Dhirubhai Ambani University

Formally known as DA-IICT

Introduction to Communication Systems (CT216)

Polar Codes

Group - 30

Mentor: Dharmi Patel
Prof: Yash Vasavada

May 6, 2025

Honor Code

- We declare that the work presented is our own work.
- We have not copied the work of someone else.
- The concepts, understanding and insights we will describe are our own.
- Where we have relied on existing work that is not our own, we have provided a proper reference citation.
- We know that a violation of this solemn promise can have grave consequences.

Group Members

1. HIYA MODI (202301011)
2. PARSHV JOSHI (202301039)
3. ARHAAN SHAH (202301048)
4. DHRUV JIGNESHKUMAR PATEL (202301095)
5. DEVANSH SHAH (202301136)
6. OM PATEL (202301163)
7. SANAGADHIYA RADHIKA(202301184)
8. PATEL DIYA MUKESHKUMAR(202301216)
9. SHARNAM SHAH (202301247)
10. PRASHAM SHAH (202301252)

1. Introduction to Polar Codes

Polar Codes, proposed by Erdal Arikan in 2008, are a revolutionary class of error-correcting codes for efficient transmission of data with high reliability using noisy communication channels [3]. Being the initial codes that are shown to realize the channel capacity of symmetric binary-input discrete memoryless channels (B-DMCs) in polynomial complexity, they provide encoding and decoding complexities of $O(N \log N)$.

This solves a key problem in coding theory: reaching Shannon's channel capacity using implementable algorithms, as opposed to Low-Density Parity-Check (LDPC) or Turbo codes, which, being efficient, have no such theoretical assurances.

The motivation behind Polar Codes came as a result of having to span theoretical channel capacity with real-world coding schemes. Arikan spent two decades leading to channel polarization, a process that converts average channels into extremal channels—highly dependable or almost worthless—allowing for data communication on the most reliable channels. It does all this with low complexity, so Polar Codes become suitable for use in 5G New Radio (NR) control channels [4].

Naturally, Polar Codes recursively group and split channels to polarize their reliability. With increasing recursions, channels divide into channels with nearly perfect reliability (capacity close to 1) and channels with nearly zero reliability (capacity close to 0). Information bits are transmitted on reliable channels, frozen bits fill unreliable ones, so that communication remains robust. Such mathematical beauty and real-world performance have rendered Polar Codes a fundamental building block of contemporary communication systems, especially 5G.

2. Polar Transform

The Polar Code encoding depends on a generator matrix G_N, where the code length is $N = 2^n$. It is formed from the application of the **Kronecker product**, an operation that takes two matrices and creates a larger block matrix. For two matrices A $(m \times n)$ and B $(p \times q)$, the Kronecker product $A \otimes B$ is an $mp \times nq$ matrix, in which every element of A is multiplied by the whole matrix B. In particular, the element at location $(i, k), (j, l)$ in $A \otimes B$ is provided by $a_{ij} b_{kl}$.

The polar transform is built on the **base matrix - kernel**:

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

This 2×2 matrix is used as the kernel for building the generator matrix G_N of Polar Codes. For code length $N = 2^n$, the generator matrix is given as:

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

where $F^{\otimes n}$ represents the n-fold Kronecker product of F by itself. Such recursive construction forms the core of the polar transform and allows the efficient encoding of Polar Codes.

2.1 Generation of the Generator Matrix

The generator matrix G_N is constructed recursively by Kronecker product. The process begins with the base matrix F, and for each increase in n, the size of the matrix is doubled. Recursive generation can be described as:

$$G_N = F \otimes G_{N/2}, \quad \text{where} \quad G_2 = F$$

This can be generalized as $G_N = F^{\otimes n}$, that is, F is tensored with itself n times. To see this, let us consider the following examples:

- **For** $N = 2$ $(n = 1)$:

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

- **For** $N = 4$ $(n = 2)$:

$$\mathrm{G}_4 = F \otimes F = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} & 0 \cdot \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \\ 1 \cdot \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} & 1 \cdot \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

- **For** $N = 8$ $(n = 3)$:

$$G_8 = F \otimes G_4$$

This produces an 8×8 matrix, in which each element of F is multiplied with the whole 4×4 matrix G_4. The general construction is:

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

This can be thought of as a hierarchical process in which every stage of recursion doubles the size of the matrix, effectively grouping together pairs of bits in an orderly fashion.

2.2 Role of the Kronecker Product

The Kronecker product is central because it mathematically captures the process of combining within the recursive structure of channels specific to polar codes. In polar coding, the physical channel is transformed into N virtual channels through combining and splitting steps. The underlying base matrix F governs the fundamental combining operation.

→ For two input bits U_1 and U_2, the mapping $X = UG_2$ (where $G_2 = F$) yields:

$$X_1 = U_1, \quad X_2 = U_1 + U_2$$

(Here, + is addition modulo 2, which is the same as the XOR operation.)

This transformation is recursively applied to groups of bits, building up a hierarchy of virtual channels. The Kronecker product $F^{\otimes n}$ captures this recursive operation:

→ $F \otimes F$ applies the transformation to two groups of bits.

→ $F \otimes (F \otimes F)$ to four groups, and so on.

Therefore, $G_N = F^{\otimes n}$ is the overall transformation for N bits, consistent with the channel polarization process underlying Polar Codes [3].

2.3 Bit-Reversal Permutation

In many practical implementations, a **bit-reversal permutation** matrix B_N is incorporated, such that:

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

The bit-reversal permutation reorders the rows of $F^{\otimes n}$ to match the natural bit order. Without B_N, the output codeword $x = uG_N$ would have its bits in a bit-reversed order relative to the input vector u. The permutation matrix B_N is defined recursively as $B_N = R_N(I_2 \otimes B_{N/2})$, where R_N is a specific reordering matrix, and B_N is symmetric ($B_N^T = B_N$) [5]. For simplicity, and as adopted in this report, $G_N = F^{\otimes n}$ is used with ordered inputs, omitting the bit-reversal step [3].

2.4 Visualizing the Polar Transform

The polar transform given by the generator matrix $G_N = F^{\otimes n}$ where $F = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ and $N = 2^n$, can be better understood with diagrams. For a smaller codes, like $N = 2$, the transformation can be represented as a graph which illustrate the transition from input bits to output bits.

In case of $N = 2$, generator matrix is $G_2 = F$, and the encoding converts input vector $u = [u_0, u_1]$ into codeword $x = uG_2 = [u_0, u_0 + u_1]$. This can be represented as:

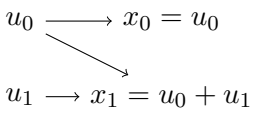

Figure 1: Polar transform for $N = 2$. The input bits u_0 and u_1 are combined to produce the codeword bits x_0 and x_1, with x_1 computed as the XOR of u_0 and u_1.

For larger N, the polar transform is constructed recursively via the Kronecker product $F^{\otimes n}$. This recursive process can be depicted in terms of a binary tree, where every

level is a step in the transformation. Take $N = 4$, where $G_4 = F \otimes F = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$, and the codeword is $x = uG_4 = [u_0, u_0 + u_1, u_0 + u_2, u_0 + u_1 + u_2 + u_3]$ for input $u = [u_0, u_1, u_2, u_3]$. The binary tree structure would be depicted below:

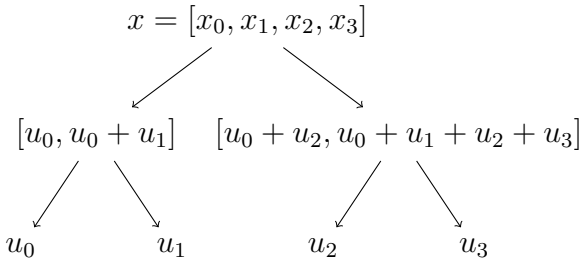

Figure 2: Binary tree representation of the polar transform for $N = 4$. The root represents the codeword x, intermediate nodes show the pairwise combinations(output of each channels after splitting), and leaves are the input bits u_0, u_1, u_2, u_3.

In this binary tree, the root symbolizes the end codeword, and every level describes how the input bits are recursively paired. At the initial level, the bits are paired together,and the transformation F is applied; the process is repeated, constructing the entire $G4$. This hierarchical organization not only explains the building of G_N but also supports the efficient encoding algorithm, which has a complexity of $O(N \log N)$, like the Fast Fourier Transform.

3. Polarization

Channel polarization is the key concept that makes Polar Codes reach channel capacity. It consists of converting N independent replicas of a target channel W into N generated bit channels $W_N^{(i)}$, $i = 1, \ldots, N$, each of which has a distinct level of reliability. As the block size N grows, these constructed channels polarize into two extremes: some become very reliable (with capacity close to 1), while others become very unreliable (with capacity close to 0). This polarization enables Polar Codes to assign information bits to the most reliable channels, essentially attaining the capacity of the original channel.

3.1 Bhattacharyya parameter

Each synthesized bit channel $W_N^{(i)}$ is defined as:

$$W_N^{(i)}(y_1^N, u_1^{i-1} | u_i)$$

where: y_1^N represents the sequence received from the N channel uses, u_1^{i-1} are the previously decoded bits (used in successive cancellation decoding), u_i is the bit transmitted through the i-th synthesized channel.

The reliability of a channel W can be graded using the Bhattacharyya parameter $Z(W)$, defined as:

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

The quantity $W(y|0)$ represents the probability that the channel outputs y when the input was 0, while $W(y|1)$ represents the same when the input was 1.[1]

This parameter can be thought as a measure of the *confusability* of the channel — a higher value indicates greater uncertainty in distinguishing the transmitted bit(whether it came from 0 or 1), and thus lower reliability. Conversely, a lower value indicates a more reliable channel.

Now, in terms of notation: although $W(y|x)$ is actually a conditional probability, we denote the channel overall — i.e., the stochastic function from inputs to outputs — using the notation W. This convention is used in order to explicitly distinguish between arbitrary probability distributions (which are often represented P) and actual channel models. In coding theory, particularly in the research of polar codes, we often work with several and changing instances of channels (e.g., generated channels $W_N^{(i)}$), and thus it is crucial to employ a special symbol such as W to denote the channel object unambiguously.

3.1.1 Reasons for the Square Root

The square root in the Bhattacharyya coefficient is not arbitrary—it serves several important purposes:

i. Normalization and Boundedness

The Bhattacharyya coefficient $Z(W)$ is designed to measure the similarity between two probability distributions, and hence it must be bounded between 0 and 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 no overlap (i.e., $W(y|0)W(y|1) = 0$ for all y).

The square root preserves this boundedness. Because $W(y|0)$ and $W(y|1)$ are probabilities (between 0 and 1), so their product $W(y|0)W(y|1)$ is also between 0 and 1, and taking the square root maintains the result within the same range. Summing over all y, and noting that $W(y|0)$ and $W(y|1)$ are normalized distributions, preserves $Z(W) \in [0, 1]$.

[1] Intuitively, both of these values indicate how probable the output y is given the two possible inputs. When both $W(y|0)$ and $W(y|1)$ are high and similar in value, the output y is very ambiguous — it is hard for a decoder to confidently decide whether 0 or 1 was sent.

ii. Comparison with Direct Product

If we applied the product directly, i.e., $\sum_y W(y|0)W(y|1)$, the value would remain between 0 and 1 (because it is the expectation of $W(y|1)$ under $W(y|0)$). But this value lacks the same geometric intuition or relation to error bounds as the Bhattacharyya coefficient. The square root "softens" the impact of small probabilities, so the measure is more sensitive to overlap between distributions.

iii. Geometric Interpretation: Connection to Hellinger Distance

The Bhattacharyya coefficient is also closely related to the Hellinger distance, a true metric between probability distributions. The Hellinger distance is defined as:

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

Or, the Hellinger distance can be written directly as:

$$H(W(y|0), W(y|1)) = \frac{1}{\sqrt{2}} \sqrt{\sum_y \left(\sqrt{W(y|0)} - \sqrt{W(y|1)} \right)^2}$$

Why the Square Root?

The expression $\left(\sqrt{W(y|0)} - \sqrt{W(y|1)} \right)^2$ in the Hellinger distance calculates the squared Euclidean distance between the square roots of the probabilities. The use of the square root makes this distance act like a proper metric (satisfying the triangle inequality) and being bounded between 0 and 1.

The Bhattacharyya coefficient $Z(W)$ is derived from this distance: Expanding the Hellinger distance formula, we get:

$$\sum_y \left(\sqrt{W(y|0)} - \sqrt{W(y|1)} \right)^2 = \sum_y W(y|0) + \sum_y W(y|1) - 2 \sum_y \sqrt{W(y|0)W(y|1)} = 2 - 2Z(W)$$

So, $H^2 = 1 - Z(W)$, and the square root in $Z(W)$ ensures that this relationship is maintained. Without the square root, the Hellinger distance would not have the same geometric properties, and the Bhattacharyya coefficient would not be as directly related to it.

iv. Connection to Error Bounds in Hypothesis Testing

In binary hypothesis testing (e.g., distinguishing between H_0 and H_1), the Bhattacharyya coefficient gives an upper bound on the Bayes error probability. For equal priors ($\pi_0 = \pi_1 = 0.5$), the Bayes error P_e is bounded as:

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

Why the Square Root?

The square root arises naturally in the derivation of this bound. The Bhattacharyya coefficient is a special case of the **Chernoff divergence** at $\alpha = 0.5$:

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

At $\alpha = 0.5$, this becomes:

$$D_{0.5} = -\ln\left(\sum_y W(y|0)^{0.5} W(y|1)^{0.5}\right) = -\ln(Z(W))$$

We use $\alpha = 0.5$ (resulting in the square root), as it regulates the contributions of $W(y|0)$ and $W(y|1)$. Since the priors are equal, it only makes sense to have both of their contributions equal.

v. Symmetry and Sensitivity

The square root ensures that the Bhattacharyya coefficient is symmetric:

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

This symmetry is important since the comparison of two distributions should not depend on their order.

The square root also makes the measure more sensitive to differences between the distributions: If we took the direct product $\sum_y W(y|0)W(y|1)$, the result would be overwhelmed by big probabilities and less sensitive to small changes in the tails of the distributions. The square root "flattens" the effect of small probabilities, so that the Bhattacharyya coefficient would be more balanced and useful for comparing distributions with different shapes.

vi. Connection to Polar Codes (Channel Reliability)

In the case of polar codes (above), the Bhattacharyya parameter $Z(W)$ is utilized to quantify the reliability of a channel. The use of the square root in the formula guarantees that $Z(W)$ has a recursive property consistent with the polarization process. For one step of polarization, the Bhattacharyya parameters of the "good" and "bad" channels are:

$$Z(W^-) = Z(W)^2$$

$$Z(W^+) \leq 2Z(W) - Z(W)^2$$

The squaring in $Z(W^-) = Z(W)^2$ directly results from the square root in the definition of $Z(W)$. In the absence of the square root, this recursive relation would fail, and the process of polarization would not be so easy to analyze. The square root ensures that $Z(W)$ acts correctly as an error probability surrogate, which is important in choosing good channels in polar code design.

vii. Mathematical Elegance: Relation to Inner Product

The Bhattacharyya coefficient can be interpreted as an inner product between the square roots of the probability distributions. Define vectors $\mathbf{p} = \left(\sqrt{W(y|0)}\right)_{y\in\mathcal{Y}}$ and $\mathbf{q} = \left(\sqrt{W(y|1)}\right)_{y\in\mathcal{Y}}$. Then:

$$Z(W) = \sum_y \sqrt{W(y|0)}\sqrt{W(y|1)} = \mathbf{p}\cdot\mathbf{q}$$

This is the dot product (or cosine similarity) between the vectors $\mathbf{p}$ and $\mathbf{q}$, which are normalized because $\sum_y W(y|0) = 1$ and $\sum_y W(y|1) = 1$. The square root ensures that $Z(W)$ behaves as a cosine similarity, ranging from 0 to 1, with 1 signifying a perfect alignment (completely similar) and 0 indicating orthogonality (no overlap). [1] [2]

3.2 Polarization Theorem

The polarization theorem, introduced by Arıkan [3], states that as $N \to \infty$, for any symmetric binary-input discrete memoryless channel W:

- The fraction of synthesized channels $W_N^{(i)}$ for which $Z(W_N^{(i)}) \to 0$ (highly reliable channels) approaches the capacity $I(W)$ of the original channel.
- The remaining channels have $Z(W_N^{(i)}) \to 1$ (completely unreliable channels).

This means that almost the entire capacity of the channel is bunched into a proportion $I(W)$ of synthesized channels, leaving the remaining to become useless for data transfer. This characteristic is imperative for Polar Codes to attain channel capacity by carrying information bits through the most stable synthesized channels.

3.3 Polarization Process

The polarization process is achieved through recursive channel combining and splitting. Consider a channel W with capacity $I(W)$. By combining N copies of W using a linear transformation, we create N synthesized channels $W_N^{(i)}$. These channels are derived recursively, with each step amplifying the differences in reliability.

To illustrate, consider a BEC with erasure probability $\epsilon = 0.5$ and block length $N = 2$:

- The first synthesized channel $W_2^{(1)}$ (for u_1) is less reliable:

$$Z(W_2^{(1)}) = 2\epsilon - \epsilon^2 = 2(0.5) - (0.5)^2 = 1 - 0.25 = 0.75$$

- The second synthesized channel $W_2^{(2)}$ (for u_2) is more reliable:

$$Z(W_2^{(2)}) = \epsilon^2 = (0.5)^2 = 0.25$$

This shows initial polarization: $W_2^{(1)}$ is less reliable than the original channel ($Z > \epsilon$), while $W_2^{(2)}$ is more reliable ($Z < \epsilon$).

For $N = 4$, the process is applied recursively. Let:

- $Z_1 = Z(W_2^{(1)}) = 2\epsilon - \epsilon^2 = 0.75$ (from the first pair of channels).
- $Z_2 = Z(W_2^{(2)}) = \epsilon^2 = 0.25$ (from the second pair of channels).

