CT-216

Introduction To Communication System

POLAR CODES

Submitted By:

Lab Group :- 2

Project Group :- 11

Group Members:

1. Diyen Pambhar 202301113
2. Yogesh Bagotia 202301114
3. Meet Dobariya l 202301115
4. Kavya Chauhan 202301116
5. Sanya Vaishnavi 202301117
6. Mudit Rungta 202301118
7. Dharva Patel 202301119
8. Krutant Jethva 202301120
9. Sri Sadana Dharavath 202301121

Instructor:

Prof. Yash Vasavada

May 6, 2025

Contents

1 Preliminaries

1.1 The Binary Erasure Channel

A *Binary Erasure Channel*, denoted BEC(p), accepts an input bit $X \in \{0, 1\}$ and outputs

$$Y = \begin{cases} X, & \text{with probability } 1 - p, \\ ?, & \text{with probability } p. \end{cases}$$

Its Shannon capacity is

$$C\big(\mathrm{BEC}(p)\big) \;=\; 1 - p.$$

1.2 Polar Transform

Given two independent uses of BEC(p) (channels 1 and 2), we form two synthesized channels W^- and W^+ by:

1. **Combine:** Define
$$U_1 = X_1 \oplus X_2, \quad U_2 = X_2,$$
and transmit U_1 through channel 1 and U_2 through channel 2.

2. **Split:**
   - W^+ decodes U_1 from (Y_1, Y_2) without knowledge of U_2.
   - W^- decodes U_2 from (Y_1, Y_2), assuming U_1 has been correctly recovered.

A straightforward calculation shows

$$W^+ = \mathrm{BEC}(p^+), \quad p^+ = p^2, \qquad W^- = \mathrm{BEC}(p^-), \quad p^- = 2p - p^2.$$

Thus W^+ is strictly *better* (lower erasure probability) and W^- is strictly *worse* (higher erasure probability) whenever $0 < p < 1$.

2 Polarization Theorem for the BEC

2.1 Basic Polarization Statement

[Polarization for BEC] Let $W = \mathrm{BEC}(p)$. For any fixed $\varepsilon \in (0, \frac{1}{2})$, after t levels of the polar transform one obtains 2^t synthesized channels $\{W^s : s \in \{+, -\}^t\}$. Define

$$\mu_t(\varepsilon) = \frac{1}{2^t} \sum_{s \in \{+,-\}^t} \mathbf{1}\big\{p(s) \in (\varepsilon,\, 1 - \varepsilon)\big\}.$$

Then

$$\lim_{t\to\infty} \mu_t(\varepsilon) \;=\; 0.$$

2.2 Ugliness as a Potential Function

To quantify "middleness," define the *ugliness* of a BEC(q) by

$$\mathrm{ugly}(q) = \sqrt{4\,q\,(1-q)}\,, \quad q \in [0,1].$$

Note that:

- $\mathrm{ugly}(0) = \mathrm{ugly}(1) = 0$.
- $\mathrm{ugly}\big(\frac{1}{2}\big) = 1$.
- If $q \in (\varepsilon, 1-\varepsilon)$ then $\mathrm{ugly}(q) \geq \sqrt{4\,\varepsilon(1-\varepsilon)} > 0$.

2.3 Contraction under One Polarization Step

Let $P = \mathrm{BEC}(q)$, and write its two children as

$$P^+ = \mathrm{BEC}(q^+), \quad q^+ = q^2, \qquad P^- = \mathrm{BEC}(q^-), \quad q^- = 2q - q^2.$$

A direct computation shows

$$\begin{aligned}
\mathrm{ugly}(q^+) &= \sqrt{4\,q^+(1-q^+)} = \mathrm{ugly}(q)\,\sqrt{q\,(1+q)}\,, \\
\mathrm{ugly}(q^-) &= \sqrt{4\,q^-(1-q^-)} = \mathrm{ugly}(q)\,\sqrt{(2-q)\,(1-q)}.
\end{aligned}$$

Hajek's observation is that the *average* ugliness contracts:

$$\frac{1}{2}\big[\mathrm{ugly}(q^+) + \mathrm{ugly}(q^-)\big] \;\leq\; \sqrt{\tfrac{3}{4}}\,\mathrm{ugly}(q), \quad \sqrt{\tfrac{3}{4}} < 1.$$

