently large block size will enable a polar code to outperform Low-Density Parity-Check (LDPC) codes. One approach to tackle this issue is to use a more sophisticated decoder. Successive Cancellation List (SCL) decoders promise better error correction performance at the the expense of higher computational complexity [TV11]. In [BPB14] they describe a way to perform all calculations with LLR values instead of LR values.

Instead of a hard bit decision for every bit, every time an information bit is decoded, the decoder follows both possible decoding paths for 0 and 1 respectively. All different decoding paths are held in a list. Every time an information bit is decoded, the list size doubles. This would be computationally very heavy. A maximum list size L is introduced to limit complexity. Maximum list size is typically chosen to be a power of 2, though that is not strictly necessary.

The SCL decoder starts decoding bits and follows each possible decoding path whenever an information bit is decoded. Whenever following all paths would result in a list size of $2L$, L paths are pruned from the list. To decide which paths should be pruned, a path metric is introduced.

The LLR-based path metric is a recursive function proposed by [BPB14]. The path metric $PM^{(i)} = \phi(PM^{(i-1)}, l^{(i)}, u_i)$ can be calculated with

$$\phi(p, l, u) = p + \ln(1 + e^{-(1-2u)l}) \tag{3.42}$$

and starts with $PM^{(-1)} = 0$. The exponential-logarithm part of the equation can be approximated by

$$\ln(1 + e^x) \approx \begin{cases} 0 & \text{if} \quad x < 0 \\ x & \text{otherwise} \end{cases} \tag{3.43}$$

which leads to an approximation

$$\tilde{\phi}(p,l,u) = \begin{cases} p \text{ if } u = \frac{1}{2}(1 - sign(l)) \\ p + |l| \text{ otherwise} \end{cases} \tag{3.44}$$

for equation (3.42). All paths are sorted by this path metric. Every time paths need to be pruned only the L paths with smallest path metric survive.

3.4.3. Other possible decoder strategies

Other sophisticated decoder strategies were proposed in several papers. Belief Propagation (BP) has been explored as an alternative to SC decoding [GQGiFS14]. It offers the potential for a fully parallelized decoder. Though it has significantly higher complexity than SC decoding and SC decoding strategies offer similar or better error-correction performance.

Another decoding strategy based on SC decoding is SC Stack decoding [NC12]. In contrast to SCL which employs a breadth-first strategy it uses a depth-first strategy.

4. Implementation

A fast polar code implementation in a Software Radio environment enables users to experiment with polar codes in possible usage scenarios and conduct tests for future use. Software Radio can be used as a technique to accelerate research and development efforts. Greater flexibility, easier implementation and better adaptability offer improvements in all parts of a radio system development process. Also, users may benefit from easier future system upgrades and improved maintainability.

This chapter will introduce the goals of the polar code implementation in GNU Radio. Major design decisions will be discussed. The implementation itself in GNU Radio will be presented with its different parts.

4.1. Objectives

The open source Software Radio framework GNU Radio shall be used to implement an encoder, a decoder and channel construction algorithms for polar codes. GNU Radio offers the FECAPI for easy integration of channel codes which shall be used. A Python implementation shall serve as a reference for a C++ implementation of encoders and decoders. Afterwards a generic C++ implementation serves as a performance reference for algorithmically and hardware optimized versions. Modern Central Processing Units (CPUs) offer vector extensions for faster Single Instruction Multiple Data (SIMD) execution. Though, they are hardware architecture specific. GNU Radio's subproject Vector-Optimized Library of Kernels (VOLK) focuses on providing a common interface for different architectures. In the x86 domain, Streaming SIMD Extensions (SSE) and its successor Advanced Vector Extensions (AVX) provide a rich set of vector optimized functions. The functions crucial to runtime of polar codes shall be implemented in VOLK.

4.2. GNU Radio

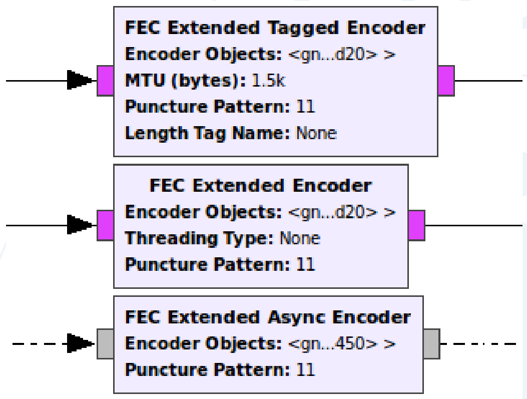
GNU Radio is an open source software-defined radio framework. It is designed around fast and flexible Digital Signal Processing (DSP). It provides an infrastructure to run multi-threaded applications, so-called flowgraphs. Flowgraphs consist of blocks which may be considered as threads of an application. GNU Radio's scheduler takes care of data passing between blocks and all other concerns that may arise from a multi-threaded environment. Users may focus on the specifics of their project in order to compose a flowgraph which contains all the processing steps necessary for their application.

GNU Radio is available as open source. This implies that users may use and alter it in order to tailor it to their needs as long as the terms of the GPL are respected. Also, users benefit from developers who contribute their work and developers benefit from users testing new software.

4.2.1. FECAPI

GNU Radio provides a specialized Application Program Interface (API) for channel coding which is referred to as *FECAPI*. It aims at separating the implementation of a specific encoder and decoder from its integration into a flowgraph. The goal is to be able to use a code implementation in different types of flowgraphs. Three different kinds of flowgraphs are supported as shown with encoders in Fig. 4.1a. They support stream-based, tagged-stream-based and message-based flowgraphs.

In Fig. 4.1b FECAPI polar code encoder and decoder variables are depicted. They may be configured in GNU Radio Companion (GRC) and then passed to the corresponding encoder or decoder skeleton block by their ID. The encoder and decoder blocks all have a parameter *Encoder Objects* and *Decoder Objects* respectively which holds that reference. The implementation presented in this chapter aims at seamless integration into this API.

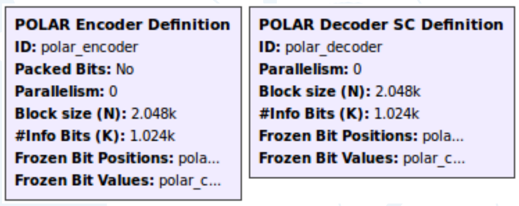


FEC Extended Tagged Encoder
Encoder Objects: <gn...d20> >
MTU (bytes): 1.5k
Puncture Pattern: 11
Length Tag Name: None

FEC Extended Encoder
Encoder Objects: <gn...d20> >
Threading Type: None
Puncture Pattern: 11

FEC Extended Async Encoder
Encoder Objects: <gn...450> >
Puncture Pattern: 11

(a) FECAPI encoders



POLAR Encoder Definition
ID: polar_encoder
Packed Bits: No
Parallelism: 0
Block size (N): 2.048k
#Info Bits (K): 1.024k
Frozen Bit Positions: pola...
Frozen Bit Values: polar_c...

POLAR Decoder SC Definition
ID: polar_decoder
Parallelism: 0
Block size (N): 2.048k
#Info Bits (K): 1.024k
Frozen Bit Positions: pola...
Frozen Bit Values: polar_c...

(b) GRC variable definitions for Polar code encoder and decoder

Figure 4.1.: GRC representation of FECAPI parts

4.2.2. VOLK

The Vector-Optimized Library of Kernels (VOLK) was developed as part of GNU Radio. Recently it was released as its own subproject [gnu15b]. The project objective is to provide a hardware abstraction layer between different Single Instruction Multiple Data (SIMD) architectures and user calls. The SIMD extensions impose additional requirements on array alignment. The objective is to abstract from hardware specifics and alignment requirements which is achieved through a kernel for every function. A kernel consists of a generic implementation which serves as a reference and fall-back in case no SIMD architecture is available. All architecture-specific implementations are tested against the generic implementation. The kernel test infrastructure complements VOLK's features.

Depending on the intended function and the available SIMD extensions several different kernel implementations exist. For the x86 family of CPUs new features were added since SIMD was first introduced. It started with MMX, went through several versions of Streaming SIMD Extensions (SSE) and the newest available SIMD extensions are Advanced Vector Extensions (AVX) and AVX2 respectively. The ARM CPU architecture family received its own set of vector extensions which is called NEON. The newly created kernels for polar codes are merged into mainline VOLK [gnu15c].

4.3. Code structure

The described polar code implementation is merged into mainline GNU Radio [git15]. It consists of encoders, decoders and tools for channel construction which are integrated into GNU Radio's *gr-fec* module. Channel construction code resides in the Python folders of *gr-fec* together with the Python test code infrastructure. C++ source code is scattered over the intended *gr-fec* subfolders. In Fig. 4.2 a Unified Modeling Language (UML) class diagram with all the classes implemented in C++ is shown. *generic_encoder* and *generic_decoder* are provided by FECAPI. All codes inherit from those two base classes which provide a common interface which the FECAPI block infrastructure expects. Most notably are the listed public methods. Though it should be mentioned that this diagram is incomplete in terms of class members.

polar_common holds configuration information like the members given as an example. *polar_encoder* and *polar_encoder_systematic* implement the actual encoders. The decoders share more information and thus *polar_encoder_common* is introduced. Again the actual implementations for the decoders are derived from this base class.