The synthesized channels for $N = 4$ are:

- $W_4^{(1)}$: Combines $W_2^{(1)}$ and $W_2^{(3)}$ (both less reliable):

$$Z(W_4^{(1)}) = 2Z_1 - Z_1^2 = 2(0.75) - (0.75)^2 = 1.5 - 0.5625 = 0.9375$$

- $W_4^{(2)}$: Combines $W_2^{(1)}$ and $W_2^{(3)}$ (more reliable combination):

$$Z(W_4^{(2)}) = Z_1^2 = (0.75)^2 = 0.5625$$

- $W_4^{(3)}$: Combines $W_2^{(2)}$ and $W_2^{(4)}$ (less reliable):

$$Z(W_4^{(3)}) = 2Z_2 - Z_2^2 = 2(0.25) - (0.25)^2 = 0.5 - 0.0625 = 0.4375$$

- $W_4^{(4)}$: Combines $W_2^{(2)}$ and $W_2^{(4)}$ (more reliable):

$$Z(W_4^{(4)}) = Z_2^2 = (0.25)^2 = 0.0625$$

For $N = 4$, the reliabilities are further polarized: $Z(W_4^{(1)}) = 0.9375$ (very unreliable), $Z(W_4^{(2)}) = 0.5625$, $Z(W_4^{(3)}) = 0.4375$, and $Z(W_4^{(4)}) = 0.0625$ (very reliable). As N increases, this polarization becomes more pronounced, with some channels approaching complete reliability ($Z \to 0$) and others approaching complete unreliability ($Z \to 1$).

For any two channels W_1 and W_2 with Bhattacharyya parameters $Z(W_1)$ and $Z(W_2)$, the synthesized channels W^- (more reliable) and W^+ (less reliable) have:

- $Z(W^+) <= Z(W_1) + Z(W_2) - Z(W_1)Z(W_2)$
- $Z(W^-) = Z(W_1)Z(W_2)$

This recursive relation drives the polarization process by ensuring that each step produces channels with more extreme reliabilities.

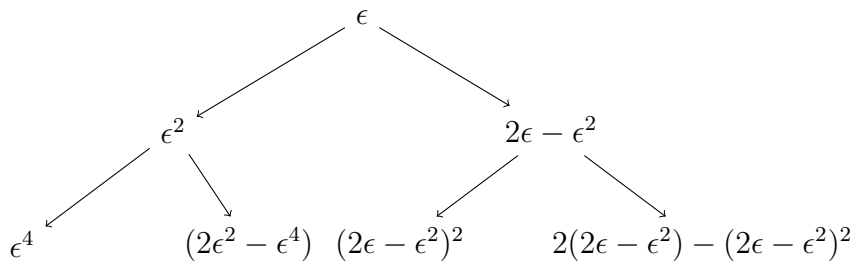

Figure 3: Binary tree representationof the polar transform for $N = 4$. The root represents the codeword x, intermediate nodes show the pairwise combinations, and leaves are the input bits u_0, u_1, u_2, u_3.

3.4 Why Polarization Occurs

Polarization arises due to the fact that the recursive process of channel splitting and combining increases the differences in reliability. When the two channels are combined together, the synthesized channels' capacities are more extreme than the original channels. Exactly:

- Total capacity is not lost:

$$I(W^+) + I(W^-) = 2I(W)$$

- But, $I(W^-) \leq I(W) \leq I(W^+)$, i.e., one channel is less trustworthy, and the other is more trustworthy.

As this round is iterated, the capacities of the produced channels tend to either 0 or 1. This is rigorously established with martingale theory [7], where the proportion of channels with capacity tending to 1 is shown to increase as $N \to \infty$. Polarization rate is quantified by an error probability that tends to diminish approximately like $2^{-\sqrt{N}}$ [6], with a relatively slow convergence in finite N.

3.5 Bit Channel Dependence

The synthesized bit channels are sequentially dependent, as would be expected in successive cancellation decoding structure:

- $W_N^{(1)}$: Is a function of only y_1^N.
- $W_N^{(2)}$: Varies with y_1^N and u_1.
- $W_N^{(3)}$: Depends on y_1^N, u_1, and u_2, etc.

This reliance guarantees that every bit u_i is recoverable from the data of decoded bits u_1^{i-1}, allowing effective decoding procedures.

Why Are Later Channels More Reliable Despite Erasures?

a. The Role of the Polar Transform

The polar transformation $\mathbf{G}_N$ is constructed to create synthetic channels such that the reliability is increased for some indices i, even though the original channel is erasure-prone:

→ The transform combines the bits u_1^N in a manner that certain synthetic channels in effect "observe" more data, and are thus more dependable.

→ For example, for the case $N = 2$:

- W^+ (for u_1) needs to decode u_1 without observing u_2, hence it is more difficult (less accurate).
- W^- (for u_2) is able to observe u_1, which gives it more information, so it is easier (more reliable).

For a BEC, the erasures in y_1 and y_2 are handled as follows:

→ In W^+, when y_1 or y_2 are erased, it becomes harder to compute u_1 as u_2 is not available. The erasure probability is increased to $2\epsilon - \epsilon^2$.

→ In W^-, since u_1 is known, u_2 can be recovered unless both y_1 and y_2 are erased. The effective erasure probability is ϵ^2, which is smaller.

This trend continues as N increases: the good channels take advantage of the structure of the transform, which compacts reliability into focused indices.

b. The Bhattacharyya Parameter Captures Ideal Reliability

The Bhattacharyya parameter $Z(W_N^{(i)})$ estimates the synthetic channel reliability under the perfect prior decisions assumption:

→ When we evaluate $Z(W_N^{(i)})$, we use the previous bits employed in SC decoding as precise.

→ For example:

- $Z(\ W_N^{(1)})$ is computed for $\ W_N^{(1)}(\ y_N^{(1)}|u_1)$, with no prior bits.
- $(\ Z(\ W_N^{(2)}))$ is calculated for $\ W_N^{(2)}(\ y_N^{(1)}, u_1|u_2)$, given the true u_1.

In practice, SC decoding may err in decoding u_1, and such errors are propagated to the decoding of u_2, u_3, etc. But:

→ The **design** of polar codes ensures that the synthetic channels $W_N^{(i)}$ for certain indices i (the "good" channels) have very small $Z\ W_N^{(i)}$.

→ This means that if the preceding bits were decoded accurately, then the error probability in u_i would be extremely low.

The reliability of the synthesized bit-channels in polar coding can be quantified using Bhattacharyya parameters. These parameters help determine which channels are suitable for carrying information and which should be frozen. Figure 4 shows the Bhattacharyya parameters for all synthesized channels, both in their original (unsorted) order and after sorting based on reliability.

c. SC Decoding Error Propagation

SC decoding suffers from **error propagation**:

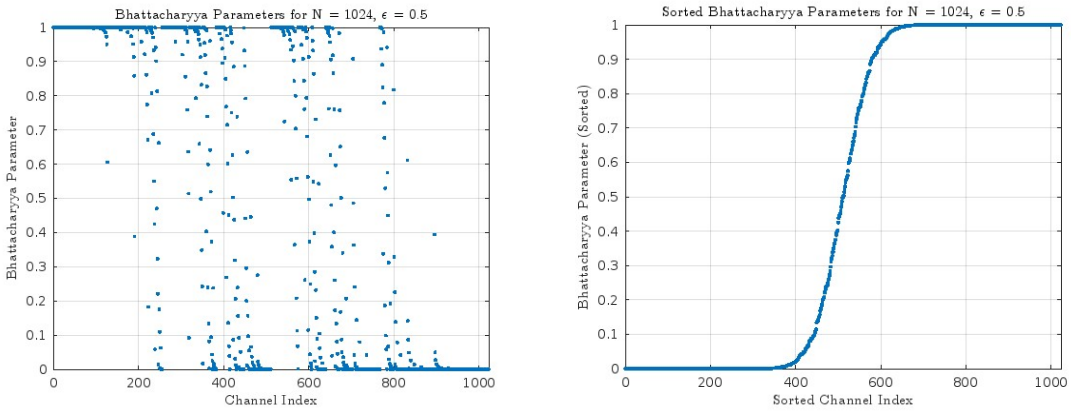

Figure 4: Bhattacharyya parameters: (left) unsorted, (right) sorted.

→ If u_1 is decoded in error ($\hat{u}_1 \neq u_1$), then $W_N 2$ will employ the erroneous $\hat{u}_1$, which can lead to a decoding error for u_2.

→ This mistake in u_2 will propagate to u_3, and so forth.

But polar codes try to minimize the impact of this:

→ **Polarization Places Most Good Channels at Later Indices**: In the recursive construction of polar codes, the good channels (small Z) are at higher indices i. For instance:

- For $N = 2$, $W_N^{(1)} = W^+$ (bad), $W_N^{(2)} = W^-$ (good).
- For $N = 4$, the channels

$$\begin{aligned} W_N^{(1)} &= (W^+)^+ \quad \text{(worst)}, \\ W_N^{(2)} &= (W^+)^-, \\ W_N^{(3)} &= (W^-)^+, \\ W_N^{(4)} &= (W^-)^- \quad \text{(best)}. \end{aligned}$$

The optimal channel is $W_N^{(4)}$, the last one.

→ **Good Channels Have Very Low Error Probability**: For good channels, $Z\ W_N^{(i)} \to 0$, i.e., the error probability is extremely low (even with erasures in $y_N^{(1)}$). If u_i is in a good channel, then it is highly probable to be decoded correctly by the decoder, assuming that previous bits are correct.

→ **Error Propagation Is Restricted**: If the majority of previous bits are in poor channels, we designate them as frozen bits (available to the decoder, say $u_i = 0$). The data bits are put in good channels, which are generally at higher indices. By the time we get to those good channels, many previous bits are frozen (and therefore correct), decreasing the likelihood of error propagation.

d. Asymptotic Behavior (Large N)

The asymptotic polarization behavior is responsible for the reliability of the later channels:

→ As $N \to \infty$, the quantity of channels such that $ZW_N^{(i)} < 2^{-N}$ (very reliable) converges towards $I(W)$.

→ For a BEC with $\epsilon = 0.4$, about 60% of channels become reliable. We use those channels for data bits.

→ SC decoder error probability for the entire codeword (block error probability) is of the order $O(2^{-N^\beta})$ for some $\beta < 1/2$, i.e., errors are improbable if N is large.

While erasures do take place, the polar transform stabilizes the good channels so much that erasures are rarely in error in such locations.

Special Case: Binary Erasure Channel (BEC)

The BEC is a particularly nice case for polar codes, and it helps illustrate why later channels are reliable:

→ In a BEC, the output can be correct (y = x) or erased (y =?). There are no "wrong" outputs (like in a BSC, where 0 might be flipped to 1).

→ For $N = 2$:

- W^+: Probability of erasure is $2\epsilon - \epsilon^2$. If either y_1 or y_2 is erased, u_1 cannot be decided (unless the two outputs are the same).
- (W^-): Erasure probability is ϵ^2. u_2 is erased only if both y_1 and y_2 are erased because having u_1 would decode u_2.

→ As we recurse, the good channels (e.g., $(W^-)^-$) have exponentially diminishing erasure probabilities (e.g., $(\epsilon^2)^2 = \epsilon^4$), while bad channels approach erasure probability 1.

For a BEC, SC decoding is especially robust since:

→ If the u_i is lost, the decoder can typically make do, for erasures will not put erroneous information there.

→ The Bhattacharyya parameter Z_{WNi} of a BEC is simply the erasure probability, so $Z_{WNi} \to 0$ indicates that erasure probability is small.

For a binary symmetric channel (BSC) with crossover probability ϵ:

→ Errors in decoding of u_1 (i.e., $u_1 = \hat{u}_1$) can propagate and affect u_2, u_3, etc.

→ But the polarization effect remains: the good channels have $Z(W_N^{(i)}) \to 0$, so their error probability (given correct prior bits) is very small.

→ The overall block error probability with SC decoding is very small for large N, since we insert data bits into the good channels, and the error probability at their positions is very small.

4. Encoding

Polar codes are linear block codes that are meant to reach the symmetric binary-input discrete memoryless channels (B-DMCs) capacity when the block length N tends to infinity [3]. The encoding operation converts K information bits into an N-bit codeword, where $N = 2^n$, with a generator matrix based on the channel polarization effect. This subsection explains the encoding process, i.e., construction of the generator matrix, information set selection, and the steps involved in encoding, along with examples and analysis of complexity.

4.1 Generator Matrix Construction

The essence of polar code encoding is the generator matrix G_N, which is built as the n-th Kronecker power of a 2x2 kernel matrix F:

$$F = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

The generator matrix is given as:

$$G_N = F^{\otimes n} = \underbrace{F \otimes F \otimes \cdots \otimes F}_{n \text{ times}}$$

where $\otimes$ is the Kronecker product. For a code of length $N = 2^n$, G_N is an $N \times N$ matrix. For instance, for $N = 4$ ($n = 2$):

$$\text{F} \otimes F = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} = G_4$$

The encoding process is a linear transformation in which the input vector $u_1^N = [u_1, u_2, \ldots, u_N]$ is multiplied by G_N:

$$x_1^N = u_1^N G_N$$

Here, $x_1^N = [x_1, x_2, \ldots, x_N]$ is the encoded codeword, and operations are carried out in the binary field (modulo-2 arithmetic).

4.2 Information Set Selection

The performance of polar codes depends on choosing the information set A, a subset of $\{1, 2, \ldots, N\}$ that has the indices of the K most reliable synthetic channels. Channel polarization, as defined above, maps the original channel W into N synthetic channels $W_N^{(i)}$, $i = 1, 2, \ldots, N$. These artificial channels have different reliabilities, quantified by their capacity $I(W_N^{(i)})$ or Bhattacharyya parameter $Z(W_N^{(i)})$, which estimates the error probability.

To choose A: - Calculate Reliability: Each artificial channel's reliability can be recursively computed. For a B-DMC W, the polarization process generates two kinds of channels at

each iteration:

$$W^-(y_1, y_2, u_2|u_1) = \sum_{u_2'} W(y_1|u_1 \oplus u_2')W(y_2|u_2')$$

$$W^+(y_1, y_2|u_2|u_1) = W(y_1|u_1 \oplus u_2)W(y_2|u_2)$$

The Bhattacharyya parameter for these channels is bounded as:

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

These recursions are used n times to calculate $Z(W_N^{(i)})$ for every i. - Rank Channels: Order the channels by decreasing reliability (rising $I(W_N^{(i)})$ or falling $Z(W_N^{(i)})$). - Choose K Channels: Select the K indices corresponding to the most credible channels to create A. The rest of the $N - K$ indices create the frozen set A^c.

Practically, for certain channels such as the binary erasure channel (BEC), the capacities may be calculated exactly, or precomputed sequences are employed for efficiency [9].

4.3 Encoding Process

The encoding algorithm for an (N, K) polar code can be described in the following steps:

1. **Define the Information Set** A: From the reliability measures (e.g., $Z(W_N^{(i)})$), choose the K most reliable synthetic channels to make up the information set A. The rest of the indices constitute the frozen set A^c.

2. **Build the Input Vector** u_1^N: Place the K information bits in the positions u_i such that $i \in A$. Set $u_i = 0$ for $i \in A^c$, considering frozen bits to be zero (other constant values can be used if agreed upon by the encoder and decoder).

3. **Perform Encoding**: Calculate the codeword:

$$x_1^N = u_1^N G_N$$

   where the matrix multiplication is done over the binary field.

Encoding complexity is $O(N \log N)$, since the topology of G_N supports quick transform like in the Fast Fourier Transform, utilising a butterfly diagram or recursive computation [3].

4.4 Encoding Example

Take an $(N = 4, K = 2)$ polar code. Assume the synthetic channels are ordered in terms of reliability, and channels $W_4^{(3)}$ and $W_4^{(4)}$ are the most reliable, so $A = \{3, 4\}$ and $A^c = \{1, 2\}$.

Let the information bits be b_1 and b_2. The input vector is formed as:

$$u_1^4 = [u_1, u_2, u_3, u_4] = [0, 0, b_1, b_2]$$

Using generator matrix G_4:

$$G_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

The codeword is:

$x_1^4 = u_1^4 G_4 = [0, 0, b_1, b_2] \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix}$ Calculating each element of x_1^4:

- $x_1 = u_1 \oplus u_3 \oplus u_4 = 0 \oplus b_1 \oplus b_2 = b_1 \oplus b_2$
- $x_2 = u_2 \oplus u_4 = 0 \oplus b_2 = b_2$
- $x_3 = u_3 = b_1$
- $x_4 = u_2 \oplus u_3 \oplus u_4 = 0 \oplus b_1 \oplus b_2 = b_1 \oplus b_2$

Therefore:

$$x_1^4 = [b_1 \oplus b_2, b_2, b_1, b_1 \oplus b_2]$$

This example illustrates how the information bits are encoded into a codeword by the linear transformation defined by G_N.

4.5 Complexity Analysis

The polar code's encoding complexity is $O(N \log N)$, which is good for long block lengths. This is because G_N can be decomposed into $n = \log_2 N$ phases, each involving $O(N)$ operations. The recursive nature of the encoding process can be represented using a butterfly diagram, with each phase having pairwise XOR operations [8].

For hardware realization, the algorithms can further minimize the XOR operations, particularly for systematic polar codes, through the exploitation of the structure of G_N. For example, matrix segmentation can obtain as much as 58.5% zero elements for $N = 2048$ and code rate 0.5 and minimize computational resources [10].

