# Polar Codes in Python: Main Reference

Brendon McBain

October 2019

# 1 Executive Summary

Polar codes are a state-of-the-art channel code used in 5G and beyond. It is the first mathematically proven capacity-achieving code, and has relatively simple algorithms for encoding and decoding. The project *A Library for Polar Coding and Shortening Algorithms in Python* is a library dedicated to implementing common algorithms used by polar codes in Python. Until now, Python enthusiasts have not had a readily-available library, only in MATLAB. Similarly, the GUI provided alongside this library is a useful addition to the polar codes community. In addition, the library includes shortening algorithms that reduce the block length of the mothercode. An application of shortening is eMBB in 5G, which requires block lengths of either 64800 bits or 1920 bits. However, conventional polar codes are restricted to block lengths using a power of two. The package is open-source and available for free on GitHub, and is intended to be used by students, designers, and researchers of polar codes that prefer Python programming.

# Contents

# 2 Introduction

The following document provides a detailed explanation of the algorithms used by *Polar Codes in Python*. A programmer new to polar codes should find this document to be a self-contained resource for them to get started with construction, encoding, decoding, and shortening. To see a broad overview of the classes, functions, and their syntax, refer to *Polar Codes in Python: Quick Reference*.

## 2.1 Notation

- $u_N$, the uncoded message with frozen bits and length $N$
- $x_N$, the codeword with length $N$
- $\mathcal{F}$, the frozen set bit indices
- $\mathcal{A}$, the information set bit indices
- $\mathcal{S}$, the coded shortened bit indices
- $\mathcal{O}$, the uncoded shortened bit indices
- $x_{\mathcal{F}}$, the vector of elements at indices from $\mathcal{F}$
- $F = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$, Arikan's kernel
- $N$, the block length
- $n = log_2(N)$, the number of bits per index
- $b(.)$, the bit-reversal of each element in the set

# 3 Log-domain Arithmetic

The recursive relationships of the algorithms in polar coding require that we work in the log-domain to avoid overflow errors in their implementations. The library uses functions that add and subtract linear variables in the log-domain. Let us define where $x' = ln(x)$ and $y' = ln(y)$, then the operations are implemented as follows.

$$lsum(x', y') \triangleq \begin{cases} x' + ln(1 + e^{y'-x'}) = x' + log1p(e^{y'-x'}) & x' \geq y' \\ y + ln(1 + e^{x'-y'}) = y' + log1p(e^{x'-y'}) & x' < y' \end{cases}$$

$$ldiff(x', y') \triangleq \begin{cases} x' + ln(1 - e^{y'-x'}) = x + log1p(-e^{y'-x'}) & x' \geq y' \\ y + ln(1 - e^{x'-y'}) = y' + log1p(-e^{x'-y'}) & x' < y' \end{cases}$$

For numerical precision, we must use $log1p$ in the implementations. Using the raw equations, the program will underflow for a large $|x' - y'|$ since the exponential term is computed. On the other hand, $log1p$ avoids this by a direct computation.

***Quick Reference 3.1***
*lsum() and ldiff() are implemented in the library as Math.logdomain_sum() and Math.logdomain_diff(), respectively.*

# 4 Construction

A polar code is based upon polarisation, whereby the capacity of all $N$ bit-channels are combined and then split into channels of capacity zero (completely unreliable), and of capacity one (very reliable). However, perfect polarisation of the capacities is only achievable for non-finite block lengths. Construction is the process of choosing the $N - K$ least reliable bit-channels, and setting them as "frozen" bits. While the frozen bit-channels cannot carry any information reliably, they are useful in decoding the remaining information bits since they are known *a priori* by the decoder. Hence, they are analagous to parity bits.

There are a variety of construction algorithms, which depend upon the type of channel, design SNR, block length, and code rate. The two algorithms described below are mothercode constructions for an AWGN channel, namely the Bhattacharrya Bounds algorithm by Arikan and the Gaussian Approximation algorithm by Trifonov.