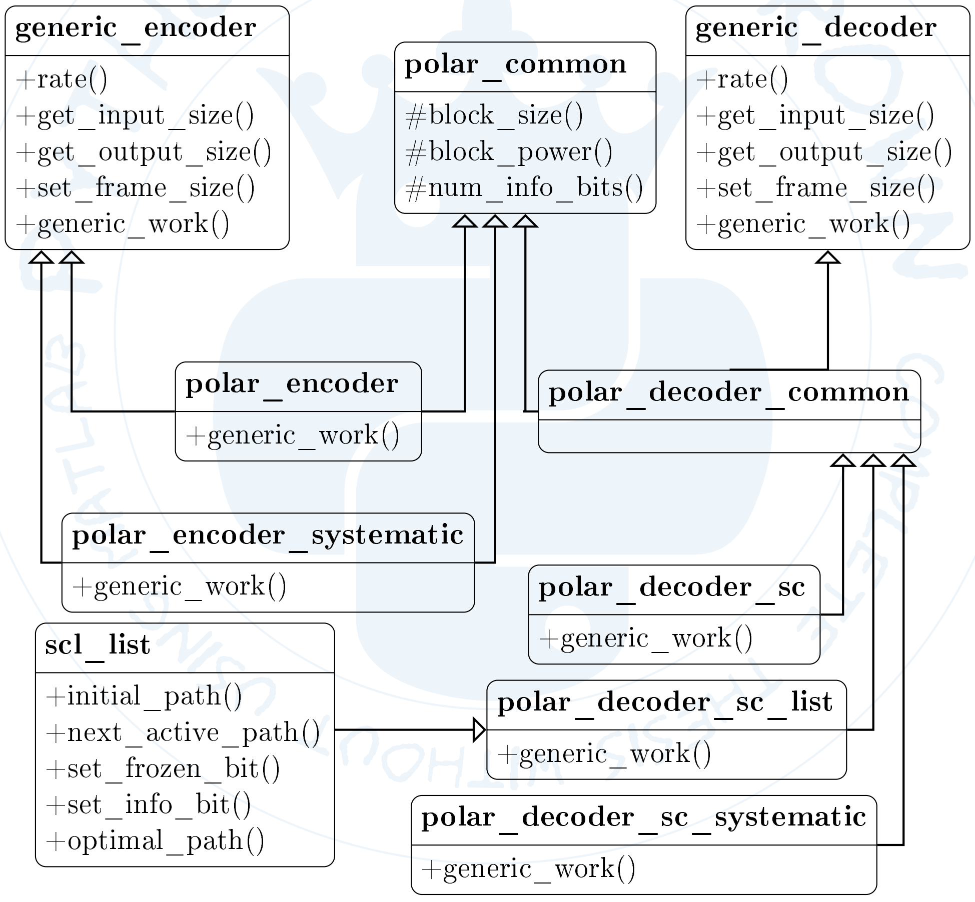


Figure 4.2.: GNU Radio Polar code class diagram

The *polar_decoder_sc_list* holds a *scl_list* object which manages the internals of the path list. Its public API provides access to the paths it manages. While *initial_path* resets the list for a new decoder pass, *optimal_path* returns the best path after decoding has completed. Internally the list takes care of copy and write operations on the paths.

FECAPI's public interface requires every code variable to define five public members. While *get_input_size* and *get_output_size* are self-explanatory, *rate* returns a value for the actual code rate R. The *rate*'s return value is used to adjust the input-output rate of the encoder and decoder skeleton blocks. *set_frame_size* is intended to adjust frame size on-the-fly but most codes do not allow for that feature. A code object is called to encode or decode a frame via *generic_work*. Thus the actual code implementations indicate that by overriding this method. Further details on the FECAPI may be found in the corresponding documentation [gnu15a] and directly in source code at [git15].

4.4. New features

Polar codes consist of three different parts, encoders, decoders and channel construction. In order to make them available in GNU Radio they are implemented and integrated into the given infrastructure. This includes the unit tests, use of the FECAPI and VOLK kernel implementations.

4.4.1. Rapid Prototyping with Python

GNU Radio established a code structure which adds unit tests for each block. This should also be the case for all newly implemented polar code blocks. In order to be usable with the FECAPI, encoder and decoder need to be variables. These variables are dropped into skeleton blocks in order to use them in flowgraphs and tests.

C++ is a very error-prone programming language. Its advantage lies in speed and being close to hardware. Python is a high-level programming language with a lot of convenience features and easier-to-understand source code. This comes at the price of considerable overhead. GNU Radio separates runtime critical tasks like DSP in C++ from flowgraph setup and control and unit tests in Python.

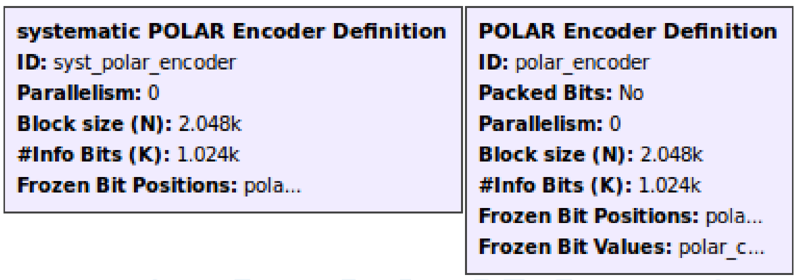
Python is used as a rapid prototyping environment to have a first implementation which is easy to understand and verify by reading source code. It is, however, not optimized for speed but for reader-friendliness. A C++ implementation is compared the Python one in order to verify correct runtime behavior and to improve application quality. It is crucial to implement those tests properly or their benefits are wasted. During later stages in development it is easier to verify if code changes did break parts of the system.

During development such a Python prototype was implemented and then the C++ encoders and decoders. Especially during refactoring in order to clean up source code after all features are implemented the test code infrastructure proves to be very valuable.

4.4.2. Encoder

The polar encoders promise low complexity and thus high throughput. The implemented polar encoders should leverage this property because this frees up resources for other parts of a communication system. On the other hand it is desirable to have as much flexibility available as possible. This implies that ideally only the restrictions imposed by theory restrict configuration options.

A systematic and a non-systematic encoder are implemented as shown in Fig. 4.3 with their GRC variable definitions. The common configuration options for both variables are *Block size*, *#Info Bits* and *Frozen Bit Positions*. Internally they are used by the *polar_common* class to prepare them for fast encoding. The *Parallelism* option is part of FECAPI.



systematic POLAR Encoder Definition
ID: syst_polar_encoder
Parallelism: 0
Block size (N): 2.048k
#Info Bits (K): 1.024k
Frozen Bit Positions: pola...

POLAR Encoder Definition
ID: polar_encoder
Packed Bits: No
Parallelism: 0
Block size (N): 2.048k
#Info Bits (K): 1.024k
Frozen Bit Positions: pola...
Frozen Bit Values: polar_c...

Figure 4.3.: FECAPI Polar encoder variables

The non-systematic encoder definition offers two additional arguments *Packed Bits* and *Frozen Bit Values*. The *Frozen Bit Values* field may be left empty and the encoder will just use an all-zero frozen bit word instead. The systematic encoder definition enforces this behavior which may enable a compiler to boost performance. *Packed Bits* presents a switch to choose between two different encoders which also effect the interface. In packed-bit mode a generic bit shift encoder is used which operates on packed bytes. In the context of GNU Radio packed bytes refers to the fact that each bit in a byte represents a value as opposed to unpacked bytes where only the least-significant-bit represents a value.

Systematic encoder and non-systematic encoder use the same functions internally in case of unpacked mode. The theoretical differences between the encoders were discussed in Section 3.2.6. While the non-systematic encoder calls the VOLK encoder function once to encode one codeword, the systematic encoder does so twice. The systematic polar encoder resets all frozen bit positions to their initial value between the two calls. The systematic polar encoder implementation omits a bit-reversal for the output vector after the second VOLK encoder call. It would be necessary to strictly follow theory here but the decoder would only have to do an additional bit-reversal operation as well. This lead to the design decision to omit this operation.

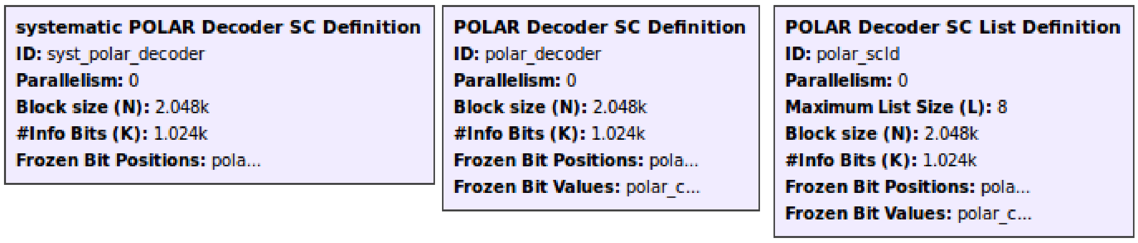
The code parts critical to runtime are implemented in VOLK kernels using SIMD extensions. The encoder variables hold configuration information and provide the connection to FECAPI. Two new VOLK kernels are implemented *volk_8u_x3_encodepolar_8u_x2* and *volk_8u_x2_encodeframepolar_8u_x2*. The former uses a frame and a frozen bit word to encode them into a codeword. The latter expects a source word to encode and returns a codeword as well. This separation is necessary in order to support systematic encoders and decoders which need an additional encoder pass before they finish their respective operations.

Several SIMD instructions from different versions of SSE are required for the encoder kernel. Most importantly *_mm_shuffle_epi8* which reorganizes bytes in a 128bit register and was only introduced with SSSE3. Compared to most other VOLK kernels the polar encoder definitions are rather complex. This leads to several pages of code in one function if they were not split into multiple ones. Originally, VOLK does not support this for a number of reasons. Some work and effort already went into this issue in order to ease the situation. This includes newly introduced headers for architecture specific functions which are also implemented.

4.4.3. Decoders

Decoders are often a bottleneck in a communication system. Polar codes offer low complexity decoders which can be used to improve this situation and were introduced in Section 3.4. All implemented polar decoders rely on the SC decoding strategy which can provide low complexity.

Fig. 4.4 shows the implemented decoders. The systematic and non-systematic SC decoder implementations differ only slightly. Firstly they both use a SC decoder strategy which is described in Section 3.4. The newly implemented *volk_32f_8u_polarbutterfly_32f* VOLK kernel facilitates calculation of all LLRs necessary for the next bit decision in a SC decoder. The final LLR for a bit is obtained and subsequently a bit decision is made. This bit information is then used for the following LLR calculations and bit decisions until all bits of a source word are estimated. While the non-systematic decoder just extracts a frame estimate from this result, the systematic decoder employs another encoder pass on that intermediate result before it extracts a frame estimate.


systematic POLAR Decoder SC Definition
ID: syst_polar_decoder
Parallelism: 0
Block size (N): 2.048k
#Info Bits (K): 1.024k
Frozen Bit Positions: pola...