The recursive nature of G_N facilitates efficient encoding with a computational complexity of $O(N \log N)$, similar to the Fast Fourier Transform (FFT). This is due to the fact that G_N can be expressed as a product of sparse matrices, and thus fast matrix-vector multiplication is possible. The encoding can be represented recursively:

- For an input vector u of length N, divide u into two halves u_L and u_R (each of length $N/2$). - Calculate the codeword $x = uG_N$ as: [$x_L = u_L G_{N/2}$, $x_R = (u_L + u_R)G_{N/2}$] - Recursively apply this process to u_L and $u_L + u_R$, until a base case ($N = 2$) is achieved.

This recursive process takes advantage of the structure of G_N, making Polar Codes computationally appealing for usage in applications like 5G communication systems [3].

4.6 Practical Considerations

In practical systems, such as 5G control channels, the information set A is often precomputed for specific channel conditions to minimize latency [11]. The frozen bits are typically set to zero, but other values can be used if both encoder and decoder agree, providing flexibility for specific applications.

5. 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.

5.1 Introduction to BPSK Modulation

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:

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

BPSK is favored in communication systems due to its simplicity and robustness against noise, making it an ideal choice for evaluating Polar Codes [24].

5.2 Additive White Gaussian Noise (AWGN)

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:

$$y = s + n \tag{2}$$

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:

$$\sigma^2 = \frac{N_0}{2}, \quad \text{where} \quad N_0 = \frac{1}{E_b/N_0} \tag{3}$$

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:

$$P_b = Q\left(\sqrt{\frac{2E_b}{N_0}}\right) \tag{4}$$

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 [29].

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 [28].

5.3 Combined Noise: AWGN + Laplace + Uniform

In this simulation, the total noise n is modeled as the sum of three independent noise components: Gaussian noise n_1, Laplace noise n_2, and Uniform noise n_3. Each component is scaled to have the same variance v. The received signal y is given by $y = s + n$, where s is the transmitted signal. The noise components are generated in MATLAB as follows:

```
% Gaussian noise
n1 = sqrt(v) * randn(1, N);

% Laplace noise
b = sqrt(v / 2);
U = rand(1, N);
n2 = b * (log(2 * U) .* (U < 0.5) - log(2 * (1 - U)) .* (U >= 0.5));

% Uniform noise
a = sqrt(3 * v);
n3 = a * (2 * rand(1, N) - 1);

% Total noise
n = n1 + n2 + n3;

% Received signal
y = s + n;
```

The objective is to derive the properties of the total noise n and assess whether it can be approximated as Additive White Gaussian Noise (AWGN) using the Central Limit Theorem (CLT).

5.3.1 Noise Component Analysis

Gaussian Noise (n_1)

The Gaussian noise component n_1 is generated as $n_1 = \sqrt{v} \cdot \text{randn}(1, N)$, where $\text{randn}(1, N)$ produces standard normal random variables with mean 0 and variance 1. Scaling by $\sqrt{v}$ adjusts the variance to $\text{Var}(n_1) = (\sqrt{v})^2 \cdot 1 = v$. Thus, $n_1 \sim \mathcal{N}(0, v)$, with mean 0 and variance v [27].

Laplace Noise (n_2)

The Laplace noise n_2 is generated using the inverse transform method. For a uniform random variable $U \sim \text{Uniform}(0, 1)$, the Laplace random variable is computed as

$$n_2 = b \cdot \begin{cases} \log(2U) & \text{if } U < 0.5, \\ -\log(2(1-U)) & \text{if } U \geq 0.5, \end{cases}$$

where $b = \sqrt{\frac{v}{2}}$. This produces a Laplace distribution with probability density function $f(x) = \frac{1}{2b} e^{-\frac{|x|}{b}}$, mean 0, and variance $2b^2$. Substituting b, we get $\text{Var}(n_2) = 2\left(\sqrt{\frac{v}{2}}\right)^2 = 2 \cdot \frac{v}{2} = v$ [26].

Uniform Noise (n_3)

The Uniform noise n_3 is generated as $n_3 = a \cdot (2 \cdot \text{rand}(1, N) - 1)$, where $a = \sqrt{3v}$ and $\text{rand}(1, N) \sim \text{Uniform}(0, 1)$. This results in $n_3 \sim \text{Uniform}(-a, a)$. The variance of a uniform distribution over $[-a, a]$ is $\frac{(a-(-a))^2}{12} = \frac{(2a)^2}{12} = \frac{4a^2}{12} = \frac{a^2}{3}$. With $a = \sqrt{3v}$, we have $\text{Var}(n_3) = \frac{(\sqrt{3v})^2}{3} = \frac{3v}{3} = v$ [25].

5.3.2 Total Noise (n)

Given the independence of n_1, n_2, and n_3, the total noise $n = n_1 + n_2 + n_3$ has mean

$$E[n] = E[n_1] + E[n_2] + E[n_3] = 0 + 0 + 0 = 0$$

and variance

$$\text{Var}(n) = \text{Var}(n_1) + \text{Var}(n_2) + \text{Var}(n_3) = v + v + v = 3v.$$

5.3.3 Approximation as AWGN

To demonstrate that the sum of random variables from Additive White Gaussian Noise (AWGN), Laplace, and Uniform distributions can be approximated as AWGN with variance $3v$, we utilize the Central Limit Theorem (CLT) and characteristic functions. The CLT states that the sum of many independent and identically distributed (i.i.d.) random variables, each with finite mean and variance, tends to follow a normal distribution as the number of summands increases, regardless of the original distributions.[30]

Consider three random variables:

- $n_1 \sim \mathcal{N}(0, v)$, representing AWGN with mean 0 and variance v.
- n_2, following a Laplace distribution with scale parameter $b = \sqrt{\frac{v}{2}}$, such that its variance is $2b^2 = v$.
- $n_3 \sim \text{Uniform}(-a, a)$, with $a = \sqrt{3v}$, so its variance is $\frac{a^2}{3} = v$.

The characteristic functions for these distributions are:

$$\phi_{n_1}(t) = e^{-\frac{vt^2}{2}},$$

$$\phi_{n_2}(t) = \frac{1}{1+b^2t^2} = \frac{1}{1+\frac{v}{2}t^2},$$

$$\phi_{n_3}(t) = \frac{\sin(at)}{at} = \frac{\sin(\sqrt{3v}t)}{\sqrt{3v}t}.$$

Now, consider the sum of these random variables, $S = n_1 + n_2 + n_3$. Since the random variables are independent, the characteristic function of the sum is the product of their individual characteristic functions:

$$\phi_S(t) = \phi_{n_1}(t) \cdot \phi_{n_2}(t) \cdot \phi_{n_3}(t) = e^{-\frac{vt^2}{2}} \cdot \frac{1}{1+\frac{v}{2}t^2} \cdot \frac{\sin(\sqrt{3v}t)}{\sqrt{3v}t}.$$

To apply the CLT, we consider the normalized sum of multiple i.i.d. copies of each distribution. For each distribution, define the sum of n i.i.d. random variables:

- For n_1: $Z_1 = \frac{N_1+N_2+\cdots+N_n}{\sqrt{n}}$, where each $N_i \sim \mathcal{N}(0, v)$.
- For n_2: $Z_2 = \frac{L_1+L_2+\cdots+L_n}{\sqrt{n}}$, where each $L_i \sim \text{Laplace}(b = \sqrt{\frac{v}{2}})$.
- For n_3: $Z_3 = \frac{U_1+U_2+\cdots+U_n}{\sqrt{n}}$, where each $U_i \sim \text{Uniform}(-\sqrt{3v}, \sqrt{3v})$.

By the CLT, as $n \to \infty$, each Z_i converges in distribution to a normal distribution with mean 0 and variance equal to the variance of the original distribution, which is v. Thus:

$$Z_i \xrightarrow{d} \mathcal{N}(0, v), \quad i = 1, 2, 3.$$

The characteristic function of a normal distribution $\mathcal{N}(0, v)$ is $e^{-\frac{vt^2}{2}}$. Therefore, for large n, the characteristic function of each Z_i approaches:

$$\phi_{Z_i}(t) \approx e^{-\frac{vt^2}{2}}.$$

Now, consider the sum $S = n_1 + n_2 + n_3$. The variance of S is the sum of the variances, since the variables are independent:

$$\text{Var}(S) = \text{Var}(n_1) + \text{Var}(n_2) + \text{Var}(n_3) = v + v + v = 3v.$$

The characteristic function of S, $\phi_S(t)$, is approximated by considering the limiting behavior. Since each component distribution, when summed and normalized, approaches a normal distribution, the sum S can be approximated as a normal distribution with mean 0 and variance $3v$. The characteristic function of $\mathcal{N}(0, 3v)$ is:

$$\phi_{\mathcal{N}(0,3v)}(t) = e^{-\frac{3vt^2}{2}}.$$

To confirm, we analyze the product $\phi_S(t)$. For small t, we can approximate the non-Gaussian terms:

- For the Laplace term, $\frac{1}{1+\frac{v}{2}t^2} \approx e^{-\frac{vt^2}{2}}$ for small t, as $\ln\left(\frac{1}{1+x}\right) \approx -x$ when $x = \frac{v}{2}t^2$ is small.
- For the Uniform term, $\frac{\sin(\sqrt{3v}t)}{\sqrt{3v}t} \approx e^{-\frac{vt^2}{2}}$, since $\frac{\sin(x)}{x} \approx 1 - \frac{x^2}{6}$, and adjusting for the variance.

Thus, for small t:

$$\phi_S(t) \approx e^{-\frac{vt^2}{2}} \cdot e^{-\frac{vt^2}{2}} \cdot e^{-\frac{vt^2}{2}} = e^{-\frac{3vt^2}{2}},$$

which matches the characteristic function of $\mathcal{N}(0, 3v)$.

Therefore, by the CLT and the properties of characteristic functions, the sum $S = n_1 + n_2 + n_3$ can be approximated as AWGN with variance $3v$, i.e., $S \approx \mathcal{N}(0, 3v)$.

6. Decoding

Polar codes are decoded using successive cancellation (SC) decoding, which achieves the channel capacity for sufficiently large block lengths. However, for finite block lengths, SC decoding may not provide optimal error correction performance. To address this limitation, Successive Cancellation List (SCL) decoding was introduced, significantly improving the performance of polar codes, especially for short and moderate block lengths. SCL decoding is the primary decoding method used for polar codes in 5G standards due to its superior error correction capabilities compared to standard SC decoding [12].

6.1 Successive Cancellation (SC) Decoding

Due to the recursive structure of channel polarization, polar codes can be efficiently decoded using a simple scheme known as **Successive Cancellation (SC)** decoding.

We assume transmission over an AWGN channel using BPSK modulation. Thus, we receive Log-Likelihood Ratios (LLRs) of transmitted bits.

6.1.1 LLR Definitions

Let r_0 and r_1 be received symbols. The corresponding LLRs are defined as:

$$L_{r_0} = \log\left(\frac{p(r_0 = 0)}{p(r_0 = 1)}\right), \tag{5}$$

$$L_{r_1} = \log\left(\frac{p(r_1 = 0)}{p(r_1 = 1)}\right). \tag{6}$$

Let's define the Log-Likelihood Ratios (LLRs) for u_0 and u_1 as follows.

$$L_{u_0} = \log\left(\frac{P(u_0 = 0)}{P(u_0 = 1)}\right) = \log\left(\frac{p_{u_0}}{1 - p_{u_0}}\right)$$

$$L_{r_1} = \log\left(\frac{P(r_1 = 0)}{P(r_1 = 1)}\right) = \log\left(\frac{p_{r_1}}{1 - p_{r_1}}\right)$$

The task of determining the LLR for the bit u_0 reduces to solving the problem of single parity check (SPC) decoding for the values r_0 and r_1. The bit u_0 is transmitted as 0 when both r_1 and r_2 are either 0 or 1. Therefore, we can express the probability p_{u_0} as:

$$p_{u_0} = p_{r_0}p_{r_1} + (1 - p_{r_0})(1 - p_{r_1})$$
$$1 - p_{u_0} = p_{r_0}(1 - p_{r_1}) + (1 - p_{r_0})p_{r_1}$$

By subtracting the second equation from the first, we obtain:

$$p_{u_0} - (1 - p_{u_0}) = (p_{r_0} - (1 - p_{r_0}))(p_{r_1} - (1 - p_{r_1}))$$

$$\Rightarrow \frac{p_{u_0} - (1 - p_{u_0})}{p_{u_0} + (1 - p_{u_0})} = \frac{(p_{r_0} - (1 - p_{r_0}))}{(p_{r_0} + (1 - p_{r_0}))} \cdot \frac{(p_{r_1} - (1 - p_{r_1}))}{(p_{r_1} + (1 - p_{r_1}))}$$

$$\Rightarrow \frac{1 - \exp(-L_{u_0})}{1 + \exp(-L_{u_0})} = \left(\frac{1 - \exp(-L_{r_0})}{1 + \exp(-L_{r_0})}\right)\left(\frac{1 - \exp(-L_{r_1})}{1 + \exp(-L_{r_1})}\right)$$

$$\Rightarrow \tanh\left(\frac{L_{u_0}}{2}\right) = \tanh\left(\frac{L_{r_0}}{2}\right)\tanh\left(\frac{L_{r_1}}{2}\right)$$

$$\Rightarrow L_{u_0} = 2\tanh^{-1}\left(\tanh\left(\frac{L_{r_0}}{2}\right)\tanh\left(\frac{L_{r_1}}{2}\right)\right)$$

This expression can be approximated by:

$$L_{u_0} = \text{sgn}(L_{r_0})\text{sgn}(L_{r_1})\min\left(|L_{r_0}|, |L_{r_1}|\right)$$

This is known as the min-sum approximation. We define the min-sum function as:

$$f(x_0, x_1) = \text{sgn}(x_0)\text{sgn}(x_1)\min\left(|x_0|, |x_1|\right)$$

Thus, we can rewrite the above equation as:

$$L_{u_0} = f(L_{r_0}, L_{r_1})$$

Based on the calculated LLR for u_0, we estimate u_0 as follows:

$$\hat{u_0} = \begin{cases} 0, & \text{if } L_{u_0} \geq 0 \\ 1, & \text{if } L_{u_0} < 0 \end{cases}$$

Once the estimate $\hat{u_0}$ is made, we can further estimate u_1. If $\hat{u_0} = 0$, then L_{r_0} and L_{r_1} will both serve as beliefs for u_1. On the other hand, if $\hat{u_0} = 1$, then L_{r_0} will act as a belief for $\bar{u_1}$, while L_{r_1} will be a belief for u_1. The task of estimating the bit u_1 now becomes a problem of performing a repetition code estimation. Assume we have $\hat{u_0} = 0$. For the repetition code, we know that the probability p_{u_1} is given by:

$$p_{u_1} = \frac{p_{r_0}p_{r_1}}{p_{r_0}p_{r_1} + (1 - p_{r_0})(1 - p_{r_1})} \tag{7}$$

$$1 - p_{u_1} = \frac{(1 - p_{r_0})(1 - p_{r_1})}{p_{r_0}p_{r_1} + (1 - p_{r_0})(1 - p_{r_1})} \tag{8}$$

Dividing equation (7) by equation (8), we get:

$$\frac{p_{u_1}}{1 - p_{u_1}} = \left(\frac{p_{r_0}}{1 - p_{r_0}}\right)\left(\frac{p_{r_1}}{1 - p_{r_1}}\right)$$

Taking the logarithm on both sides yields:

$$\log\left(\frac{p_{u_1}}{1 - p_{u_1}}\right) = \log\left(\frac{p_{r_0}}{1 - p_{r_0}}\right) + \log\left(\frac{p_{r_1}}{1 - p_{r_1}}\right)$$

$$\Rightarrow L_{u_1} = L_{r_0} + L_{r_1} \quad \text{(from equations (5) and (6))}$$

Similarly, when $\hat{u_0} = 1$, we can derive the following expression using the same steps:

$$L_{u_1} = L_{r_1} - L_{r_0}$$

By combining the two equations above, we obtain:

$$L_{u_1} = L_{r_1} + (-1)^{\hat{u_0}} L_{r_0}$$

We can now define a function $g(x)$ as follows:

$$g(x_0, x_1, y) = x_1 + (-1)^y x_0$$

Thus, the equation for L_{u_1} can be rewritten as:

$$L_{u_1} = g(r_0, r_1, \hat{u_0})$$

After computing the LLR for u_1, we can estimate u_1 as:

$$\hat{u_1} = \begin{cases} 0, & \text{if } L_{u_1} \geq 0 \\ 1, & \text{if } L_{u_1} < 0 \end{cases}$$

Now that we have an understanding of how decoding works sequentially, we can fully grasp the functionality of the SC decoder as illustrated in figure ??.

The process begins by passing the received LLRs through node f, after which we make the decision for bit u_0. Based on this decision, the values are then passed to node g, and the decision for bit u_1 is made. Finally, the decoding process is complete.

6.1.2 SC Decoder for Length 4 Code

The structure is recursively extended for a length-4 code. Decoding proceeds step by step using a combination of f and g functions.

Stepwise Decoding Process

1. **Step 1:** Use LLRs and pass them through f nodes to estimate $\hat{u}_0$:

$$L_{u_0} = f(L_{r_0}, L_{r_1}) = \text{sgn}(L_{r_0}) \cdot \text{sgn}(L_{r_1}) \cdot \min(|L_{r_0}|, |L_{r_1}|)$$

2. **Step 2:** Use $\hat{u}_0$ and g function to compute L_{u_1} and estimate $\hat{u}_1$:

$$L_{u_1} = g(L_{r_0}, L_{r_1}, \hat{u}_0) = L_{r_1} + (-1)^{\hat{u}_0} L_{r_0}$$

3. **Step 3:** Use $\hat{u}_0$ and $\hat{u}_1$ to compute intermediate values and estimate $\hat{u}_2$:

$$L_{u_2} = f(L_{r_2}, L_{r_3}) \quad \text{(using similar logic as in Step 1)}$$

4. **Step 4:** Use previous estimates and compute:

$$L_{u_3} = g(L_{r_2}, L_{r_3}, \hat{u}_2)$$

6.1.3 Generalization

Due to the recursive nature of polar codes, the same structure and logic can be generalized to any code length $N = 2^n$. SC decoding complexity is $\mathcal{O}(N \log N)$, making it efficient and practical.

