Dhirubhai Ambani University

Introduction to Communication Systems
(CT216)

Polar Codes

Group - 31

Prof: Yash Vasavada

Mentor: Dharmi Patel

Honor Code:

We, Group 31, declare that

- the work that we are presenting is our own work.
- we have not copied the work (Matlab code, results, etc.) that someone else has done.
- concepts, understanding, and insights we will be describing are our own.
- wherever we have relied on existing work that is not our own, we have provided a proper reference citation.
- we make this pledge truthfully. We know that a violation of this solemn promise can have grave consequences.

Group Members:

1. YUVA UNDAVIYA (ID : 202301426)
2. AALOK THAKKAR (ID : 202301429)
3. JAY MANISHKUMAR (ID : 202301430)
4. JAS MEHTA (ID : 202301432)
5. DHRUVIL KATHAROTIYA (ID : 202301434)
6. VRAJKUMAR MAKWANA (ID : 202301436)
7. MAANAV GURUBAXANI (ID : 202301438)
8. VRAJ PARIKH (ID: 202301440)
9. HRITHIK PATEL (ID : 202301441)
10. SHUBHAM RAMOLIYA (ID : 202301442)

Contents

1 Polar Codes

Polar Codes are a type of **Linear Block Code** that were invented by Erdal Arıkan in 2009. They represent the first *proved capacity-achieving code*[1] with an explicit construction, while maintaining relatively low encoding and decoding complexity.

The key principle behind Polar Codes lies in the concept of **Channel Polarization**. This is the fundamental process that enables the polar coding scheme to approach *Shannon's Capacity*.

The polar coding scheme is highly structured and block-based, which makes it favorable for analysis and performance prediction. Another significant advantage is its compatibility with other coding schemes, such as **Cyclic Redundancy Check (CRC)** or even **Low-Density Parity-Check (LDPC)** codes in hybrid designs.

1.1 Applications

- 5G New Radio (NR)[2]
- NASA Deep-space Communication Systems Research

These applications benefit from the **low latency** and **efficient hardware implementation** capabilities of Polar Codes.

1.2 Channel Polarization in Polar Codes

Polar Codes are primarily based on the principle of **Channel Polarization**, a process that transforms a set of identical, independent binary-input discrete memoryless channels into a new set of polarized channels that are either highly reliable or highly unreliable.[5][4]

This transformation occurs through two key steps: **Channel Combining** and **Channel Splitting**.

- **Channel Combining** involves taking multiple identical channels and recursively combining them using a specific transformation—typically the Kronecker product of a basic kernel matrix (like the polar encoding using G matrix), such as

$$G_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

  This operation creates a larger virtual channel from multiple physical channels, effectively entangling their inputs.

- **Channel Splitting** is then applied to the combined channel. It decomposes the new larger channel into a set of subchannels , each representing a specific bit position in the input vector. These synthesized channels are no longer identical; instead, they exhibit a wide range of reliabilities due to the polarization effect.

As the block length $N = 2^n$ increases, the subchannels become more polarized: some approach a state of *perfect reliability* (ideal for sending information bits), while others approach *zero reliability* (unsuitable for transmission). This phenomenon is known as **channel polarization**.

Consequently, information bits are transmitted through the highly reliable (good) channels, while the unreliable (bad) channels are assigned fixed, known values (often 0) — these are called **frozen bits**. This allocation allows the code to approach the channel capacity as the block length grows, fulfilling Shannon's theoretical limit.

Note: This block diagram illustrates the Polar Code encoding and decoding process for $N = 2$. While the figure mentions the SCL (Successive Cancellation List) decoding method to indicate different types of decoders, for such a small block length ($N = 2$), we typically use only the SC (Successive Cancellation) decoding method. The SCL method is more relevant for longer code lengths where improved performance is necessary.

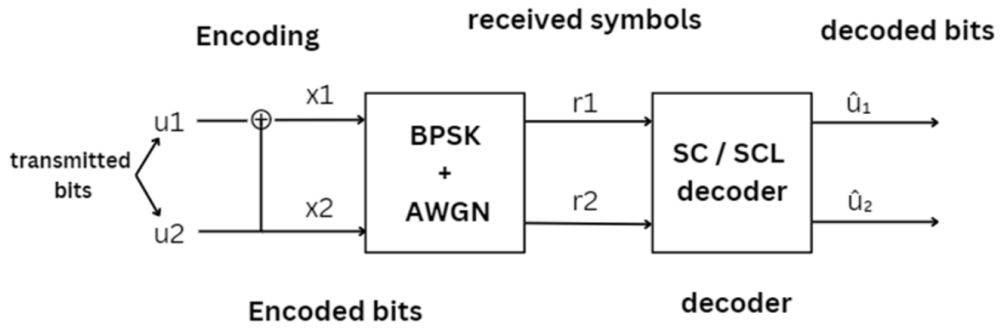


Figure 1: Block Diagram for Polar Code

Here are Channel block Diagrams for N values 4, and 8 respectively to see how it scales.

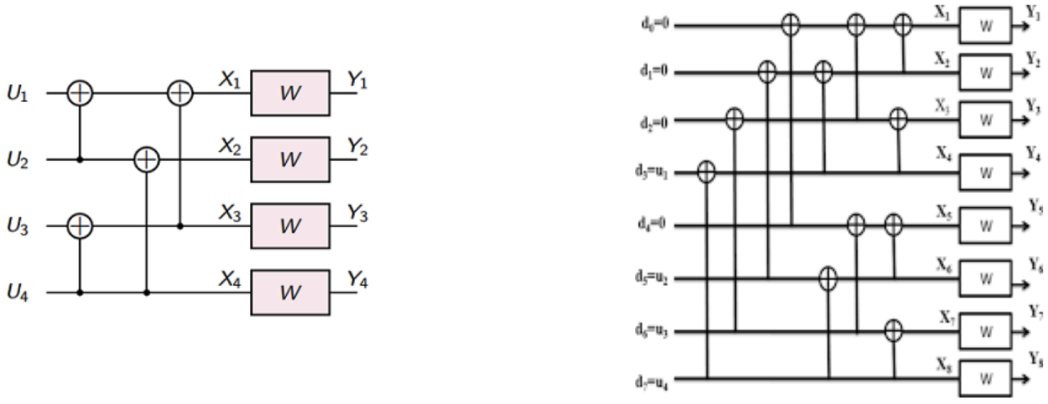


Figure 2: Channel block diagrams for Polar Codes with $N = 4$ and $N = 8$, showing how the structure scales with increasing block length.

1.3 Channel Reliability and Bhattacharyya Parameter

To construct the Polar Code, we must select the K most reliable channels to send the information bits through. This can be done by evaluating the reliability of each of the channels and sorting them accordingly. For specific channels like the Binary Erasure Channel (BEC) and Binary Symmetric Channel (BSC), the **Bhattacharyya parameter** is used as a measure of channel reliability.

Bhattacharyya Parameter

The Bhattacharyya parameter is denoted by $Z(W)$. A higher value of $Z(W)$ indicates a less reliable channel, and vice versa. In particular:

- $Z(W) = 0$ models a **perfectly reliable** channel
- $Z(W) = 1$ models a **totally unreliable** channel

By sorting the channels based on their Bhattacharyya parameters, we can select the best K channels (with the lowest $Z(W)$ values) to transmit the information bits, while the remaining channels are used as frozen bits.

2 Polar Encoding

2.1 Matrix-Based Polar Encoding

In polar coding, the encoding of the input vector $\mathbf{u}$, which contains both message and frozen bits, is performed using a generator matrix G_N. The generator matrix is an $N \times N$ binary matrix derived using the **Kronecker product** of a base matrix G_2, where $N = 2^n$ for some integer n.[3]

The base matrix is defined as:

$$G_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

To construct the generator matrix for a polar code of length N, we compute:

$$G_N = G_2^{\otimes n}$$

where $\otimes$ represents the Kronecker product and $G_2^{\otimes n}$ denotes the n-fold Kronecker product of G_2 with itself.

For example:

$$G_4 = G_2^{\otimes 2} = G_2 \otimes G_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Once G_N is obtained, the encoded vector $\mathbf{x}$ is computed as:

$$\mathbf{x} = \mathbf{u} \cdot G_N$$

where all operations are carried out in modulo-2 arithmetic (i.e., binary addition and multiplication).

This generator matrix-based approach forms the basis of the polar encoding process and ensures the structured transformation of the input vector into a codeword suitable for transmission over a noisy communication channel.

2.2 Tree-Based Polar Encoding for $N = 8$

In the matrix-based approach to polar encoding, the encoded output is obtained by multiplying the input vector $\mathbf{u}$ (containing message and frozen bits) with the generator matrix G_N, using modulo-2 arithmetic.

For example, when $N = 4$, the encoding using G_4 yields:

$$[u_1, u_2, u_3, u_4] \cdot G_4 = [u_1 + u_2 + u_3 + u_4,\ u_2 + u_4,\ u_3 + u_4,\ u_4]$$

Here, u_1, u_2, u_3, u_4 are the message bits, and the addition is modulo-2 (binary XOR operation).

Now consider a polar code with parameters $(N, K) = (8, 4)$. Based on the channel reliability sequence, we select $N - K = 4$ frozen bit positions and place the message bits in the remaining $K = 4$ most reliable positions. An example of such an input vector is:

$$\text{Msg} = [0,\ 0,\ 0,\ m_1,\ 0,\ m_2,\ m_3,\ m_4]$$

This placement ensures that the most reliable synthetic channels are used to transmit actual data, while unreliable ones are fixed (frozen) to known values.

In the next slide, we illustrate the **tree-based encoding approach** for $N = 8$, which provides an alternative visualization of the encoding process. Instead of direct matrix multiplication, this method uses a binary tree structure, where encoding is performed through a sequence of XOR operations along the branches of the tree, following the polar transform logic.

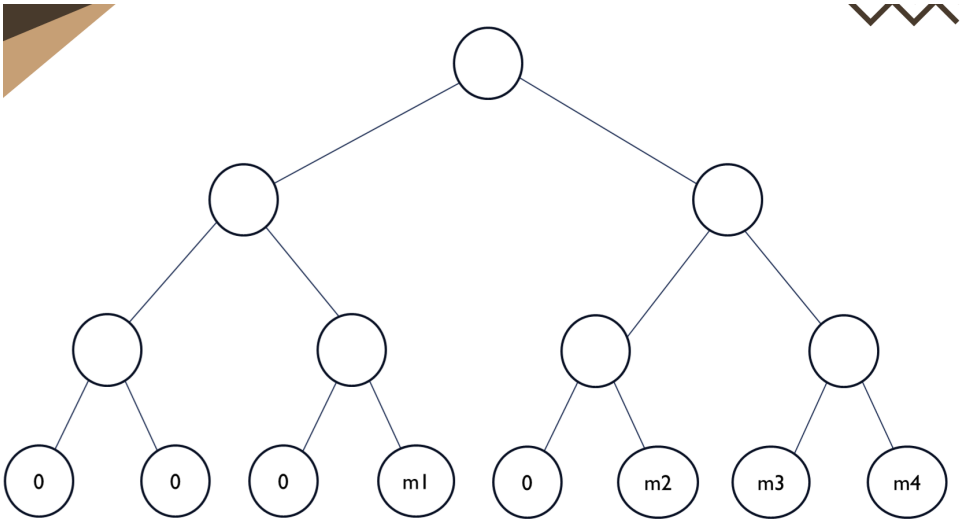


Figure 3: Binary tree diagram with mixed leaf nodes

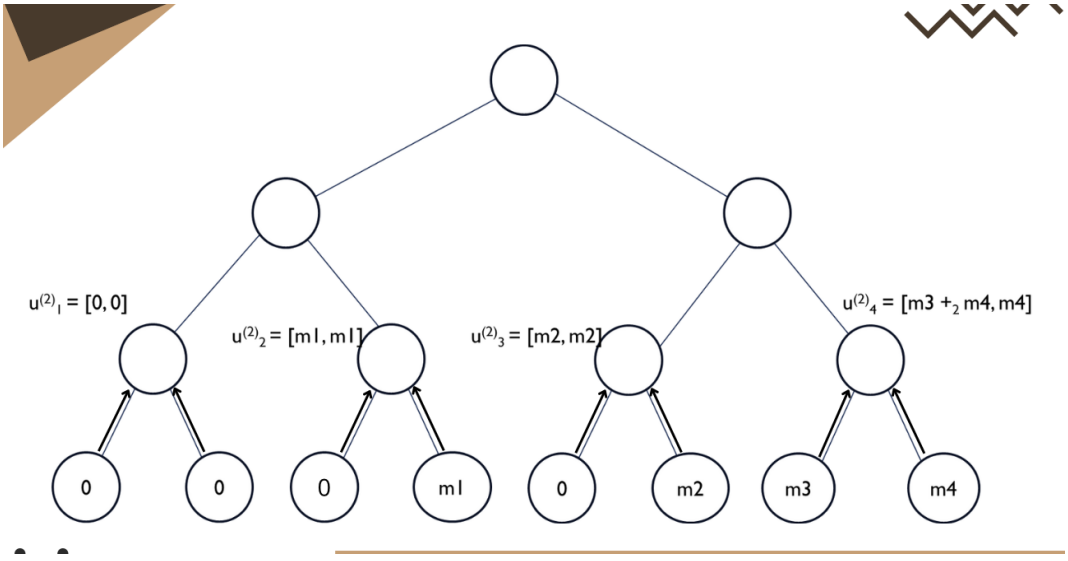


Figure 4: Binary tree with upward message passing annotations $u_i^{(2)}$

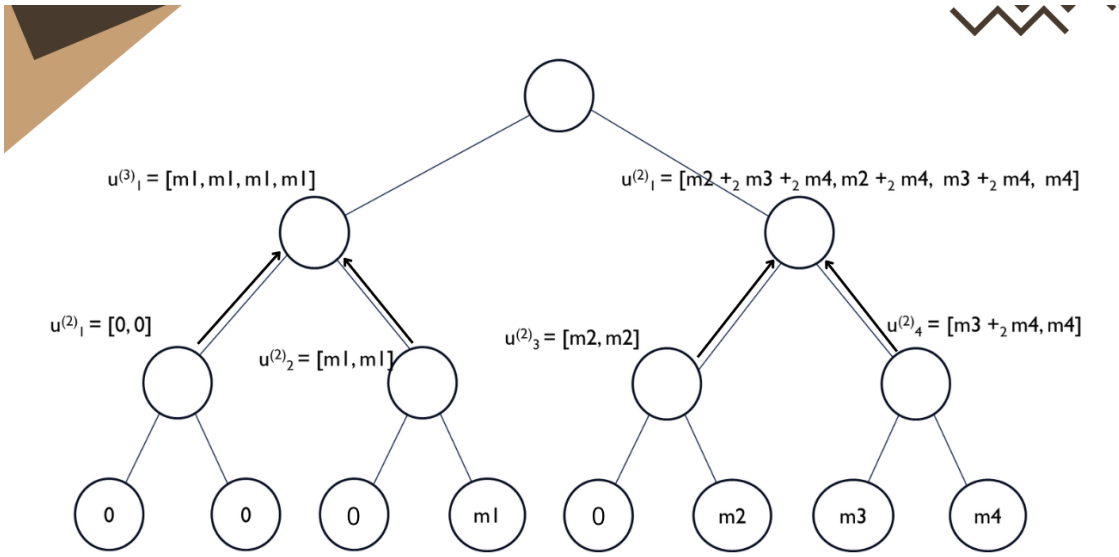


Figure 5: Final stage of upward message passing showing $u_1^{(3)}$ derived from lower-level computations

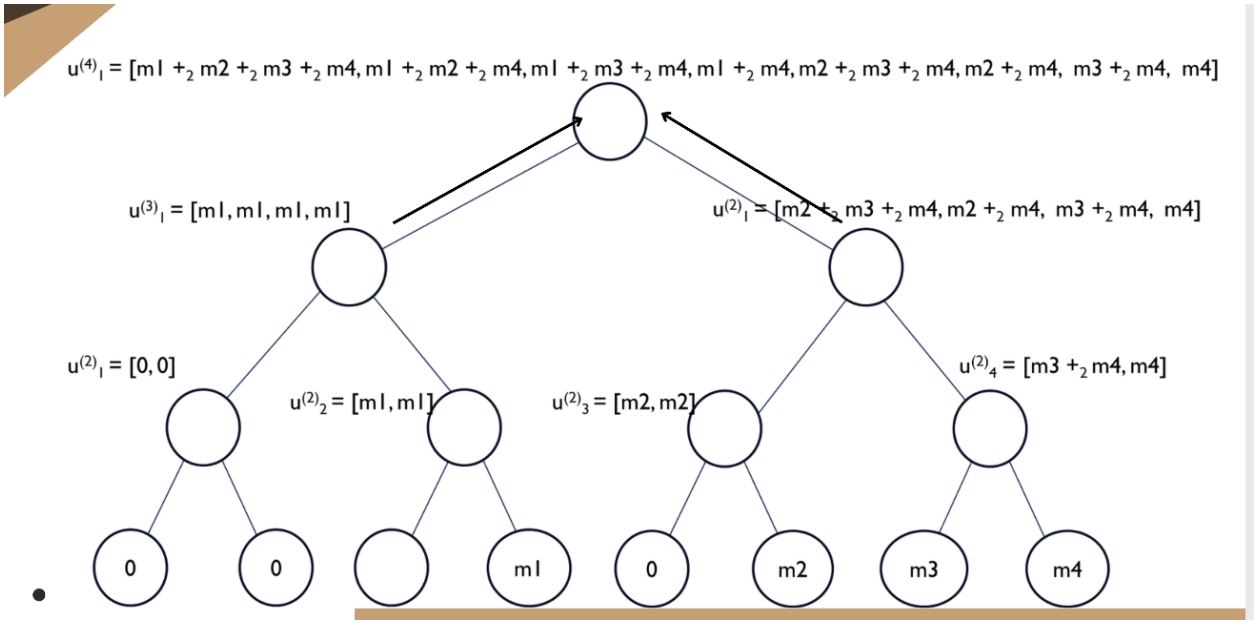


Figure 6: Final upward pass result $u_1^{(4)}$ after binary tree message passing combining all m values

2.3 Comparing G matrix and Tree approach

The **G-Matrix approach** for encoding is a preferred method for understanding, as it simply and directly represents the polar transform. It is also much easier to implement. As a result, it is suggested for simulating and prototyping.