POLAR Decoder SC Definition
ID: polar_decoder
Parallelism: 0
Block size (N): 2.048k
#Info Bits (K): 1.024k
Frozen Bit Positions: pola...
Frozen Bit Values: polar_c...

POLAR Decoder SC List Definition
ID: polar_scld
Parallelism: 0
Maximum List Size (L): 8
Block size (N): 2.048k
#Info Bits (K): 1.024k
Frozen Bit Positions: pola...
Frozen Bit Values: polar_c...

Figure 4.4.: FECAPI Polar decoder variables

The SCL decoder extends this process by maintaining a list of decoder paths. One path represents a specific intermediate decoding result. For each bit to be decoded, the LLR for the next bit in each path is calculated. If the decoded bit is at a frozen bit position, this bit is set in each path and the metric is updated according to equation (3.44). Otherwise all paths are split in two representing the possible bit decisions and the path metrics are updated accordingly. If this process grows the list beyond maximum list size L, the paths with the highest path metrics are pruned from the list. List updates and path management are implemented in the *scl_list* class while the *polar_decoder_sc_list* class provides the interface to FECAPI as well as glue code to use the VOLK kernel.

In Section 3.4 SC decoders are introduced and their use of LLRs. LLRs are mostly represented as float values which use 32bit. All versions of SSE operate on 128bit registers which allows to do four float calculations in parallel. AVX introduces 256bit wide registers for float values, thus enabling eight parallel calculations. Values must be loaded into those registers from a consecutive data array which motivates the use natural bit order graphs.

While the polar encoder VOLK kernel cannot benefit from AVX because it does not use floats, the decoder does. With respect to the available hardware and the objective to implement the fastest possible decoder, the VOLK kernel for polar decoders is implemented for AVX. The basis for implementations for other target architectures is readily available. The decision to calculate only LLRs for one bit is motivated by the fact that SCL decoders need to perform list management after each decoded bit.

4.4.4. Channel construction

Polar codes need frozen bit positions $\mathcal{A}_\mathcal{C}$ and values $u_{\mathcal{A}_\mathcal{C}}$ to be completely defined. Those two sets of parameters are calculated for a specific block size and the assumption of a specific channel model with its defining parameters. This defining parameter is usually given as design-SNR (dSNR) and then converted to the needed parameter for the channel model by (3.25).

In case the assumed channel model is a BEC channel construction is feasible to be calculated during flowgraph instantiation with the Bhattacharyya bounds method, compare Section 3.2.2. Other underlying channel models require a more complex channel construction algorithm. This algorithm was introduced in Section 3.2.4 and requires a channel quantization value μ in addition to the other polar code and channel parameters.

Channel construction algorithms are implemented and accessible through two different interfaces. A command-line tool *polar_channel_construction* provides access through a shell. The *POLAR code Configurator* shown in Fig. 4.5 is the other interface specifically designed for use with GRC. The command-line tool prints the calculated synthetic channel values to the console.

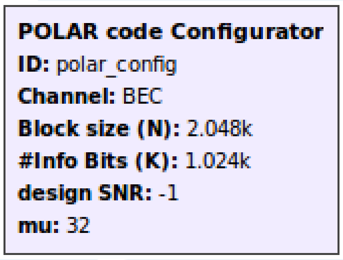

POLAR code Configurator
ID: polar_config
Channel: BEC
Block size (N): 2.048k
#Info Bits (K): 1.024k
design SNR: -1
mu: 32

Figure 4.5.: FECAPI Polar configurator

The *POLAR code Configurator* expects a *#Info Bits (K)* parameter. Internally the synthetic channel Bhattacharyya parameters are calculated and then the k most reliable positions are returned as a set for frozen bit positions $\mathcal{A}_\mathcal{C}$. Along with the frozen bit positions the configurator returns a list of frozen bit values $u_{\mathcal{A}_\mathcal{C}}$ and the design parameters for channel construction in a Python dictionary. These parameters may be used in any polar encoder and decoder afterwards.

The channel construction calculates the parameters with the Bhattacharyya bounds method whenever it is called again. This approach is unfeasible for other channel construction methods. In this case the parameters obtained from channel construction are cached persistently in '$HOME/.gnuradio/polar'. Before a new set of values is calculated the system checks if the chosen parameters are already cached and loads them instead.

All algorithms related to channel construction are implemented in Python and NumPy. This can be seen in contrast to the encoders and decoders which are tweaked for performance and implemented in C++. The motivation to implement channel construction in Python stems from the fact that this is an off-line task which only needs to be performed once. It is thus considered to be more important to make the algorithms easier to understand and less error-prone.

5. Experimental

Polar codes are known to asymptotically achieve channel capacity. It is, however, of major interest how those codes perform under suboptimal configurations because their optimality is only reached for very large block sizes. Channel codes are mostly compared in terms of error correction performance or in other terms their BER.

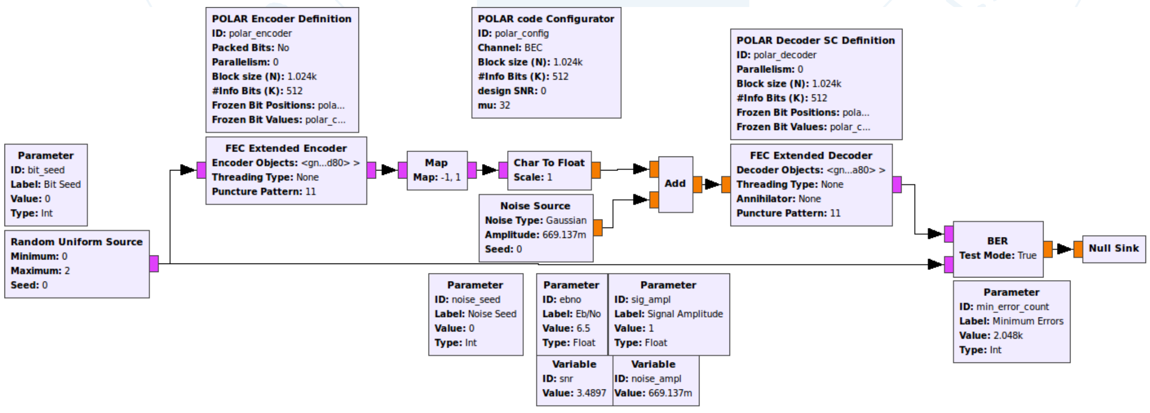
In this chapter a simulation environment is introduced and then used to analyze BER curves by varying different code parameters. This extends to different channels for channel construction and even different implementations. Ido Tal provided his channel construction implementation which is optimized in terms of speed and configuration options. Its results were used in some of the simulations. Unfortunately it is not available as open source and thus other channel construction algorithms are of interest.

Also, polar code performance is compared to state-of-the-art LDPC codes. Especially in comparison to LDPC codes it is interesting to observe throughput because channel coding is often a performance critical task in Software Radio.

5.1. Simulation environment

It is important to conduct simulations carefully, otherwise results may be biased and incomparable. In Fig. 5.1 the simulation environment is shown. The extended encoder and decoder skeleton blocks hold encoder and decoder variables respectively. Simulations are intended to be performed for different codes and also different decoders shall be compared.

The flowgraph in Fig. 5.1 produces new uniformly random bits with its *Random Uniform Source*. This block was implemented for the following simulations but is also already merged into GNU Radio mainline. Together with *Noise Source* it uses the same Boost Random library [boo10] to generate new random values.


POLAR Encoder Definition
ID: polar_encoder
Packed Bits: No
Parallelism: 0
Block size (N): 1.024k
#Info Bits (K): 512
Frozen Bit Positions: pola...
Frozen Bit Values: polar_c...

POLAR code Configurator
ID: polar_config
Channel: BEC
Block size (N): 1.024k
#Info Bits (K): 512
design SNR: 0
mu: 32

POLAR Decoder SC Definition
ID: polar_decoder
Parallelism: 0
Block size (N): 1.024k
#Info Bits (K): 512
Frozen Bit Positions: pola...
Frozen Bit Values: polar_c...

Parameter
ID: bit_seed
Label: Bit Seed
Value: 0
Type: Int

FEC Extended Encoder
Encoder Objects: <gn...d80> >
Threading Type: None
Puncture Pattern: 11

Map
Map: -1, 1

Char To Float
Scale: 1

Add

FEC Extended Decoder
Decoder Objects: <gn...a80> >
Threading Type: None
Annihilator: None
Puncture Pattern: 11

Noise Source
Noise Type: Gaussian
Amplitude: 669.137m
Seed: 0

Random Uniform Source
Minimum: 0
Maximum: 2
Seed: 0

BER
Test Mode: True

Null Sink

Parameter
ID: noise_seed
Label: Noise Seed
Value: 0
Type: Int

Parameter
ID: ebno
Label: Eb/No
Value: 6.5
Type: Float

Parameter
ID: sig_ampl
Label: Signal Amplitude
Value: 1
Type: Float

Parameter
ID: min_error_count
Label: Minimum Errors
Value: 2.048k
Type: Int

Variable
ID: snr
Value: 3.4897

Variable
ID: noise_ampl
Value: 669.137m

Figure 5.1.: GNU Radio simulation environment

In order to make codes more comparable, the actual code rate must be considered. *Noise Source* expects a noise amplitude A_N which is calculated from $\frac{E_b}{N_0}$ and code rate R

$$A_N = A_S \cdot 10^{\frac{E_b}{N_0}/20} \cdot R^2 \tag{5.1}$$

where A_S is the signal amplitude. The equation is a reformulation from

$$\frac{S}{N} = \frac{E_b}{N_0} \cdot \frac{R}{B} \tag{5.2}$$

where the system bandwidth B is assumed to be 1 constantly and will be pruned from the equation. This equation is then reformulated for logarithmic values

$$\frac{S}{N} = \frac{E_b}{N_0} + 10 \log R. \tag{5.3}$$

Simulations are running continuously until a minimum amount of errors is reached. This limit is passed on to the *BER* block which returns *WORK_DONE* as soon as the limit is reached and thus shuts down the flowgraph. A BER can be obtained from the *BER* block by querying it for its *error count* and *item count*. A public getter to the blocks error count variable was also added for the purpose of the following simulations.