Successive Cancellation decoding takes advantage of the recursive construction of polar codes. By using a combination of basic decoding functions f and g, it successively estimates each bit using previously decoded values. Despite its simplicity, SC decoding provides reliable performance for large block lengths and is the foundational algorithm behind modern polar code decoders.

6.1.4 Simulation Results

This section presents the simulation results for the Bit Error Rate (BER), Block Error Rate (BLER), and the probability of successful decoding for polar codes decoded using the Successive Cancellation (SC) decoder. The polar code parameters are $N = 1024$, $K = 512$, and the code rate is $R = 0.5$. The simulations are conducted over an AWGN channel with BPSK modulation.

6.2 Successive Cancellation List (SCL) Decoding

SCL decoding enhances standard SC decoding by maintaining a list of the most likely candidate paths instead of a single path. This approach allows the decoder to explore multiple possible bit decisions, thereby improving error correction capabilities. The list size L is a critical parameter that controls the trade-off between decoding performance and computational complexity. Common list sizes in practice include $L = 8, 16, 32$, with larger values improving performance at the cost of increased complexity [12].

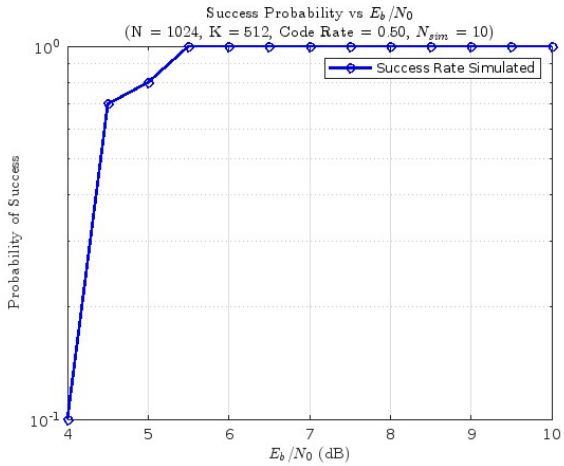

Figure 5: Probability of Success vs E_b/N_0 for SC decoding of polar code with $N_{\text{sim}} = 10$

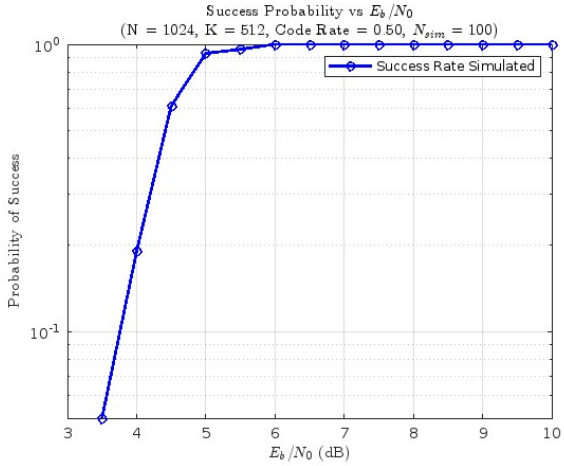

Figure 6: Probability of Success vs E_b/N_0 for SC decoding of polar code with $N_{\text{sim}} = 100$

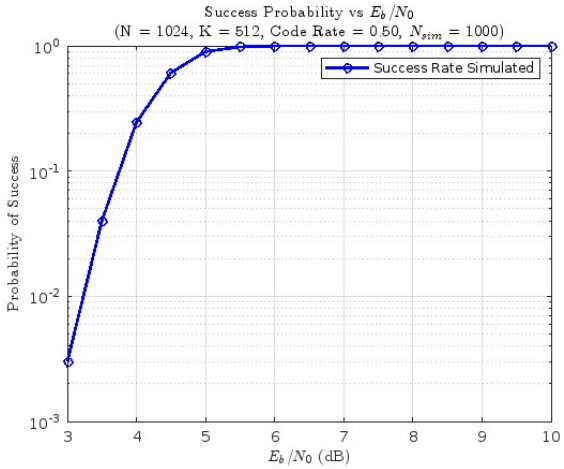

Figure 7: Probability of Success vs E_b/N_0 for SC decoding of polar code with $N_{\text{sim}} = 1000$

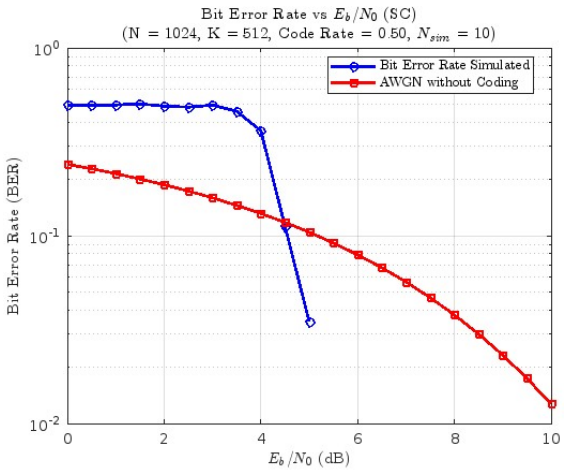

Figure 8: BER vs E_b/N_0 for SC decoding of polar code with $N_{\text{sim}} = 10$

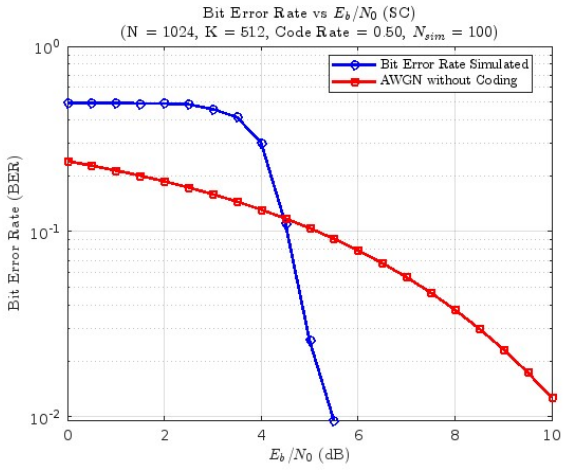

Figure 9: BER vs E_b/N_0 for SC decoding of polar code with $N_{\text{sim}} = 100$

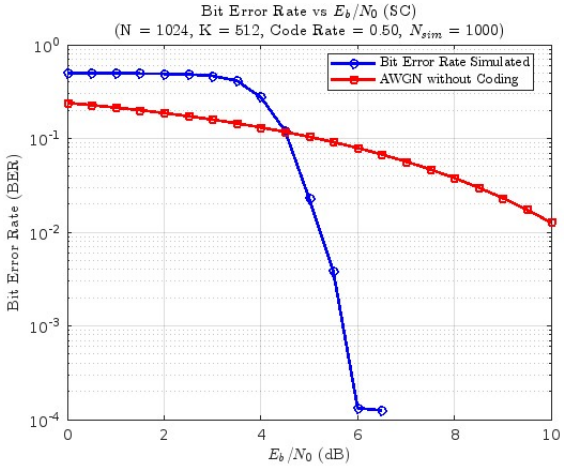

Figure 10: BER vs E_b/N_0 for SC decoding of polar code with $N_{\text{sim}} = 1000$

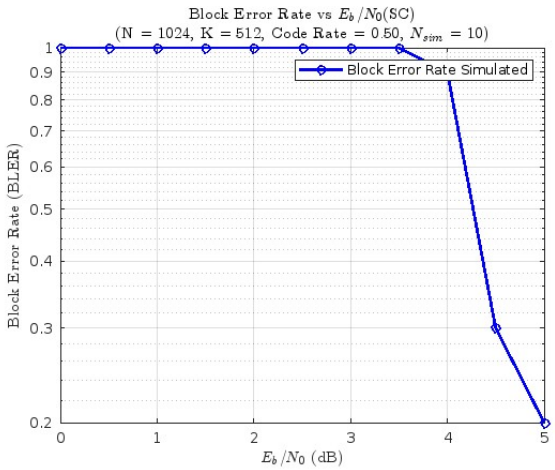

Figure 11: BLER vs E_b/N_0 for SC decoding of polar code with $N_{\text{sim}} = 10$

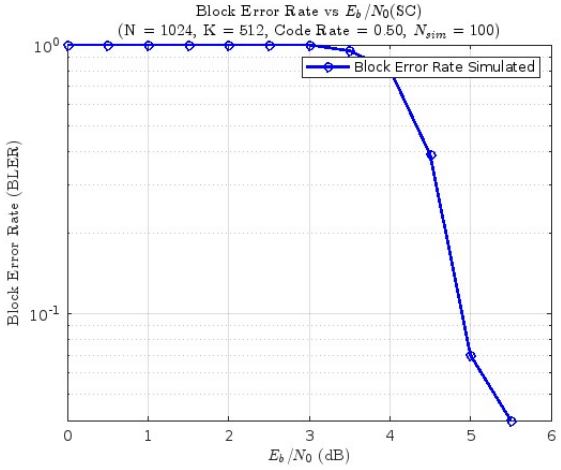

Figure 12: BLER vs E_b/N_0 for SC decoding of polar code with $N_{\text{sim}} = 100$

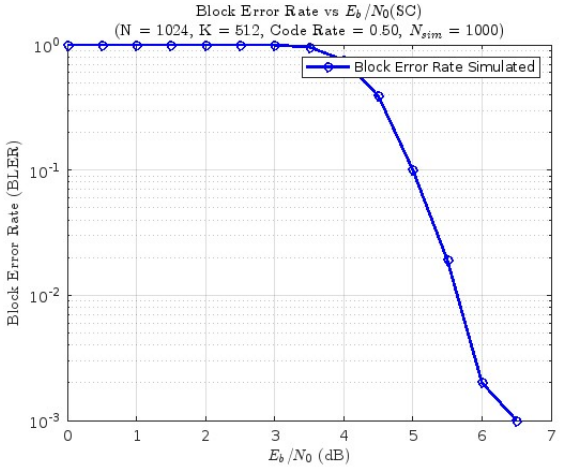

Figure 13: BLER vs E_b/N_0 for SC decoding of polar code with $N_{\text{sim}} = 1000$

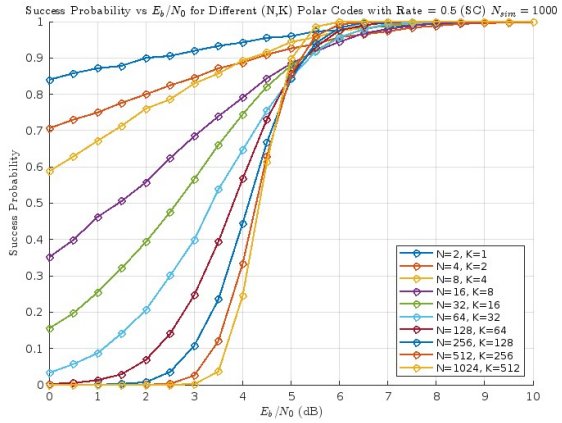

Figure 14: Probability of Success vs E_b/N_0 for SC decoding of polar code with different values of (N, K)

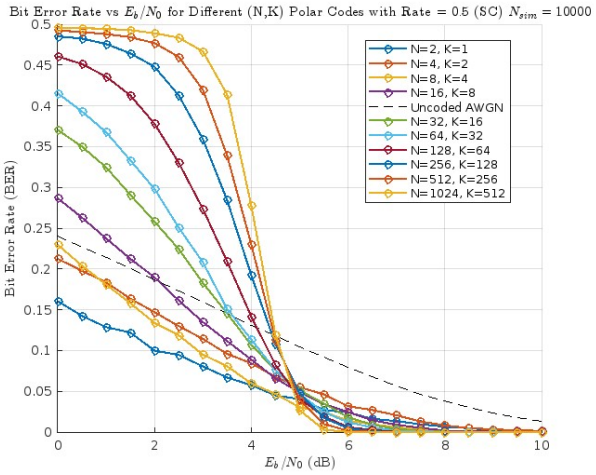

Figure 15: BER vs E_b/N_0 for SC decoding of polar code with different values of (N, K)

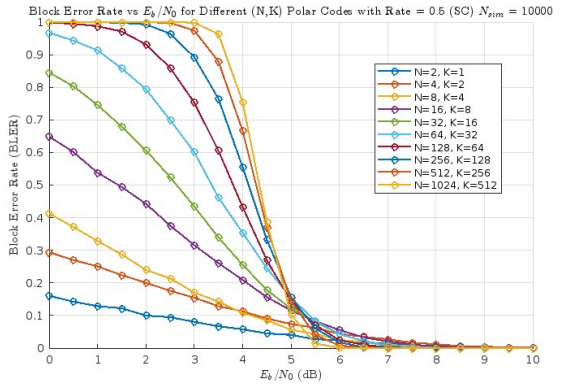

Figure 16: BLER vs E_b/N_0 for SC decoding of polar code with different values of (N, K)

6.2.1 Algorithm

The SCL decoding algorithm can be described as follows:

1. **Initialization**: Start with a single path (the root of the decoding tree) with a path metric of zero.
2. **Extension**: For each bit position i from 1 to N:
   - If i is a frozen bit ($i \in A^c$), set $\hat{u}_i = 0$ for all paths in the list.
   - If i is an information bit ($i \in A$), for each path in the current list, create two new paths: one with $\hat{u}_i = 0$ and one with $\hat{u}_i = 1$.
3. **Pruning**: After extending the paths, if the number of paths exceeds the list size L, keep only the L paths with the highest metrics (or lowest path metrics, depending on the implementation).
4. **Selection**: At the end of the decoding process, select the path with the highest metric as the final estimate. If a cyclic redundancy check (CRC) is used, select the path that passes the CRC check.

This algorithm ensures that the most likely candidate paths are retained at each step, increasing the likelihood of correcting errors compared to SC decoding [12].

6.2.2 Path Metric

In SCL decoding, each path is associated with a path metric that measures its likelihood. The path metric is computed using the log-likelihood ratio (LLR), defined as:

$$L_N^{(i)}(y_1^N, \hat{u}_1^{i-1}) = \log\left(\frac{W_N^{(i)}(y_1^N, \hat{u}_1^{i-1}|u_i = 0)}{W_N^{(i)}(y_1^N, \hat{u}_1^{i-1}|u_i = 1)}\right)$$

Here, $W_N^{(i)}$ represents the transition probability of the i-th synthetic channel in the polar code, y_1^N is the received channel output, and $\hat{u}_1^{i-1}$ are the previously estimated bits. The path metric for a path l is updated as:

$$PM_l = PM_l - \log\left(1 + e^{-(1-2\hat{u}_i)L_N^{(i)}}\right)$$

This update rule ensures that the path metric reflects the cumulative likelihood of the decisions made along the path. To reduce computational complexity, min-sum approximations can be applied, which simplify the logarithmic and exponential operations while maintaining good performance [13].

6.2.3 CRC-Aided SCL Decoding

To further enhance the performance of SCL decoding, CRC-Aided SCL (CA-SCL) decoding is employed. In CA-SCL, a cyclic redundancy check (CRC) is appended to the information bits, and the decoder selects the path that passes the CRC check. This approach significantly improves error correction by leveraging the CRC to identify the correct path among the candidate paths.

In 5G standards, polar codes typically use CRC lengths of 11 or 24 bits, depending on the specific use case, such as control channels or ultra-reliable low-latency communications (URLLC). The use of CRC allows CA-SCL decoding to approach maximum likelihood (ML) performance, making it highly effective for practical applications [14].

6.2.4 Data Structures

Efficient implementation of SCL decoding requires careful management of data structures to store and process the list of paths. The key data structures include:

- **Probability Arrays**: Store the LLR values for each bit position across all paths.
- **Bit Arrays**: Store the bit estimates ($\hat{u}_i$) for each path.
- **Path Index Stack**: Manages the list of paths, often using techniques like lazy-copy to reduce memory usage and improve efficiency [12].

These structures ensure that the decoder can efficiently track and update multiple candidate paths during the decoding process.

6.2.5 Complexity

The computational complexity of SCL decoding is higher than that of SC decoding due to the maintenance of multiple paths. The complexity is characterized as follows:

- **Time Complexity**: $O(LN \log N)$, where L is the list size and N is the block length. The $\log N$ factor arises from the recursive structure of polar codes, similar to SC decoding.
- **Space Complexity**: $O(LN)$, as storage is required for L paths, each of length N.

Despite the increased complexity, SCL decoding is feasible for practical applications, particularly with optimizations that reduce computational overhead [12].

6.2.6 Optimizations

Several optimizations have been proposed to improve the efficiency of SCL decoding, making it more suitable for real-time applications:

- **Fast SCL Decoding**: Exploits the structure of polar codes to decode special nodes (such as rate-0, rate-1, repetition, and single-parity-check nodes) more efficiently, reducing the number of computations [13].

- **Adaptive List Size**: Dynamically adjusts the list size L based on channel conditions or decoding progress, reducing unnecessary computations when a smaller list size is sufficient [17].
- **Logarithmic Decoding**: Uses logarithmic representations of probabilities to simplify computations and reduce hardware complexity, particularly in hardware implementations [15].

These optimizations enhance the practicality of SCL decoding, especially in resource-constrained environments like 5G devices.

6.2.7 Performance

SCL decoding, particularly when combined with CRC, achieves near-maximum likelihood (ML) performance, making it highly effective for error correction. For example, for a polar code with a block length $N = 256$ and a list size $L = 32$, SCL decoding is within 0.1 dB of ML decoding in terms of frame error rate (FER) [12].

In 5G applications, such as machine-type communications (mMTC), polar codes with SCL decoding provide coding gains of 8.3–10 dB and require 1–3 dB less E_b/N_0 to achieve a block error rate (BLER) of 10^{-4} compared to low-density parity-check (LDPC) codes [16]. This performance advantage makes SCL decoding a preferred choice for 5G control channels and URLLC scenarios.