The **tree-based method** for polar encoding is an efficient approach for hardware and real-world system design. It reduces the complexity of both encoding and decoding, and thus supports larger values of N.[7]

For code implementation within this project, we have chosen the **Matrix method**, as it offers lower encoding complexity and higher speed, making it better suited for simulation purposes.

2.4 Calculation and comaparison of Time complexities

2.4.1 Matrix-Based Encoding Approach: Complexity Analysis

Let us define:

- $N = 2^n$ (block length),
- $G_N = B_N F^{\otimes n} \in \mathbb{F}_2^{N \times N}$, where B_N is the bit-reversal permutation matrix and $F = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.

(a) Naive Matrix Multiplication To compute the codeword:

$$x = uG_N$$

where $u \in \mathbb{F}_2^N$, we perform a multiplication of a $1 \times N$ row vector with an $N \times N$ matrix. This naive method requires:

$$T_{\text{naive}}(N) = O(N^2)$$

because:

- Each of the N output bits requires a full inner product with a row of G_N,
- Total $= N \times N = N^2$ bit-wise operations.

(b) Exploiting Kronecker Product Structure Recall that:

$$G_N = B_N F^{\otimes n}$$

Using the recursive Kronecker product structure, we can implement polar encoding in a divide-and-conquer manner. At each of the $\log_2 N$ stages, N bits undergo $N/2$ XOR operations.

Hence, the fast implementation gives:

$$T_{\text{fast}}(N) = O(N \log N)$$

Conclusion:

- **Naive G-matrix encoding:** $O(N^2)$
- **Fast recursive implementation:** $O(N \log N)$

2.4.2 Tree-Based Encoding Approach: Complexity Analysis

This method relies on the recursive structure of polar encoding. At each step, two bits are combined using the kernel F as:

$$(u_0, u_1) \mapsto (u_0 \oplus u_1, u_1)$$

Recursive Structure Let $T(N)$ be the number of XOR operations required for a code of length N. Then:

$$T(N) = 2 \cdot T(N/2) + N/2$$

with base case:

$$T(2) = 1$$

Solving the Recurrence We solve this using a recursion tree:

$$\begin{aligned} T(N) &= 2T(N/2) + N/2 \\ &= 2(2T(N/4) + N/4) + N/2 = 4T(N/4) + N \\ &= 8T(N/8) + \frac{3N}{2} \\ &\dots \\ &= 2^k T(N/2^k) + k \cdot \frac{N}{2} \end{aligned}$$

When $k = \log_2 N$, we reach the base case $T(2) = 1$. Thus:

$$T(N) = N \cdot T(1) + \frac{N}{2} \log_2 N = O(N \log N)$$

Conclusion:

- **Tree-based recursive encoding:** $O(N \log N)$

3 Analysis

3.1 Bhattacharyya Parameter

For a binary-input discrete memoryless channel (B-DMC) $W : \{0, 1\} \to \mathcal{Y}$, the Bhattacharyya parameter is defined as:

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

This measures how confusable the outputs of the channel are — a lower $Z(W)$ indicates a more reliable channel.

3.1.1 Derivation of the Bhattacharyya Parameter $Z(W)$

The Bhattacharyya parameter quantifies the reliability of a communication channel W. For a binary-input discrete memoryless channel, it is defined as:

$$Z(W) = \sum_y \sqrt{W(y|0)W(y|1)} \tag{1}$$

This quantity measures the **confusability** of the channel: the higher the value of $Z(W)$, the more difficult it is to distinguish whether the received symbol originated from bit 0 or 1.

3.1.2 Why Use the Square Root?

The square root in $Z(W)$ is crucial and serves several purposes:

1. **Normalization and Boundness:**
   - Ensures $Z(W) \in [0, 1]$.
   - $Z(W) = 1$ when $W(y|0) = W(y|1)$ (identical distributions).
   - $Z(W) = 0$ when $W(y|0)$ and $W(y|1)$ have disjoint support.
2. **Comparison with Direct Product:**
   - Without the square root, i.e., $\sum_y W(y|0)W(y|1)$, the measure would still be between 0 and 1 but lacks geometric or probabilistic interpretability.
3. **Geometric Interpretation (Hellinger Distance):**

$$H(W(y|0), W(y|1)) = \sqrt{1 - Z(W)} \tag{2}$$

   - Makes $Z(W)$ a proper metric-like measure tied to the Hellinger distance.
4. **Error Bounds in Hypothesis Testing:**

$$P_e \leq \frac{1}{2} Z(W) \tag{3}$$

   - Bounds the Bayesian probability of error.
5. **Chernoff Divergence Derivation:**

$$D_\alpha(W(y|0) \| W(y|1)) = -\ln \left(\sum_y W(y|0)^\alpha W(y|1)^{1-\alpha} \right) \tag{4}$$

   - With $\alpha = 0.5$, this becomes:

$$D_{0.5} = -\ln(Z(W))$$

   - The square root arises naturally here.
6. **Symmetry and Sensitivity:**

$$Z(W) = \sum_y \sqrt{W(y|0)W(y|1)} = \sum_y \sqrt{W(y|1)W(y|0)} \tag{5}$$

   - Symmetric in its arguments.
   - Flattens small values, preventing tail-dominated errors.
7. **Connection to Polar Codes:**

- Channel transformation in polar coding gives:

$$Z(W^-) = 2Z(W) - Z(W)^2, \quad Z(W^+) = Z(W)^2$$

- The square root ensures $Z(W)$ behaves well under polarization.

3.2 BPSK and Noise

This section explores the Binary Phase Shift Keying (BPSK) modulation scheme and the noise models used to evaluate the performance of Polar Codes in various channel conditions. Specifically, we consider two noise scenarios: Additive White Gaussian Noise (AWGN), and a combination of AWGN with Laplace and Uniform noise. Each model provides insights into different aspects of communication system performance.

Binary Phase Shift Keying (BPSK) is a digital modulation technique that encodes binary data by altering the phase of a carrier signal. In this report, we map the bit '0' to a signal value of $+1$ (corresponding to a 0-degree phase) and the bit '1' to -1 (corresponding to a 180-degree phase). Mathematically, for a binary bit $b \in \{0, 1\}$, the transmitted symbol s is given by:

$$s = 1 - 2b \tag{6}$$

BPSK is favored in communication systems due to its simplicity and robustness against noise, making it an ideal choice for evaluating Polar Codes [**?**].

3.3 BI-AWGN Channel Model

Additive White Gaussian Noise (AWGN) is a fundamental noise model used to represent random perturbations in communication channels, such as thermal noise in electronic components. The term "additive" indicates that the noise is superimposed on the transmitted signal, "white" denotes a constant power spectral density across all frequencies, and "Gaussian" refers to the noise amplitude following a normal distribution.

In an AWGN channel, the received signal y is expressed as:

$$y = s + n \tag{7}$$

where s is the transmitted BPSK symbol ($s = \pm 1$), and n is the Gaussian noise with zero mean and variance σ^2. The noise variance is typically determined from the signal-to-noise ratio (SNR) or the ratio of bit energy to noise power spectral density (E_b/N_0). For BPSK with $s = \pm 1$, the bit energy $E_b = 1$, and the noise variance is set as:

$$\sigma^2 = \frac{N_0}{2}, \quad \text{where} \quad N_0 = \frac{1}{E_b/N_0} \tag{8}$$

The AWGN model is widely used because it allows for tractable mathematical analysis. For instance, the bit error rate (BER) for BPSK in an AWGN channel is given by:

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) \tag{9}$$

where $Q(\cdot)$ is the Q-function, defined as $Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-t^2/2} dt$. This closed-form expression facilitates performance evaluation and system design [**?**].

The AWGN channel is a good approximation for scenarios with minimal multipath effects, such as satellite and deep-space communications. However, it is less suitable for terrestrial wireless systems where fading and interference are significant [**?**].

We assume the following model:

- Input $x \in \{0, 1\}$ is modulated using BPSK:

$$0 \mapsto +\sqrt{E_s}, \quad 1 \mapsto -\sqrt{E_s}$$

- Received symbol:

$$y = x + n, \quad n \sim \mathcal{N}(0, \sigma^2)$$

- Signal-to-noise ratio (SNR):

$$\text{SNR} = \frac{E_s}{\sigma^2}$$

3.4 Derivation

We substitute into the definition:

$$Z(W) = \int_{-\infty}^{\infty} \sqrt{W(y|0)W(y|1)}\, dy$$

The conditional densities are:

$$W(y|0) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y-\sqrt{E_s})^2}{2\sigma^2}}, \quad W(y|1) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y+\sqrt{E_s})^2}{2\sigma^2}}$$

Now compute:

$$\begin{aligned} Z(W) &= \int_{-\infty}^{\infty} \sqrt{W(y|0)W(y|1)}\, dy \\ &= \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{4\sigma^2}\left[(y-\sqrt{E_s})^2 + (y+\sqrt{E_s})^2\right]} dy \end{aligned}$$

Simplify the exponent:

$$\begin{aligned} (y - \sqrt{E_s})^2 + (y + \sqrt{E_s})^2 &= y^2 - 2y\sqrt{E_s} + E_s + y^2 + 2y\sqrt{E_s} + E_s \\ &= 2y^2 + 2E_s \end{aligned}$$

So the integrand becomes:

$$\frac{1}{\sqrt{2\pi\sigma^2}} \cdot e^{-\frac{y^2}{2\sigma^2}} \cdot e^{-\frac{E_s}{2\sigma^2}}$$

Now factor out the constant:

$$Z(W) = e^{-\frac{E_s}{2\sigma^2}} \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{y^2}{2\sigma^2}}\, dy$$

But the integral is the total area under a Gaussian PDF:

$$\int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{y^2}{2\sigma^2}} dy = 1$$

3.5 Final Result

$$\boxed{Z(W) = e^{-\frac{E_s}{2\sigma^2}} = e^{-\frac{\text{SNR}}{2}}}$$

3.6 Interpretation

- As $\text{SNR} \to \infty$, $Z \to 0$: channel becomes perfectly reliable.
- As $\text{SNR} \to 0$, $Z \to 1$: channel becomes completely noisy.

This expression is widely used in the Gaussian approximation of polar code construction, where it serves as an initialization for recursive reliability estimation of synthetic channels.

Log-Likelihood Ratio (LLR) for AWGN Channel

Consider a Binary Input AWGN channel where the transmitted symbol $X \in \{+1, -1\}$, and the received symbol is:

$$Y = X + N, \quad \text{where } N \sim \mathcal{N}(0, \sigma^2)$$

The Log-Likelihood Ratio (LLR) for a received symbol y is defined as:

$$\text{LLR}(y) = \log \frac{P(Y = y \mid X = +1)}{P(Y = y \mid X = -1)}$$

Using the Gaussian conditional probabilities:

$$P(Y = y \mid X = x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - x)^2}{2\sigma^2}\right)$$

Substituting into the LLR expression:

$$\text{LLR}(y) = \log \left(\frac{\exp\left(-\frac{(y-1)^2}{2\sigma^2}\right)}{\exp\left(-\frac{(y+1)^2}{2\sigma^2}\right)} \right)$$

$$= \frac{-(y-1)^2 + (y+1)^2}{2\sigma^2} = \frac{2y}{\sigma^2}$$

Thus, the LLR for a Binary Input AWGN channel is:

$$\boxed{\text{LLR}(y) = \frac{2y}{\sigma^2}}$$

By examining the expression for the LLR in an AWGN channel, we observe that since the received symbol $Y \sim \mathcal{N}(\pm 1, \sigma^2)$, the LLR given by

$$\text{LLR}(Y) = \frac{2Y}{\sigma^2}$$

is a linear transformation of a Gaussian random variable. Therefore, the LLR itself also follows a Gaussian distribution. Specifically:

- If $X = +1$ was transmitted, then $Y \sim \mathcal{N}(1, \sigma^2)$, and hence:

$$\text{LLR}(Y) \sim \mathcal{N}\left(\frac{2}{\sigma^2}, \frac{4}{\sigma^4} \cdot \sigma^2\right) = \mathcal{N}\left(\frac{2}{\sigma^2}, \frac{4}{\sigma^2}\right)$$

Thus, the LLR follows a Gaussian distribution with:

$$\boxed{\text{Mean} = \frac{2}{\sigma^2}, \quad \text{Variance} = \frac{4}{\sigma^2}}$$

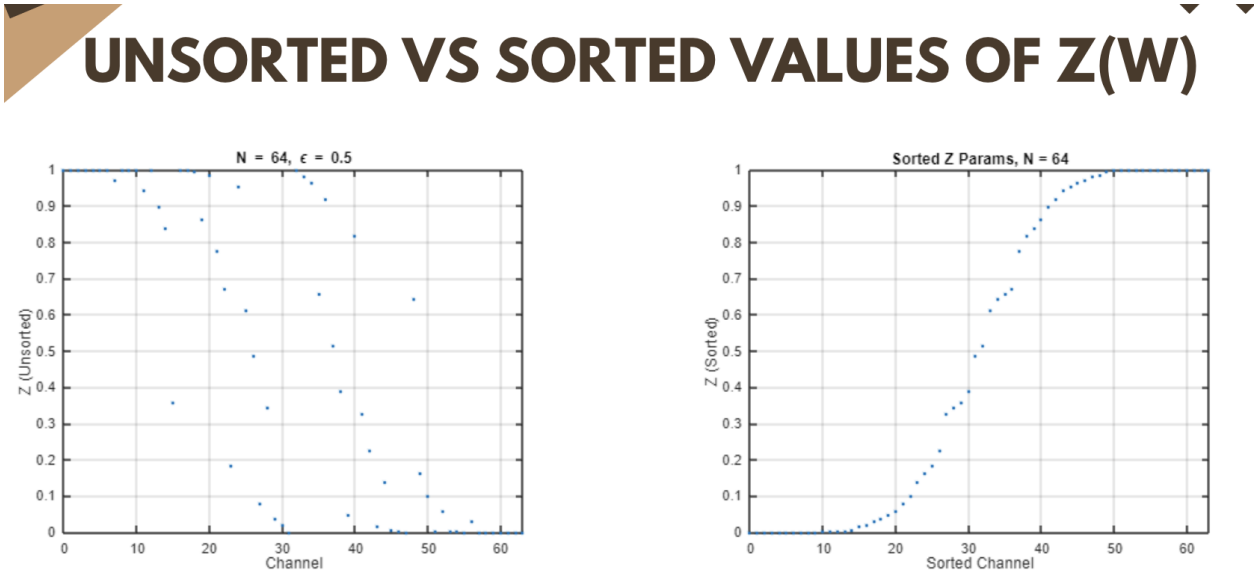


Figure 7: Comparison between unsorted and sorted values of the Bhattacharyya parameters $Z(W)$ for $N = 64$ and erasure probability $\epsilon = 0.5$. The left plot shows the raw distribution of Z values across channels, while the right plot shows the same values after sorting.

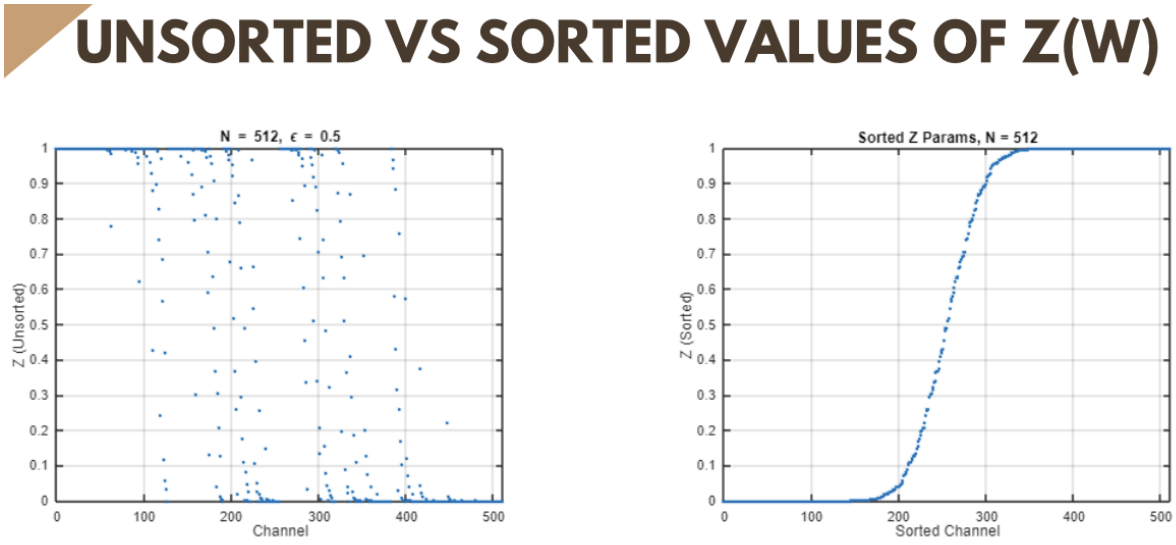


Figure 8: Comparison between unsorted and sorted values of the Bhattacharyya parameters $Z(W)$ for $N = 512$ and erasure probability $\epsilon = 0.5$. The left plot shows the raw distribution of Z values across the synthesized channels, while the right plot shows their sorted version, illustrating the polarization effect.