5.2. Impact of block size on BER

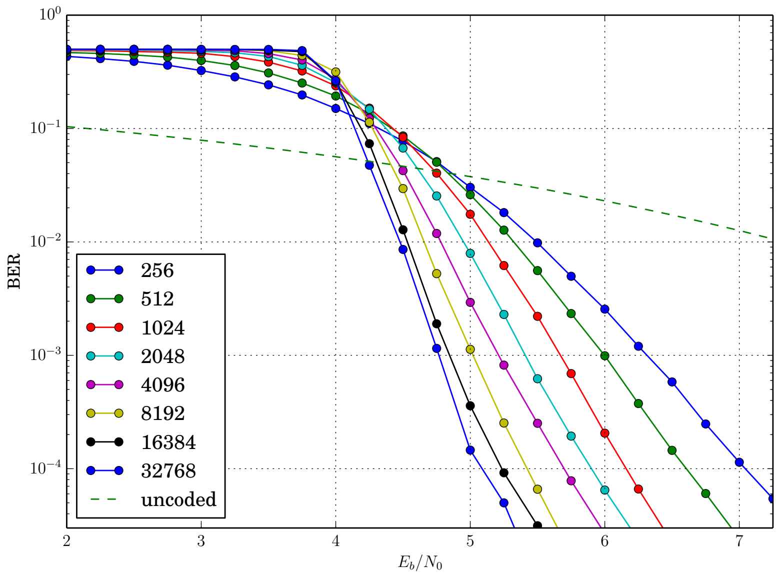


Figure 5.2.: block size N comparison

Polar codes exploit the channel polarization effect which increases with block size N, Section 3.2.3. A larger block size requires that more symbols need to be received before a codeword can be decoded which increases latency. Depending on the application different code sizes may be favorable and their influence on BER need to be known beforehand in order to make sane design decisions. The influence of block size and thus polarization on BER is analyzed.

In Fig. 5.2 a variety of block sizes are compared to each other. All simulations were conducted with a code rate $R = 0.5$ and an AWGN channel with a dSNR of 0.0 dB, and $\mu = 256$ constructed with Ido Tal's channel construction implementation.

It is observable that higher block sizes correspond to a steeper decent in BERs. Also, uncoded transmission yields better performance in case $\frac{E_b}{N_0} < 4.0$ dB and polar coded transmission performs better above 5.0 dB. A larger block size leads to stronger polarization which in turn lowers BER. During system design a suitable trade-off between latency and BER needs to be chosen.

5.3. Impact of code rate on BER

The code rate defines how much a code expands on the required bit rate or how much it lowers information bit rate compared to the raw system bit rate. While low SNR environments make it favorable to use a lower code rate and gain higher reliability, this might be a waste of resources in other cases.

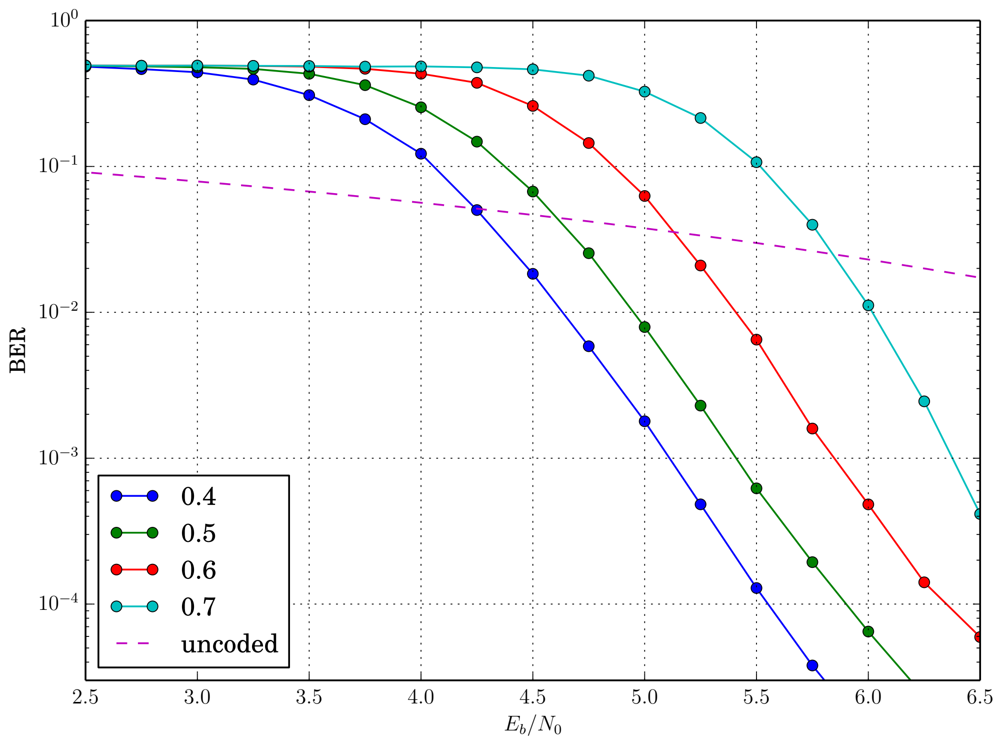


Figure 5.3.: code rate R comparison

In Fig. 5.3 different code rates R of a $N = 2048$ polar code can be observed. The polar code was obtained with a dSNR of 0.0 dB and $\mu = 256$ with Ido Tal's channel construction algorithm. A higher code rate increases BER at the same $\frac{E_b}{N_0}$.

After channel construction is performed for polar codes they exhibit a degree of freedom because their code rate can be chosen freely. A communication system may leverage this flexibility in order to improve efficient use of available resources.

5.4. Impact of AWGN channel design parameters on BER

In Section 3.2.4 an efficient algorithm for channel construction is introduced. A higher value for the value μ promises more accuracy for channel construction and thus better results because it allows for less quantization. It is, however, preferable to use a small value for μ because that reduces complexity and speeds up channel construction.

Fig. 5.4 shows the results for a wide range of μ values. The other parameters for this polar code are block size $N = 8192$ and code rate $R = 0.5$ and dSNR $0.0dB$ with Ido Tal's channel construction algorithm assuming an underlying AWGN channel model. The BER curve for $\mu = 4$ differs significantly from the others. All the other curves are closer together. It is, however, observable that for $\mu = 16$ the BER curve indicates a higher BER for higher values of $\frac{E_b}{N_0}$. For $\mu = 32$ and $\mu = 256$ the BER curves are so close that one can not come to a sound conclusion about differences.

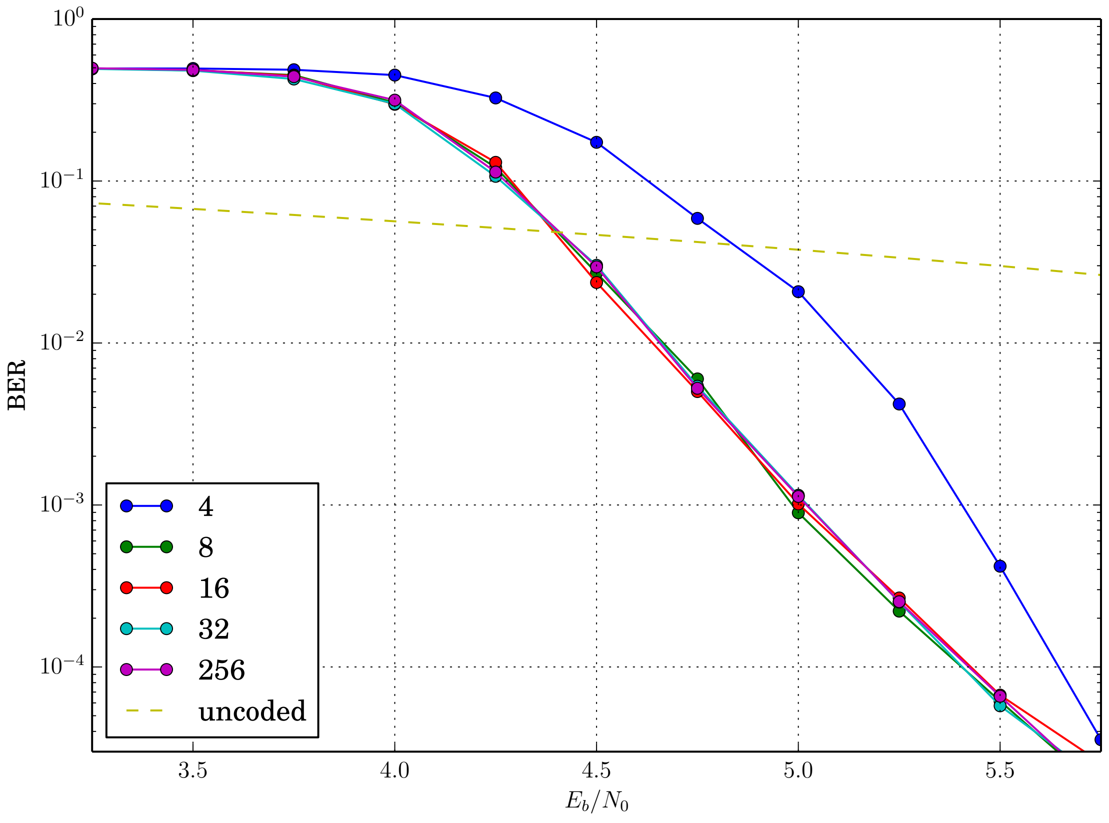


Figure 5.4.: μ comparison

This result is especially interesting because for $\mu = 32$ the channel construction complexity is rather low. It takes significantly more time to construct channels with $\mu = 256$, while the results indicate that this is not necessary because one would not expect measurable improvements.