Table 1: SCL Decoding Performance

Feature	**SCL Decoding**
List Size	8, 16, 32
Error Correction	Near-ML with CRC
Complexity	$O(LN \log N)$ time, $O(LN)$ space
Applications	5G control channels, URLLC

6.2.8 Simulation Results

This section presents the simulation results for the Bit Error Rate (BER), Block Error Rate (BLER), and the probability of successful decoding for polar codes decoded using the Successive Cancellation List (SCL) decoder with Cyclic Redundancy Check (CRC-SCL).

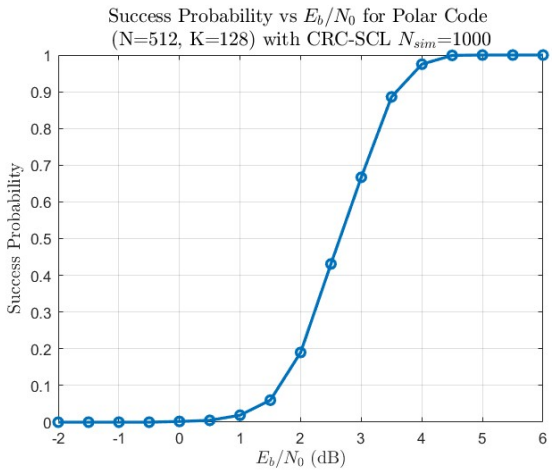

Figure 17: Probability of Success vs E_b/N_0 for CRC-SCL decoding of polar code with $N_{\text{sim}} = 100$

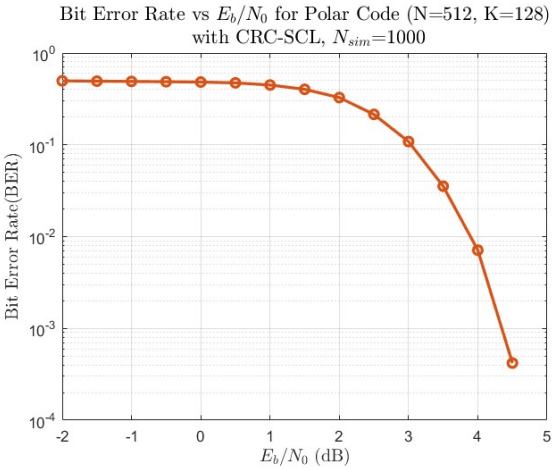

Figure 18: BER vs E_b/N_0 for CRC-SCL decoding of polar code with $N_{\text{sim}} = 100$

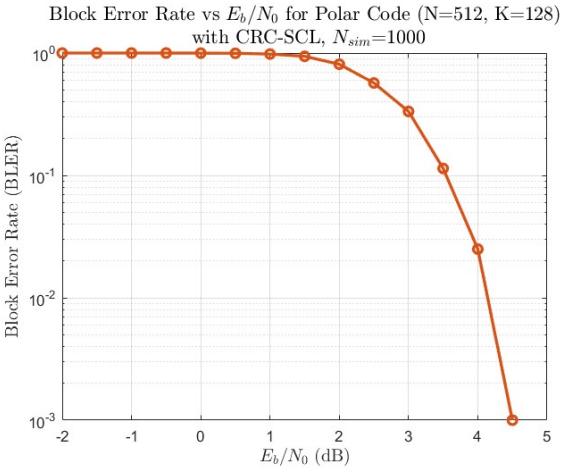

Figure 19: BLER vs E_b/N_0 for CRC-SCL decoding of polar code with $N_{\text{sim}} = 100$

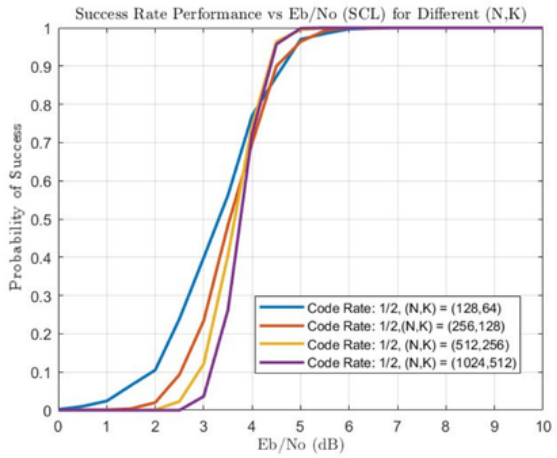

Figure 20: Probability of Success vs E_b/N_0 for CRC-SCL decoding of polar code with different values of (N, K)

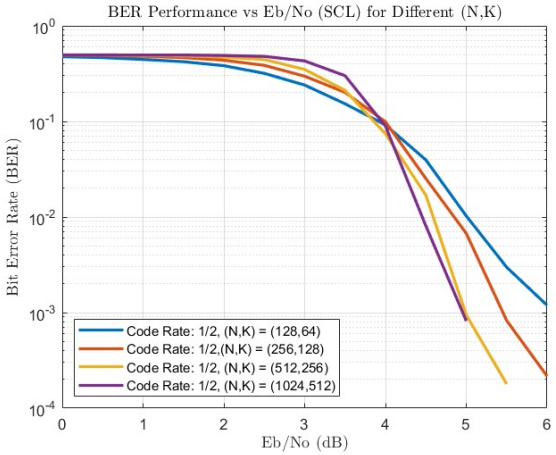

Figure 21: BER vs E_b/N_0 for CRC-SCL decoding of polar code with different values of (N, K)

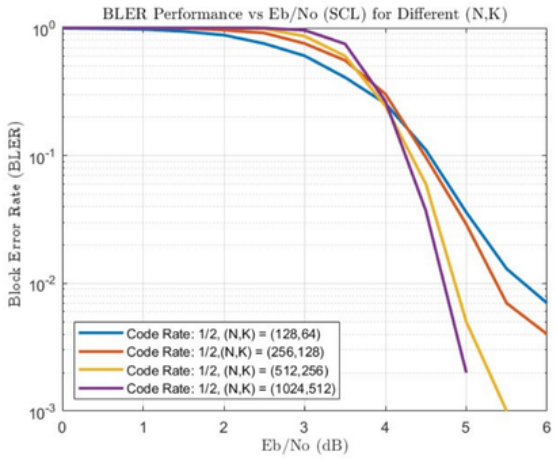

Figure 22: BLER vs E_b/N_0 for CRC-SCL decoding of polar code with different values of (N, K)

7. 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, analyzes their performance, and compares them with Low-Density Parity-Check (LDPC) codes, with citations provided at the end.

7.1 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 [22].

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

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

where $\mathcal{A}$ is the information set. As $N \to \infty$, $P_e \to 0$, proving Polar Codes achieve $I(W)$ with complexity $O(N \log N)$ for encoding and decoding [3].

7.3 Analysis and Comparison with LDPC Codes

Polar Codes explicitly achieve capacity, but their finite-length performance under SC decoding is suboptimal. For a binary erasure channel (BEC) with erasure probability ϵ, $I(W) = 1 - \epsilon$. Simulations show that for $N = 1024$ and $R = 0.5$, Polar Codes have higher frame error rates (FER) than optimized LDPC codes due to SC decoding's sequential nature.

LDPC codes, defined by sparse parity-check matrices, approach capacity iteratively via belief propagation decoding [23]. For the same BEC, an LDPC code with degree distribution optimized via density evolution can achieve FER closer to the Shannon limit at $N = 1024$. However, LDPC codes lack an explicit capacity-achieving proof and require complex optimization.

Polar Codes excel theoretically with predictable scaling ($P_e \sim 2^{-N^\beta}$), while LDPC codes offer practical advantages (parallel decoding, flexibility). Enhanced Polar Code variants (e.g., SCL decoding) close the gap but increase complexity to $O(LN \log N)$, where L is the list size.

7.4 Performance Evaluation

Performance is evaluated via Monte Carlo simulations, measuring BLER/BER versus E_b/N_0, compared to:

Shannon Limit For code rate r:

$$\left(\frac{E_b}{N_0}\right)_{\min} = \frac{2^r - 1}{r}$$

or in dB:

$$10 \log_{10}\left(\frac{2^r - 1}{r}\right)$$

Normal Approximation Error probability:

$$P_{N,e} = Q\left(\sqrt{\frac{N}{V}}\left(C - r + \frac{\log_2 N}{2N}\right)\right)$$

where $C = \frac{1}{2}\log_2(1 + \text{SNR})$, V is channel dispersion. LDPC Comparison Polar Codes excel for short blocks ($N = 128$–512), requiring 1–3 dB less E_b/N_0 for BLER 10^{-4} in mMTC

Table 2: Performance Comparison for $N = 512$, $r = 1/2$

Code	**Decoder**	E_b/N_0 **(dB) for BLER** 10^{-4}
Polar	CA-SCL ($L = 32$)	3.5–4.5
LDPC	LBP	4.5–7.5

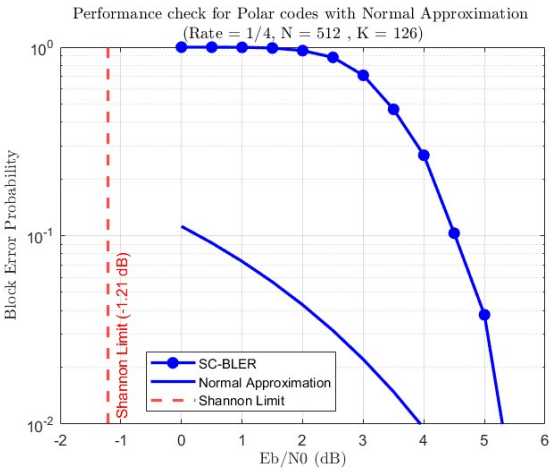

Figure 23: Performance Evaluation with Normal Approximation and Shannon Limit for Polar Codes

7.5 Performance Comparison of SC vs CRC-SCL Decoder

The simulation results clearly show that CRC-aided SCL decoding performs much better than SC decoding. For the same code parameters and conditions, CRC-SCL achieves lower Bit Error Rates (BER) and Block Error Rates (BLER), especially at low signal-to-noise ratios. This is because CRC-SCL checks multiple decoding paths and uses the CRC to choose the most likely correct one.

SC decoding is faster and simpler, making it good for low-latency applications, but it struggles in noisy environments. CRC-SCL, though more complex, is much more reliable and comes closer to ideal (maximum likelihood) performance. It also has a higher probability of correctly decoding the message, making it a better choice when accuracy is more important than speed.

The following graphs compare the performance of polar code using CRC aided Successive Cancellation List (SCL) decoding and Succesive Cancellagtion(SC) decoder.

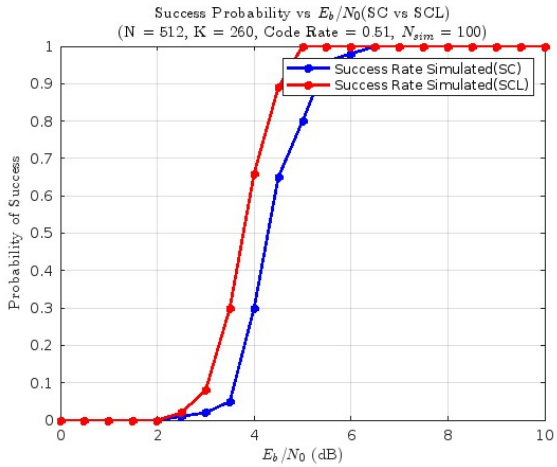

Figure 24: Probability of Success comparison between SC and CRC-SCL decoding. CRC-SCL maintains a higher success rate, especially in noisy conditions.

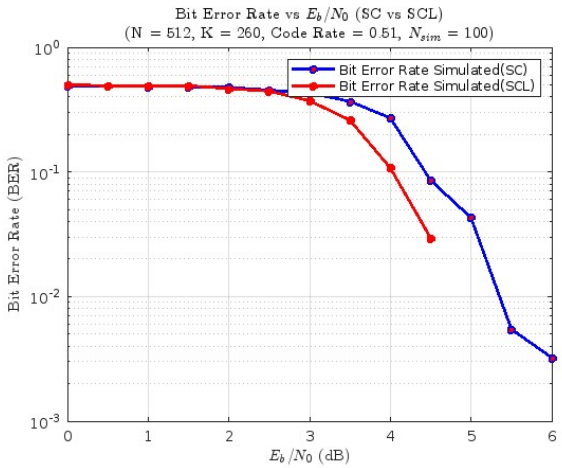

Figure 25: BER comparison between SC and CRC-SCL decoding. CRC-SCL shows lower bit error rates across all E_b/N_0 values.

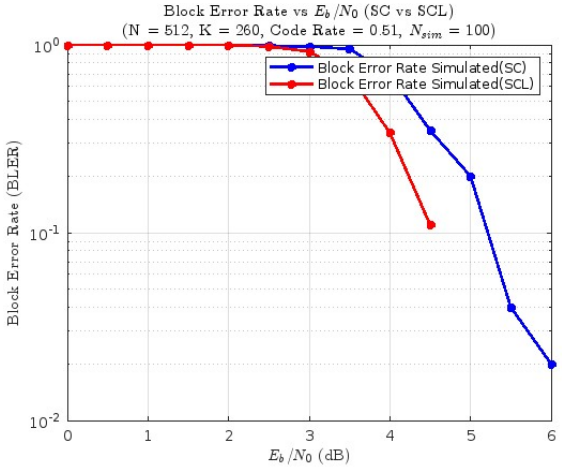

Figure 26: BLER comparison between SC and CRC-SCL decoding. CRC-SCL achieves better block-level accuracy due to CRC-assisted path selection.

7.6 Performance Comparison of CRC-SCL Decoder between Random Noise Channel and BPSK-AWGN Channel

We compared the performance of the CRC-aided SCL decoder over two types of channels: a simple random noise channel and a BPSK-modulated AWGN channel. While the random noise flips bits independently, the BPSK-AWGN channel adds Gaussian noise to modulated signals.

Thanks to the Central Limit Theorem, large combinations of random noise can resemble Gaussian noise, making the AWGN model a realistic approximation. Our results show that the decoder performs much better in the BPSK-AWGN channel, with lower BER and BLER, and higher success rates. The structure and predictability of BPSK over AWGN clearly help the decoder work more efficiently.

The following graphs compare the performance of polar code using CRC aided Successive Cancellation List (SCL) decoding under mixed noise and AWGN.

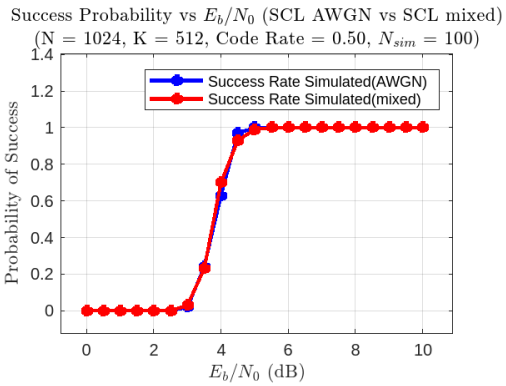

Figure 27: Probability of Success comparison between AWGN and Random Noise Channel.

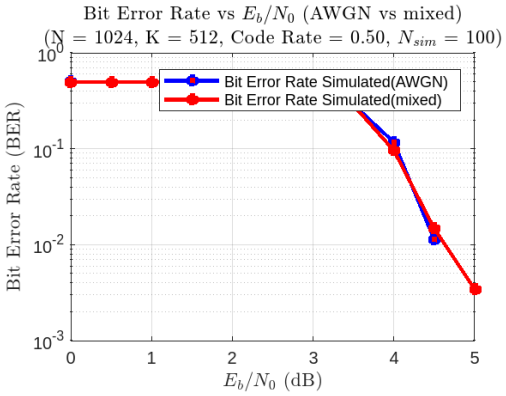

Figure 28: BER comparison between AWGN and Random Noise Channel.

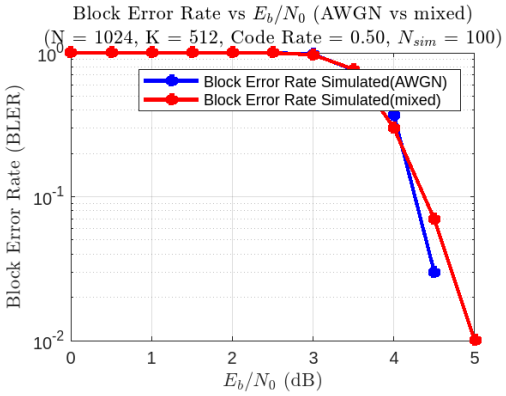

Figure 29: BLER comparison between AWGN and Random Noise Channel.

References

[1] F. Nielsen, "Generalized Bhattacharyya and Chernoff upper bounds on Bayes error using quasi-arithmetic means," arXiv preprint arXiv:1401.4788, 2014. [Online]. Available: https://arxiv.org/abs/1401.4788

[2] C. P. Kitsos and C.-S. Nisiotis, "Considering Distance Measures in Statistics," ResearchGate, 2022. [Online]. Available: https://www.researchgate.net/publication/361686693_Considering_distance_measures_in_Statistics

[3] E. 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, pp. 3051–3073, July 2009.

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

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

[6] E. Arıkan and E. Telatar, "On the Rate of Channel Polarization," *Proc. IEEE International Symposium on Information Theory (ISIT)*, Seoul, South Korea, June 2009.

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

[8] Alexios Balatsoukas-Stimming, Masoud Bagheri, and Andreas Burg. LLR-based successive cancellation list decoding of polar codes. In *2015 Asilomar Conference on Signals, Systems and Computers*, pages 435–439. IEEE, 2015.

[9] Kai Niu, Kai Chen, and Jia-Ru Lin. Beyond turbo codes: Rate-compatible punctured polar codes. In *2012 IEEE International Conference on Communications (ICC)*, pages 3423–3427. IEEE, 2012.