## 4.1 Bhattacharyya Bounds

The Bhattacharyya bounds are the simplest way to construct a polar code, and was the first method proposed by Arikan [1]. The construction algorithm

evolves Bhattacharyya parameters through the butterfly diagram, and returns upper bounds of FER for all $N$ channels. The Bhattacharyya bounds can be computed iteratively by the following.

$$\begin{cases} Z_N^{(2i-1)} = Z_{N/2}^{(2i-1)} + Z_{N/2}^{(2i)} - Z_{N/2}^{(2i-1)} Z_{N/2}^{(2i)} \\ Z_N^{(2i)} = Z_{N/2}^{(2i-1)} Z_{N/2}^{(2i)} \end{cases}$$

$$\overset{log\ domain}{\rightarrow} \begin{cases} z_N^{(2i-1)} = ldiff(lsum(z_{N/2}^{(2i-1)}, z_{N/2}^{(2i)}), z_{N/2}^{(2i-1)} + z_{N/2}^{(2i)}) \\ z_N^{(2i)} = z_{N/2}^{(2i-1)} + z_{N/2}^{(2i)} \end{cases}$$

where $1 \leq i \leq N$. In the case of an AWGN channel, $z_1^{(i)} = -E_b/N_o$.

To implement this algorithm with support for shortening, we must replace these principal equations with their limits for different combinations of inputs of $z_n^{(i)}$. These values have been presented in Table 1.

| $z_{n-1}^{(2j)}$ | $z_{n-1}^{(2j+1)}$ | $z_n^{(2j)}$ | $z_n^{(2j+1)}$ |
|---|---|---|---|
| $-\infty$ | finite | $z_{n-1}^{(2j+1)}$ | $-\infty$ |
| finite | $-\infty$ | $z_{n-1}^{(2j)}$ | $-\infty$ |
| $-\infty$ | $-\infty$ | $-\infty$ | $-\infty$ |

Table 1: An extension to the Bhattacharyya Bounds equations for non-finite Bhattacharyya parameters.

***Quick Reference 4.1***
*Bhattacharyya Bounds construction is implemented in the library as Construct.general_pcc().*

## 4.2 Gaussian Approximation

A more accurate construction is to evolve the densities and estimate the precise reliability of each channel, rather than an upper bound. This is what the Gaussian Approximation algorithm does, using an approximation of the gaussian pdf to perform density evolution of the log-likelihood ratios at the receiver. The densities are computed iteratively by the following, as presented in [5].

$$\begin{cases} m_N^{(2i-1)} = \phi^{-1}(1 - [1 - f(m_{N/2}^{(2i-1)})][1 - f(m_{N/2}^{(2i)})]) \\ m_N^{(2i)} = m_{N/2}^{(2i-1)} + m_{N/2}^{(2i)} \end{cases}$$

where

$$\phi(x) \triangleq \begin{cases} \exp\{-0.4527x^{0.86} + 0.0218\} & 0 \leq x < 10 \\ \sqrt{\frac{\pi}{x}}(1 - \frac{10}{7x}) \exp\{-x/4\} & x \geq 10 \end{cases}, \ 1 \leq i \leq N, \ m_1^{(i)} = 4E_b/N_0$$

In terms of implementation, the biggest difference between the Gaussian Approximation and Bhattacharyya Bounds is that we also have to compute the inverse of $\phi(x)$, which is a piecewise function that does not have a closed-form. Hence, we have to resort to numerical methods and solve for the roots of $\phi(x) - y$ with respect to $x$.

> ***Quick Reference 4.2***
> *Gaussian Approximation construction is implemented in the library as Construct.general_ga().*

### 4.3 Implementation

The following is a generic algorithm that can be used to implement common polar code constructions, simply by swapping out functions $f$ and $g$.

**Algorithm 1** Generalised construction template

**procedure** CONSTRUCTION($z_0$)
  $z = zeros(N, n+1)$
  $z[:, 0] \leftarrow z_0$
  **for** $j \in [1 : n]$ **do**
    $u \leftarrow 2^j$
    **for** $t \in [0 : u : N]$ **do**
      **for** $s \in [0 : u/2]$ **do**
        $k \leftarrow t + s$
        $z[k, j] \leftarrow f(z[k, j-1], z[k+u/2, j-1])$
        $z[k+u/2, j] \leftarrow g(z[k, j-1], z[k+u/2, j-1])$
  $\mathcal{R} \leftarrow argsort(z[:, n])$
  **return** $\mathcal{R}$ ▷ Reliabilities in ascending order

Note that the densities for the Gaussian Approximation are inversely proportional to the Bhattacharyya parameters used in the Bhattacharyya Bounds construction. Therefore, you must sort in ascending order for the Gaussian Approximation, and descending order for the Bhattaryya Bounds. More detailed explanations and pseudo-code is presented in [3].