5.5. Code optimization for specific SNR environments

Polar codes may be adopted for specific use cases. Mainly the underlying channel properties may be altered. Though, channel conditions are unknown at channel construction time. It is thus advisable to carefully choose the design parameters beforehand. The goal is to find a dSNR for AWGN and BEC channel construction for which BERs are minimized. Fig. 5.5 shows BER curves for a wide range of dSNRs for an underlying BEC. For this simulation the fixed parameters are block size $N = 2048$ and code rate $R = 0.5$.

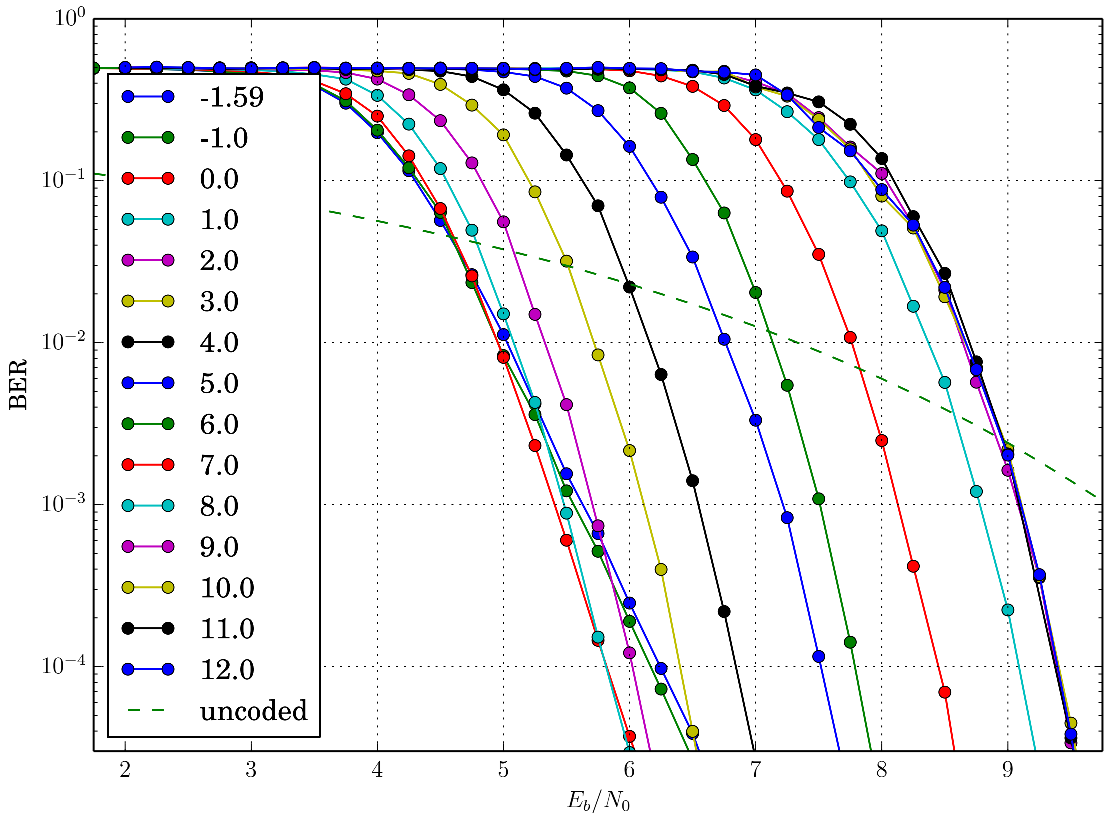


Figure 5.5.: BEC dSNR comparison

It can be observed that higher dSNRs tend to produce higher BERs. Though a dSNR of 9.0 dB or higher does not continue to impair BER any further. A smaller dSNR results in a flatter BER curve. Small dSNRs exhibit special behavior regarding BER curves. This is discussed in the Section 5.6 in conjunction with Fig. 5.7.

The BER curves presented in this Section indicate that a dSNR > 3.0 dB does not expose any beneficial features in terms of error correction performance. This result motivates the dSNR chosen for the other simulations.

5.6. Comparison of channel construction algorithms

Three different channel construction algorithms are available for the following simulation. Ido Tal's channel construction, GNU Radio's channel construction and the Bhattacharyya BEC method, or Bhattacharyya bounds method. After a BER performance evaluation of all three, comparable results were chosen in order to compare them in Fig. 5.6. The chosen parameters are block size $N = 2048$ and code rate $R = 0.5$ and $\mu = 64$ where applicable.

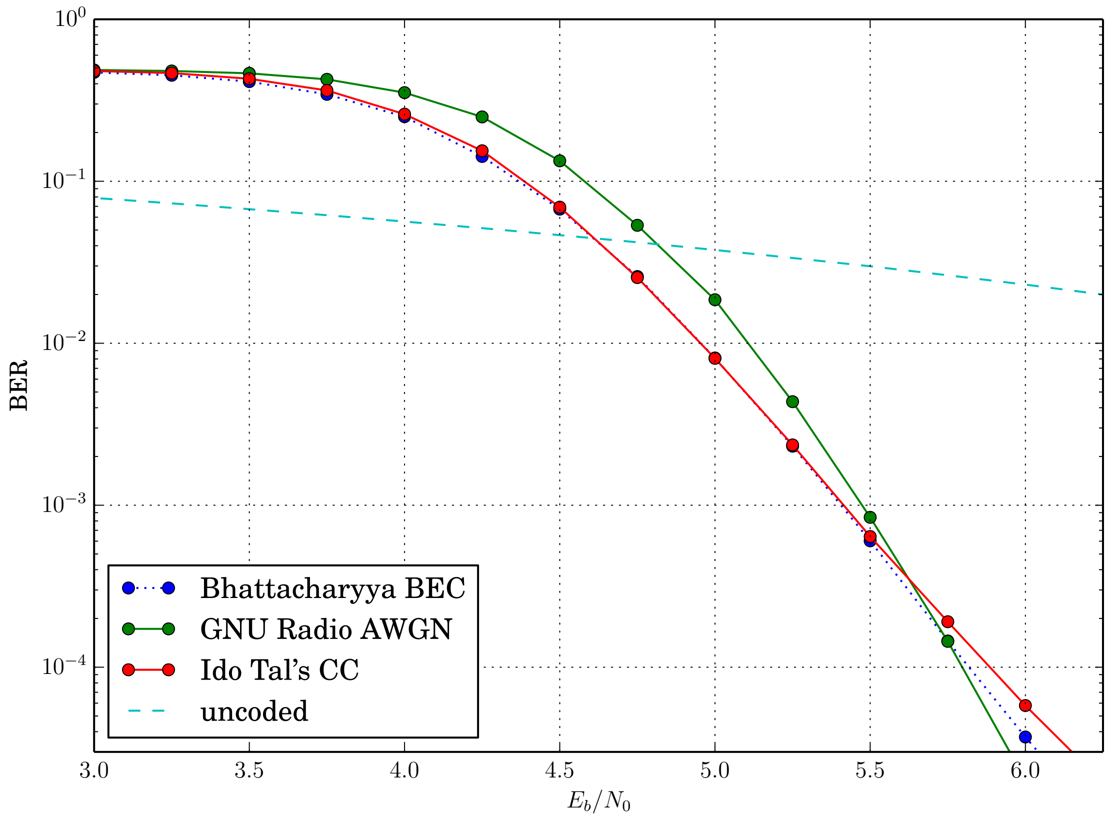


Figure 5.6.: BER curve comparison for channel construction algorithms

GNU Radio's channel construction implementation yields worse error correction performance then the other channel construction algorithms. It is unclear if the observed performance degradation is due to software bugs in the implementation or if simplifications result in a degraded result.

The results for Ido Tal's channel construction algorithm and the Bhattacharyya BEC method are very close. A more detailed comparison for different dSNRs is shown in Fig. 5.7. Channel construction with Bhattacharyya bounds method for BEC improves with a lower dSNR, though the curves get flatter. Ido Tal's channel construction algorithm produces steeper curves for higher dSNRs.

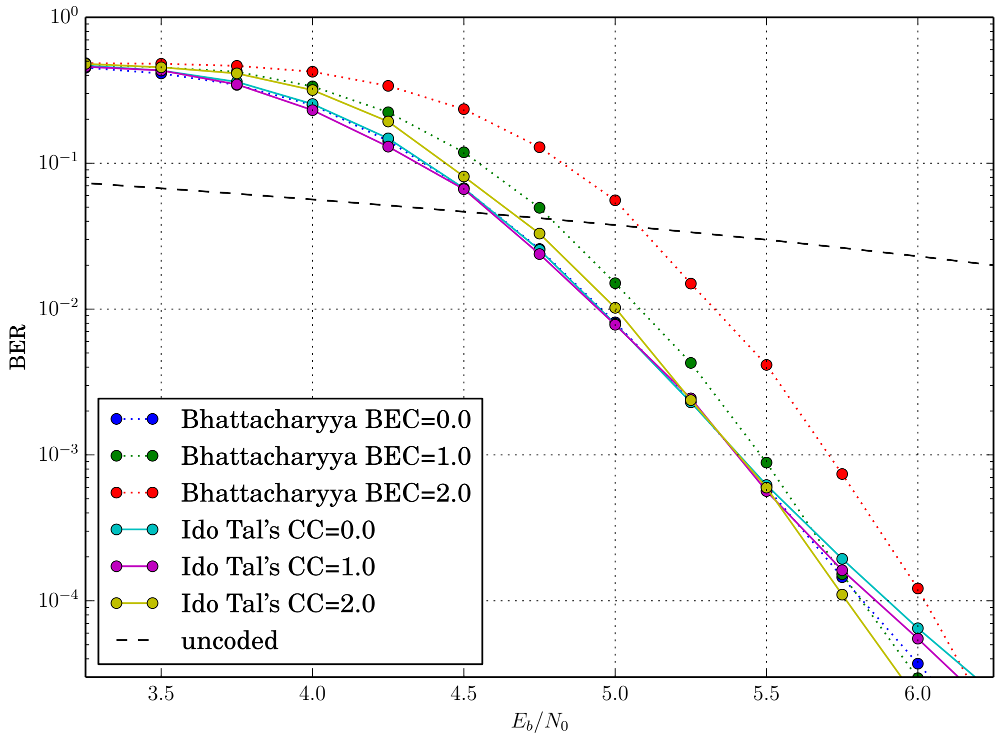


Figure 5.7.: Bhattacharyya BEC vs. Ido Tal's channel construction for AWGN

The results suggest to use Bhattacharyya bounds method with a dSNR of 0.0 dB or to carefully choose the design parameters in case Ido Tal's implementation is used. In case very low BER regions are targeted Ido Tal's implementation with a dSNR of 2.0 dB promises superior results. All other cases suggest using the Bhattacharyya bounds method because it is on par or even better then the other channel constructions.