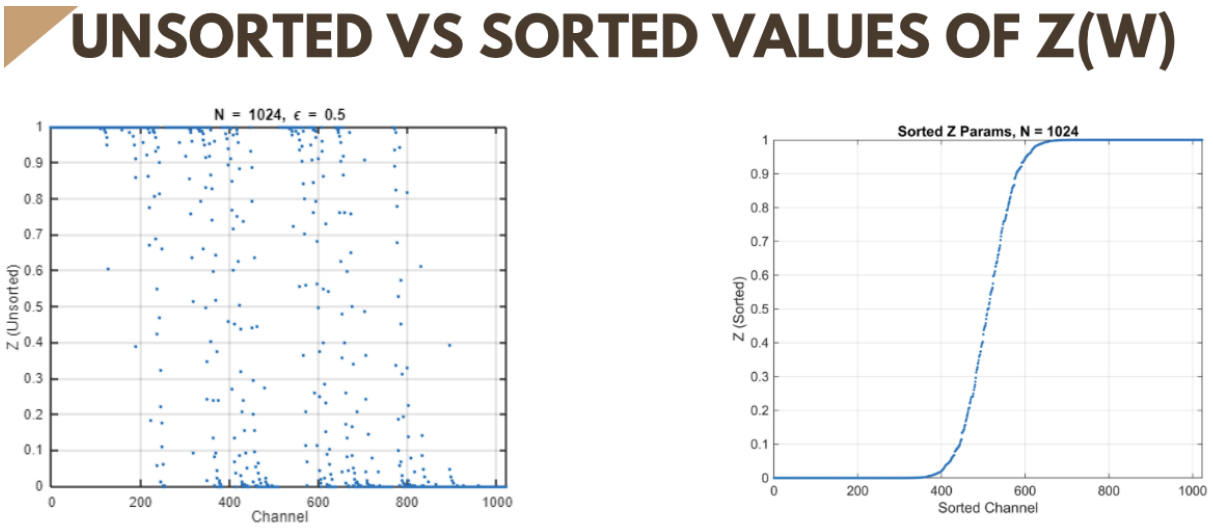


Figure 9: Unsorted vs sorted Bhattacharyya parameters $Z(W)$ for $N = 1024$ and erasure probability $\epsilon = 0.5$. The left plot shows the scattered reliability values across channels, while the right plot illustrates the polarization effect as channels are sorted in ascending order of reliability.

Reliability Sequence

In polar coding, the reliability of each bit-channel is determined using the Bhattacharyya parameter, which quantifies the likelihood of error in each synthetic channel. The bit-channels are then sorted from most to least reliable based on their Bhattacharyya parameters, resulting in a *reliability sequence*. This sequence is fixed for a given block length N.

For instance, when $N = 16$, one possible reliability sequence (from most to least reliable) is:

$$[1, 2, 3, 5, 9, 4, 6, 10, 7, 11, 13, 8, 12, 14, 15, 16]$$

To form a polar code, we assign the K most reliable positions to carry the **message bits**, and the remaining $N - K$ positions to **frozen bits**, which are usually set to 0. This forms the basis of the polar coding mechanism, effectively converting a noisy communication channel into a mixture of highly reliable and unreliable bit-channels.

Consider a message vector:

$$\mathbf{u} = [1, 1, 1, 1, 1, 1, 1, 1]$$

containing 8 message bits. Using the reliability sequence, we place frozen bits at the first 8 least reliable positions and map the message bits to the 8 most reliable ones. This results in the following encoded vector:

$$\text{Encoded Vector} = [0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1]$$

This coding strategy results in a $(16, 8)$ polar code, where:

- $N = 16$: total number of bits,
- $K = 8$: number of message bits,
- $N - K = 8$: number of frozen bits.

This demonstrates the concept of **channel polarization**, in which the bit-channels either become highly reliable (approaching noiseless) or extremely unreliable (approaching pure noise).

4 Polar Decoding

The two primary methods for decoding polar codes are:

- **Successive Cancellation Decoder (SC Decoder)**
- **Successive Cancellation List Decoder (SCL Decoder)**

4.1 Successive Cancellation Decoding

Introduction

Successive Cancellation (SC) decoding is one of the fundamental algorithms used for decoding polar codes. It operates in a sequential manner, processing one bit at a time in a specific order. For each bit, the decoder makes a decision based on previously decoded bits and the received channel output.

The decoding process can be visualized as a traversal through a binary tree, where each node corresponds to a decision point for a particular bit. The traversal path is influenced by earlier decisions, as the decoding of each bit depends on the reliability of the channel and the previously determined bit values. Due to this recursive and structured approach, SC decoding is relatively simple in terms of complexity, making it suitable for practical hardware implementations, especially in systems with constrained resources.

However, the sequential nature of SC decoding can also be a limitation in scenarios requiring low latency or very high reliability, since early decision errors can propagate through the decoding tree.

Key Concept

A core principle in polar coding, and specifically in SC decoding, is the distinction between **frozen bits** and **information bits**. This concept leverages the phenomenon of *channel polarization*, where individual channels (or bit positions) become either highly reliable or highly unreliable as the block length increases.

- **Frozen bits** are placed in the least reliable positions as determined by the channel polarization process. These bits are fixed to known values (usually 0) and are shared between the encoder and decoder beforehand. Since their values are known, they do not need to be decoded and are instead used to assist the decoding of other bits by providing known references.
- **Information bits**, on the other hand, are assigned to the most reliable positions in the codeword. These bits contain the actual user data and must be accurately decoded at the receiver. Their placement in highly reliable positions increases the likelihood of correct recovery even in the presence of channel noise.

By strategically assigning frozen and information bits based on the reliability of each channel, SC decoding exploits the structure of polar codes to maximize error-correcting performance while maintaining low complexity. Although SC decoding is not optimal in terms of error performance compared to more advanced decoders, it serves as a foundational and efficient method in the family of polar code decoders.

- $\mathbf{u} = [u_1, u_2]$: The input bit vector, where u_1 and u_2 are the individual message or frozen bits.
- $\mathbf{x} = [x_1, x_2]$: The encoded output vector obtained from applying the polar transform to $\mathbf{u}$.

Min-Sum Function in SC Decoding

In the Successive Cancellation (SC) decoding algorithm for polar codes, decoding is performed in a recursive manner based on the received soft information from the channel.

Let r_1 and r_2 be the soft beliefs (e.g., log-likelihood ratios or LLRs) corresponding to the received values x_1 and x_2, respectively.

To estimate the bit u_1, a function known as the **min-sum** function is used to combine the beliefs r_1 and r_2. This function approximates the operation used in belief propagation decoders and is computationally efficient.

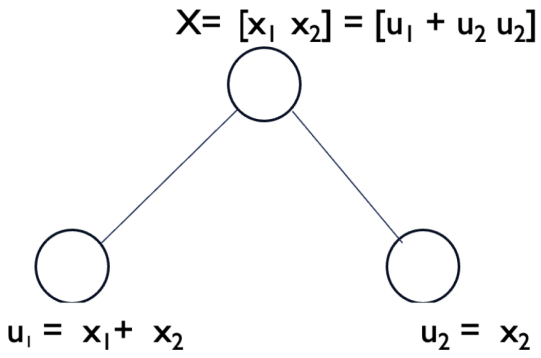


Figure 10: Binary tree structure in SC decoding.

Min-Sum Function

The min-sum function $f(r_1, r_2)$ is defined as:

$$f(r_1, r_2) = \text{sgn}(r_1) \cdot \text{sgn}(r_2) \cdot \min\left(|r_1|, |r_2|\right)$$

Where:

- $\text{sign}(r)$ gives the sign of the value r
- $|r|$ denotes the absolute value
- The output $f(r_1, r_2)$ is the soft belief for u_1

The hard decision for u_1 is then obtained by applying a threshold at 0:

$$\hat{u}_1 = \begin{cases} 0, & \text{if } f(r_1, r_2) \geq 0 \\ 1, & \text{if } f(r_1, r_2) < 0 \end{cases}$$

This means that if the soft belief suggests that u_1 is more likely to be 0 (i.e., $f \geq 0$), it is decoded as 0, otherwise as 1.

Decoding u_2 Based on the Estimated $\hat{u}_1$ (Case Analysis)

After estimating the first bit $\hat{u}_1$ using the min-sum function, the Successive Cancellation (SC) decoder assumes this estimate is correct and proceeds to decode the second bit u_2.

- **If** $\hat{u}_1 = 0$:
  From the encoding rule $x_1 = u_1 + u_2$ and $x_2 = u_2$, if $u_1 = 0$, then:

  $$x_1 = u_2, \quad x_2 = u_2 \Rightarrow \mathbf{x} = [u_2, u_2]$$

  This implies that both r_1 and r_2 are soft beliefs (e.g., LLRs) corresponding to u_2. Therefore, the decoder adds these two beliefs to produce a final belief for u_2:

  $$g(r_1, r_2, \hat{u}_1) = r_2 + r_1$$

- **If** $\hat{u}_1 = 1$:
  In this case:

  $$x_1 = 1 + u_2, \quad x_2 = u_2 \Rightarrow \mathbf{x} = [1 + u_2, u_2]$$

Here, r_1 represents the belief about the complement of u_2, and r_2 is still the belief for u_2. Thus, to realign both beliefs to refer to u_2, we negate r_1, giving:

$$g(r_1, r_2, \hat{u}_1) = r_2 - r_1$$

Generalized Function

Both cases are captured compactly by the following function:

$$g(r_1, r_2, \hat{u}_1) = r_2 + (1 - 2\hat{u}_1) \cdot r_1$$

This equation dynamically adjusts the sign of r_1 based on the value of $\hat{u}_1$:

- If $\hat{u}_1 = 0$: $g = r_2 + r_1$
- If $\hat{u}_1 = 1$: $g = r_2 - r_1$

Hard Decision for u_2

The soft belief $g(r_1, r_2, \hat{u}_1)$ is then converted into a hard decision using a threshold at 0:

$$\hat{u}_2 = \begin{cases} 0, & \text{if } g(r_1, r_2, \hat{u}_1) \geq 0 \\ 1, & \text{if } g(r_1, r_2, \hat{u}_1) < 0 \end{cases}$$

This means that the bit u_2 is estimated to be 0 if the soft belief is non-negative, and 1 otherwise.

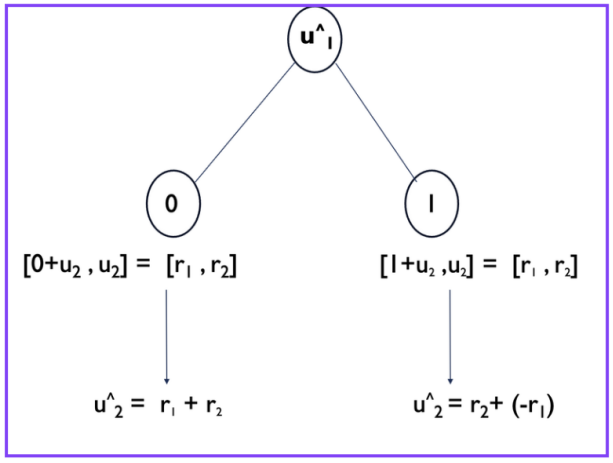


Figure 11:

Tree-Based Decoding Using Recursive Traversal

The decoding algorithm follows a recursive traversal of a binary tree, comprising the following three steps:

Step 1: Left Traversal (L)

- Start at the root node and traverse to the left child.
- Apply the **minsum** (or f) function to compute the beliefs passed to the left child.
- Repeat this step recursively until a leaf node is reached.
- At a leaf node, compute the belief for the first bit and return it to the parent node.

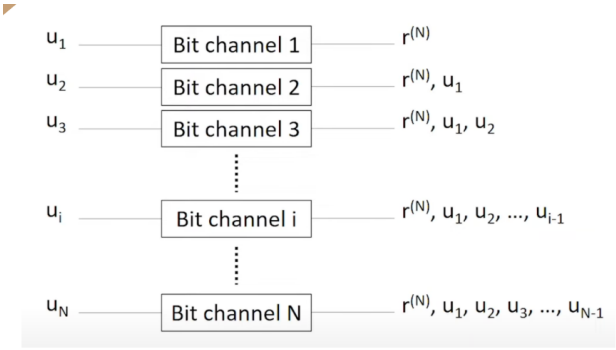


Figure 12: General Results for N-bit Channel: At each node, decoding uses previous results, as illustrated in the figure above.

$$L(u, v) = f(u, v) = \text{sgn}(u) \cdot \text{sgn}(v) \cdot \min(|u|, |v|)$$

Step 2: Right Traversal (R)

- After receiving the belief from the left child, traverse to the right child.
- Use the g function, which takes:
  - The parent node's incoming belief (u),
  - The decoded bit from the left child ($\hat{u}_1$),

  to compute the belief for the right child.

$$R(u, v, \hat{u}_1) = g(u, v, \hat{u}_1) = v + (1 - 2\hat{u}_1) \cdot u$$

Step 3: Upward Move (U)

- After receiving the belief from the right child, the node now has decoded values from both children.
- It combines both results and sends the combined belief upward to its parent node.
- This allows decoding to proceed up the tree.

$$U(\hat{u}_1, \hat{u}_2) = (\hat{u}_1 \oplus \hat{u}_2, \hat{u}_2)$$

Limitations of SC Decoding

Successive cancellation (SC) decoding, despite its theoretical guarantees, suffers from several critical limitations that require more sophisticated approaches. The main problems with SC decoding being:

- **Error Propagation:** SC decoding makes hard decisions for each bit sequentially. If an early bit is decoded incorrectly, this error propagates through the decoding tree, affecting all subsequent bits. This is because SC decoding does not review previous decisions or consider alternative decoding paths.
- **Finite-Length Performance Gap:** While SC decoding achieves capacity asymptotically as block length approaches infinity, it exhibits poor performance at practical block lengths, i.e., for 256,512,1024.
- **Sensitivity to Channel Conditions:** SC performance deteriorates rapidly in moderate to low SNR environments, making it unsuitable for real-world applications.
- **Lack of Error Detection:** Basic SC provides no built-in mechanism to detect decoding failures.

These limitations lead to a significant performance gap compared to other modern coding techniques like LDPC and Turbo codes, especially at finite block lengths.

4.2 SCL Decoding Principle

Successive Cancellation List (SCL) decoding addresses these limitations by maintaining multiple candidate paths simultaneously, effectively exploring the most promising branches of the decoding tree.

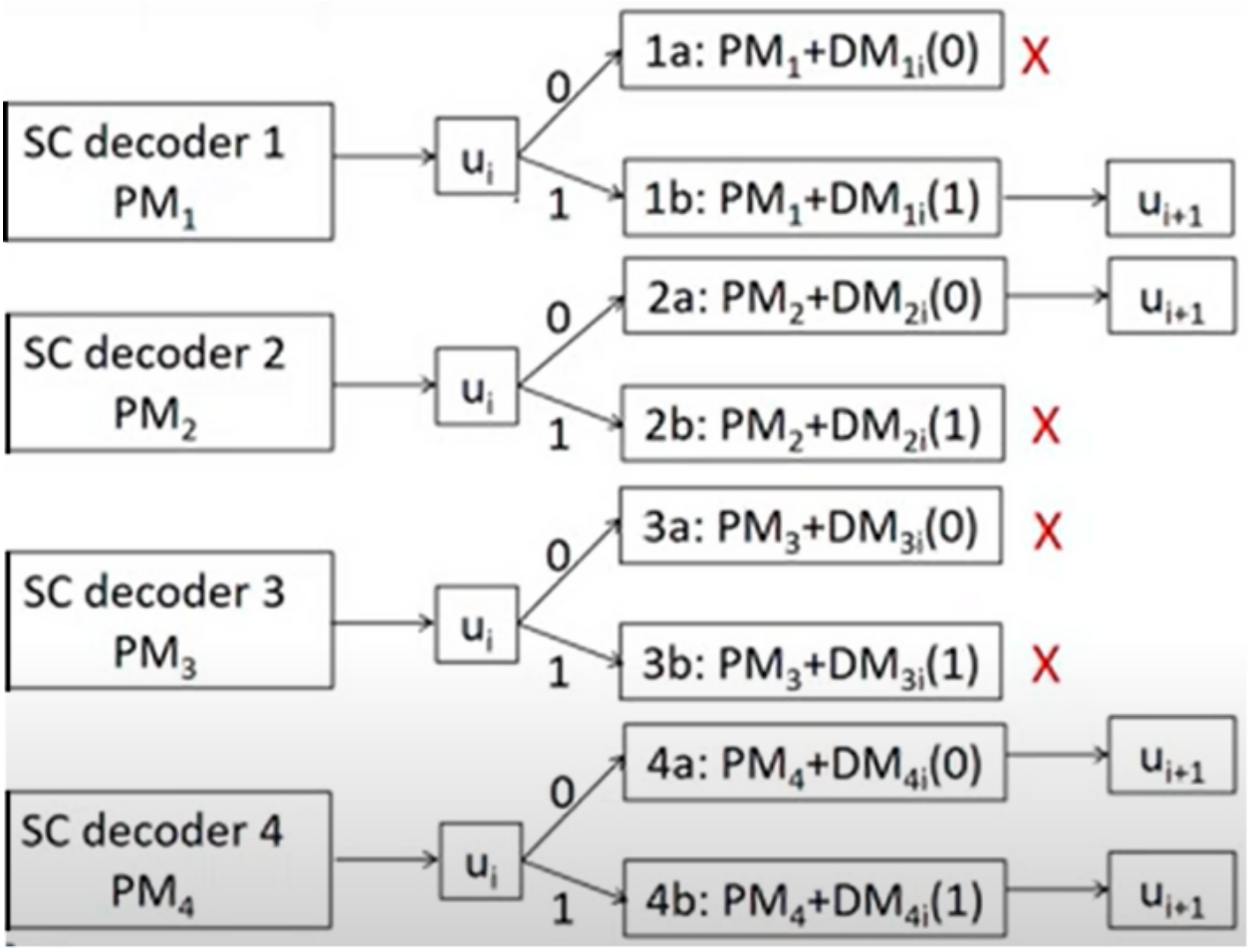


Figure 13: SCL decoding with list size $L = 4$. Paths marked with X are pruned based on path metrics.