## 5 Shortening

Polar codes are restricted to powers of two, which can be inconvenient in some applications. Therefore, we need an algorithm that can reduce $N$ to $M$ such that the performance is as close to the mothercode as possible. By the nature of shortening, we lose the capacity of $N - M$ bit channels since we do not send bits on these channels. Shortening algorithms try to do this intelligently by optimising $(\mathcal{F}, \mathcal{S})$, using density evolution or heuristic algorithms. The challenge of shortening is the exponentially large solution space, so this is often

restricted.Unlike puncturing, shortening will assign values to the bits that are not sent, of which the decoder will know *a priori*. Hence, the shortened bits will have bit-channels with likelihoods of $\infty$.

There are threes types of shortenings algorithms, and each vary in computational complexity:

- Type I: Optimise $\mathcal{F}$, which is dependent on $\mathcal{S}$.
- Type II: Optimise $\mathcal{S}$, which is dependent on $\mathcal{F}$.
- Type III: Jointly optimise $(\mathcal{F}, \mathcal{S})$.

It is worth noting that shortening polar codes is unique in that the size of the information set $K$ need not be reduced with the block length, which is an advantage of having the so-called frozen set.

## 5.1 Shortening Patterns

Type I shortening algorithms optimise $\mathcal{F}$ using a chosen shortening pattern $\mathcal{S}$. By introducing coded bits that are known as $x_{\mathcal{S}} = 0$, it can be proven that there are corresponding uncoded bits where $u_{\mathcal{O}} = 0$, often called overcapable bits. However, if these overcapable bits are not a subset of the frozen set, then the message will surely have an error. The main advantage of Type I shortening is that we can choose shortening patterns where the overcapable set is easy to compute, and satisfies $\mathcal{S} \subseteq \mathcal{F}$. When this is not satisfied, the pattern is said to be "catastrophic".

**Algorithm 2** Find $\mathcal{F}$ from a shortening pattern

**procedure** FROZEN_FROM_PATTERN($z_0$)
  Set the shortening pattern $\mathcal{S}$.
  $\mathcal{R}_0 \leftarrow \mathcal{R}_m \backslash \mathcal{S}$ ▷ $\mathcal{R}_m$ is the mothercode reliability set
  $\mathcal{F}_0 \leftarrow \mathcal{R}_0[0 : M - K - 1]$
  $\mathcal{F} \leftarrow \mathcal{F}_0 \cup \mathcal{S}$.
  **return** $\mathcal{F}$

***Quick Reference 5.1***
*Algorithm 2 is implemented in the library as Shorten.frozen_from_pattern().*

## 5.2 Catastrophic Patterns

A catastrophic shortening pattern is one that introduces shortened bits into the information set, which lead to block errors almost surely. We can detect if a shortening pattern, or puncturing pattern, is catastrophic using boolean functions. This was proven in [6] by Hong and Hui.

The boolean functions are based on assuming perfect bit-channels, where shortened bit-channels have a capacity of 1 and the rest are 0. Hence, we can construct a look-up table using Algorithm 1 that approximates which indices are the uncoded shortened bits.

Given the following "perfect" construction

$$Z(\overline{p}_0) = \begin{cases} p_N^{(2i-1)} = \overline{p}_{N/2}^{(2i-1)} \ + \ \overline{p}_{N/2}^{(2i)} \\ p_N^{(2i)} = \overline{p}_{N/2}^{(2i-1)} \cdot \overline{p}_{N/2}^{(2i)} \end{cases} , \ 1 \leq i \leq N$$

Then shortening pattern $p_0$ is non-catastrophic if $\{i \in \mathcal{A} : Z^{(i)}(\overline{p}_0) = 0\}$ is satisfied. The "shortening rule" in the literature [5] characterises where each zero from the uncoded bits are mapped, hence there are valid and invalid shortening patterns.