5.7. Systematic polar codes

Systematic polar codes need a second encoder and decoder pass as introduced in Section 3.2.6. This increases the computational complexity and is only justified if the error correction performance is improved. A range of different block sizes for both systematic and non-systematic polar codes is shown in Fig. 5.8a. The chosen parameters are a code rate $R = 0.5$ with Ido Tal's channel construction algorithm with $\mu = 256$ and an underlying AWGN channel with a dSNR of 0.0 dB.

It can be observed that systematic polar codes outperform their non-systematic counterparts with twice the block size. Systematic polar codes do not tend towards a BER of 0.5 for lower $\frac{E_b}{N_0}$ but a lower value.

The results indicate that systematic polar codes should always be favored in comparison to their non-systematic counterparts. This also motivates their use when polar codes are compared to other error correcting codes.

Systematic polar codes show superior behavior for BER compared to non-systematic polar

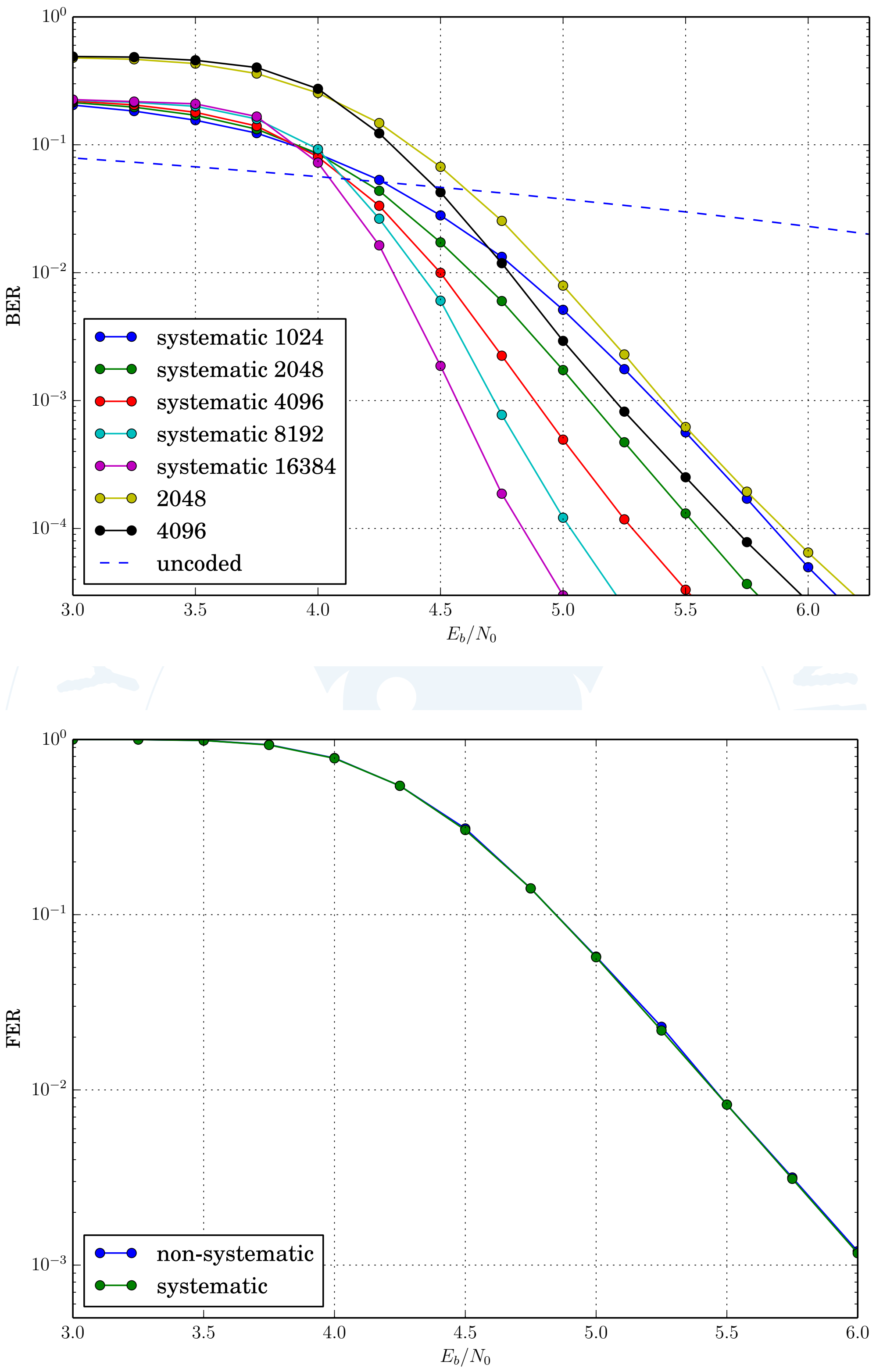


Figure 5.8.: BER and FER curve comparison for systematic and non-systematic polar codes

codes. In [Ari11] it was noted that systematic and non-systematic polar codes show the same results when compared by FER. Fig. 5.8b shows a FER comparison between systematic and non-systematic polar codes with a block size $N = 2048$ and code rate $R = 0.5$. A frame is considered erroneous if at least one bit is erroneous. This also explains the fact that the FER may reach 1.0 in contrast to the BER which has its maximum at 0.5.

5.8. Comparison of maximum list sizes in SCL decoders

The SCL decoder strategy does not perform immediate bit decisions but follows all possible code paths up to a given maximum. This maximum is the maximum list size L which is also a trade-off between error correction performance and complexity. Fig. 5.9 shows the simulation results for a polar code with block size $N = 2048$, code rate $R = 0.5$ and constructed with the Bhattacharyya bounds algorithm for a dSNR of $0.0dB$.

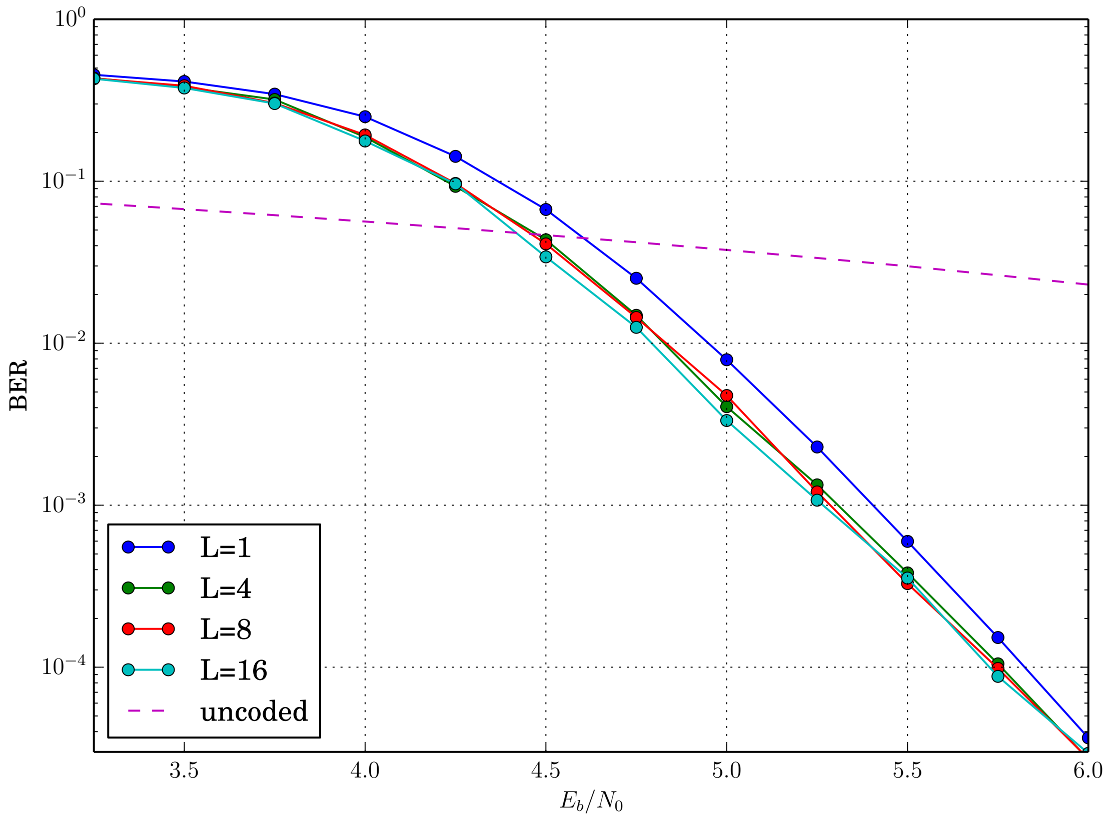


Figure 5.9.: BER curves for different decoder list sizes

The standard SC decoder exhibits the highest BERs. A higher maximum list size tends to deliver better BERs but this effect diminishes as the curves tend towards lower BERs. Small maximum list sizes promise better BER performance. Increasing this list size to a higher value is not advisable, though, because the improvements shrink considerably.

5.9. Polar codes in perspective

Polar codes are asymptotically capacity achieving. But the need for packetized transmissions and low latency requirements limit the usage of polar codes with large block sizes. In this section polar codes are compared to a state-of-the-art LDPC code.

5.9.1. LDPC codes

LDPC codes are block codes which are defined by sparse matrices which are available in a format called AList files [HS15]. Decoders for LDPC codes use BP to decode codewords. A decoder updates the bits of a received codeword iteratively in order to use the transmitted redundancy to correct bit errors. One iteration a decoder performs corresponds to N updates where N is the codeword size. After each iteration a check if errors persist is performed until either a correct codeword is detected or the maximum number of iterations is reached.