Decision Metrics and Path Metrics

The path management in SCL decoding[7][14] relies on two critical metrics that quantify the reliability of bit decisions and complete decoding paths:

Decision Metric (DM) Decision metrics quantify the reliability cost associated with assigning a specific bit value (0 or 1) at a particular position:

- **Definition:** A measure that represents how "costly" or "unlikely" a particular bit decision is based on channel information.
- **Mathematical Formulation:** For LLR-based SCL decoding, the decision metric is calculated as:

$$\mathrm{DM}_i(\hat{u}_i) = \ln(1 + e^{-(1-2\hat{u}_i)\cdot L_i}) \tag{10}$$

where $\hat{u}_i$ is the bit value (0 or 1) and L_i is the log-likelihood ratio at position i.

- **Min-Sum Approximation:** For computational efficiency, DM can be approximated as:

$$\mathrm{DM}_i(\hat{u}_i) \approx \max(0, -(1 - 2\hat{u}_i) \cdot L_i/2) \tag{11}$$

Path Metric (PM) Path metrics represent the cumulative reliability of an entire candidate decoding path:

- **Definition:** Sum of decision metrics along a path, indicating its overall likelihood.
- **Recursive Calculation:** Updated for each new bit decision:

$$\mathrm{PM}_{\mathrm{new}} = \mathrm{PM}_{\mathrm{old}} + \mathrm{DM}_i(\hat{u}_i) \tag{12}$$

- **Selection Principle:** Paths with smaller metrics are more reliable (lower penalty) and therefore maintained in the list.

As shown in Figure 13, each SC decoder path (e.g., PM_1, PM_2) maintains its own path metric. When processing bit u_i, each path branches into paths for bit values 0 and 1, with updated metrics (e.g., PM_1+$\mathrm{DM}_{1,i}(0)$, PM_1+$\mathrm{DM}_{1,i}(1)$). The paths marked with X (1a, 2b, 3a, 3b) are pruned because their metrics exceed the threshold required to maintain list size $L = 4$.

SCL Algorithm

The SCL decoder maintains a list of L most likely decoding paths. For each bit position:

1. **Path Expansion:** For each existing path in the list, create two extended paths corresponding to the bit value 0 and 1 (for information bits) or only value 0 (for frozen bits).
2. **Path Pruning:** Select the L paths with the smallest path metrics (highest likelihoods) and discard the rest.

As shown in Figure 13, the decoding process starts with L SC decoders. Each decoder splits into two paths when decoding an information bit, doubling the number of candidates. After pruning, only the L best paths continue to the next bit.

4.3 CRC-Aided SCL Decoding (CA-SCL)

To further improve the performance of SCL decoding, Cyclic Redundancy Check (CRC) is incorporated as an outer code.

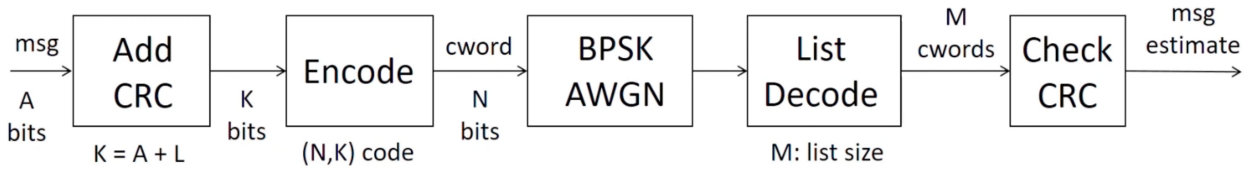


Figure 14: Block diagram of CRC-aided SCL decoding system

CRC Integration

The CRC-aided SCL approach follows these steps:

1. **Encoding Phase:** Append CRC bits to the information bits before polar encoding

$$K = A + L \tag{13}$$

where A is the number of information bits and L is the CRC length.

2. **Decoding Phase:** After SCL decoding produces M candidate codewords, validate each using CRC
3. **Final Selection:** Choose the most reliable path among those that pass CRC validation

Mathematical Formulation

In LLR-based SCL decoding, the path metric update becomes:

$$\text{PM}_{\text{new}} = \text{PM}_{\text{old}} + \ln(1 + e^{-(1-2\hat{u}_i)L_i}) \tag{14}$$

This can be simplified using the min-sum approximation:

$$\text{PM}_{\text{new}} \approx \text{PM}_{\text{old}} + \max(0, -(1-2\hat{u}_i)L_i/2) \tag{15}$$

For CRC validation, a path $\mathbf{u}$ is considered valid if:

$$\text{CRC}(\mathbf{u}) \equiv 0 \pmod{g(x)} \tag{16}$$

where $g(x)$ is the generator polynomial of the CRC.

Tree-Based Implementation

The SCL decoding process can be visualized as a tree search, where each level represents a bit position and branches represent bit decisions.

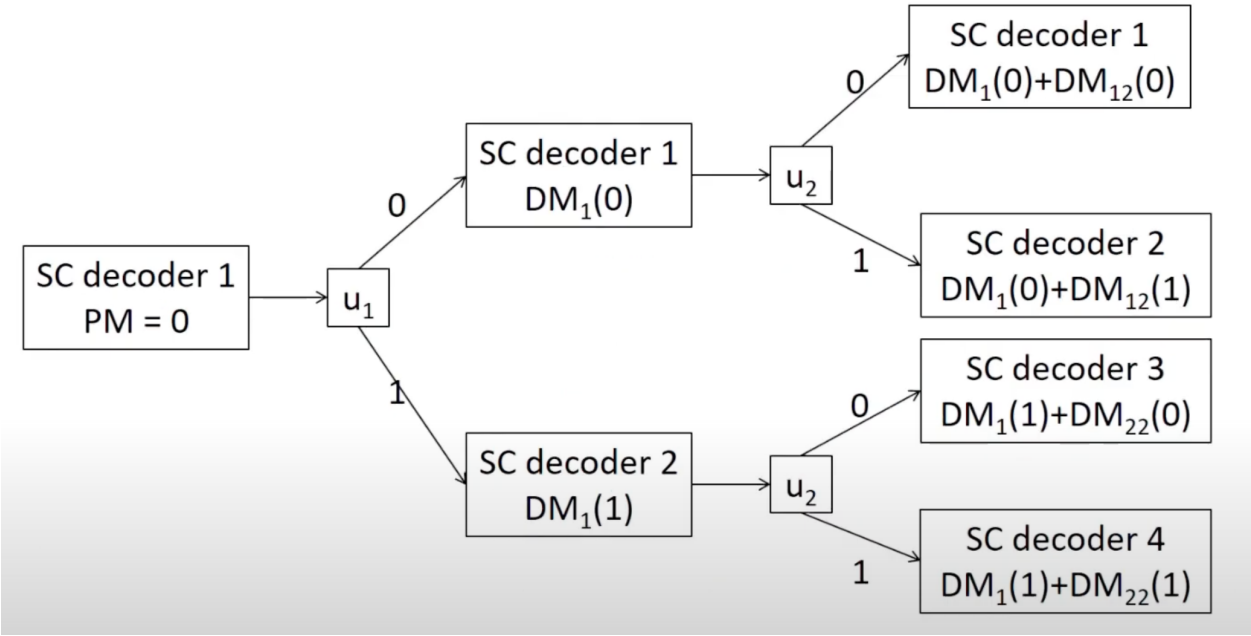


Figure 15: Tree representation of SCL decoding for the first two bits

As shown in Figure 15, the decoding process:

- Starts with a single path (PM = 0)
- Branches for each bit decision (0 or 1)
- Accumulates decision metrics (DM) along paths

- Prunes paths exceeding list size L

This tree-based visualization helps understand how SCL explores multiple candidate paths simultaneously, significantly improving error-correction performance compared to conventional SC decoding.

Performance Analysis

CRC-aided SCL decoding offers significant advantages:

- **Error-Correction Performance:** Approaches maximum-likelihood decoding with reasonable list sizes ($L = 8$ or $L = 16$)
- **Error-Detection Capability:** CRC validation identifies and eliminates incorrect paths
- **Computational Efficiency:** LLR-based implementation reduces memory requirements by 50% compared to likelihood-based approaches
- **Practical Viability:** Enabled adoption in 5G NR for control channel coding

The performance improvement comes at the cost of increased complexity, which scales linearly with the list size L.

4.4 Conclusion

Successive Cancellation List decoding with CRC validation bridges the gap between theoretical capacity and practical performance for polar codes. The ability to maintain multiple candidate paths and select the most reliable one using CRC significantly enhances error-correction capability, enabling polar codes to compete with state-of-the-art error-correcting codes like LDPC and Turbo codes.

5 Simulations

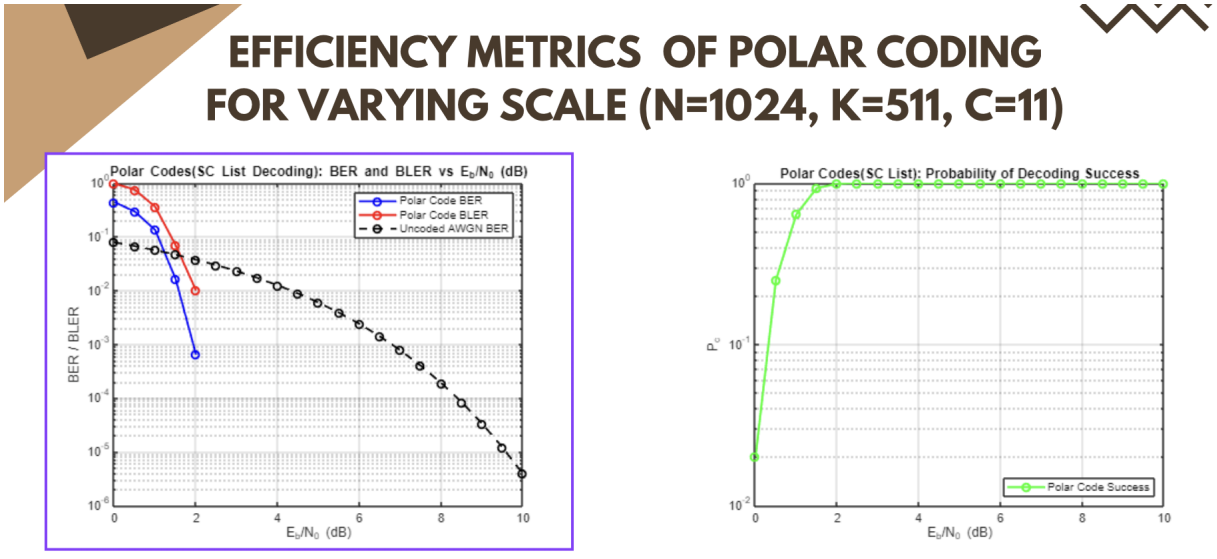


Figure 16: Efficiency metrics of polar coding for varying scale ($N = 1024, K = 511, C = 11$). The left graph compares the BER and BLER performance of Polar Codes under SC List decoding with uncoded AWGN channel BER. The right graph shows the probability of successful decoding as a function of E_b/N_0.

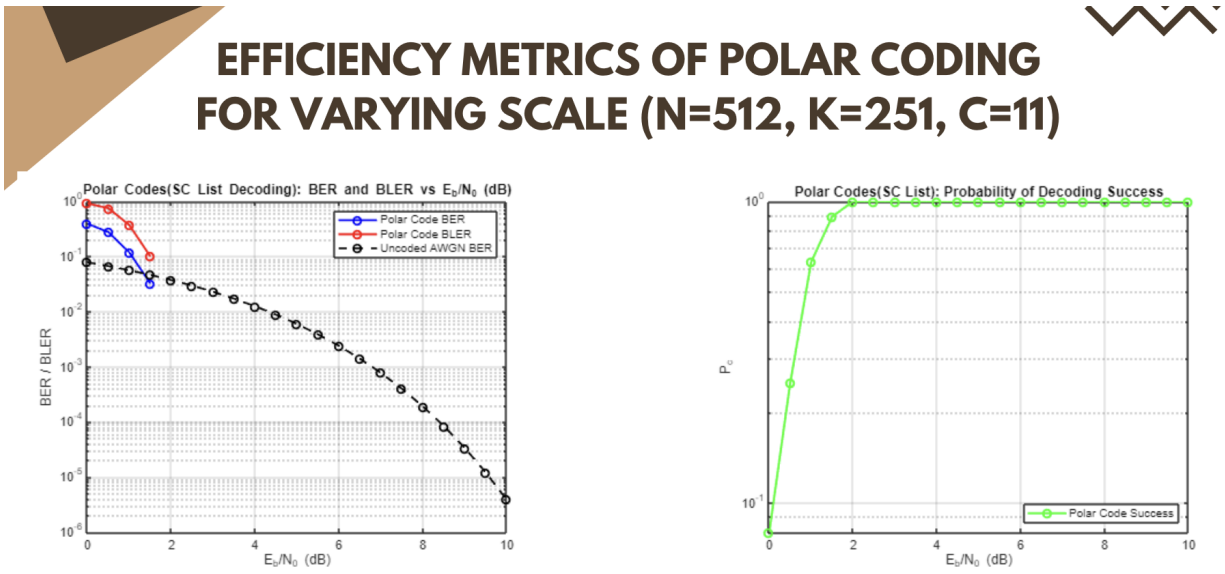


Figure 17: Efficiency Metrics of Polar Coding for Varying Scale (N=512, K=251, C=11)

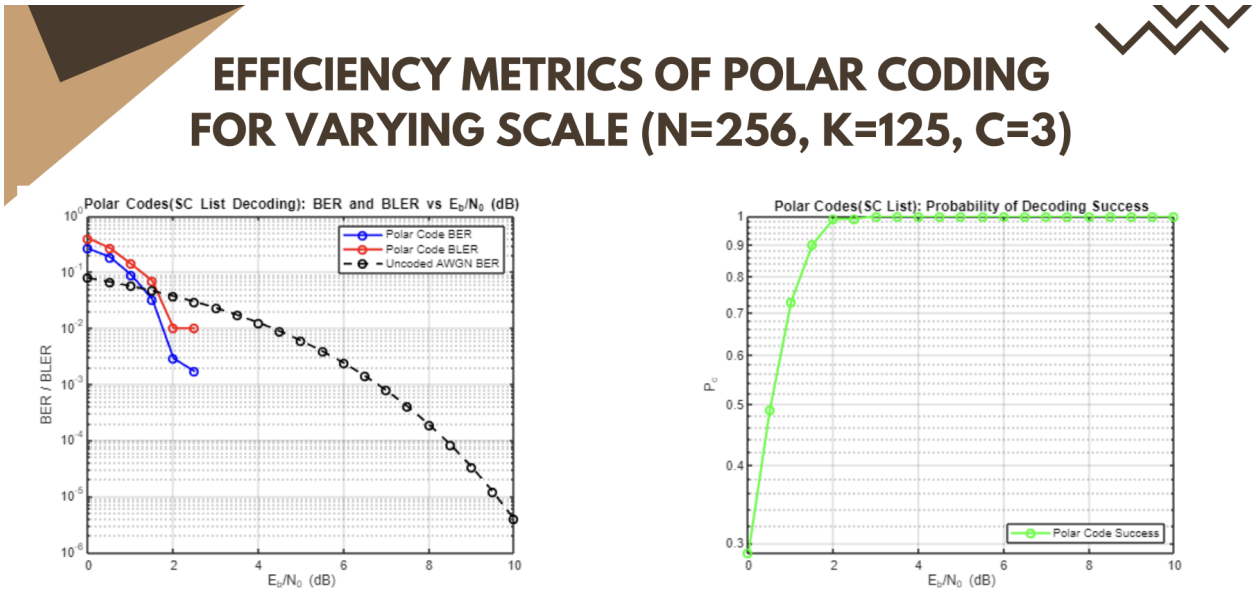


Figure 18: Efficiency Metrics of Polar Coding for Varying Scale (N=256, K=125, C=3)