***Quick Reference 5.2***
*Use Construct.perfect_pcc() from the library to compute $Z(p_0)$ and verify a catastrophic pattern.*

## 5.3 Wang-Liu Shortening

The Wang-Liu shortening algorithm [4] finds shortening patterns that satisfy $\mathcal{S} \subseteq \mathcal{F}$ and $\mathcal{O}(\mathcal{S}) = \mathcal{S}$. Using the generator matrix of the mothercode, the algorithm finds which coded bit can be shortened such that it is mapped to an uncoded bit with the same index. Then, it freezes and shortens it. Each iteration of the algorithm shortens one bit-channel by selecting a row index of the generator matrix with a hamming weight of 1. Therefore, we can find a valid shortening pattern for any block length. By choosing the greatest index value with a hamming weight of 1 at each iteration, we arrive at the most common output of the Wang-Liu algorithm, where $S_{WLS} = \{N-M, N-M+1, ..., N-1\}$.

***Quick Reference 5.3***
*The Wang-Liu shortening algorithm is implemented in the library as Shorten.wang_liu(), and the WLS shortening pattern is return by Shorten.last_pattern().*

## 5.4 Bit-Reversal Shortening

Bit-reversal shortening is a very simply and well-performing shortening pattern. It is sometimes called RQUP, derived from the similarly successful QUP puncturing pattern. By definition, it is simply $S_{BRS} = b(S_{WLS})$, the bit-reversal of each index in the Wang-Liu shortening pattern. At present, this is the best performing shortening pattern that satisfies $\mathcal{O}(\mathcal{S}) = \mathcal{S}$, in a wide range of code rates.

***Quick Reference 5.4***
*Bit-reversal shortening pattern is returned from Shorten.brs_pattern().*

# 6 Encoding

Traditionally, the encoder transforms a message of dimension $K$ to a block length of $N$ by adding parity bits. However, polar coding appends $N - K$ so-called frozen bits to the $K$ message bits before encoding. The information and frozen indices are specified by $\mathcal{F}$ from the output of the construction algorithm.

## 6.1 Non-systematic Encoder

The encoder of this library is a non-systematic element-wise encoder that is optimised from the more natural and slower recursive counterpart. The encoder evaluates $x_N = F^{\otimes n} u_N$. Note that the literature often uses the bit-reversed indices of Arikan's kernel and reverses the matrix multiplication.

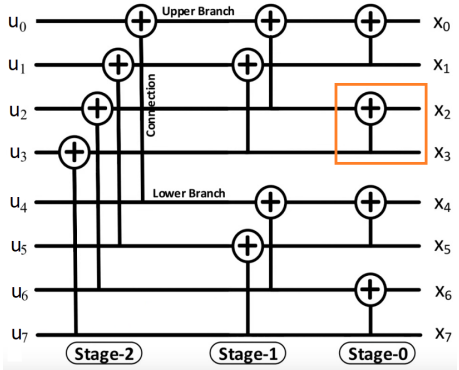


Figure 1: Code tree for for $N = 8$. The kernel $F$ is represented inside the orange box.

***Quick Reference 6.1***
*The non-recursive encoding algorithm is implemented in the library as Encode.polar_encode(), and the recursive version as Encode.polar_encode2().*

# 7 Transmission

## 7.1 Modulation

The modulation scheme used in this library is BPSK, where a "0" gets mapped to $+\sqrt{E_b}$ and a "1" gets mapped to $-\sqrt{E_b}$. Hence, each codeword is modulated by

$$\hat{x} = (2x - 1)\sqrt{E_b}$$

## 7.2 Channel

The channel model in this library is AWGN. Hence, the received signal will be of the form $y = \hat{x} + n$, where $n \sim \mathcal{N}(0, N_0/2)$ and $N_0$ is the double-sided noise power. We typically set $N_0$ to one for simplicity, then $E_b$ becomes the design SNR.