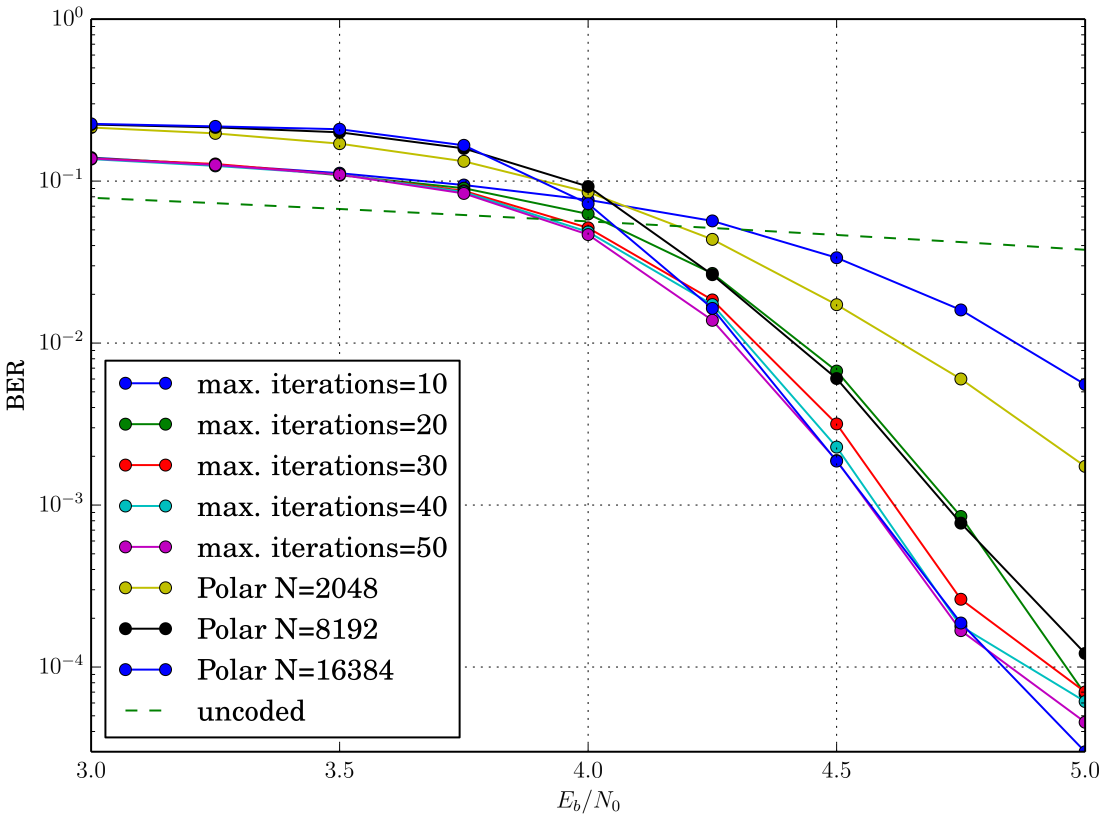


Figure 5.10.: Polar codes and LDPC codes in comparison

LDPC codes are considered to be state-of-the-art channel codes. Polar codes need to compete with them in terms of error correction performance. Fig. 5.10 shows the $(2304, 1152)$ LDPC code which is used in the WiMAX standard with different maximum iterations and systematic polar codes of different block size. The code rate is fixed to $R = 0.5$ for all codes.

Polar code channel construction was performed by Ido Tal's channel construction with a dSNR of $0.0dB$ and $\mu = 256$.

It can be observed that the polar code with smallest block size performance better compared to the LDPC code with 10 maximum iterations. More iterations and thus additional computational complexity improve LDPC performance. The curves for LDPC codes converge for higher maximum iterations. A polar code with block size $N = 16384$ can outperform all other codes shown in Fig. 5.10.

The DVB-S2 standard uses LDPCs codes with block size 16200 and 64800. It would be interesting to compare polar codes with approximately the same block size to them. Unfortunately, the available LDPC implementation is too slow to conduct meaningful simulations with bigger block sizes.

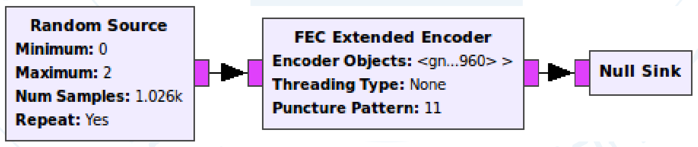
5.10. Throughput

Polar codes have low complexity encoders and decoders. They were optimized for speed which may be measured in terms of throughput. GNU Radio comes with ControlPort. It enables monitoring of flowgraphs remotely and offers a lot of standardized performance metrics. One of the standardized performance metrics is an average item throughput. The average throughput $R_{T/P} = \frac{N_{tot}}{T_{mon}}$ is calculated with the total number of output items N_{tot} a block already processed divided by the time it is monitored T_{mon}.

Throughput measurements are hardware dependent. Furthermore the operating system introduces uncertainties because the scheduler may allocate more or less resources to the process under test. All tests are performed on the same machine with Ubuntu 14.04.3 LTS 64bit with 16 GB RAM and an Intel Core i7-3630QM CPU.

All encoder and decoder implementations run in one thread on one core. Other operations can be performed in parallel on the same machine, making it feasible to run the whole transmitter and or receiver chain on it. This is an important distinction from possible multi-core implementations which may consume all resources on a given machine.

The measurements are conducted with a stripped down flowgraph shown in Fig. 5.11. Only the encoder and decoder skeleton blocks are swapped out for the different measurements.


Random Source
Minimum: 0
Maximum: 2
Num Samples: 1.026k
Repeat: Yes

FEC Extended Encoder
Encoder Objects: <gn...960> >
Threading Type: None
Puncture Pattern: 11

Figure 5.11.: Example flowgraph for throughput measurements

5.10.1. Polar code encoders

Three different polar code encoders are available. The unpacked non-systematic encoder only needs one encoder pass, while the systematic encoder needs two plus a frozen bit reset. Also, a comparison between packed and unpacked encoders should reveal which one should be the encoder of choice.

Encoder	$R_{T/P,info}$	$R_{T/P,code}$
non-syst. unpacked	275.4	550.4
non-syst. packed	135.5	271.0
syst. unpacked	156.6	313.2

(a) Polar encoders with $N = 2048$

block size N	$R_{T/P,info}$	$R_{T/P,code}$
1024	286.8	573.5
2048	275.4	550.4
4096	260.1	521.1

(b) Non-systematic unpacked polar encoder

Table 5.1.: Polar encoder throughput $R_{T/P}$ measurements in $Mbit/s$

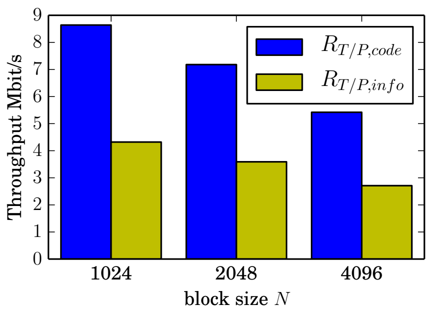
The block size $N = 2^m$ may also influence throughput because the encoders are said to have complexity $\mathcal{O}(N \log_2 N)$. In order to encode the same number of bits, this complexity would add 2^{m+1} additional runtime when comparing block size 2^m with 2^{m+1}.

Tables 5.1 show the throughput results in $Mbit/s$ for different setups. In Table 5.1b the throughput impact of a block size increase is shown. A shorter block size results in a higher throughput as expected. The performance hit is $\sim 4\%$ when the block size is increased from $N = 1024$ to $N = 2048$. Another block size increase results in an additional $\sim 5\%$ throughput drop.

Table 5.1a shows a comparison between different polar code encoders for a block size $N = 2048$. The non-systematic unpacked encoder exhibits the highest throughput and the systematic polar encoder is faster than the packed non-systematic one. This result suggests that the SIMD optimized VOLK kernels for the unpacked encoders are always preferable compared to a the generic one. The systematic encoder requires two calls for the VOLK kernel for every encoder pass. Throughput does not drop below half of that of the non-systematic encoder which suggests that the non-systematic encoder is bounded by some other condition.

5.10.2. Polar code decoders

Polar decoders are expected to be the bottleneck in a communication system. Three different polar decoders exist with different complexity. The non-systematic polar decoder can be regarded as the baseline all other decoders are compared against. The systematic polar decoder adds overhead with its additional encoder pass after decoding. The SCL decoder effectively decodes multiple different paths simultaneously which adds calculations to the decoding process.



(a) Non-systematic polar decoder

Decoder	$R_{T/P,info}$	$R_{T/P,code}$
systematic SC	3.59	7.18
non-systematic SC	3.59	7.18
SCL $L = 2$	0.2139	0.4278
SCL $L = 4$	0.0850	0.1700
SCL $L = 8$	0.0411	0.0822
SCL $L = 16$	0.0191	0.0382

(b) Polar decoders with $N = 2048$

Figure 5.12.: Polar decoder throughput $R_{T/P}$ measurements in $Mbit/s$

Encoder	$R_{T/P,info}$	$R_{T/P,code}$
LDPC	0.75	1.49
non-syst. polar	275.4	550.4

(a) LDPC and non-systematic encoder

Decoder	$R_{T/P,info}$	$R_{T/P,code}$
non-syst. polar	3.59	7.18
LDPC m.i. 10	0.1177	0.2354
LDPC m.i. 20	0.0608	0.1216
LDPC m.i. 30	0.0408	0.0816
LDPC m.i. 40	0.0305	0.0610
LDPC m.i. 50	0.0248	0.0496

(b) Non-systematic polar decoder and LDPC decoders with maximum iterations

Table 5.2.: Throughput $R_{T/P}$ measurements in $Mbit/s$

Similar to the encoder throughput measurements the polar decoder measurements swap a decoder skeleton for an encoder skeleton and use the decoder variables. Table 5.12b shows the results for the decoders with different parameters. A comparison between systematic and non-systematic decoders shows that they are on par in terms of throughput. The additional encoder pass in the systematic decoder is fast enough to not impact throughput any further, making the use of systematic polar codes preferable because of their superior BER performance.