[10] Ya-Ping Zhang, Hao-Yu Li, and Hai-Peng Zhao. An optimized encoding algorithm for systematic polar codes. *EURASIP Journal on Wireless Communications and Networking*, 2019(1):1–11, July 2019.

[11] Ebyte IoT. What exactly is Polar code in 5G communication? *IoT Module Shop Manufacturer Factory Blog*, April 2024.

[12] I. Tal and A. Vardy, "List decoding of polar codes," *IEEE Transactions on Information Theory*, vol. 61, no. 5, pp. 2213–2226, May 2015, arXiv:1206.0050.

[13] S. A. Hashemi, C. Condo, and W. J. Gross, "Fast and flexible successive-cancellation list decoding for polar codes," *IEEE Transactions on Signal Processing*, vol. 65, no. 22, pp. 5756–5769, Nov. 2017, DOI:10.1109/TSP.2017.2740204.

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

[15] C. Leroux, I. Tal, A. Vardy, and W. J. Gross, "Hardware implementation of successive-cancellation decoders for polar codes," *Journal of Signal Processing Systems*, vol. 69, no. 3, pp. 305–315, Dec. 2012, DOI:10.1007/s11265-012-0688-6.

[16] R. S. Prayogo, T. Adiono, and S. Fuada, "Performance evaluation of polar codes for 5G mMTC applications," in *Proc. International Symposium on Electronics and Smart Devices (ISESD)*, Oct. 2018, pp. 1–6, DOI:10.1109/ISESD.2018.8605482.

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

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

[19] 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, Available at: http://www.journalmc.com/en/article/id/56c5cfad-6ec3-4616-9e9f-cf0bd12ca75c.

[20] C. Chen, B. Bai, X. Ma, and X. Wang, "Improved successive cancellation decoding of polar codes," *Electronics Letters*, vol. 48, no. 9, pp. 500–502, Apr. 2012, DOI:10.1049/el.2012.0234.

[21] Q. Zhang, A. Liu, X. Pan, and Y. Zhang, "Fast successive-cancellation list decoder for polar codes with early stopping criterion," in *Proc. IEEE International Conference on Integrated Circuits, Technologies and Applications (ICTA)*, Nov. 2018, pp. 1–2, DOI:10.1109/CICTA.2018.8705938.

[22] C. E. Shannon, "A Mathematical Theory of Communication," *Bell System Technical Journal*, vol. 27, pp. 379–423, 623–656, 1948.

[23] R. G. Gallager, "Low-Density Parity-Check Codes," *IRE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, January 1962.

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

[25] Wikipedia, "Uniform distribution (continuous)," https://en.wikipedia.org/wiki/Uniform_distribution_(continuous).

[26] Wikipedia, "Laplace distribution," https://en.wikipedia.org/wiki/Laplace_distribution.

[27] Wikipedia, "Normal distribution," https://en.wikipedia.org/wiki/Normal$_d$*istribution*.

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

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

[30] "Central Limit Theorem," Wikipedia, https://en.wikipedia.org/wiki/Central_limit_theorem.

Computing Reliability Sequence using Bhattacharyya Parameters

```
function [Z_list] = plot_bhattacharyya_parameters(N, epsilon)
    if mod(log2(N), 1) ~= 0
        error('N must be a power of 2.');
    end
    if epsilon < 0 || epsilon > 1
        error('epsilon must be between 0 and 1.');
    end

    Z_list = epsilon;
    n = log2(N);
    for i = 1:n
        new_Z_list = zeros(1, 2^i);
        for j = 1:length(Z_list)
            Z = Z_list(j);
            Z_minus = 2 * Z - Z^2;  % Z(W^-)
            Z_plus = Z^2;           % Z(W^+)
            new_Z_list(2*j - 1) = Z_minus;
            new_Z_list(2*j) = Z_plus;
        end
        Z_list = new_Z_list;
    end
end
```

```
clearvars;
close all;
clc;

p = 0.5;
N = 2:10;

for n = N
    N_val = 2^n;
    polarized_channels = plot_bhattacharyya_parameters(N_val, p);

    % Figure 1: Unsorted Bhattacharyya parameters
    figure;
    plot(0:N_val-1, polarized_channels, '.', 'LineWidth', 1.5);
    xlabel('Channel Index');
    ylabel('Bhattacharyya Parameter');
    title(['Bhattacharyya Parameters for N = ', num2str(N_val), ', \epsilon = ',
num2str(p)]);
    grid on;
    xlim([0, N_val-1]);

    % Compute reliability sequence
    [~, reliability_sequence] = sort(polarized_channels, 'descend');
```

```
    sorted_polarized_channels = sort(polarized_channels);
    reliability_sequence = reliability_sequence - 1;
    disp(['Reliability Sequence for N = ', num2str(N_val)]);
    disp(reliability_sequence);

    % Figure 2: Sorted Bhattacharyya parameters
    figure;
    plot(0:N_val-1, sorted_polarized_channels, '.', 'LineWidth', 1.5);
    xlabel('Sorted Channel Index');
    ylabel('Bhattacharyya Parameter (Sorted)');
    title(['Sorted Bhattacharyya Parameters for N = ', num2str(N_val), ', \epsilon
= ', num2str(p)]);
    grid on;
    xlim([0, N_val-1]);

    drawnow;
    pause(0.1);
end
```

Encoder

```
function [x] = polar_encode(N, K, reliability_sequence, message)
    % Input validation
    if mod(log2(N), 1) ~= 0
        error('N must be a power of 2.');
    end
    if K > N
        error('K cannot be greater than N.');
    end
    if length(reliability_sequence) ~= N
        error('Reliability sequence length must equal N.');
    end

    % Select K most reliable indices (last K in reliability sequence, 0-based

    % Generate polar code generator matrix G = F^(⊗n)
    F = [1 0; 1 1];
    n = log2(N);
    G = 1;
    for i = 1:n
        G = kron(G, F);
    end
    % Compute codeword x = u * G (mod 2)
    x = mod(message * G, 2);
end
```

BPSK Modulator and AWGN Channel

```
function y = bpsk_awgn_channel(code_word, Eb_No, K, N)
    if ~isvector(code_word) || length(code_word) ~= N
        error('code_word must be a 1x%d vector.', N);
    end
    if ~all(ismember(code_word, [0 1]))
        error('code_word must be binary (0 or 1).');
    end
    if K <= 0 || K > N
        error('K must be between 1 and N.');
    end

    s = 1 - 2 * code_word;
    Es_No = (K / N) * Eb_No;
    sigma = sqrt(1 / Es_No);
    AWGN = sigma * randn(1, N);
    y = s + AWGN;
end
```

Successive Cancellation Decoder

- **LLR Update Function For Left Child**
- **LLR Update Function For Right Child**
- **SC Decoder**

```
function L_left = f_function(L1, L2)
    L_left = sign(L1) .* sign(L2) .* min(abs(L1), abs(L2));
end
```

```
function L_right = g_function(L1, L2, u_left)
    L_right = L2 + (1 - 2*u_left) .* L1;
end
```

```
function sc_decode_awgn(message_bpsk,depth,N,Frozen_bits)
    global L;
    global u_hat;
    global nodes;

    node = nodes(depth);
    if depth == N+1
        nodes(depth) = nodes(depth) + 1;
        if any(Frozen_bits == (node+1))
            u_hat(N+1,node+1) = 0;
        else
            if L(N+1,node+1) >= 0
```

```
                u_hat(N+1,node+1) = 0;
            else
                u_hat(N+1,node+1) = 1;
            end
        end
        return;
    end
    temp=2^(N-depth+1);
    a = message_bpsk(1:temp/2);
    b = message_bpsk(temp/2+1:end);
    L(depth+1,temp/2*node+1:temp/2*(node+1)) = f_function(a,b);

    new_set = f_function(a,b);
    nodes(depth) = nodes(depth) + 1;

    sc_decode_awgn(new_set, depth+1, N, Frozen_bits);

    uhat_left = u_hat(depth+1, temp/2*node+1:temp/2*(node+1));
    node = nodes(depth);
    L(depth+1, temp/2*node+1:temp/2*(node+1)) = g_function(a,b,uhat_left);
    nodes(depth) = nodes(depth) + 1;

    new_set_2 = g_function(a,b,uhat_left);

    sc_decode_awgn(new_set_2,depth+1,N,Frozen_bits);

    uhat_right = u_hat(depth+1,temp/2*node+1:temp/2*(node+1));
    uhat_left = u_hat(depth+1,(((temp/2)*node)-(temp/2)+1):temp/2*(node));

    u_hat(depth, (((temp/2)*node)-(temp/2)+1):temp/2*(node+1)) =[mod(uhat_left +
uhat_right, 2) uhat_right];

end
```

Plots Of Probability of Success, BLER Rate, BER Rate And Shannon Limit For Specififc Code Rate (N,K) With SC Decoder

```
clearvars;
close all;
clc;
tic
load('Reliability_Sequence.mat');
p = 0.5;  % Erasure probability for BEC
N_val = 512;
N = log2(N_val); % Exponents for N = 2^n
K = 128;
R = K/N_val;
Sequence = Reliability_Sequence(Reliability_Sequence <= N_val);
```

```
Frozen_bits = Sequence(1:N_val-K);

N_sim = 1000;
Eb_No_dB = 0:0.5:10;
Eb_No_Range = 10.^(Eb_No_dB/10);
P_Succ = zeros(1, length(Eb_No_Range));
Bit_Error_Rate = zeros(1, length(Eb_No_Range));
Block_Error_Rate = zeros(1, length(Eb_No_Range));
Iteration_index = 1;
theoretical = qfunc(sqrt(R * Eb_No_Range));
for Eb_No = Eb_No_Range
    Successful_Decoding = 0;
    Nsim_block_errors = 0;
    Nsim_bit_errors = 0;
    for nblock = 1:N_sim
        info_bits = randi([0 1], 1, K);
        message = zeros(1, N_val);
        message(Sequence(N_val-K+1:end)) = info_bits;

        %Encoding
        encodded_msg = polar_encode(N_val, K, Sequence, message);
        message_bpsk = bpsk_awgn_channel(encodded_msg,Eb_No,K,N_val);
        global L;
        global u_hat;
        global nodes;

        L = zeros(N+1,N_val);%Belifs
        L(1,:) = message_bpsk;%Belif of root
        u_hat = zeros(N+1,N_val);
        nodes = zeros(N+1,1);

        sc_decode_awgn(message_bpsk, 1, N, Frozen_bits);
        message_cap = u_hat(N+1, Sequence(N_val-K+1:end));

        Evident_errors = sum(info_bits ~= message_cap);
        if(Evident_errors == 0)
                Successful_Decoding = Successful_Decoding + 1;
        else
                Nsim_bit_errors = Nsim_bit_errors + Evident_errors;
                Nsim_block_errors = Nsim_block_errors + 1;
        end
    end
    P_Succ(Iteration_index) = (1/N_sim)*Successful_Decoding;
    Bit_Error_Rate(Iteration_index) = (Nsim_bit_errors/K)/N_sim;
    Block_Error_Rate(Iteration_index) = Nsim_block_errors/N_sim;
    Iteration_index = Iteration_index + 1;

end
```

```
% Plot Probability of Success
figure;
plot(Eb_No_dB, P_Succ, 'bo-', 'LineWidth', 2);
xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex');
ylabel('Probability of Success', 'Interpreter', 'latex');
title(['Success Probability vs $E_b/N_0$' newline '(N = ', num2str(N_val), ', K =
', num2str(K),', Code Rate = ', num2str(R, '%.2f'),', $N_{sim}$ = ',num2str(N_sim),
')'],'Interpreter', 'latex');
legend('Success Rate Simulated');
grid on;

% Plot Bit Error Rate
figure;
plot(Eb_No_dB, Bit_Error_Rate, 'bo-', 'LineWidth', 2);
hold on;

plot(Eb_No_dB, theoretical, 'sr-', 'LineWidth', 2);

xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex');
ylabel('Bit Error Rate (BER)', 'Interpreter', 'latex');
title(['Bit Error Rate vs $E_b/N_0$ (SC)' newline '(N = ', num2str(N_val),
', K = ', num2str(K), ', Code Rate = ', num2str(R, '%.2f'), ', $N_{sim}$ =
',num2str(N_sim),')'],'Interpreter', 'latex');

grid on;
legend('Bit Error Rate Simulated', 'Location', 'Best', 'Interpreter', 'latex');  %
Legend for both plots

% Plot Block Error Rate
figure;
plot(Eb_No_dB, Block_Error_Rate, 'bo-', 'LineWidth', 2);
hold on;

xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex');
ylabel('Block Error Rate (BER)', 'Interpreter', 'latex');
title(['Block Error Rate vs $E_b/N_0$ (SC)' newline '(N = ', num2str(N_val),
', K = ', num2str(K), ', Code Rate = ', num2str(R, '%.2f'), ', $N_{sim}$ =
',num2str(N_sim),')'],'Interpreter', 'latex');

grid on;
legend('Block Error Rate Simulated', 'Location', 'Best', 'Interpreter', 'latex');
% Legend for both plots


% Performance Check usng shannon limit
figure;
semilogy(Eb_No_dB, Block_Error_Rate, 'bo-', 'LineWidth', 2, 'MarkerFaceColor',
'b'); hold on;
r = 1/4;
K_s = 128;
```

```
N_s = K_s / r;

Pe_NA = zeros(size(Eb_No_dB));

for i = 1:length(Eb_No_dB)
    P = r * Eb_No_Range(i);
    C = log2(1 + P);
    V = (log2(exp(1)))^2 * (P * (P + 2)) / (2 * (P + 1)^2);
    term = sqrt(N/V) * (C - r + log2(N) / (2 * N));
    Pe_NA(i) = qfunc(term);
end

% Shannon Limit in dB
shannon_limit_dB = 10 * log10((2^r - 1)/r);

% Plot
semilogy(Eb_No_dB, Pe_NA, 'b-', 'LineWidth', 2); hold on;
xline(shannon_limit_dB, 'r--', 'LineWidth', 2, ...
    'Label', sprintf('Shannon Limit (%.2f dB)',
shannon_limit_dB),'LabelVerticalAlignment', 'bottom');
grid on;
xlabel('Eb/N0 (dB)','Interpreter','latex');
ylabel('Block Error Probability','Interpreter','latex');
title('Performance check for Polar codes with Normal Approximation (Rate = 1/4, N =
512 , K = 126)','Interpreter','latex');
legend('SC-BLER','Normal Approximation', 'Shannon Limit', 'Location', 'SouthWest');

xlim([-2,6]);
ylim([10e-3,10e-1]);
grid on;

toc
```

Comparison of SC Decoder For Different (N,K) Polar Codes with Rate = 0.5

```
clearvars;
close all;
clc;
tic

load('Reliability_Sequence.mat');
p = 0.5;  % Erasure probability for BEC

Eb_No_dB = 0:0.5:10;
Eb_No_Range = 10.^(Eb_No_dB/10);
N_sim = 10000;  % reduce to make simulations faster

% Store plots
figure_success = figure;
```

```
hold on;
figure_BER = figure;
hold on;
figure_BLER = figure;
hold on;

colors = lines(10); % distinct colors for plotting

% Loop over different N values
for expN = 1:1:10
    N_val = 2^expN;
    K = N_val/2;  % Set K to keep Code Rate ~ 0.5
    R = K / N_val;
    Sequence = Reliability_Sequence(Reliability_Sequence <= N_val);
    Frozen_bits = Sequence(1:N_val-K);

    P_Succ = zeros(1, length(Eb_No_Range));
    Bit_Error_Rate = zeros(1, length(Eb_No_Range));
    Block_Error_Rate = zeros(1, length(Eb_No_Range));
    theoretical = qfunc(sqrt(R * Eb_No_Range));
    Iteration_index = 1;

    for Eb_No = Eb_No_Range
        Successful_Decoding = 0;
        Nsim_block_errors = 0;
        Nsim_bit_errors = 0;

        for nblock = 1:N_sim
            info_bits = randi([0 1], 1, K);
            message = zeros(1, N_val);
            message(Sequence(N_val-K+1:end)) = info_bits;

            encodded_msg = polar_encode(N_val, K, Sequence, message);
            message_bpsk = bpsk_awgn_channel(encodded_msg,Eb_No,K,N_val);

            global L u_hat nodes;
            L = zeros(expN+1, N_val);
            L(1,:) = message_bpsk;
            u_hat = zeros(expN+1, N_val);
            nodes = zeros(expN+1, 1);

            sc_decode_awgn(message_bpsk, 1, expN, Frozen_bits);
            message_cap = u_hat(expN+1, Sequence(N_val-K+1:end));

            Evident_errors = sum(info_bits ~= message_cap);
            if Evident_errors == 0
                Successful_Decoding = Successful_Decoding + 1;
            else
                Nsim_bit_errors = Nsim_bit_errors + Evident_errors;
                Nsim_block_errors = Nsim_block_errors + 1;
```

```
            end
        end

        P_Succ(Iteration_index) = Successful_Decoding / N_sim;
        Bit_Error_Rate(Iteration_index) = Nsim_bit_errors / (K * N_sim);
        Block_Error_Rate(Iteration_index) = Nsim_block_errors / N_sim;
        Iteration_index = Iteration_index + 1;
    end

    % Plot all results
    label_str = sprintf('N=%d, K=%d', N_val, K);

    % --- Plot: Success Rate ---
    figure(figure_success);
    semilogy(Eb_No_dB, P_Succ, 'o-', 'Color', colors(expN-3,:), 'DisplayName',
label_str, 'LineWidth', 1.5);

    % --- Plot: BER ---
    figure(figure_BER);
    semilogy(Eb_No_dB, Bit_Error_Rate, 'o-', 'Color', colors(expN-3,:),
'DisplayName', label_str, 'LineWidth', 1.5);

    % Optional: Add theoretical uncoded BER
    if expN == 4  % Only once
        plot(Eb_No_dB, qfunc(sqrt(R * Eb_No_Range)), 'k--', 'DisplayName', 'Uncoded
AWGN');
    end

    % --- Plot: BLER ---
    figure(figure_BLER);
    semilogy(Eb_No_dB, Block_Error_Rate, 'o-', 'Color', colors(expN-3,:),
'DisplayName', label_str, 'LineWidth', 1.5);
end

% Finalize Success Plot
figure(figure_success);
xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex');
ylabel('Success Probability', 'Interpreter', 'latex');
title('Success Probability vs $E_b/N_0$ for Different (N,K) Polar Codes with Rate =
0.5 (SC)', 'Interpreter', 'latex');
legend('show', 'Location', 'southEast');
grid on;

% Finalize BER Plot
figure(figure_BER);
xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex');
ylabel('Bit Error Rate (BER)', 'Interpreter', 'latex');
title('Bit Error Rate vs $E_b/N_0$ for Different (N,K) Polar Codes with Rate = 0.5
(SC)', 'Interpreter', 'latex');
legend('show', 'Location', 'southWest');
```

```
grid on;

% Finalize BLER Plot
figure(figure_BLER);
xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex');
ylabel('Block Error Rate (BLER)', 'Interpreter', 'latex');
title('Block Error Rate vs $E_b/N_0$ for Different (N,K) Polar Codes with Rate =
0.5 (SC)', 'Interpreter', 'latex');
legend('show', 'Location', 'southWest');
grid on;

toc
```