The decoder implementation requires log-likelihoods for numerical accuracy and mathematical convenience, and are defined as

$$\mathcal{L} = ln\left(\frac{p(x=0)}{p(x=1)}\right) = -\frac{2y}{N_0}\sqrt{E_b} \overset{unity\ noise}{\rightarrow} -2y\sqrt{E_b/N_0}$$

where

$$p(x) = p(y|\hat{x}) = \frac{1}{\sqrt{\pi N_0}} \exp\{(y-\hat{x})/N_0\}, \hat{x} \in \{\pm\sqrt{E_b}\}$$

To be able to compare codes, the design SNR must be normalised by its code rate $R = K/N$ to ensure $(\frac{E_b}{N_0})^{code} = \frac{1}{R}(\frac{E_b}{N_0})^{message}$. Therefore, the message in any codeword will have the same energy. When puncturing we must use $N = M$, since only $M$ bits are being sent through the channel.

> ***Quick Reference 7.1***
> *The AWGN channel model is implemented in the library as class AWGN. It has functions AWGN.modulation() for modulation, and AWGN.noise() for generating random gaussian noise.*

# 8 Decoding

This section is dedicated to the design and implementation of a successive cancellation decoder, and how to adapt the principal decoding equations for shortening.

## 8.1 Successive Cancellation Decoder

The successive cancellation decoder is the very first decoder for polar codes. It is based upon successively decoding each bit, and using the observation of previously decoded bits to improve future decoded bit estimates. Specifically, the decoded bit at the top branch of each iteration must be known before decoding the bottom branch. The top branches have indices $2i-1$, and the bottom branches have indices $2i$, where $1 \leq i \leq N$. Hence, parallelisation of such an algorithm is not feasible, but its low computational complexity is very attractive.

The likelihoods are updated by

$$\begin{cases} l_N^{(2i-1)} = f(l_{N/2}^{(2i-1)}, l_{N/2}^{(2i)}) \\ l_N^{(2i)} = g(l_{N/2}^{(2i-1)}, l_{N/2}^{(2i)}, b_N^{(2i-1)}) \end{cases}$$

and the bits are updated by

$$\begin{cases} b_N^{(2i-1)} = b_{N/2}^{(2i-1)} \oplus b_{N/2}^{(2i)} \\ b_N^{(2i)} = b_{N/2}^{(2i)} \end{cases}$$

The functions $f$ and $g$ are used by the principal SCD equations to find the likelihoood ratios for each bit-channel, and were proven by Arikan in [1]. To avoid overflow, we will convert them into the log-domain and use the operations defined in the Math section for a high-precision algorithm implementation.

$$f(L_1, L_2) = \frac{L_1 * L_1 + 1}{L_1 + L_2} \overset{log\ domain}{\rightarrow} f(l_1, l_2) = ln(\frac{e^{l_1+l_2}+1}{e^{l_1}+e^{l_2}}) = lsum(l_1+l_2, 0) - lsum(l_1, l_2)$$

$$g(L_1, L_2, b_1) = \begin{cases} L_1 * L_2 & b_1 = 0 \\ L_1 / L_2 & b_1 = 1 \end{cases} \overset{log\ domain}{\rightarrow} g(l_1, l_2, b_1) = \begin{cases} l_1 + l_2 & b_1 = 0 \\ l_1 - l_2 & b_1 = 1 \end{cases}$$

However, since this library must support shortening, the equations must support non-finite log-likelihood ratios. Using the linear scale equations, we can take the limit for all input cases of non-finite values and find their corresponding log-likelihood ratios as in Table 2. The cases for when $b_1 = 1$ can be ignored, since all frozen bits are set to zero and the shortened bits are a subset of them. Therefore, the undefined value is avoided in the implementation.

| | | $f(l_1, l_2)$ | $g(l_1, l_2, b_1)$ | |
|---|---|---|---|---|
| $l_1$ | $l_2$ | all b | $b_1 = 0$ | $b_1 = 1$ |
| finite | $\infty$ | $l_1$ | $\infty$ | 0 |
| $\infty$ | finite | $l_2$ | $\infty$ | $\infty$ |
| $\infty$ | $\infty$ | $\infty$ | $\infty$ | undef |

Table 2: An extension to the SCD equations for non-finite log-likelihood ratios.

## 8.2 Implementation

Pseudo-code for a simple SCD implementation can be found in [2]. However, this is composed of recursive functions, which are often slow. The following algorithms show how to implement a non-recursive version of the aforementioned pseudo-code.