The SCL decoder experiences a performance meltdown compared to the other decoders. The smallest possible list size $L = 2$ already leads to a 16-fold throughput decrease. After the list size is quadrupled to $L = 8$ the decoder takes another fivefold throughput hit. This hints at serious performance issues within list management code. The target of most optimization efforts was the SC decoder which the throughput results reflect.

The non-systematic polar encoder suffers from a $\sim 4\%$ performance hit after the block size is increased from $N = 1024$ to $N = 2048$. Whereas the non-systematic polar decoder takes a $\sim 20\%$ performance hit when going through the same block size increase.

5.10.3. Polar codes vs. LDPC codes

Polar codes offer low complexity encoders and decoders together with asymptotically achieving capacity. These features make them compete with state-of-the-art channel codes. GNU Radio offers LDPC codes for channel coding. Their throughput is compared with polar code throughput. The iterative nature of LDPC decoders makes it interesting to compare them with different maximum iterations.

A non-systematic polar code with block size $N = 2048$ is compared to the previously introduced WiMAX LDPC code. The encoders throughput results are shown in Fig. 5.2a. The polar encoder has a ~ 367 times higher throughput than the LDPC encoder. The LDPC decoder with a maximum of 10 iterations is is ~ 30 times slower than the competing polar decoder, compare Table 5.2b. This gap widens to a ~ 144 times slower LDPC decoder if maximum iterations is increased to 50.

Polar codes may not always perform better in terms of error correction performance if compared to LDPC codes but they are orders of magnitude faster which may compensate for potential BER losses.

6. Conclusion

In the channel coding domain the objective is to reach Shannon capacity with low complexity encoders and decoders. This enables users to have higher data rates and to use sparse resources more efficiently. Polar codes are asymptotically capacity achieving codes with low complexity encoders and decoders which makes them suitable for software radio applications.

Polar codes can be designed for a wide range of different channel models. These channel models were introduced along with a common notation used throughout this thesis. The communication system for which polar codes are intended and which parts are to be expected are defined.

The channel polarization effect is exploited for polar codes as shown in Chapter 3. This effect was introduced accompanied by its properties. Channel construction calculates the effect of polarization in order to select the best possible synthetic channels for information transmission. The results from channel construction are then used to parameterize specific polar codes in order to deliver the best possible polar codes. Polar code encoders and decoders have different options how they can be described. This extended to systematic polar codes and SCL decoders. Further discussions focused on interesting properties for code validation and efficient channel construction.

The software radio framework GNU Radio received a polar code implementation for its FECAPI. The features and their usage were explained together with the implementation in Chapter 4. A number of design decisions already affected how the theory of polar codes was presented. Most notably, the decision to use natural bit order instead of bit reversed order for the encoders and decoders influenced how their graphs were presented. Channel construction is required to obtain the best possible error correction performance for polar codes. The respective algorithms needed for channel construction were implemented as well in order to simplify usage of the new polar encoders and decoders. Polar codes have low complexity encoders and decoders, thus optimization can extend on this advantage. The implemented polar encoders and decoders were optimized with SIMD functions in order to leverage modern CPU architectures.

Numerous parameters for polar codes offer a great flexibility in terms of parameterization. The effects of parameterization decisions were evaluated for an optimal set under given constraints in Chapter 5. It was shown that different channel construction algorithms may provide similar performance. The expected polarization effect intensifies at higher block sizes. Also, it was shown that polar codes can outperform state-of-the-art LDPC codes if they are constructed carefully. Furthermore polar codes have orders of magnitude higher throughput than the available LDPC encoder and decoder in GNU Radio.

With this thesis a polar code implementation in GNU Radio was presented. It covers all aspects necessary for their usage. Polar codes yield very good error correction performance which makes them a candidate for future communication systems. Even under suboptimal conditions they show excellent error correction performance. The low complexity of the polar encoder and decoder offer superior throughput. The polar code implementation in

GNU Radio leverages all the benefits in order to offer high throughput while still being usable with any polar code.

6.1. Future work

A Polar code implementation is available as open source software in GNU Radio. This implementation is already optimized for software to a certain extent. Potential users and contributors can extend on the given source in order to improve and customize polar codes to their needs.

The decoder can be reimplemented using 8 bit integers for LLRs instead of floats in order to boost throughput. Several extensions to the original SC decoder are available which further enhance the decoder. The decoder may recognize different kinds of constituent codes to simplify the decoding process. This will lead to the Fast-SSC implementation for polar codes [GSL+15].

The SCL decoder will benefit from optimized list management and eliminating implementation quirks. Other options for more sophisticated decoders exist as well. Investigating the benefits of BP decoders or SC stack decoders is a task at hand.

The original efficient channel construction algorithm is not available to the general public and thus a GNU Radio implementation will be useful. This channel construction implementation can be extended in several ways. Additional channel models may be supported in order to have a more diverse set of options. The available channel model options would benefit from more fine grained configuration options in order to optimize their outcome. Finally, the basic channel construction algorithm may be optimized for speed.

A. Abbreviations

API	Application Program Interface
AWGN	Additive White Gaussian Noise
AVX	Advanced Vector Extensions
BEC	Binary Erasure Channel
BER	Bit-Error-Rate
BDMC	binary DMC
BP	Belief Propagation
BSC	Binary Symmetric Channel
CPU	Central Processing Unit
DMC	Discrete Memoryless Channel
dSNR	design-SNR
DSP	Digital Signal Processing
Fast-SSC	Fast-Simplified SC
FER	Frame Error Rate
FFT	Fast Fourier Transform
GPP	General Purpose Processor
GRC	GNU Radio Companion
LDPC	Low-Density Parity-Check
SIMD	Single Instruction Multiple Data
VOLK	Vector-Optimized Library of Kernels
LR	Likelihood Ratio
LLR	Log Likelihood Ratio
ML	Maximum Likelihood
NRZ	Non-Return-to-Zero

RV	Random Variable
symmetric-BDMC	symmetric BDMC
SC	Successive Cancellation
SCL	Successive Cancellation List
SNR	Signal-to-Noise-Ratio
SPC	Single-Parity-Check
SSE	Streaming SIMD Extensions
UML	Unified Modeling Language

Bibliography

[Ari09] Erdal Arikan. Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. *Information Theory, IEEE Transactions on*, 55(7):3051–3073, July 2009.

[Ari11] Erdal Arikan. Systematic polar coding. *IEEE communications letters*, 15(8):860–862, 2011.

[boo10] Boost Random. `http://www.boost.org/doc/libs/1_59_0/doc/html/boost_random.html`, 2010. Accessed 2015-11-03.

[BPB14] Alexios Balatsoukas-Stimming, Mani Bastani Parizi, and Andreas Burg. Llr-based successive cancellation list decoding of polar codes. *CoRR*, abs/1401.3753, 2014.

[Dij08] Edsger W. Dijkstra. Why numbering should start at zero. `http://www.cs.utexas.edu/users/EWD/transcriptions/EWD08xx/EWD831.html`, 2008. Accessed 2015-10-26.

[Fri] Bernd Friedrichs. Error-control coding. `http://www.berndfriedrichs.de/buecher_d.htm#ecc`. Accessed 2015-11-06.

[git15] GNU Radio repository. `https://github.com/gnuradio/gnuradio`, 2015. Accessed 2015-11-02.

[gnu15a] FECAPI documentation. `https://gnuradio.org/doc/doxygen/page_fec.html`, 2015. Accessed 2015-11-03.

[gnu15b] Vector-Optimized Library of Kernels. `http://libvolk.org/`, 2015. Accessed 2015-11-19.

[gnu15c] Vector-Optimized Library of Kernels. `https://github.com/gnuradio/volk`, 2015. Accessed 2015-11-19.

[GQGiFS14] Jing Guo, Minghai Qin, A. Guillen i Fabregas, and P.H. Siegel. Enhanced belief propagation decoding of polar codes through concatenation. In *Information Theory (ISIT), 2014 IEEE International Symposium on*, pages 2987–2991, June 2014.

[GSL$^+$15] Pascal Giard, Gabi Sarkis, Camille Leroux, Claude Thibeault, and Warren J. Gross. Low-latency software polar decoders. *CoRR*, abs/1504.00353, 2015.

[HS15] Michael Helmling and Stefan Scholl. Database of Channel Codes and ML Simulation Results. `www.uni-kl.de/channel-codes`, 2015. Accessed 2015-11-19.

[NC12] K. Niu and K. Chen. Stack decoding of polar codes. *Electronics Letters*, 48(12):695 – 697, 2012.

[RMP+15] P. Rost, A. Maeder, H. Paul, D. Wübben, A. Dekorsy, G. Fettweis, Ignacio Berberana, Vinay Suryaprakash, and Matthew Valenti. Benefits and challenges of virtualization in 5g radio access networks. *accepted for publications in Special Issue Advanced Cloud & Virtualization Techniques for 5G Networks of the IEEE Communications Magazine*, Dec 2015.

[RU08] Tom Richardson and Ruediger Urbanke. *Modern Coding Theory*. Cambridge University Press, New York, NY, USA, 2008.

[Sha49] Claude E. Shannon. Communication in the Presence of Noise. In *Proceedings of the IRE*, January 1949.

[STG+15] Gabi Sarkis, Ido Tal, Pascal Giard, Alexander Vardy, Claude Thibeault, and Warren J. Gross. Flexible and low-complexity encoding and decoding of systematic polar codes. *CoRR*, abs/1507.03614, 2015.

[TV11] I. Tal and A. Vardy. List decoding of polar codes. In *Information Theory Proceedings (ISIT), 2011 IEEE International Symposium on*, pages 1–5, July 2011.

[TV13] I. Tal and A. Vardy. How to construct polar codes. *Information Theory, IEEE Transactions on*, 59(10):6562–6582, Oct 2013.

[VVH15] Harish Vangala, Emanuele Viterbo, and Yi Hong. A comparative study of polar code constructions for the AWGN channel. *CoRR*, abs/1501.02473, 2015.