Successive Cancellation List Decoder With Cyclic Redundancy Check

- **Select Best Path Algorithm**
- **CRC Aided SCL Decoder**

```
function [y, pos] = select_best_paths(x, k)
    [xs, I] = sort(x);
    y = xs(1:k);
    pos = I(1:k);
end
```

```
function CA_SCL_decode(Depth, Node, n, Frozen_bits)
    global L;
    global u_hat;
    global Path_matrix;
    global List_size;
    global Node_states;

    % disp(length(u_hat));
    Npos = (2^Depth-1) + Node + 1;
    if Depth == n
       Node_states(Npos) = 3;
       Decision_matrix = squeeze(L(:, Depth+1, Node+1));
       if any(Frozen_bits == (Node+1))
            u_hat(:, Depth+1, Node+1) = 0;
            Path_matrix = Path_matrix + abs(Decision_matrix).*(Decision_matrix < 0);
       else
            Decisions = Decision_matrix < 0;
            Path_matrix_2 = [Path_matrix; Path_matrix + abs(Decision_matrix)];

            [Path_matrix, Position] = select_best_paths(Path_matrix_2, List_size);

            Surviving_Position = Position > List_size;
```

```
            Position(Surviving_Position) = Position(Surviving_Position) - List_size;
            Decisions = Decisions(Position);
            Decisions(Surviving_Position) = 1 - Decisions(Surviving_Position);

            L = L(Position, :, :);
            u_hat = u_hat(Position, :, :);
            u_hat(:, Depth+1, Node+1) = Decisions;
        end
        return;
    end
    temp = 2^(n-Depth);
    received_bits = squeeze(L(:, Depth+1, temp*Node+1:temp*(Node+1)));
    if(Node_states(Npos) == 0)
        Node_states(Npos) = 1;
        a = received_bits(:, 1:temp/2);
        b = received_bits(:, temp/2+1:end);
        L(:, Depth+2, (temp/2)*(2*Node)+1:(temp/2)*((2*Node)+1)) = f_function(a,b);
        CA_SCL_decode(Depth+1, 2*Node, n, Frozen_bits);
    end

    received_bits = squeeze(L(:, Depth+1, temp*Node+1:temp*(Node+1)));
    if(Node_states(Npos) == 1)

        Node_states(Npos) = 2;

        a = received_bits(:, 1:temp/2);
        b = received_bits(:, (temp/2)+1:end);
        Left_u_hat = squeeze(u_hat(:, Depth+2, (temp/2)*(2*Node)+1:(temp/
2)*((2*Node)+1)));

        L(:, Depth+2, (temp/2)*(2*Node+1)+1:(temp/2)*((2*Node+1)+1)) =
g_function(a, b, Left_u_hat);

        CA_SCL_decode(Depth+1, 2*Node+1, n, Frozen_bits);
    end

    if(Node_states(Npos) == 2)

        Node_states(Npos) = 3;
        Left_u_hat = squeeze(u_hat(:, Depth+2, (temp/2)*(2*Node)+1:(temp/
2)*((2*Node)+1)));
        Right_u_hat = squeeze(u_hat(:, Depth+2, (temp/2)*(2*Node+1)+1:(temp/
2)*((2*Node+1)+1)));

        u_hat(:, Depth+1, temp*Node+1:temp*(Node+1)) = [mod(Left_u_hat +
Right_u_hat, 2) Right_u_hat];
    end
end
```

Plots Of Probability of Success, BLER Rate, BER Rate And Shannon Limit For Specififc Code Rate (N,K) With CRC Aided SCL Decoder

```
clearvars;
close all;
clc;
load('Reliability_Sequence.mat');
N = 512;
n = log2(N);
A = 124;
CRC_Length = 4;
CRC_Polynomial = fliplr([1 0 0 1 1]);
K = A + CRC_Length;
Rate = A/N;

global List_size;
List_size = 8;

Sequence = Reliability_Sequence(Reliability_Sequence <= N);
Frozen_bits = Sequence(1:N-K);

Nsim = 1000;
Eb_No_dB = -2:0.5:6;
Eb_No_Range = 10.^(Eb_No_dB/10);
P_succ = zeros(1, length(Eb_No_dB));
Bit_Error_Rate = zeros(1, length(Eb_No_dB));
Block_Errors_Rate = zeros(1, length(Eb_No_dB));
Iteration_index = 1;
for Eb_No = Eb_No_Range
    Successful_Decoding = 0;
    Nsim_block_errors = 0;
    Nsim_bit_errors = 0;

    for block = 1:Nsim
        info_bits = randi([0 1], 1, A);
        [quotient, remainder] = gfdeconv([zeros(1, CRC_Length) fliplr(info_bits)],
CRC_Polynomial);
        CRC_padded_message = [info_bits fliplr([remainder zeros(1, CRC_Length -
length(remainder))])];
        message = zeros(1, N);
        message(Sequence(N-K+1:end)) = CRC_padded_message;

        code_word = polar_encode(N, n, Sequence, message);
        code_word = bpsk_awgn_channel(code_word,Eb_No,A,N);

        global L;
        global u_hat;
        global Path_matrix;
```

```
        global Node_states;

        L = zeros(List_size, n+1, N);
        L(:,1,:) = repmat(code_word, List_size, 1, 1);
        u_hat = zeros(List_size, n+1, N);
        Node_states = zeros(1, 2*N-1);
        Path_matrix = Inf*ones(List_size, 1);
        Path_matrix(1) = 0;

        CA_SCL_decode(0, 0, n, Frozen_bits);

        message_cap_temp = squeeze(u_hat(:,n+1, Sequence(N-K+1:end)));
        selected_code_word = 1;
        for count = 1:List_size
             [new_quotient, new_remainder] =
gfdeconv(fliplr(message_cap_temp(count,:)), CRC_Polynomial);
             if (isequal(new_remainder, 0))
                 selected_code_word = count;
                 break;
             end
        end
        message_cap = message_cap_temp(selected_code_word, 1:A);
        count = 0;
        for i = 1:length(message_cap)
            if message_cap(i) ~= info_bits(i)
                Nsim_bit_errors = Nsim_bit_errors + 1;      % bit error count
                count = 1;               % mark block error
            end
        end
        Nsim_block_errors = Nsim_block_errors + count;
    end
    P_succ(Iteration_index) = 1 - (1/Nsim)*Nsim_block_errors;
    Bit_Error_Rate(Iteration_index) = Nsim_bit_errors/(A*Nsim);
    Block_Errors_Rate(Iteration_index) = Nsim_block_errors/Nsim;
    Iteration_index = Iteration_index + 1;
end
```

```
%% Plot Success Probability (Psucc)
figure;
plot(Eb_No_dB, P_succ, 'o-', 'LineWidth', 2, 'Color', [0 0.4470 0.7410]);
xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex', 'FontSize', 12);
ylabel('Success Probability', 'Interpreter', 'latex', 'FontSize', 12);
title(sprintf('Success Probability vs $E_b/N_0$ for Polar Code (N=%d, K=%d) with
CRC-SCL, $N_{sim}$=%d', N, K, Nsim), ...
      'Interpreter', 'latex', 'FontSize', 13);
grid on;

%% Plot Block Error Rate (BLER)
figure;
```

```
semilogy(Eb_No_dB, Block_Errors_Rate, 'o-', 'LineWidth', 2, 'Color', [0.8500 0.3250
0.0980]);
xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex', 'FontSize', 12);
ylabel('Block Error Rate (BLER)', 'Interpreter', 'latex', 'FontSize', 12);
title(sprintf('Block Error Rate vs $E_b/N_0$ for Polar Code (N=%d, K=%d) with CRC-
SCL, $N_{sim}$=%d', N, K, Nsim),'Interpreter', 'latex', 'FontSize', 13);
grid on;
set(gca, 'YScale', 'log');

%% Plot Bit Error Rate (BER)
figure;
semilogy(Eb_No_dB, Bit_Error_Rate, 'o-.', 'LineWidth', 2, 'Color', [0.4660 0.6740
0.1880]);
xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex', 'FontSize', 12);
ylabel('Bit Error Rate (BER)', 'Interpreter', 'latex', 'FontSize', 12);
title(sprintf('Bit Error Rate vs $E_b/N_0$ for Polar Code (N=%d, K=%d) ,Rate = %.2f
with CRC-SCL, $N_{sim}$=%d', N, K,Rate,Nsim),'Interpreter', 'latex', 'FontSize',
13);
grid on;
set(gca, 'YScale', 'log');

%% Performance Check with Shannon Limit
figure;
semilogy(Eb_No_dB, Block_Errors_Rate, 'o-.', 'LineWidth', 2, 'Color', [0.4660
0.6740 0.1880]);hold on;
r = 1/4;
K_s = 128;
N_s = K_s / r;

Pe_NA = zeros(size(Eb_No_dB));

for i = 1:length(Eb_No_dB)
    P = r * Eb_No_Range(i);
    C = log2(1 + P);
    V = (log2(exp(1)))^2 * (P * (P + 2)) / (2 * (P + 1)^2);
    term = sqrt(N/V) * (C - r + log2(N) / (2 * N));
    Pe_NA(i) = qfunc(term);
end

% Shannon Limit in dB
shannon_limit_dB = 10 * log10((2^r - 1)/r);

semilogy(Eb_No_dB, Pe_NA, 'b-', 'LineWidth', 2); hold on;
xline(shannon_limit_dB, 'r--', 'LineWidth', 2, ...
    'Label', sprintf('Shannon Limit (%.2f dB)',
shannon_limit_dB),'LabelVerticalAlignment', 'bottom');
ylim([10e-3,10e-1]);
xlabel('Eb/N0 (dB)','Interpreter','latex');
ylabel('Block Error Probability','Interpreter','latex');
```

```
title('Performance check for Polar codes with Normal Approximation (Rate = 1/4, N =
512 , K = 126)','Interpreter','latex');
legend('SC-BLER','Normal Approximation', 'Shannon Limit', 'Location', 'SouthWest');
grid on;
```

Comparison of CA-SCL Decoder For Different (N,K) Polar Codes with Rate = 0.5 (List Size = 8)

```
clearvars;
close all;
clc;

tic
load('Reliability_Sequence.mat');
N = [128 256 512 1024];
size = length(N);
A = [57 120 246 501];
CRC_Length = [7 8 10 11];
CRC_Polynomial_Storage = {[1 0 1 1 0 1 0], [1 1 1 1 0 0 0 0], [1 1 0 0 1 0 1 1 0 1
0], [1 1 1 0 0 0 1 0 0 0 0 1]};
K = A + CRC_Length;
Rate = A./N;

global List_size;
List_size = 8;

Nsim = 1000;
Eb_No_Range = 0:0.5:10;
P_succ = zeros(size, length(Eb_No_Range));
Bit_Error_Rate = zeros(size, length(Eb_No_Range));
Block_Errors_Rate = zeros(size, length(Eb_No_Range));

for N_index = 1:size
    CRC_Polynomial = fliplr(CRC_Polynomial_Storage(N_index));
    Sequence = Reliability_Sequence(Reliability_Sequence <= N(N_index));
    Frozen_bits = Sequence(1:N(N_index)-K(N_index));
    Iteration_index = 1;
    n = log2(N(N_index));

    for Eb_No = Eb_No_Range
        Successful_Decoding = 0;
        Nsim_block_errors = 0;
        Nsim_bit_errors = 0;

        for block = 1:Nsim
            info_bits = randi([0 1], 1, A(N_index));
            [quotient, remainder] = gfdeconv([zeros(1, CRC_Length(N_index))
fliplr(info_bits)], cell2mat(CRC_Polynomial));
```

```
            CRC_padded_message = [info_bits fliplr([remainder zeros(1,
CRC_Length(N_index) - length(remainder))])];
            message = zeros(1, N(N_index));
            message(Sequence(N(N_index)-K(N_index)+1:end)) = CRC_padded_message;

            code_word = polar_encode(N(N_index),n,Sequence,message);
            Eb_No_l = 10^(Eb_No/10);
            code_word = bpsk_awgn_channel(code_word, Eb_No_l, A(N_index),
N(N_index));

            global L;
            global u_hat;
            global Path_matrix;
            global Node_states;

            L = zeros(List_size, n+1, N(N_index));
            L(:,1,:) = repmat(code_word, List_size, 1, 1);
            u_hat = zeros(List_size, n+1, N(N_index));
            Node_states = zeros(1, 2*(N(N_index))-1);
            Path_matrix = Inf*ones(List_size, 1);
            Path_matrix(1) = 0;

            CA_SCL_decode(0, 0, n, Frozen_bits);
            message_cap_temp = squeeze(u_hat(:,n+1, Sequence(N(N_index)-K(N_index)
+1:end)));
            selected_code_word = 1;
            for count = 1:List_size
                [new_quotient, new_remainder] =
gfdeconv(fliplr(message_cap_temp(count,:)), cell2mat(CRC_Polynomial));
                if (isequal(new_remainder, 0))
                    selected_code_word = count;
                    break;
                end
            end
            message_cap = message_cap_temp(selected_code_word, 1:A(N_index));
            count = 0;
            for i = 1:length(message_cap)
                if message_cap(i) ~= info_bits(i)
                   Nsim_bit_errors = Nsim_bit_errors + 1;      % bit error count
                   count = 1;               % mark block error
                end
            end
            Nsim_block_errors = Nsim_block_errors + count;
        end

        P_succ(N_index, Iteration_index) = 1 - (1/Nsim)*Nsim_block_errors;
        Bit_Error_Rate(N_index, Iteration_index) = Nsim_bit_errors/A(N_index)/Nsim;
        Block_Errors_Rate(N_index, Iteration_index) = Nsim_block_errors/Nsim;
        Iteration_index = Iteration_index + 1;
    end
```

```
end
```

```
semilogy(Eb_No_Range, Bit_Error_Rate, 'LineWidth', 2);
title('BER Performance vs Eb/No for Various (N,K)','Interpreter','latex');
xlabel('Eb/No (dB)','Interpreter','latex');
ylabel('Bit Error Rate (BER)','Interpreter','Latex');
legend('Code Rate: 1/2, (N,K) = (128,64)', ...
       'Code Rate: 1/2,(N,K) = (256,128)', ...
       'Code Rate: 1/2, (N,K) = (512,256)', ...
       'Code Rate: 1/2, (N,K) = (1024,512)', ...
       'Location', 'southwest');
grid on;
```

```
semilogy(Eb_No_Range, Block_Errors_Rate, 'LineWidth', 2);
title('BLER Performance vs Eb/No for Various (N,K)','Interpreter','latex');
xlabel('Eb/No (dB)','Interpreter','latex');
ylabel('Block Error Rate (BLER)','Interpreter','Latex');
legend('Code Rate: 1/2, (N,K) = (128,64)', ...
       'Code Rate: 1/2,(N,K) = (256,128)', ...
       'Code Rate: 1/2, (N,K) = (512,256)', ...
       'Code Rate: 1/2, (N,K) = (1024,512)', ...
       'Location', 'southwest');
grid on;
```

```
plot(Eb_No_Range, P_succ, 'LineWidth', 2);
title('Success Rate Performance vs Eb/No for Various (N,K)','Interpreter','latex');
xlabel('Eb/No (dB)','Interpreter','latex');
ylabel('Probability Of Success','Interpreter','Latex');
legend('Code Rate: 1/2, (N,K) = (128,64)', ...
       'Code Rate: 1/2,(N,K) = (256,128)', ...
       'Code Rate: 1/2, (N,K) = (512,256)', ...
       'Code Rate: 1/2, (N,K) = (1024,512)', ...
       'Location', 'southwest');
grid on;
```