**Algorithm 3** Successive cancellation decoder

```
procedure UPDATE_LLR(y)
    L ← full((N, n + 1), np.nan)
    B ← full((N, n + 1), np.nan)
    for i ∈ [0 : N − 1] do
        l ← b(i)
        Update_LLR(L, B, l)
        if l ∈ F then
            B[l, 0] = 0
        else
            if L[l, 0] ≥ 0 then
                B[l, 0] = 0
            else
                B[l, 0] = 1
        Update_Bits(B, l)
    x ← B[:, 0]
    u ← x_F
    return u                                   ▷ The decoded message
```

**Algorithm 4** Update log-likelihood ratios in L (non-recursive)

```
procedure UPDATE_LLR(L, B, x)
    for j ∈ [n − 1 : −1 : 0] do
        s ← 2^(n−j)
        t ← s/2
        for i ∈ [x : N : s] do
            if t > i  mod s then
                L[i, j] ← f(L[i, j + 1], L[i + t, j + 1])
            else
                L[i, j] ← g(L[i, j + 1], L[i − t, j + 1], B[i − t, j])
```

**Algorithm 5** Update bits in B (non-recursive)

```
1: procedure UPDATE_BITS(B, x)
2:     b ← [x]
3:     for j ∈ [0 : n − 1] do
4:         s ← 2^{n−j}
5:         t ← s/2
6:         b_next ← []
7:         for i ∈ b do
8:             if t ≤ i  mod s then
9:                 B[i − t, j + 1] ← B[i, j] ⊕ B[i − t, j]
10:                B[i, j + 1] ← B[i, j]
11:            b_next.append(i, i − t)
12:        b ← b_next
```

***Quick Reference 8.1***
*The SCD algorithm is implemented in the library as Decode.polar_decode().*

# 9 Results

The following figures demonstrate the typical performance of the mother and shortened codes using algorithms produced by the *Polar Codes in Python* package.

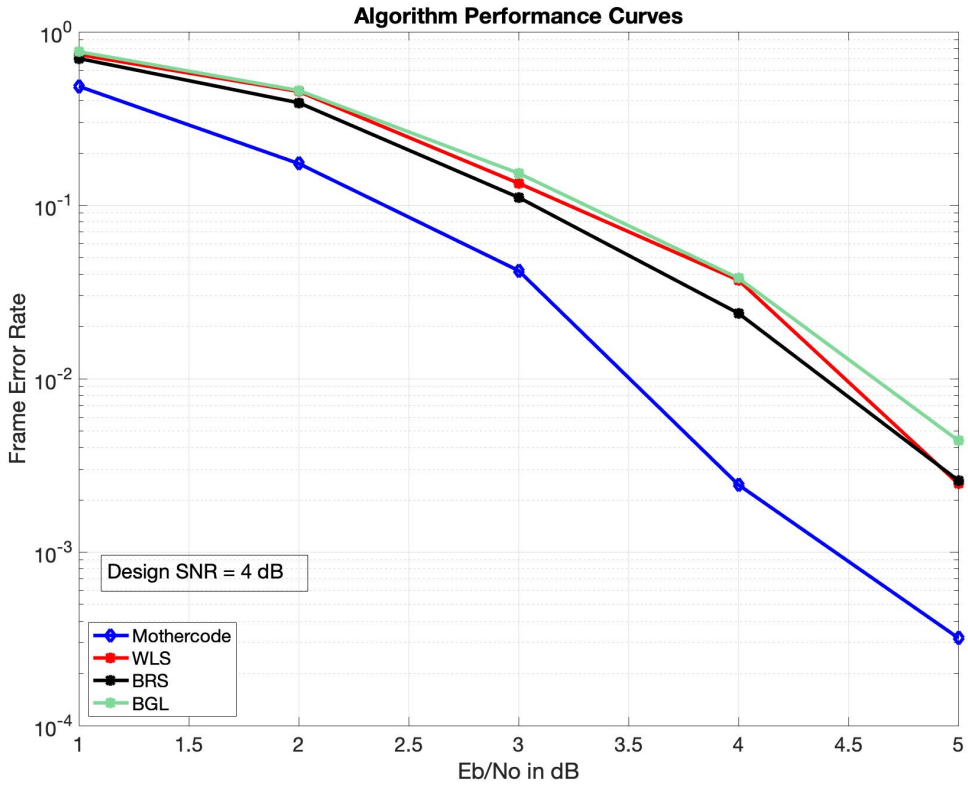


Figure 2: Shortening a (128,70) mothercode to (98,70).