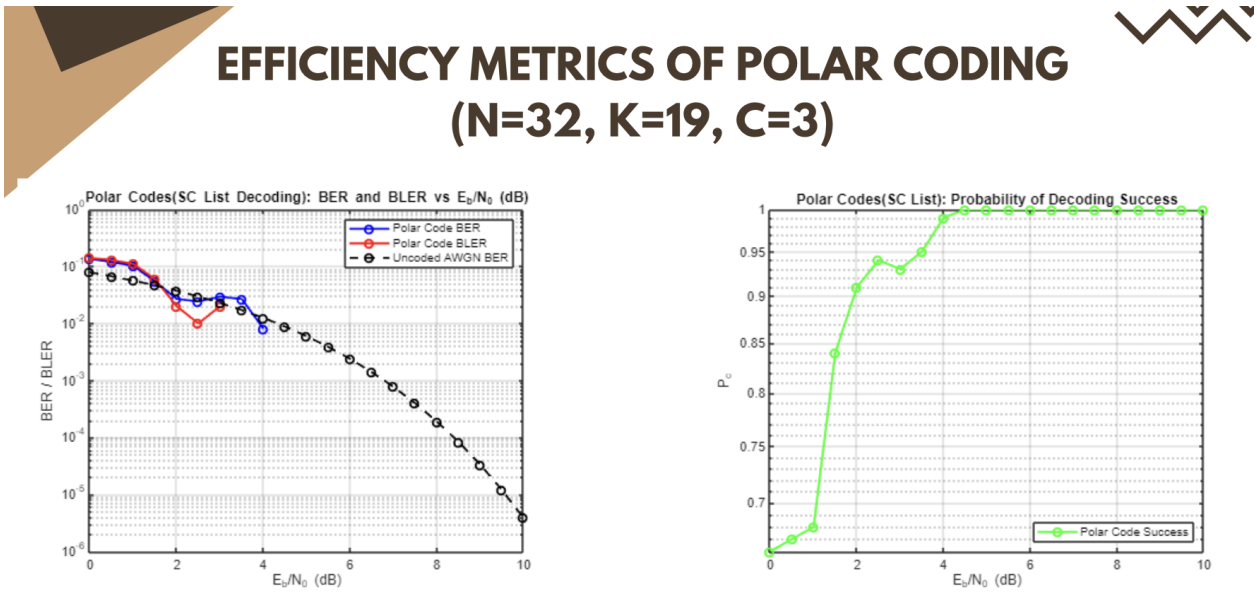


Figure 19: Efficiency Metrics of Polar Coding for Varying Scale (N=32, K=19, C=3)

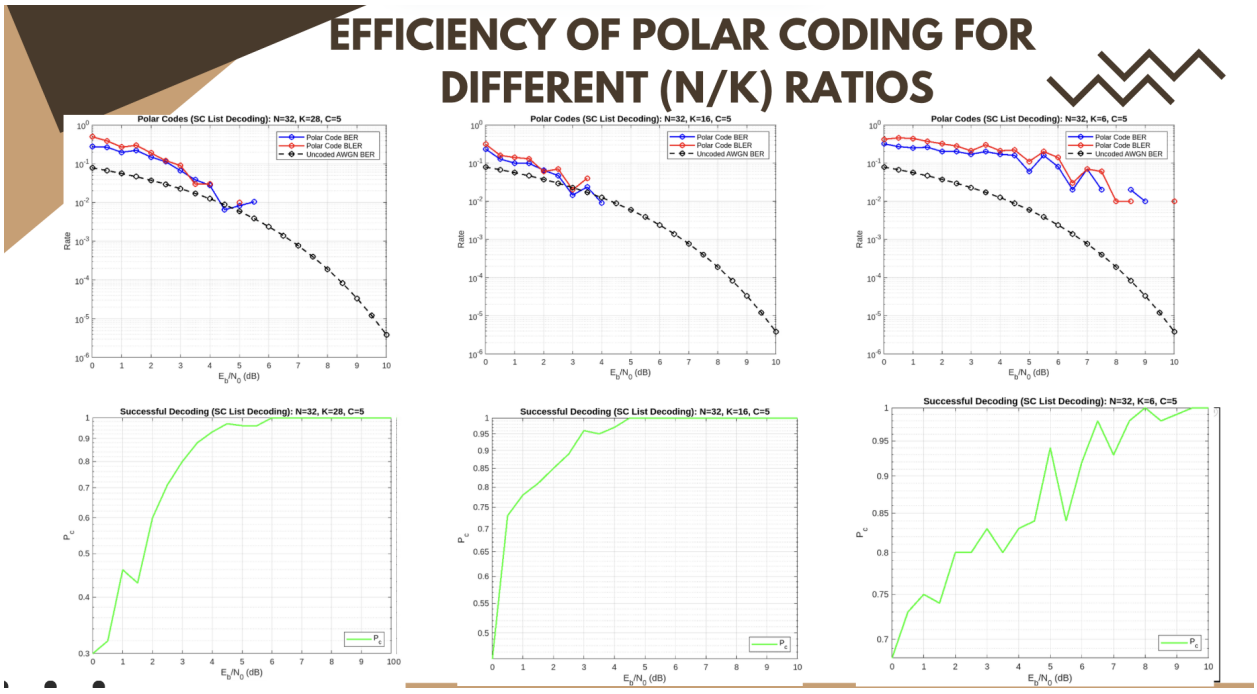


Figure 20: Efficiency of Polar Coding for Different (N/K) Ratios

6 Shannon's Capacity

6.1 Achieving Channel Capacity

Polar Codes, introduced by Erdal Arıkan, are a class of error-correcting codes designed to achieve the symmetric capacity of binary-input discrete memoryless channels (B-DMCs). This section derives how Polar Codes strive to achieve Shannon's channel capacity, and analyzes their performance.

6.2 Shannon's Channel Capacity Theorem

Shannon's channel capacity theorem defines the maximum rate at which reliable communication is possible over a noisy channel. For a B-DMC W with input $x \in \{0,1\}$ and output $y \in \mathcal{Y}$, the symmetric capacity $I(W)$ is:

$$I(W) = \sum_{y\in\mathcal{Y}} \sum_{x\in\{0,1\}} \frac{1}{2} W(y|x) \log_2 \left(\frac{2W(y|x)}{W(y|0) + W(y|1)} \right).$$

This mutual information $I(W)$ (in bits per channel use) is the capacity when inputs are equiprobable. Shannon proved that for any rate $R < I(W)$, there exists a code of block length N such that the error probability $P_e \to 0$ as $N \to \infty$ under optimal decoding [16].

6.3 Derivation of Polar Codes Achieving Capacity

Polar Codes achieve $I(W)$ through *channel polarization*. Consider $N = 2^n$ uses of W. The polarization transform constructs N synthetic channels $W_N^{(i)}$ via a recursive process. Define the basic transformation for two channels:

$$\text{- } W_2^{(1)}(y_1, y_2|u_1) = \sum_{u_2\in\{0,1\}} \frac{1}{2} W(y_1|u_1 \oplus u_2) W(y_2|u_2) \quad \text{(worse channel)},$$
$$\text{- } W_2^{(2)}(y_1, y_2, u_1|u_2) = \frac{1}{2} W(y_1|u_1 \oplus u_2) W(y_2|u_2) \quad \text{(better channel)},$$

where u_1, u_2 are inputs, and $\oplus$ denotes modulo-2 addition. The generator matrix $G_N = F^{\otimes n}$, with $F = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, applies this transformation recursively.

The key property is that as $N \to \infty$, the capacities $I(W_N^{(i)})$ polarize:

$$I(W_N^{(i)}) \to \begin{cases} 1 & \text{for a fraction } I(W) \text{ of indices,} \\ 0 & \text{for a fraction } 1 - I(W) \text{ of indices.} \end{cases}$$

This is quantified using the Bhattacharyya parameter $Z(W)$, where $Z(W_N^{(i)}) \approx 0$ for reliable channels and ≈ 1 for unreliable ones. For a code of rate $R = K/N < I(W)$, select the K indices with the lowest $Z(W_N^{(i)})$ for information bits, freezing the rest to 0.

The error probability under successive cancellation (SC) decoding is bounded by:

$$P_e \leq \sum_{i\in\mathcal{A}} Z(W_N^{(i)}) \leq N \cdot 2^{-N^\beta}, \quad \beta < \frac{1}{2}.$$

Where $\mathcal{A}$ is the set of information bit indices. As $N \to \infty, P_e \to 0$, proving Polar Codes achieve $I(W)$ with complexity $O(N \log N)$ for encoding and decoding [1].

7 Analysis on Large-small Sorting for Successive Cancellation List Decoding of Polar Codes

Introduction

- Successive Cancellation List (SCL) decoding is used for finite length error correction in polar codes using a metric sorting method.
- As the list size increases, sorting metrics increases overall latency.
- To reduce latency, Kyungpil Lee and In-Cheol Park proposed a method in their paper *"Large-small Sorting for Successive Cancellation List Decoding of Polar Codes"* that reduces latency for larger list sizes.

Notations

- $\mathbf{m} = [m_0, m_1, m_2, \ldots, m_{2L-1}]$ – metrics of $2L$ child candidates to be sorted.
- $\mathbf{n} = [n_0, n_1, n_2, \ldots, n_{L-1}]$ – metrics of L parent candidates from the previous decoding step.
- $\mathbf{a} = [a_0, a_1, a_2, \ldots, a_{L-1}]$ – absolute LLR values representing incremental reliability used in computing child path metrics.

2L-to-L Metric Sorting Method

- A metric is simply a numerical measure which helps to calculate reliability of a particular decoding path.
- In the implementation based on LLR, 2L child candidates are calculated from the LLR value of parent.
- Metric m is calculated as:[18]

$$m_{2l} = n_l \quad \text{for } l = 0, 1, \ldots, L-1 \tag{*}$$

$$m_{2l+1} = n_l + a_l \quad \text{for } l = 0, 1, \ldots, L-1 \tag{*}$$

  Since $a_l > 0$, we know that $m_{2l} \leq m_{2l+1}$.
  If the parent metrics $\mathbf{n}$ are sorted, then $m_{2l} \leq m_{2(l+1)}$.
- Only L smallest metrics from m are selected as parents for the next decoding step.
- This is known as the "2L-to-L metric sorting" process.

Mathematical Explanation Behind Comparison in Sorting

Consider $L = 2$:

$$\begin{aligned} Given : n_0 &\leq n_1 \\ a_0, a_1 &\geq 0 \\ So, m_0 = n_0, &\quad m_2 = n_1 \\ m_1 = n_0 + a_0, &\quad m_3 = n_1 + a_1 \end{aligned}$$

Then:

$$m_0 \leq m_1 \quad \text{and} \quad m_1 \leq m_3 \Rightarrow m_0 \leq m_3$$

Hence, only one comparison m_1 and m_2 is needed to determine the best two of the four.

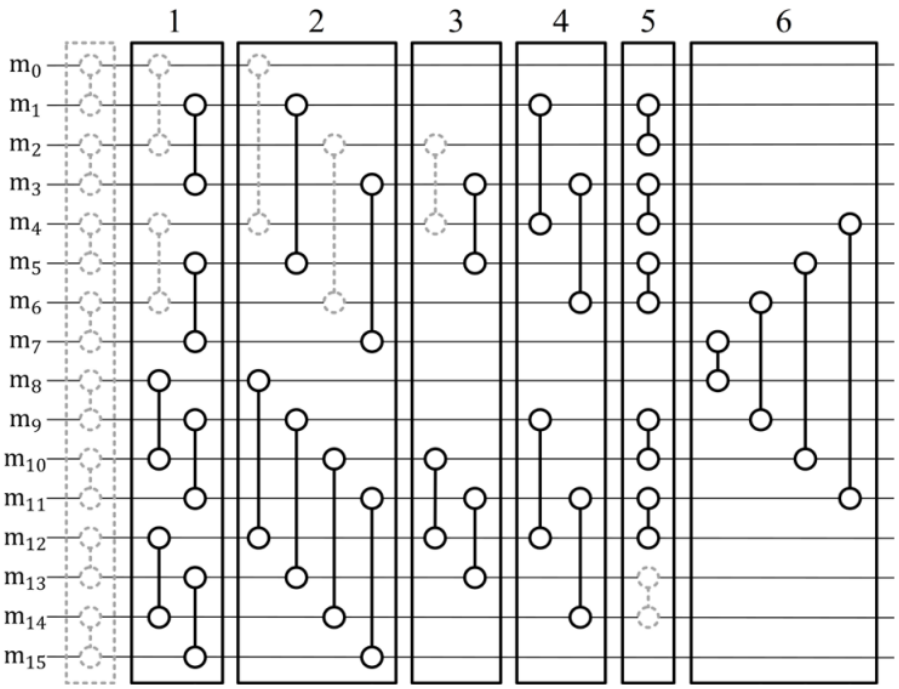


Figure 2. Half-sorted pairwise metric sorting (HS-PMS) architecture [20] for *L* = 8. A rectangular box represents a comparison stage and the dotted line denotes a pruned comparison.

Extending to $L = 8$, we observe:

- Some comparisons are pruned (not needed) based on transitivity.
- A graphical illustration (from the referenced article) shows necessary (solid lines) vs. pruned (dotted lines) comparisons.
- This supports the half-sorting pairwise metric sorting architecture.

Conclusion

1. Each pair of siblings (m_{2l}, m_{2l+1}) is already ordered for $l = 0, 1, .., L-1$.
2. The metrics indexed even in $\mathbf{m}$ are globally ordered.
3. Total number of comparisons is reduced.

8 Simplified Decoding of Polar Codes by Identifying Reed-Muller Constituent Codes

Main contributions of the paper:

- Identifying first-order Reed-Muller nodes, along with their sub-codes, as special nodes in the decoding tree of polar codes.
- Incorporating the Hadamard decoder into SCL decoding, and generating a list of candidates at the identified Reed-Muller nodes.
- Reducing the decoding latency of polar codes, while improving the error-correction performance.

Performance Improvements:

- **Block Error Improvement:** The simulations clearly conclude the improvements by considering RM nodes.
- **Latency Improvements:** The plot shows total decode cycles for the $N = 512$, 432-bit output code as a function of k. On average, RM-SSCL cuts latency by about 10.7% when $P = 16$ and by 20.5% when $P = 64$, thanks to frequent small RM nodes.

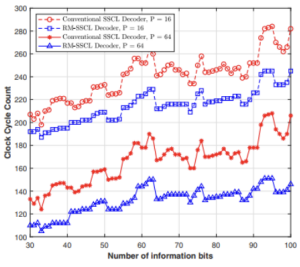


Fig. 7: Clock cycle count of the two decoders for a polar code of mother code length $N = 512$ with rate-matched output of 432 bits, assuming $P = 16$ and $P = 64$ processing elements.

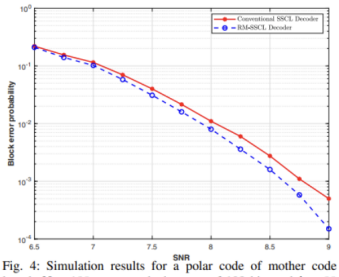


Fig. 4: Simulation results for a polar code of mother code length $N = 128$, rate-matched output of 108 bits and $k = 80$ information bits.

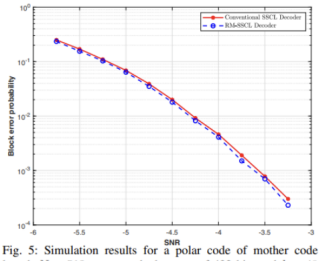


Fig. 5: Simulation results for a polar code of mother code length $N = 512$, rate-matched output of 432 bits and $k = 40$ information bits.

Figure 21: *

Fig. 7: Clock cycle count of the two decoders for a polar code of mother code length $N = 512$ with rate-matched output of 432 bits, assuming $P = 16$ and $P = 64$ processing elements.
Fig. 4: Simulation results for a polar code of mother code length $N = 128$, rate-matched output of 108 bits and $k = 80$ information bits.
Fig. 5: Simulation results for a polar code of mother code length $N = 512$, rate-matched output of 432 bits and $k = 40$ information bits.

References

[1] E. Arikan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Trans. Inf. Theory*, vol. 55, no. 7, pp. 3051–3073, July 2009.

[2] EventHelix, "Polar codes - 5G NR - Medium," *Medium*, Apr. 2019. [Online]. Available: https://medium.com/5g-nr/polar-codes-703336e9f26b

[3] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. Baltimore, MD: Johns Hopkins University Press, 2012.

[4] E. Arıkan and E. Telatar, "On the Rate of Channel Polarization," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Seoul, South Korea, June 2009.

[5] E. Şaşoğlu, *Polarization and Polar Codes*, Now Publishers Inc, 2012.

[6] A. Balatsoukas-Stimming, M. Bagheri, and A. Burg, "LLR-based successive cancellation list decoding of polar codes," in *Proc. Asilomar Conf. Signals, Syst., Comput.*, 2015, pp. 435–439.

[7] Y.-P. Zhang, H.-Y. Li, and H.-P. Zhao, "An optimized encoding algorithm for systematic polar codes," *EURASIP J. Wireless Commun. Netw.*, vol. 2019, no. 1, pp. 1–11, July 2019.

[8] I. Tal and A. Vardy, "List decoding of polar codes," *IEEE Trans. Inf. Theory*, vol. 61, no. 5, pp. 2213–2226, May 2015. [Online]. Available: https://arxiv.org/abs/1206.0050

[9] W. Xu, B. Feng, H. Zhou, and A. Vardy, "Demystifying polar codes for 5G NR: Practical designs and implementation insights," *arXiv preprint arXiv:2005.00555*, 2020.

[10] R. S. Prayogo, T. Adiono, and S. Fuada, "Performance evaluation of polar codes for 5G mMTC applications," in *Proc. Int. Symp. Electron. Smart Devices (ISESD)*, Oct. 2018, pp. 1–6.

[11] D. Tse, I. Du, and Y. Li, "Adaptive successive cancellation list decoding of polar codes," Stanford University Technical Report, 2019. [Online]. Available: https://web.stanford.edu/dntse/papers/lst.pdf

[12] C. Condo, S. A. Hashemi, and W. J. Gross, "Efficient LLR-based successive-cancellation list decoding of polar codes," *arXiv preprint arXiv:1411.7282*, 2014.

[13] Y. Zhang, Q. Zhang, X. You, and C. Zhang, "A serial successive-cancellation list decoder for polar codes," *Microelectronics & Computer*, vol. 36, no. 8, pp. 1–5, 2019. [Online]. Available: http://www.journalmc.com/en/article/id/56c5cfad-6ec3-4616-9e9f-cf0bd12ca75c

[14] C. Chen, B. Bai, X. Ma, and X. Wang, "Improved successive cancellation decoding of polar codes," *Electron. Lett.*, vol. 48, no. 9, pp. 500–502, Apr. 2012.