Comparison of CRC-SCL Decoder Vs SC Decoder

```
%% SC Decoder
clearvars;
close all;
clc;
tic
load('Reliability_Sequence.mat');
p = 0.5;  % Erasure probability for BEC
N_val = 1024;
N = log2(N_val); % Exponents for N = 2^n
```

```
K = 508;
R = K/N_val;
Sequence = Reliability_Sequence(Reliability_Sequence <= N_val);
Frozen_bits = Sequence(1:N_val-K);

N_sim = 10;
Eb_No_dB = 0:0.5:10;
Eb_No_Range = 10.^(Eb_No_dB/10);
P_Succ_SC = zeros(1, length(Eb_No_Range));
Bit_Error_Rate_SC = zeros(1, length(Eb_No_Range));
Block_Error_Rate_SC = zeros(1, length(Eb_No_Range));
Iteration_index = 1;
for Eb_No = Eb_No_Range
    Successful_Decoding = 0;
    Nsim_block_errors = 0;
    Nsim_bit_errors = 0;
    for nblock = 1:N_sim
        info_bits = randi([0 1], 1, K);
        message = zeros(1, N_val);
        message(Sequence(N_val-K+1:end)) = info_bits;

        %Encoding
        encodded_msg = polar_encode(N_val, K, Sequence, message);
        message_bpsk = bpsk_awgn_channel(encodded_msg,Eb_No,K,N_val);
        global L;
        global u_hat;
        global nodes;

        L = zeros(N+1,N_val);%Belifs
        L(1,:) = message_bpsk;%Belif of root
        u_hat = zeros(N+1,N_val);
        nodes = zeros(N+1,1);

        %Successive Cancellation Decoder
        sc_decode_awgn(message_bpsk, 1, N, Frozen_bits);
        message_cap = u_hat(N+1, Sequence(N_val-K+1:end));

        Evident_errors = sum(info_bits ~= message_cap);
        if(Evident_errors == 0)
                Successful_Decoding = Successful_Decoding + 1;
        else
                Nsim_bit_errors = Nsim_bit_errors + Evident_errors;
                Nsim_block_errors = Nsim_block_errors + 1;
        end
    end
    P_Succ_SC(Iteration_index) = (1/N_sim)*Successful_Decoding;
    Bit_Error_Rate_SC(Iteration_index) = (Nsim_bit_errors/K)/N_sim;
    Block_Error_Rate_SC(Iteration_index) = Nsim_block_errors/N_sim;
    Iteration_index = Iteration_index + 1;
end
```

```
%% CRC-SCL Decoder
load('Reliability_Sequence.mat');
N = 1024;
n = log2(N);
A = 508;
CRC_Length = 4;
CRC_Polynomial = fliplr([1 0 0 1 1]);
K = A + CRC_Length;
Rate = A/N;

global List_size;
List_size = 4;

Sequence = Reliability_Sequence(Reliability_Sequence <= N);
Frozen_bits = Sequence(1:N-K);

Nsim = 10;
Eb_No_dB = 0:0.5:10;
Eb_No_Range = 10.^(Eb_No_dB/10);
P_succ_SCL = zeros(1, length(Eb_No_dB));
Bit_Error_Rate_SCL = zeros(1, length(Eb_No_dB));
Block_Error_Rate_SCL = zeros(1, length(Eb_No_dB));


%Successive Cancellation List Decoder
Iteration_index = 1;
for Eb_No = Eb_No_Range
    Successful_Decoding = 0;
    Nsim_block_errors = 0;
    Nsim_bit_errors = 0;

    for block = 1:Nsim
        info_bits = randi([0 1], 1, A);
        [quotient, remainder] = gfdeconv([zeros(1, CRC_Length) fliplr(info_bits)],
CRC_Polynomial);
        CRC_padded_message = [info_bits fliplr([remainder zeros(1, CRC_Length -
length(remainder))])];
        message = zeros(1, N);
        message(Sequence(N-K+1:end)) = CRC_padded_message;

        code_word = polar_encode(N, n, Sequence, message);
        code_word = bpsk_awgn_channel(code_word,Eb_No,A,N);
        global L
        global u_hat;
        global Path_matrix;
        global Node_states;

        L = zeros(List_size, n+1, N);
        L(:,1,:) = repmat(code_word, List_size, 1, 1);
```

```
        u_hat = zeros(List_size, n+1, N);
        Node_states = zeros(1, 2*N-1);
        Path_matrix = Inf*ones(List_size, 1);
        Path_matrix(1) = 0;

        CA_SCL_decode(0, 0, n, Frozen_bits);

        message_cap_temp = squeeze(u_hat(:,n+1, Sequence(N-K+1:end)));
        selected_code_word = 1;
        for count = 1:List_size
             [new_quotient, new_remainder] =
gfdeconv(fliplr(message_cap_temp(count,:)), CRC_Polynomial);
             if (isequal(new_remainder, 0))
                 selected_code_word = count;
                 break;
             end
        end
        message_cap = message_cap_temp(selected_code_word, 1:A);
        count = 0;
        for i = 1:length(message_cap)
            if message_cap(i) ~= info_bits(i)
                Nsim_bit_errors = Nsim_bit_errors + 1;      % bit error count
                count = 1;               % mark block error
            end
        end
        Nsim_block_errors = Nsim_block_errors + count;
    end
    P_succ_SCL(Iteration_index) = 1 - (1/Nsim)*Nsim_block_errors;
    Bit_Error_Rate_SCL(Iteration_index) = Nsim_bit_errors/(A*Nsim);
    Block_Error_Rate_SCL(Iteration_index) = Nsim_block_errors/Nsim;
    Iteration_index = Iteration_index + 1;
end
```

```
% Plot Probability of Success
figure;
plot(Eb_No_dB, P_Succ_SC, 'bo-', 'LineWidth', 2,'MarkerFaceColor',
'b','MarkerSize',5);hold on;
plot(Eb_No_dB, P_succ_SCL, 'ro-', 'LineWidth', 2,'MarkerFaceColor',
'r','MarkerSize',5);
xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex');
ylabel('Probability of Success', 'Interpreter', 'latex');
title(['Success Probability vs $E_b/N_0$(SC vs SCL)' newline '(N = ',
num2str(N_val), ', K = ', num2str(K),', Code Rate = ', num2str(R, '%.2f'),',
$N_{sim}$ = ',num2str(N_sim), ')'],'Interpreter', 'latex');
legend('Success Rate Simulated(SC)','Success Rate Simulated(SCL)');
grid on;

% Plot Bit Error Rate
```

```
figure;
semilogy(Eb_No_dB, Bit_Error_Rate_SC, 'bo-', 'LineWidth', 2,'MarkerFaceColor',
'r','MarkerSize',5);hold on;
semilogy(Eb_No_dB, Bit_Error_Rate_SCL, 'ro-', 'LineWidth', 2,'MarkerFaceColor',
'r','MarkerSize',5);
xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex');
ylabel('Bit Error Rate (BER)', 'Interpreter', 'latex');
title(['Bit Error Rate vs $E_b/N_0$ (SC vs SCL)' newline '(N = ', num2str(N_val),
', K = ', num2str(K), ', Code Rate = ', num2str(R, '%.2f'), ', $N_{sim}$ =
',num2str(N_sim),')'],'Interpreter', 'latex');
legend('Bit Error Rate Simulated(SC)','Bit Error Rate Simulated(SCL)'),grid on;

figure;
semilogy(Eb_No_dB, Block_Error_Rate_SC, 'bo-', 'LineWidth', 2, 'MarkerFaceColor',
'b','MarkerSize',5); hold on;
semilogy(Eb_No_dB, Block_Error_Rate_SCL, 'ro-', 'LineWidth', 2, 'MarkerFaceColor',
'r','MarkerSize',5);
xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex');
ylabel('Block Error Rate (BLER)', 'Interpreter', 'latex');
title(['Block Error Rate vs $E_b/N_0$ (SC vs SCL)' newline '(N = ', num2str(N_val),
', K = ', num2str(K),', Code Rate = ', num2str(R, '%.2f'), ', $N_{sim}$ =
',num2str(N_sim), ')'],'Interpreter', 'latex');
legend('Block Error Rate Simulated(SC)','Block Error Rate Simulated(SCL)'),
grid on;
toc
```

Random Noise Channel

```
function y = bpsk_mixed_channel(code_word, Eb_No, K, N)
    % Check inputs
    if ~isvector(code_word) || length(code_word) ~= N
        error('code_word must be a 1x%d vector.', N);
    end
    if ~all(ismember(code_word, [0 1]))
        error('code_word must be binary (0 or 1).');
    end
    if K <= 0 || K > N
        error('K must be between 1 and N.');
    end

    % Compute signal
    s = 1 - 2 * code_word;

    % Compute Es_No and sigma
    R = K / N;
    Es_No = R * Eb_No;
    sigma = sqrt(1 / Es_No);
    sigma2 = sigma^2;
```

```
    % Variance for each noise source
    v = sigma2 / 3;

    % Generate noise sources
    % Gaussian noise
    n1 = sqrt(v) * randn(1, N);

    % Laplace noise
    b = sqrt(v / 2);
    U = rand(1, N);
    n2 = b * (log(2 * U) .* (U < 0.5) - log(2 * (1 - U)) .* (U >= 0.5));

    % Uniform noise
    a = sqrt(3 * v);
    n3 = a * (2 * rand(1, N) - 1);

    % Total noise
    n = n1 + n2 + n3;

    % Received signal
    y = s + n;
end
```

Comparison of Random Noise Channel and AWGN Channel With CRC-SCL Decoder

```
%% CRC-SCL Decoder for AWGN Channel
clearvars;
close all;
clc;
tic
load('Reliability_Sequence.mat');
N = 1024;
n = log2(N);
A = 508;
CRC_Length = 4;
CRC_Polynomial = fliplr([1 0 0 1 1]);
K = A + CRC_Length;
Rate = A/N;

global List_size;
List_size = 4;

Sequence = Reliability_Sequence(Reliability_Sequence <= N);
Frozen_bits = Sequence(1:N-K);

Nsim = 100;
Eb_No_dB = 0:0.5:10;
```

```
Eb_No_Range = 10.^(Eb_No_dB/10);
P_succ_AWGN = zeros(1, length(Eb_No_dB));
Bit_Error_Rate_AWGN = zeros(1, length(Eb_No_dB));
Block_Error_Rate_AWGN = zeros(1, length(Eb_No_dB));

%Successive Cancellation List Decoder with CRC Check
Iteration_index = 1;
for Eb_No = Eb_No_Range
    Successful_Decoding = 0;
    Nsim_block_errors = 0;
    Nsim_bit_errors = 0;

    for block = 1:Nsim
        info_bits = randi([0 1], 1, A);
        [quotient, remainder] = gfdeconv([zeros(1, CRC_Length) fliplr(info_bits)],
CRC_Polynomial);
        CRC_padded_message = [info_bits fliplr([remainder zeros(1, CRC_Length -
length(remainder))])];
        message = zeros(1, N);
        message(Sequence(N-K+1:end)) = CRC_padded_message;

        code_word = polar_encode(N, n, Sequence, message);
        code_word = bpsk_awgn_channel(code_word,Eb_No,A,N);
        global L
        global u_hat;
        global Path_matrix;
        global Node_states;

        L = zeros(List_size, n+1, N);
        L(:,1,:) = repmat(code_word, List_size, 1, 1);
        u_hat = zeros(List_size, n+1, N);
        Node_states = zeros(1, 2*N-1);
        Path_matrix = Inf*ones(List_size, 1);
        Path_matrix(1) = 0;

        CA_SCL_decode(0, 0, n, Frozen_bits);

        message_cap_temp = squeeze(u_hat(:,n+1, Sequence(N-K+1:end)));
        selected_code_word = 1;
        for count = 1:List_size
             [new_quotient, new_remainder] =
gfdeconv(fliplr(message_cap_temp(count,:)), CRC_Polynomial);
             if (isequal(new_remainder, 0))
                 selected_code_word = count;
                 break;
             end
        end
        message_cap = message_cap_temp(selected_code_word, 1:A);
        count = 0;
        for i = 1:length(message_cap)
```

```
            if message_cap(i) ~= info_bits(i)
                Nsim_bit_errors = Nsim_bit_errors + 1;      % bit error count
                count = 1;                % mark block error
            end
        end
        Nsim_block_errors = Nsim_block_errors + count;
    end
    P_succ_AWGN(Iteration_index) = 1 - (1/Nsim)*Nsim_block_errors;
    Bit_Error_Rate_AWGN(Iteration_index) = Nsim_bit_errors/(A*Nsim);
    Block_Error_Rate_AWGN(Iteration_index) = Nsim_block_errors/Nsim;
    Iteration_index = Iteration_index + 1;
end
```

```
%% CRC-SCL Decoder for Random Noise Channel
load('Reliability_Sequence.mat');
N = 1024;
n = log2(N);
A = 508;
CRC_Length = 4;
CRC_Polynomial = fliplr([1 0 0 1 1]);
K = A + CRC_Length;
Rate = A/N;

global List_size;
List_size = 4;

Sequence = Reliability_Sequence(Reliability_Sequence <= N);
Frozen_bits = Sequence(1:N-K);

Nsim = 100;
Eb_No_dB = 0:0.5:10;
Eb_No_Range = 10.^(Eb_No_dB/10);
P_succ_mixed = zeros(1, length(Eb_No_dB));
Bit_Error_Rate_mixed = zeros(1, length(Eb_No_dB));
Block_Error_Rate_mixed = zeros(1, length(Eb_No_dB));


%Successive Cancellation List Decoder with CRC Check
Iteration_index = 1;
for Eb_No = Eb_No_Range
    Successful_Decoding = 0;
    Nsim_block_errors = 0;
    Nsim_bit_errors = 0;

    for block = 1:Nsim
        info_bits = randi([0 1], 1, A);
        [quotient, remainder] = gfdeconv([zeros(1, CRC_Length) fliplr(info_bits)],
CRC_Polynomial);
```

```
        CRC_padded_message = [info_bits fliplr([remainder zeros(1, CRC_Length -
length(remainder))])];
        message = zeros(1, N);
        message(Sequence(N-K+1:end)) = CRC_padded_message;

        code_word = polar_encode(N, n, Sequence, message);
        code_word = bpsk_mixed_channel(code_word,Eb_No,A,N);
        global L
        global u_hat;
        global Path_matrix;
        global Node_states;

        L = zeros(List_size, n+1, N);
        L(:,1,:) = repmat(code_word, List_size, 1, 1);
        u_hat = zeros(List_size, n+1, N);
        Node_states = zeros(1, 2*N-1);
        Path_matrix = Inf*ones(List_size, 1);
        Path_matrix(1) = 0;

        CA_SCL_decode(0, 0, n, Frozen_bits);

        message_cap_temp = squeeze(u_hat(:,n+1, Sequence(N-K+1:end)));
        selected_code_word = 1;
        for count = 1:List_size
             [new_quotient, new_remainder] =
gfdeconv(fliplr(message_cap_temp(count,:)), CRC_Polynomial);
             if (isequal(new_remainder, 0))
                 selected_code_word = count;
                 break;
             end
        end
        message_cap = message_cap_temp(selected_code_word, 1:A);
        count = 0;
        for i = 1:length(message_cap)
            if message_cap(i) ~= info_bits(i)
                Nsim_bit_errors = Nsim_bit_errors + 1;      % bit error count
                count = 1;               % mark block error
            end
        end
        Nsim_block_errors = Nsim_block_errors + count;
    end
    P_succ_mixed(Iteration_index) = 1 - (1/Nsim)*Nsim_block_errors;
    Bit_Error_Rate_mixed(Iteration_index) = Nsim_bit_errors/(A*Nsim);
    Block_Error_Rate_mixed(Iteration_index) = Nsim_block_errors/Nsim;
    Iteration_index = Iteration_index + 1;
end
```

```
% Plot Probability of Success
```

```
figure;
plot(Eb_No_dB, P_succ_AWGN, 'bo-', 'LineWidth', 2, 'MarkerFaceColor', 'b',
'MarkerSize', 5);
hold on;
plot(Eb_No_dB, P_succ_mixed, 'ro-', 'LineWidth', 2, 'MarkerFaceColor', 'r',
'MarkerSize', 5);
xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex');
ylabel('Probability of Success', 'Interpreter', 'latex');
title(['Success Probability vs $E_b/N_0$ (SCL AWGN vs SCL mixed)' newline ...
       '(N = ', num2str(N), ', K = ', num2str(K), ', Code Rate = ', num2str(Rate,
'%.2f'), ...
       ', $N_{sim}$ = ', num2str(Nsim), ')'], 'Interpreter', 'latex');
legend('Success Rate Simulated(AWGN)', 'Success Rate Simulated(mixed)');
grid on;

% Plot Bit Error Rate (BER)
figure;
semilogy(Eb_No_dB, Bit_Error_Rate_AWGN, 'bo-', 'LineWidth', 2, 'MarkerFaceColor',
'r', 'MarkerSize', 5);
hold on;
semilogy(Eb_No_dB, Bit_Error_Rate_mixed, 'ro-', 'LineWidth', 2, 'MarkerFaceColor',
'r', 'MarkerSize', 5);
xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex');
ylabel('Bit Error Rate (BER)', 'Interpreter', 'latex');
title(['Bit Error Rate vs $E_b/N_0$ (AWGN vs mixed)' newline ...
       '(N = ', num2str(N), ', K = ', num2str(K), ', Code Rate = ', num2str(Rate,
'%.2f'), ...
       ', $N_{sim}$ = ', num2str(Nsim), ')'], 'Interpreter', 'latex');
legend('Bit Error Rate Simulated(AWGN)', 'Bit Error Rate Simulated(mixed)');
grid on;

% Plot Block Error Rate (BLER)
figure;
semilogy(Eb_No_dB, Block_Error_Rate_AWGN, 'bo-', 'LineWidth', 2, 'MarkerFaceColor',
'b', 'MarkerSize', 5);
hold on;
semilogy(Eb_No_dB, Block_Error_Rate_mixed, 'ro-', 'LineWidth', 2,
'MarkerFaceColor', 'r', 'MarkerSize', 5);
xlabel('$E_b/N_0$ (dB)', 'Interpreter', 'latex');
ylabel('Block Error Rate (BLER)', 'Interpreter', 'latex');
title(['Block Error Rate vs $E_b/N_0$ (AWGN vs mixed)' newline ...
       '(N = ', num2str(N), ', K = ', num2str(K), ', Code Rate = ', num2str(Rate,
'%.2f'), ...
       ', $N_{sim}$ = ', num2str(Nsim), ')'], 'Interpreter', 'latex');
legend('Block Error Rate Simulated(AWGN)', 'Block Error Rate Simulated(mixed)');
grid on;
toc
```