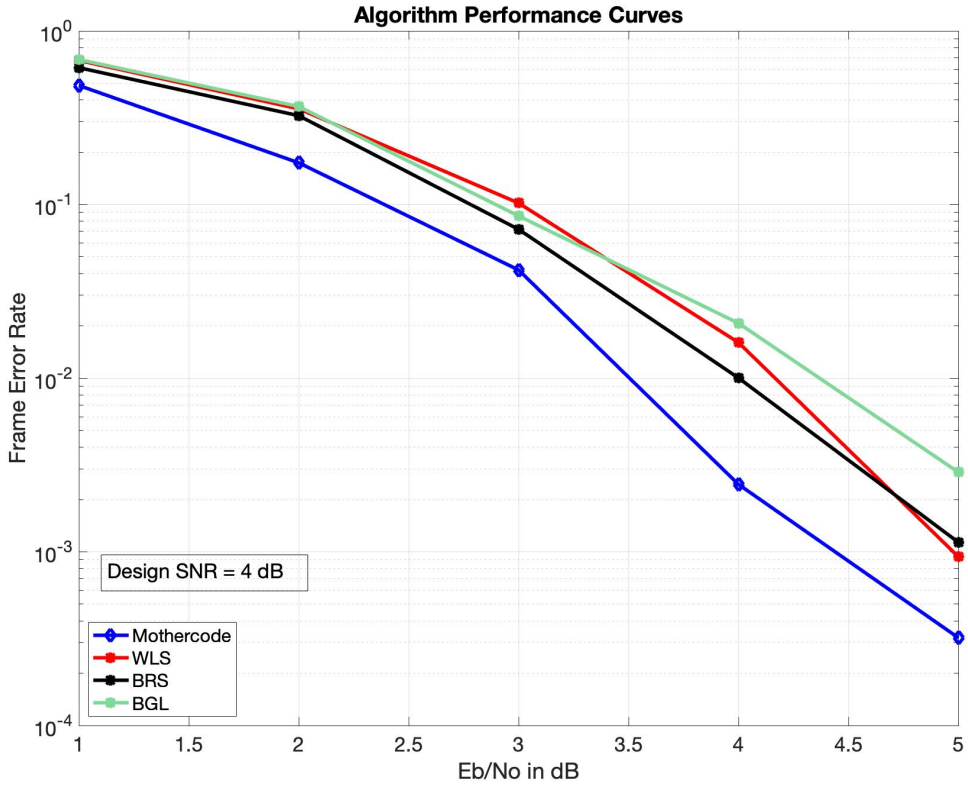


Figure 3: Shortening a (128,70) mothercode to (108,70).

# References


[1] Erdal Arikan. *Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels*. IEEE Transactions on Information Theory, Vol. 55, No. 7, July 2009.

[2] Harish Vangala, Emanuele Viterbo, and Yi Hong. *Permuted Successive Cancellation Decoder*. ISITA2014, Melbourne, Australia, October 26-29, 2014.

[3] Harish Vangala, Emanuele Viterbo, and Yi Hong. *A Comparative Study of Polar Code Constructions for the AWGN Channel*. arXiv:1501.02473 [cs.IT], 2015.

[4] Runxin Wang and Rongke Liu. *A novel puncturing scheme for polar codes*. IEEE Communications Letters, Vol. 18, No. 12, December 2014.

[5] Prakash Chaki and Norifumi Kamiya. *On the properties of bit-reversal shortening in polar codes*. ISITA2018, Singapore, October 28-31, 2018.

[6] Song-Nam Hong and Min-Oh Jeong. *On the analysis of puncturing for finite-length polar codes: Boolean function approach*. IEEE Transactions on Communications, Vol. 66, No. 11, November 2018.