Iterating this contraction over t levels yields

$$\frac{1}{2^t} \sum_{s\in\{+,-\}^t} \mathrm{ugly}\big(p(s)\big) \;\leq\; \Big(\sqrt{\tfrac{3}{4}}\Big)^t \mathrm{ugly}(p) \;=\; \Big(\tfrac{3}{4}\Big)^{t/2} \mathrm{ugly}(p) \;\longrightarrow\; 0.$$

Since each ε-mediocre channel contributes at least $\sqrt{4\,\varepsilon(1-\varepsilon)} > 0$ ugliness, their fraction $\mu_t(\varepsilon)$ must also tend to zero.

3 Polar–Coding Scheme and Its Limitations

3.1 Construction of Polar Codes

To communicate over a BEC(p) of capacity $1-p$ at any rate $R < 1-p$, proceed as follows:

1. Fix the number of polarization levels $t \in \mathbb{N}$, and set the block-length
$$n = 2^t.$$

2. Apply Arıkan's combine–split transform for t levels to synthesize n *bit-channels*
$$\big\{ W^{s_1 \dots s_t} = \mathrm{BEC}\big(p(s_1 \dots s_t)\big) : s_i \in \{+,-\}\big\},$$
where $p(s_1 \dots s_t)$ denotes the erasure probability of the channel indexed by the sign-sequence $s = (s_1, \dots, s_t)$.

3. Sort these n channels in ascending order of $p(s)$.

4. Choose
$$k = \lfloor n\, R \rfloor$$
information channels (those with the smallest $p(s)$). Freeze the remaining $n-k$ channels by fixing their inputs to a known constant (e.g. 0).

3.2 Union–Bound on Block–Error Probability

Under successive-cancellation decoding, a block error occurs if any of the k information-carrying channels erases its bit. By the union bound,

$$P_{\text{block error}} \;=\; \Pr\{\text{any selected channel erases}\} \;\le\; \sum_{s \in \mathcal{A}} p(s) \;\le\; k \max_{s \in \mathcal{A}} p(s) \;\le\; n\,\varepsilon,$$

where

- $\mathcal{A}$ is the set of selected channels,
- ε is an upper bound on each $p(s)$ for $s \in \mathcal{A}$.

3.3 Limitation of a Fixed ε

The basic polarization theorem guarantees that for any fixed $\varepsilon \in (0, 1/2)$,

$$\frac{\big|\{\, s : p(s) \le \varepsilon\}\big|}{n} \;\longrightarrow\; 1-p \quad \text{as } t \to \infty.$$

However, if $\varepsilon > 0$ is held constant, then

$$P_{\text{block error}} \;\leq\; n\,\varepsilon \;\longrightarrow\; \infty.$$

Therefore, to achieve vanishing error probability one must allow ε to *decay* with n.

$$A_i \;=\; \mathrm{Rot}\big(z,\theta_i\big)\;\mathrm{Trans}\big(z,d_i\big)\;\mathrm{Trans}\big(x,a_{i-1}\big)\;\mathrm{Rot}\big(x,\alpha_{i-1}\big) \tag{1}$$

[a4paper,12pt]article [utf8]inputenc [T1]fontenc geometry margin=1in amsmath,amsfonts,amssymb sectsty graphicx caption booktabs hyperref colorlinks=true, linkcolor=blue, filecolor=magenta, urlcolor=cyan, enumitem noto

4 Polar Codes: Achieving Shannon's Capacity, Proof, Bhattacharyya Parameters, and Polarization

4.1 Introduction

Polar codes, pioneered by Erdal Arikan, mark a pivotal advancement in coding theory, enabling the achievement of Shannon's channel capacity for various channels, including binary discrete memoryless channels (B-DMCs) and additive white Gaussian noise (AWGN) channels. This section introduces the Shannon capacity, explains how polar codes attain this capacity through channel polarization, provides a detailed step-by-step proof, and explores the roles of Bhattacharyya parameters and polarization. Emphasizing the polar transform's ability to segregate channels into highly informative and information-less subsets.