[15] Q. Zhang, A. Liu, X. Pan, and Y. Zhang, "Fast successive-cancellation list decoder for polar codes with early stopping criterion," in *Proc. IEEE Int. Conf. Integr. Circuits, Technol. Appl. (ICTA)*, Nov. 2018, pp. 1–2.

[16] C. E. Shannon, "A Mathematical Theory of Communication," *Bell Syst. Tech. J.*, vol. 27, pp. 379–423, 623–656, 1948.

[17] J. G. Proakis and M. Salehi, *Digital Communications*, 5th ed., McGraw-Hill, 2008.

[18] K. Lee and I.-C. Park, "Large-small Sorting for Successive Cancellation List Decoding of Polar Codes," *IEEE*, [Online]. Available: <insertURLifavailable>

[19] Wikipedia, "Normal distribution," [Online]. Available: https://en.wikipedia.org/wiki/Normal_distribution

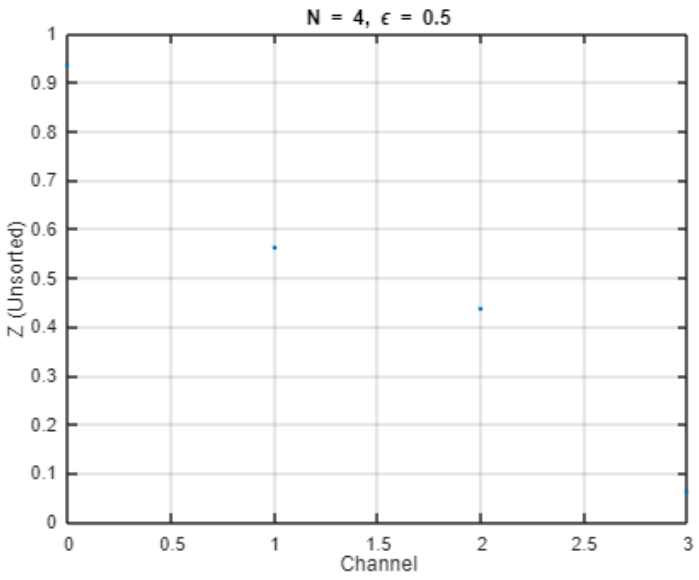
[20] "Additive White Gaussian Noise Channels," [Online]. Available: http://www.wirelesscommunication.nl/reference/chaptr05/digimod/awgn.htm

[21] Wikipedia, "Additive white Gaussian noise," [Online]. Available: https://en.wikipedia.org/wiki/Additive_white_Gaussian_noise

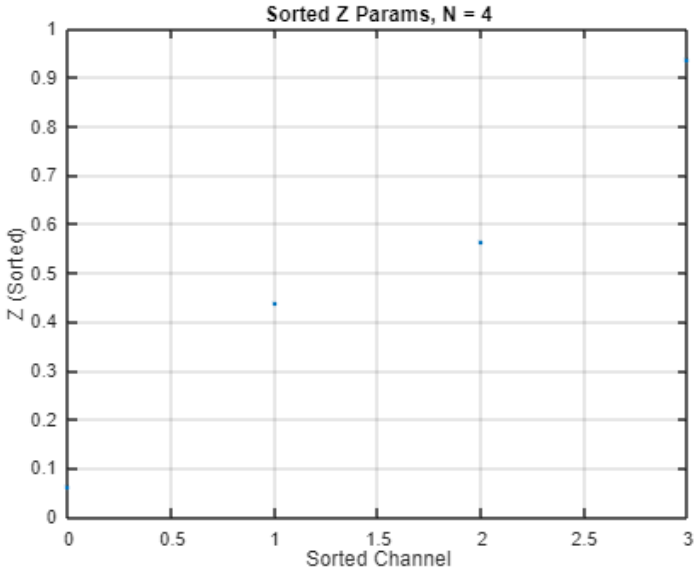
9 Codes

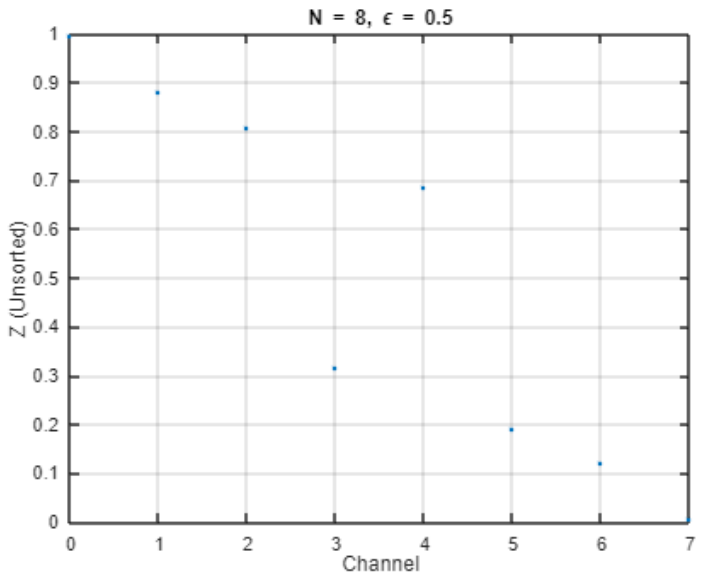
9.1 Channel Polarization

```
function [Z_out] = bhattacharyya_rec(N_dim, eps_val)
    Z_out = eps_val;
    levels = log2(N_dim);
    for depth = 1:levels
        updated = zeros(1, 2^depth);
        idx = 1;
        for z = Z_out
            updated(idx) = 2*z - z^2;
            updated(idx+1) = z^2;
            idx = idx + 2;
        end
        Z_out = updated;
    end
end

base_p = 0.5;
block_exponents = 2:10;

for n_exp = block_exponents
    blklen = 2^n_exp;
    Z_vals = bhattacharyya_rec(blklen, base_p);

    figure;
    plot(0:blklen-1, Z_vals, '.', 'LineWidth', 1.25);
    xlim([0 blklen-1]);
    grid on;
    xlabel('Channel');
    ylabel('Z (Unsorted)');
    title(['N = ', num2str(blklen), ', \epsilon = ', num2str(base_p)]);

    [~, rel_idx] = sort(Z_vals, 'descend');
    rel_idx = rel_idx - 1;
    disp(['Rel. Seq. (N=', num2str(blklen), '):']);
    disp(rel_idx);

    Z_sorted = sort(Z_vals);
    figure;
    plot(0:blklen-1, Z_sorted, '.', 'LineWidth', 1.25);
    xlim([0 blklen-1]);
    grid on;
    xlabel('Sorted Channel');
    ylabel('Z (Sorted)');
    title(['Sorted Z Params, N = ', num2str(blklen)]);

    drawnow;
    pause(0.1);
end
```

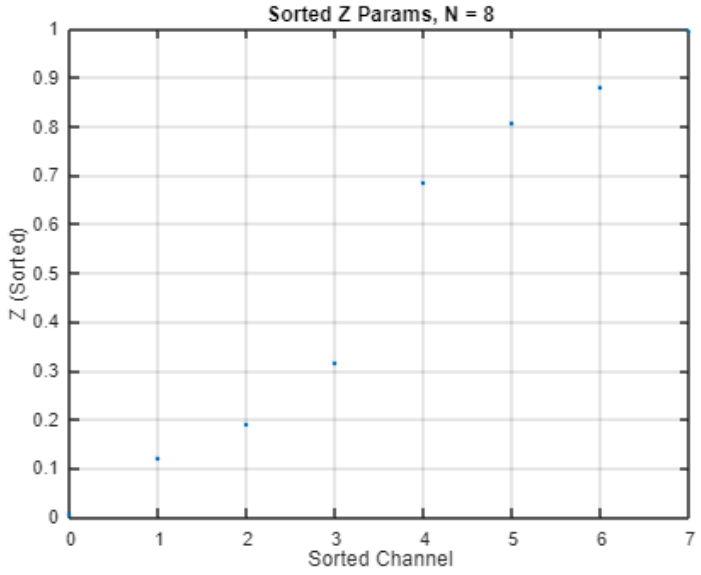


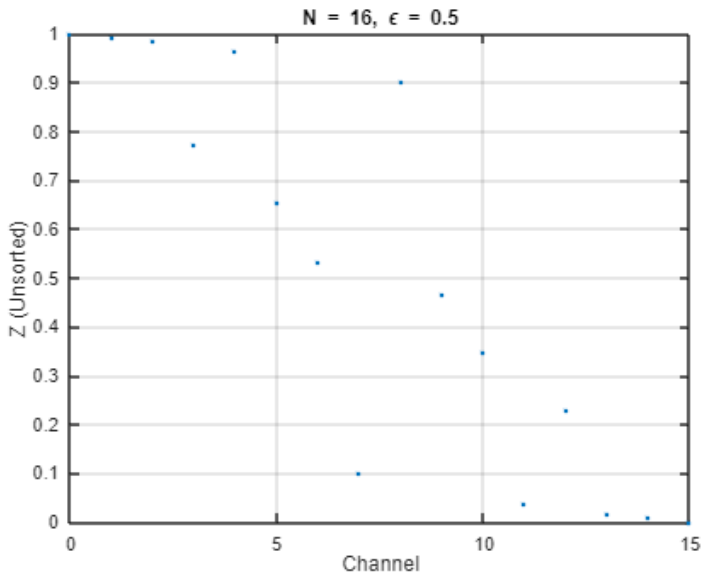

```
Rel. Seq. (N=4):
     0     1     2     3
```



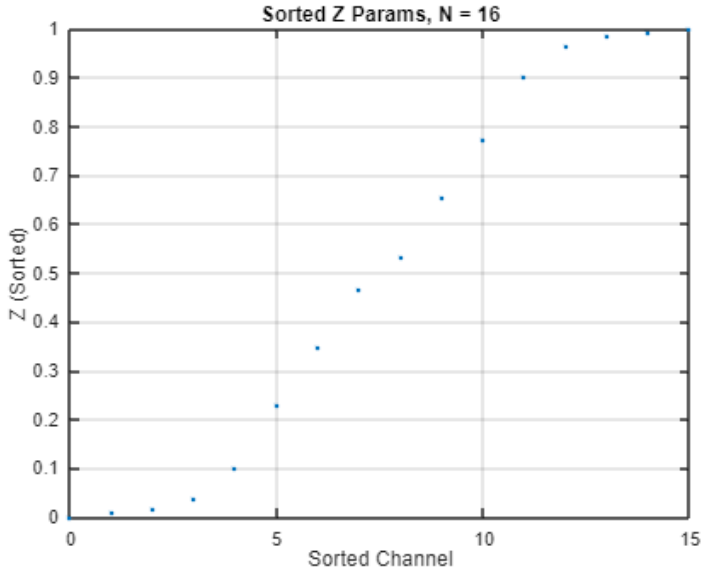


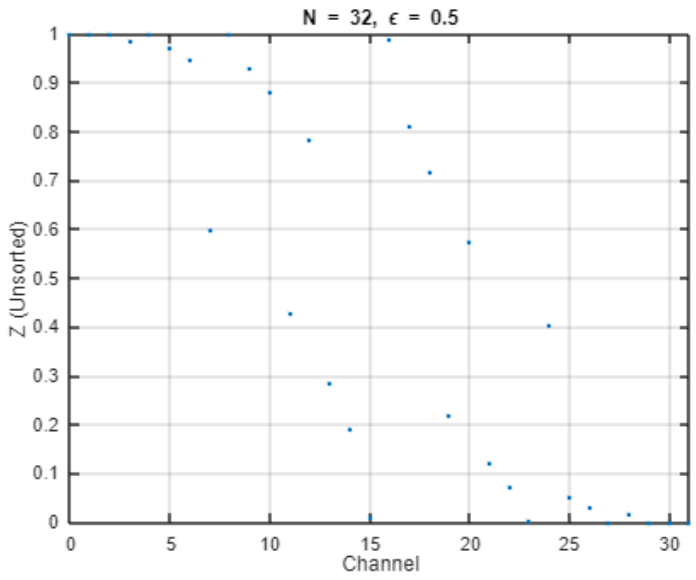

```
Rel. Seq. (N=8):
     0     1     2     4     3     5     6     7
```



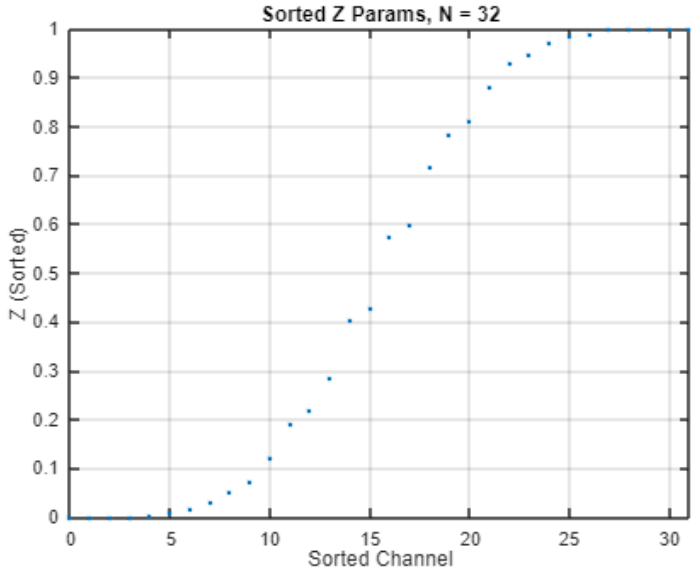


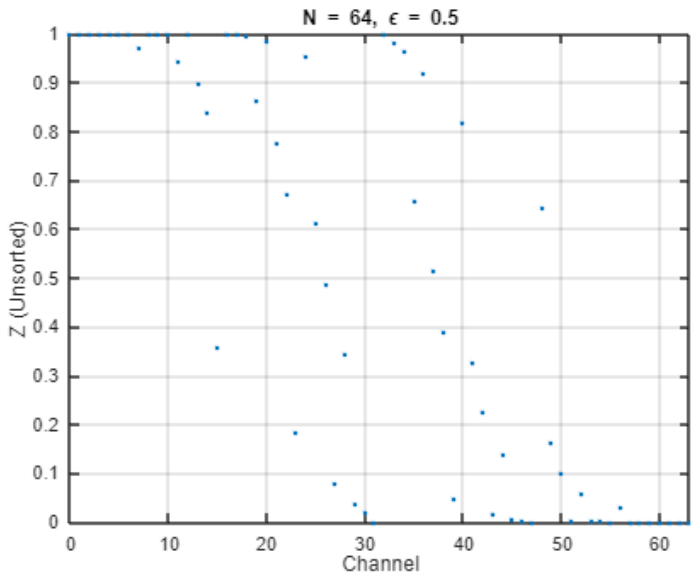

```
Rel. Seq. (N=16):
     0     1     2     4     8     3     5     6     9    10    12     7    11    13    14    15
```



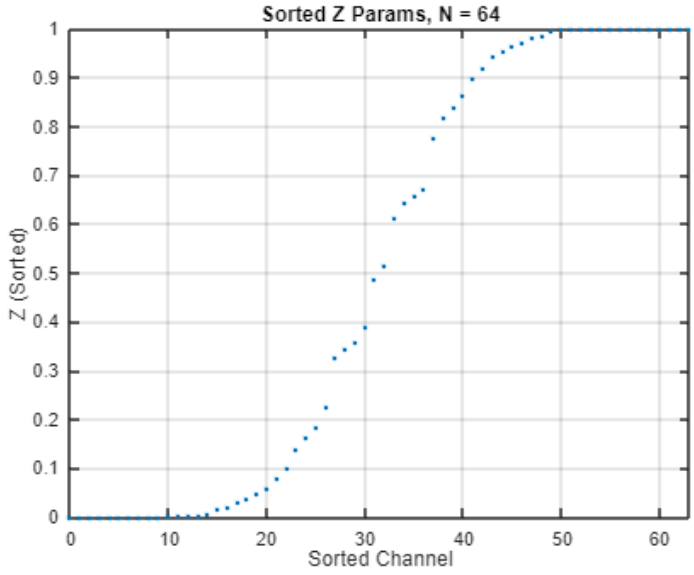


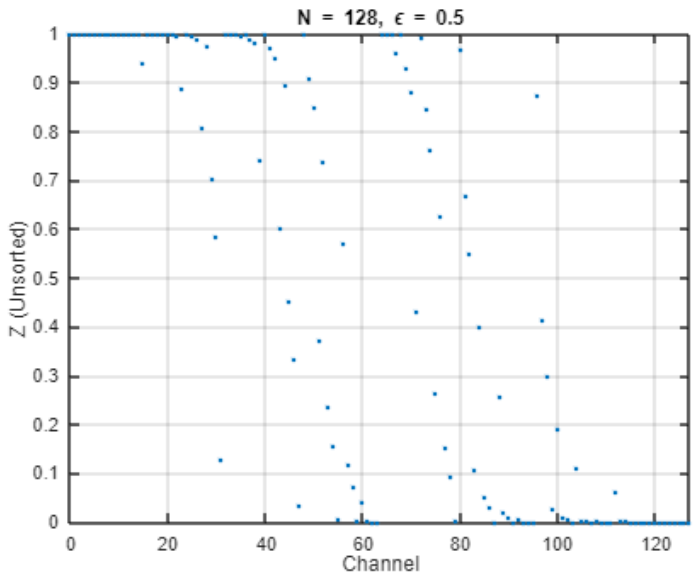

```
Rel. Seq. (N=32):
     0     1     2     4     8    16     3     5     6     9    10    17    12    18     7    20     1
```



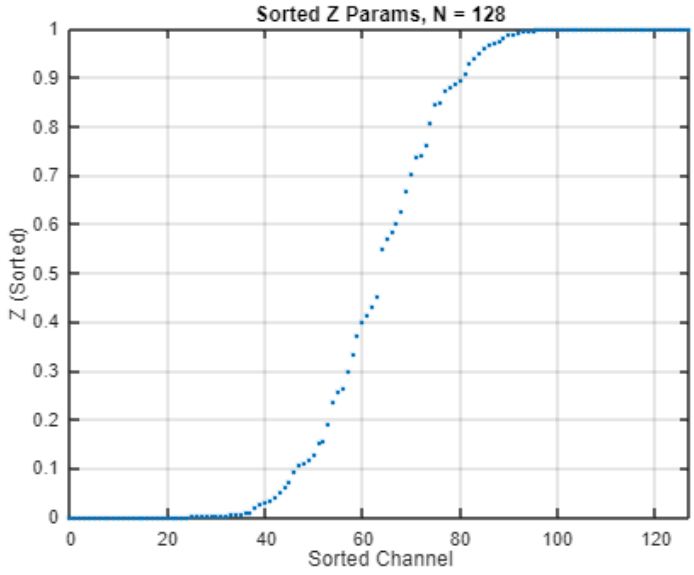


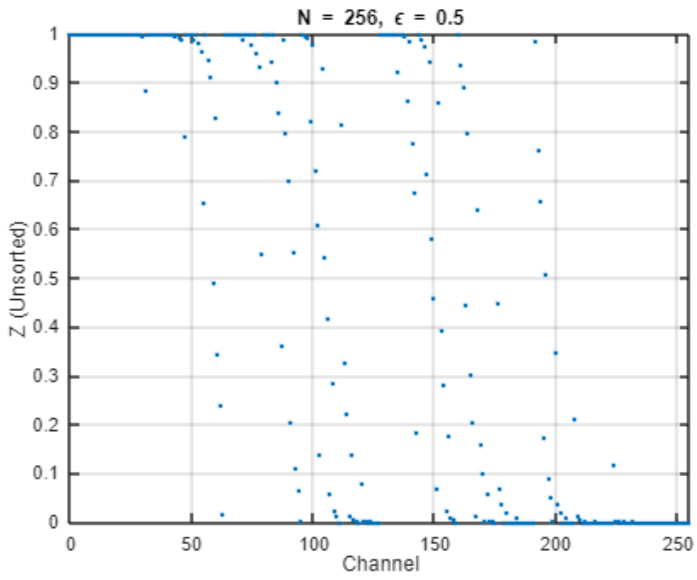

```
Rel. Seq. (N=64):
     0     1     2     4     8    16     3    32     5     6     9    10    12    17    18    20     3
```



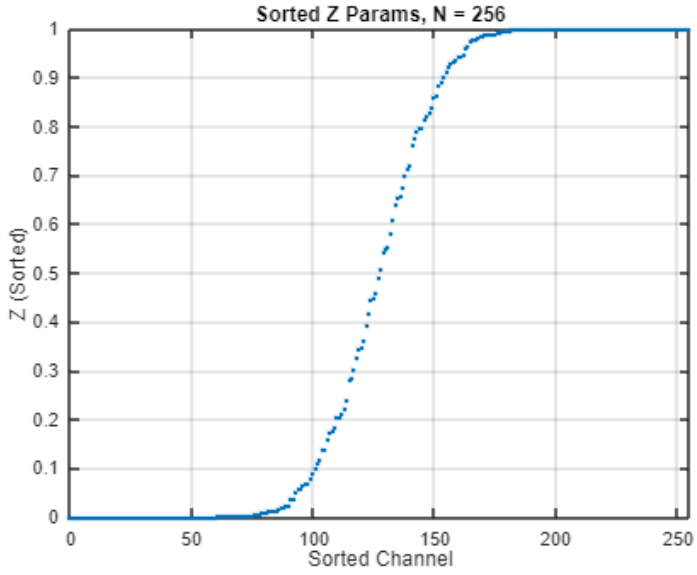


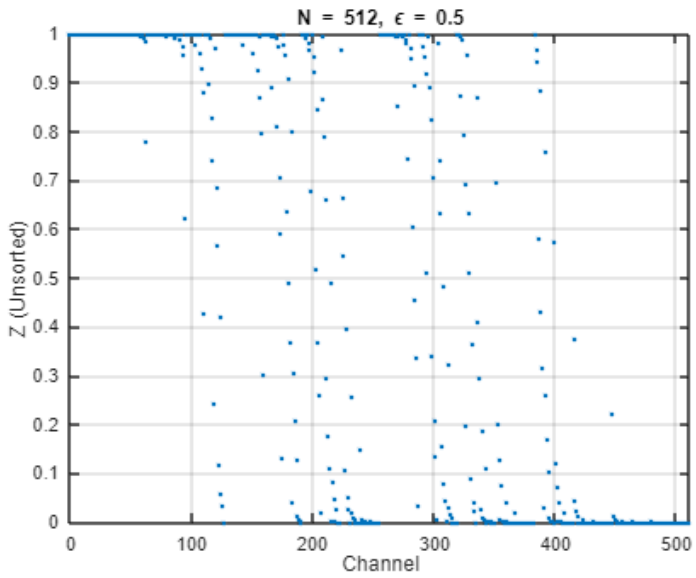

```
Rel. Seq. (N=128):
     0     1     2     4     8    16    32     3     5     6     9    64    10    12    17    18     2
```



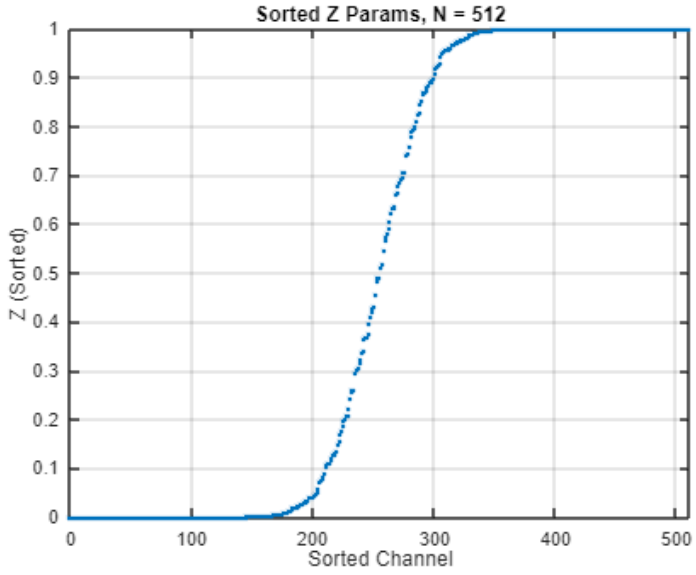


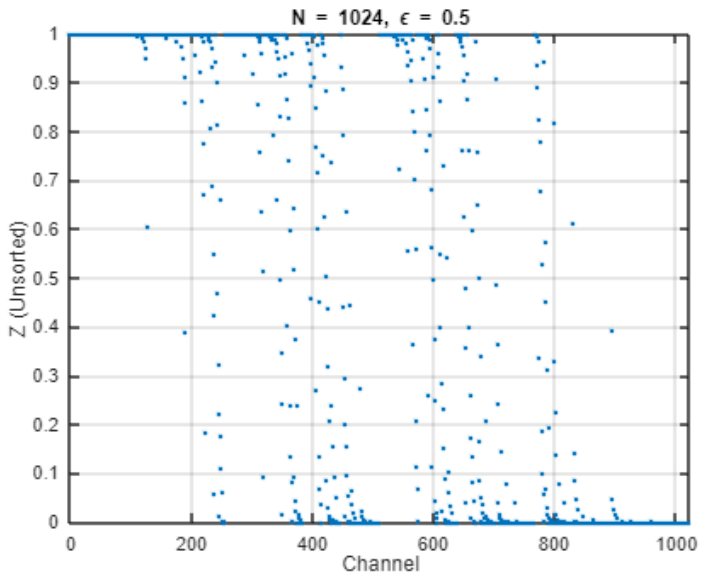

```
Rel. Seq. (N=256):
     0     1     2     3     4     5     6     8     9    10    12    16    17    18    32    64    12
```



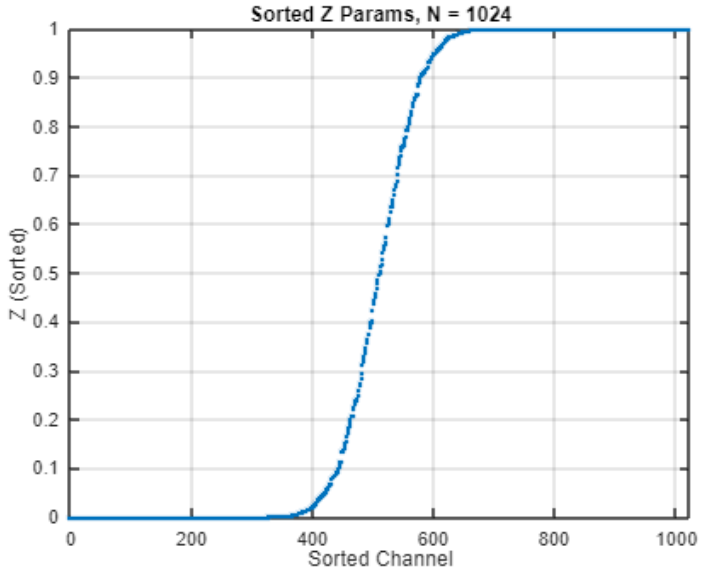



```
Rel. Seq. (N=512):
     0     1     2     3     4     5     6     7     8     9    10    11    12    13    14    16     1
```






```
Rel. Seq. (N=1024):
           0           1           2           3           4           5           6           7
```



9.2 Encoding

```
% --------------------------
% Parameters
% --------------------------
N = 32;             % Code length (must be power of 2)
K = 16;              % Number of message bits
u_msg = randi([0,1],1,K);  % Message bits extended to K

% --------------------------
% Step 1: Define frozen and info indices
% ----------------------
% ----
% Generate reliability sequence (example: based on a simple pattern)
reliability = [0 1 2 4 8 16 32 3 5 64 9 6 17 10 18 128 12 33 65 20 256 34 24 36 7 129
66 512 11 40 68 130 19 13 48 14 72 257 21 132 35 258 26 513 80 37 25 22 136 260 264
38 514 96 67 41 144 28 69 42 516 49 74 272 160 520 288 528 192 544 70 44 131 81 50 73
15 320 133 52 23 134 384 76 137 82 56 27 97 39 259 84 138 145 261 29 43 98 515 88 140
30 146 71 262 265 161 576 45 100 640 51 148 46 75 266 273 517 104 162 53 193 152 77 164
768 268 274 518 54 83 57 521 112 135 78 289 194 85 276 522 58 168 139 99 86 60 280 89
290 529 524 196 141 101 147 176 142 530 321 31 200 90 545 292 322 532 263 149 102 105
304 296 163 92 47 267 385 546 324 208 386 150 153 165 106 55 328 536 577 548 113 154 79
269 108 578 224 166 519 552 195 270 641 523 275 580 291 59 169 560 114 277 156 87 197
116 170 61 531 525 642 281 278 526 177 293 388 91 584 769 198 172 120 201 336 62 282
143 103 178 294 93 644 202 592 323 392 297 770 107 180 151 209 284 648 94 204 298 400
608 352 325 533 155 210 305 547 300 109 184 534 537 115 167 225 326 306 772 157 656 329
110 117 212 171 776 330 226 549 538 387 308 216 416 271 279 158 337 550 672 118 332 579
540 389 173 121 553 199 784 179 228 338 312 704 390 174 554 581 393 283 122 448 353 561
203 63 340 394 527 582 556 181 295 285 232 124 205 182 643 562 286 585 299 354 211 401
185 396 344 586 645 593 535 240 206 95 327 564 800 402 356 307 301 417 213 568 832 588
186 646 404 227 896 594 418 302 649 771 360 539 111]+1;
% Random permutation as a basic reliability measure

arr = zeros(1, N);
ind=1;
for i=1:1024
    if(reliability(i)<=N)
        arr(ind)=reliability(i);
        ind=ind+1;
    end
    if(ind==N)
        break;
    end
end

frozen_indices = arr(1:(N-K));
info_indices = setdiff(1:N, frozen_indices);

u = zeros(1, N);
u(info_indices(1:K)) = u_msg;
```

```
F = [1 0; 1 1];

% Recursive Kronecker product to get G_N
G = 1;  % Start with identity
n = log2(N);
for i = 1:n
    G = kron(G, F);
end


% Optional: reverse bit order in rows/columns (depends on convention)
% G = bitrevorder(G);

% --------------------------
% Step 4: Encode
% --------------------------
x = mod(u * G, 2);  % Polar encoding using matrix multiplication

% --------------------------
% Output
% --------------------------

disp(u_msg);
```

```
     1     1     1     1     1     0     1     0     1     0     0     1     1     0     0     0
```

```
disp('Input vector u:');
```

```
Input vector u:
```

```
disp(u);
```

```
     0     0     0     0     0     0     0     1     0     0     0     1     0     1     1     1
     0     0     0     0     0     1     0     1     0     0     0     1     1     0     0     0
```

```
disp('Encoded vector x:');
```

```
Encoded vector x:
```

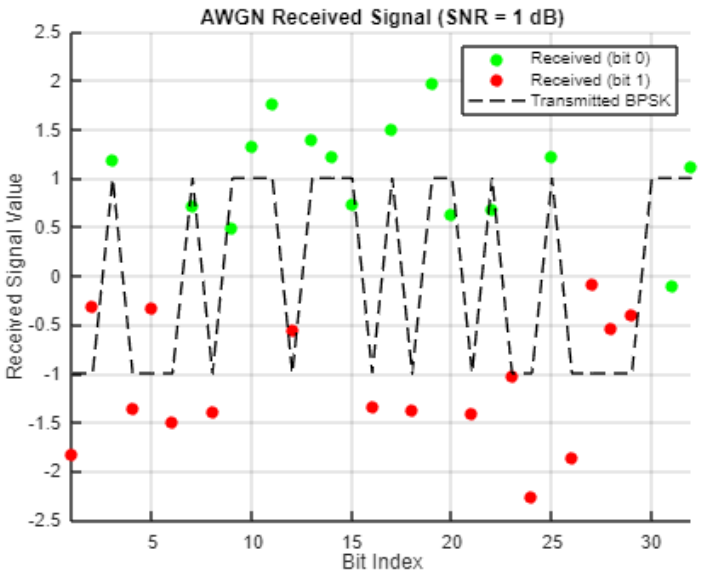
```
disp(x);
```

```
     1     1     0     1     1     1     0     1     0     0     0     1     0     0     0     1
     0     1     0     0     1     0     1     1     0     1     1     1     1     0     0     0
```

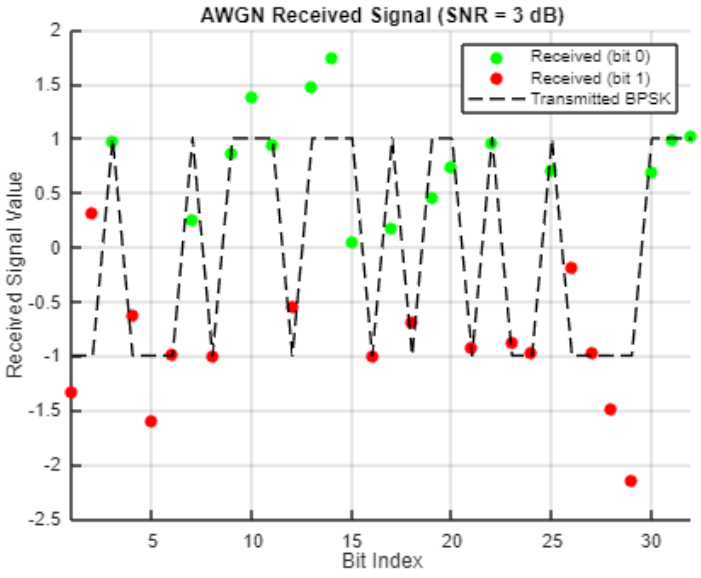
```
s = 1 - 2 * x;  % BPSK Mapping: 0 -> +1, 1 -> -1


SNRs = [1,3,5];
% SNR in dB (adjust as needed)
for i=1:3
```

```
    SNR_dB=SNRs(i);
SNR = 10^(SNR_dB / 10);      % Linear SNR
sigma = sqrt(1 / (2 * SNR)); % Noise standard deviation (unit energy symbols)

noise = sigma * randn(1, N); % AWGN noise vector
r = s + noise;               % Received noisy signal

disp(s);
disp(r);

figure;
hold on; grid on;
title(['AWGN Received Signal (SNR = ', num2str(SNR_dB), ' dB)']);
xlabel('Bit Index');
ylabel('Received Signal Value');
xlim([1 N]);

idx_0 = find(x == 0);
idx_1 = find(x == 1);

plot(idx_0, r(idx_0), 'go', 'MarkerFaceColor', 'g', 'DisplayName', 'Received (bit 0)');
plot(idx_1, r(idx_1), 'ro', 'MarkerFaceColor', 'r', 'DisplayName', 'Received (bit 1)');

plot(1:N, s, 'k--', 'LineWidth', 1.2, 'DisplayName', 'Transmitted BPSK');
legend('Location', 'northeast');
hold off;
end
```

```
    -1    -1     1    -1    -1    -1     1    -1     1     1     1    -1     1     1     1    -1
    1    -1     1     1    -1     1    -1    -1     1    -1    -1    -1    -1     1     1     1
      -1.8208     -0.31251       1.1792      -1.3486     -0.33237      -1.5016      0.71615
      -1.4003      0.48146       1.3178       1.7631     -0.55853       1.3992       1.2298
      0.73568      -1.3387       1.5038      -1.3794       1.9797       0.6373      -1.4172
      0.67764      -1.0278      -2.2663       1.2305      -1.8602      -0.0888     -0.54263
      -0.40646       2.0243      -0.1014       1.1217
```

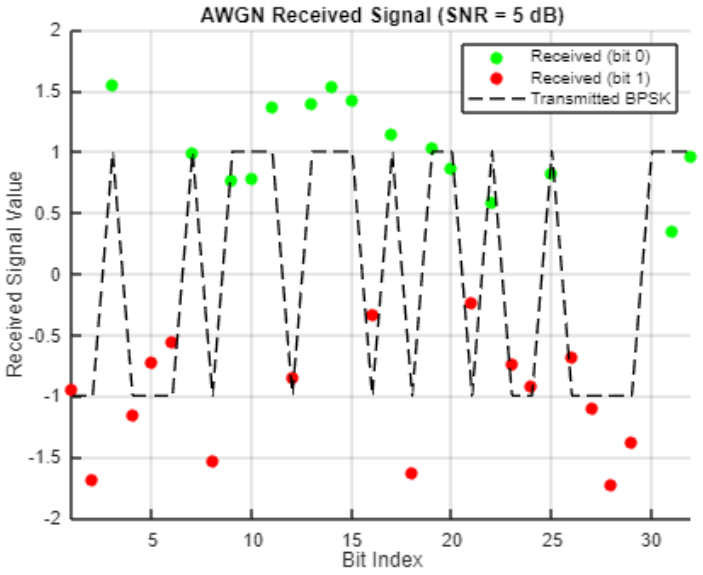



```
-1    -1     1    -1    -1    -1     1    -1     1     1     1    -1     1     1     1    -1
 1    -1     1     1    -1     1    -1    -1     1    -1    -1    -1    -1     1     1     1
  -1.3235      0.31494      0.97749     -0.62435      -1.597     -0.97853      0.25505
  -1.0047      0.86376       1.3849      0.95051     -0.54267       1.4713       1.7392
  0.053031       -1.006       0.1781     -0.68496      0.45944      0.73659     -0.92136
  0.95783     -0.87421     -0.97322      0.70204     -0.19092     -0.96404      -1.4866
  -2.1524      0.69688       0.9933       1.0226
```




```
-1    -1     1    -1    -1    -1     1    -1     1     1     1    -1     1     1     1    -1
 1    -1     1     1    -1     1    -1    -1     1    -1    -1    -1    -1     1     1     1
 -0.94446      -1.6853       1.5444      -1.1484     -0.72969     -0.55464      0.99362
 -1.535      0.77056      0.77998       1.3645     -0.84642       1.3944       1.5319
  1.4169     -0.33059       1.1402      -1.6248       1.0336      0.86063     -0.23771
 0.59322     -0.73414     -0.92289      0.82775     -0.68009      -1.1061      -1.7315
```

```
-1.3825        1.7207        0.35041        0.96559
```



9.3 Decoding

```
% Rel_Seq is the Bit-Channel Reliability Sequence
Rel_Seq =[0 1 2 4 8 16 32 3 5 64 9 6 17 10 18 128 12 33 65 20 256 34 24 36 7 129
66 512 11 40 68 130 19 13 48 14 72 257 21 132 35 258 26 513 80 37 25 22 136 260 264
38 514 96 67 41 144 28 69 42 516 49 74 272 160 520 288 528 192 544 70 44 131 81 50 73
15 320 133 52 23 134 384 76 137 82 56 27 97 39 259 84 138 145 261 29 43 98 515 88 140
30 146 71 262 265 161 576 45 100 640 51 148 46 75 266 273 517 104 162 53 193 152 77 164
768 268 274 518 54 83 57 521 112 135 78 289 194 85 276 522 58 168 139 99 86 60 280 89
290 529 524 196 141 101 147 176 142 530 321 31 200 90 545 292 322 532 263 149 102 105
304 296 163 92 47 267 385 546 324 208 386 150 153 165 106 55 328 536 577 548 113 154 79
269 108 578 224 166 519 552 195 270 641 523 275 580 291 59 169 560 114 277 156 87 197
116 170 61 531 525 642 281 278 526 177 293 388 91 584 769 198 172 120 201 336 62 282
143 103 178 294 93 644 202 592 323 392 297 770 107 180 151 209 284 648 94 204 298 400
608 352 325 533 155 210 305 547 300 109 184 534 537 115 167 225 326 306 772 157 656 329
110 117 212 171 776 330 226 549 538 387 308 216 416 271 279 158 337 550 672 118 332 579
540 389 173 121 553 199 784 179 228 338 312 704 390 174 554 581 393 283 122 448 353 561
203 63 340 394 527 582 556 181 295 285 232 124 205 182 643 562 286 585 299 354 211 401
185 396 344 586 645 593 535 240 206 95 327 564 800 402 356 307 301 417 213 568 832 588
186 646 404 227 896 594 418 302 649 771 360 539 111];

G = [1, 0; 1, 1]; % Polar Transforms
N = 1024; % Length of Code-word
A = 500; % Length of Message Bits
crcN = 11; % Length of CRC
K = A+crcN; % Total Length of Message Bits and CRC bits
crcP = fliplr([1 1 1 0 0 0 1 0 0 0 0 1]); % CRC for 11 bits
% crcP = fliplr([1 1 0 1]); % CRC for 3 bits
Rate = A/N; % Rate
n = log2(N);
Ls = 4; % Size of List
EbNodB = 0:0.5:10; % Value of Eb/No in dB
BER = zeros(1, size(EbNodB, 2)); % Storing BER values for plotting against Eb/No(dB)
Pc = zeros(1, size(EbNodB, 2)); % Storing Probability of Successful Decoding for
                                % plotting against Eb/No(dB)
ind=1;

for i = EbNodB
    EbNo = 10^(i/10); % Value of Eb/No
    sigma = 1/(sqrt(2*Rate*EbNo));

    % To find Polar Transform for N bits
    GN = Nbit_PolarTransform(G, n);

    Rel_SeqN = Rel_Seq(Rel_Seq<=N); % Reliability Sequence for N bits
    Frozen = Rel_SeqN(1:N-K); % Frozen Positions

    % Simulating the Code
    Nsim=100;
    BitErrors=0;
    BlockErrors=0;
```

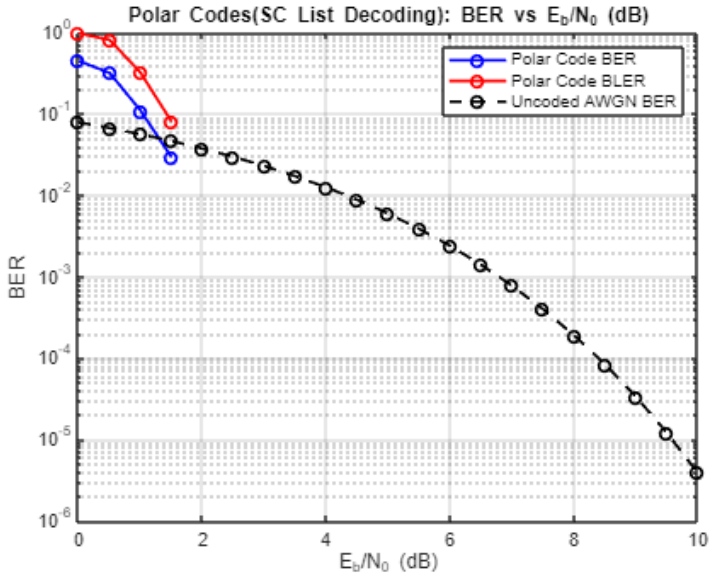
```
F_ni = 0;

for Block = 1:Nsim
    message = randi([0 1], 1, A); % K-bit message
    [quot,rem] = gfdeconv([zeros(1,crcN) fliplr(message)],crcP);
    % Finding remainder of message/CRC
    messagecrc = [message fliplr([rem zeros(1,crcN-length(rem))])];
    % Appending remainder to message

    u = zeros(1, N);

    u(Rel_SeqN(N-K+1:end)) = messagecrc; % assigning message bits

    code_word = mod((u*GN), 2); % generating code word

    s = BPSK_Modulation(code_word); % BPSK Modulation on the codeword
    r = AWGN_Channel(s, sigma, N); % Passing through AWGN Channel

    % Successive Cancelation List Decoder
    LLR = repmat(r, Ls, 1, 1); % Taking the r as beliefs values for our decoder

    PathMetrics = Inf*ones(Ls,1); % Path Metrics
    PathMetrics(1) = 0; % Activating First Path Metric

    node = 1:N;
    [u_capL, PathMetrics, x_cap, positions] = Polar_Decode(LLR, PathMetrics, Frozen,
    node, Ls);

    % Selecting message by Cyclic Redundancy Check (CRC)
    message_capL = u_capL(:,Rel_SeqN(N-K+1:end)); % To get all Ls messages
    ans_ind = 1; % The accepted index of message if none of them passes CRC
     crc_pass=0;
    for in = 1:Ls
        [qoutient, remainder] = gfdeconv(fliplr(message_capL(in, :)),crcP);
        if (remainder==0) % Checking if CRC Passes
            ans_ind = in; % Selecting the First message that passes CRC
            crc_pass = 1;
            break
        end
    end
    message_cap = message_capL(ans_ind,1:A); % Extracting the message from accepted
    sequence

    % Finding total Bits with Error
    errors = sum(message ~= message_cap);
    BitErrors = BitErrors + errors;
    if (errors==0)
        F_ni = F_ni + 1; % Adding 1 to F_ni if message is decoded successfully
    end
    if (crc_pass==0) % Checking whether the CRC has passed
        BlockErrors = BlockErrors + 1; % Counting total block errors
    end
```

```
    end

    BER_simulation = BitErrors/(A*Nsim);
    BER(ind) = BER_simulation; % Storing the values of BER_simulation for a given
    Eb/No(dB)
    BLER_simulation = BlockErrors/Nsim; % Calculating BLER_simulation for a given
    Eb/No(dB)
    BLER(ind) = BLER_simulation; % Storing the values of BLER_simulation for a given
    Eb/No(dB)
    BER_AWGN(ind) = qfunc(sqrt(2*EbNo));
    Pc(ind) = F_ni/Nsim; % Storing the value of Probability of Successful decoding for
    a given Eb/No(dB)
    ind = ind+1;
    format short g;
    disp([i BER_simulation BitErrors (A*Nsim)]);
end
```

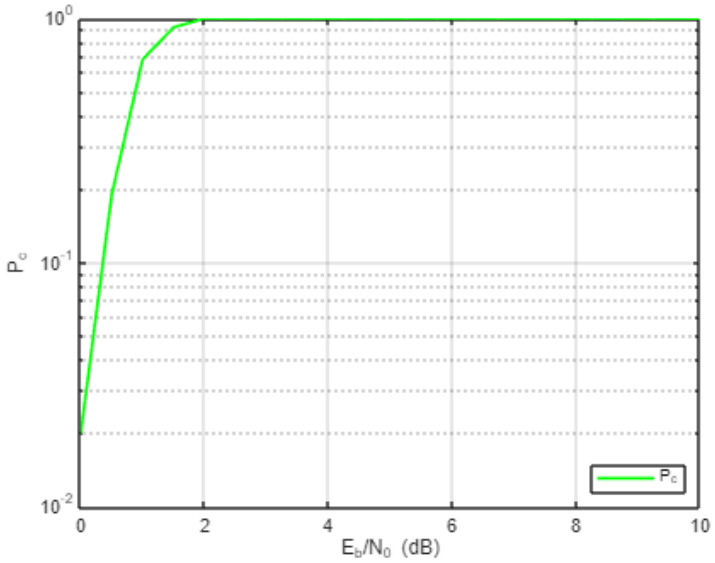
```
            0      0.44724        22362        50000

          0.5      0.31832        15916        50000

            1      0.10962         5481        50000

          1.5      0.02922         1461        50000

            2            0            0        50000

          2.5            0            0        50000

            3            0            0        50000

          3.5            0            0        50000

            4            0            0        50000

          4.5            0            0        50000

            5            0            0        50000

          5.5            0            0        50000

            6            0            0        50000

          6.5            0            0        50000

            7            0            0        50000

          7.5            0            0        50000

            8            0            0        50000

          8.5            0            0        50000

            9            0            0        50000
```

```
    9.5            0            0        50000

    10             0            0        50000
```

```
semilogy(EbNodB, BER, 'DisplayName', 'Polar Code BER', 'LineWidth', 1.5, 'Color', 'b',
'Marker', 'o');
hold on;
semilogy(EbNodB, BLER, 'DisplayName', 'Polar Code BLER', 'LineWidth', 1.5, 'Color', 'r',
'Marker', 'o');
semilogy(EbNodB, BER_AWGN, 'DisplayName', 'Uncoded AWGN BER', 'LineWidth', 1.5, 'Color',
'k', 'LineStyle', '--', 'Marker', 'o');
grid on;
hold off;
xlabel('E_b/N_0 (dB)');
ylabel('BER');
title('Polar Codes(SC List Decoding): BER vs E_b/N_0 (dB)');
legend('show','location','northeast');
```




```
semilogy(EbNodB, Pc, 'DisplayName', 'P_c', 'LineWidth', 1.5, 'Color', 'g');
grid on;
xlabel('E_b/N_0 (dB)');
ylabel('P_c');
legend('show','location','southeast');
```




```
function [u_cap, PathMetrics, x_cap, positions] = Polar_Decode(LLR, PathMetrics, Frozen,
node, Ls)
    N = size(LLR, 2);
    if (N==1)
        if any(Frozen==node) % To check if node is frozen
                x_cap = zeros(Ls, 1); % Assigning 0 if node is frozen
                u_cap = zeros(Ls, 1); % Assigning 0 if node is frozen
                PathMetrics = PathMetrics + abs(LLR).*(LLR < 0); % Adding |LLR| value
                if it is negative
                positions = 1:Ls; % No extra Path Metrics Added, So Positions
                will remain Unchanged
        else
            decision = LLR < 0; % Decisions as per LLR values

            % Creating All Path Metrics such that, first Ls are: PathMetrics
            % Next Ls are (opposite to Decision Metrics): Pathmetrics + |LLR|
            AllPathMetrics = [PathMetrics; PathMetrics+abs(LLR)];
            [PathMetrics, positions] = mink(AllPathMetrics, Ls);

            greater_pos = positions > Ls; % Positions which are opposite of Decision
            Metrics
            positions(greater_pos) = positions(greater_pos) - Ls;
            % Adjusting to their
            actual Index in List

            decision = decision(positions); % Decisions of Selected Ls Decoders
            decision(greater_pos) = 1 - decision(greater_pos);
            % Flipping the Decisions opposite to Decision Metrics

            x_cap = decision; % Assigning Decisions
            u_cap = decision; % Assigning Decisions
```

```
        end
    else
        % Partitioning Beliefs into two parts
        L1 = LLR(:, 1:N/2);
        L2 = LLR(:, N/2+1:N);

        % Finding the Beliefs for Left Part by SPC Decoding
        L_Left = SPC_Decode(L1, L2);

        % Recursive function to get u1cap x1cap, PathMetrics and positions
        [u1_cap, PathMetrics, x1_cap, positions1] = Polar_Decode(L_Left, PathMetrics,
        Frozen, node(1:N/2), Ls);

        % Adjusting L1 and L2 according to received positions of Decision
        L1 = L1(positions1, :);
        L2 = L2(positions1, :);

        % Finding the Beleifs for Right Part by Repetition Decoding
        L_Right = Rep_Decode(L1, L2, x1_cap);

        % Recursive function to get u1_cap x1_cap, PathMetrics and positions
        [u2_cap, PathMetrics, x2_cap, positions2] = Polar_Decode(L_Right, PathMetrics,
        Frozen, node(N/2+1:N), Ls);

        % Adjusting u1_cap and x1_cap according to received positions of Decision
        u1_cap = u1_cap(positions2, :);
        x1_cap = x1_cap(positions2, :);

        x_cap = [mod((x1_cap+x2_cap), 2) x2_cap];
        u_cap = [u1_cap u2_cap];
        positions = positions1(positions2); % Final Position according to the Decisions
    end
end

function C = KroneckerProduct(A, B)
    [m, n] = size(A);
    [p, q] = size(B);
    C = zeros(m*p, n*q);
    for i=1:m
        for j=1:n
            % rows from r1 to r2
            r1 = 1 + p*(i-1);
            r2 = p*i;

            % columns from c1 to c2
            c1 = 1 + q*(j-1);
            c2 = q*j;
            C(r1:r2, c1:c2) = A(i, j)*B;
        end
    end
end

function GN = Nbit_PolarTransform(G, n)
```

```
    GN = G;
    for i = 1:n-1
        GN = KroneckerProduct(G, GN);
    end
end

function L = SPC_Decode(a, b)
    % Calculating the sign of 'a' and 'b' respectively
    signA = (1-2*(a<0));
    signB = (1-2*(b<0));

    % Calculating the minimum of the absolute values of 'a' and 'b'
    minAbs = min(abs(a), abs(b));

    % Calculating the final result
    L = signA.*signB.*minAbs;
end

function L = Rep_Decode(a, b, c)
    % c=0 --> L = b+a
    % c=1 --> L = b-a
    L = b + (1-2*c).*a;
end

function s = BPSK_Modulation(code_word)
    % 0 --> 1       1 --> -1
    s = 1 - (2*code_word);
end

function r = AWGN_Channel(s, sigma, N)
    % Additive White Gaussian Distributed Noise added to symbols
    r = s + sigma*randn(1, N);
end
```