4.2 Achieving Shannon's Capacity

Shannon's noisy coding theorem posits that a channel W has a capacity C or $I(W)$, representing the maximum rate for reliable communication with vanishing error probability. For a general channel with input X and output Y, the Shannon capacity is defined as the maximum mutual information over all possible input distributions:

$$I(W) = \max_{p(x)} I(X;Y),$$

where $I(X;Y) = H(Y) - H(Y|X)$ is the mutual information, and $H(\cdot)$ denotes entropy. Specifically, for a binary discrete memoryless channel (B-DMC):

- For a binary symmetric channel (BSC) with error probability p and uniform input distribution (Bernoulli $q = 0.5$), the capacity is:

$$I(W) = 1 - H_2(p),$$

where $H_2(p) = -p \log_2 p - (1-p) \log_2 (1-p)$ is the binary entropy function.

- For a binary erasure channel (BEC) with erasure probability p, the capacity is:

$$I(W) = 1 - p.$$

For an additive white Gaussian noise (AWGN) channel, the Shannon capacity is given by:

$$C = B \cdot \log_2 \left(1 + \frac{S}{N}\right),$$

where C is the channel capacity (bits/sec), B is the bandwidth (Hz), and $\frac{S}{N}$ is the signal-to-noise ratio (SNR).

Polar codes achieve this capacity through *channel polarization*, a process that transforms N copies of a channel W into N bit-channels $W_N^{(i)}$ with extreme reliability characteristics. Polarization occurs by applying a polar transform, defined by the matrix $\mathbf{G}_N = \mathbf{G}_2^{\otimes n}$ where $\mathbf{G}_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ and $n = \log_2 N$. This transform recursively splits the channel into bit-channels, where some become highly reliable (nearly noiseless) and others become highly unreliable (noisy), a phenomenon often likened to the "Matthew Effect" (the rich get richer, the poor get poorer). The detailed proof of how this leads to achieving Shannon's capacity is provided in the next section.

4.3 Step-by-Step Proof of Capacity Achievement

The proof that polar codes achieve Shannon's capacity hinges on the polar transform and channel polarization. Below is a detailed, step-by-step proof.

4.3.1 Step 1: Define the Polar Transform

The polar transform is defined by a transformation matrix. For $N = 2$, the matrix is:

$$\mathbf{G}_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

For a general block length $N = 2^n$, the transformation matrix is the n-th Kronecker product of $\mathbf{G}_2$:

$$\mathbf{G}_N = \mathbf{G}_2^{\otimes n},$$

where $n = \log_2 N$. This matrix transforms an input vector $u = [u_1, u_2, \ldots, u_N]$ into an output vector:

$$Y = u \cdot \mathbf{G}_N,$$

where $Y = [y_1, y_2, \ldots, y_N]$ is transmitted through N copies of the channel W.

4.3.2 Step 2: Channel Polarization for $N = 2$

Consider a B-DMC W with input alphabet $X = \{0, 1\}$, output alphabet Y, and transition probabilities $W(y|x)$. For $N = 2$, define a two-input, two-output channel W_2:

$$W_2(y_1, y_2 \mid u_1, u_2) = W(y_1 \mid u_1 \oplus u_2) \cdot W(y_2 \mid u_2),$$

where $\oplus$ denotes modulo-2 addition (XOR). The polar transform applies $\mathbf{G}_2$, mapping inputs $u = [u_1, u_2]$ to:

$$Y = [y_1, y_2] = u \cdot \mathbf{G}_2 = [u_1 \oplus u_2, u_2].$$

This creates two virtual bit-channels:

- **Minus channel** (W^-): $W^- : U_1 \to (Y_1, Y_2)$, transmitting u_1 given outputs y_1, y_2.
- **Plus channel** (W^+): $W^+ : U_2 \to (Y_1, Y_2, U_1)$, transmitting u_2 given y_1, y_2, u_1.

For a BEC with erasure probability p, the erasure probabilities are:

- W^+: Erasure probability $= p^2$.
- W^-: Erasure probability $= p(2 - p)$.

The capacity of a BEC is $I(W) = 1 - p$. Assuming inputs follow a Bernoulli distribution with $q = 0.5$ (uniform), the mutual information is:

- Original channel: $I(W) = H_2(q)(1 - p) = 1 - p$.
- Plus channel: $I(W^+) = H_2(q)(1 - p^2) = 1 - p^2$.
- Minus channel: $I(W^-) = H_2(q)(1 - (2p - p^2)) = 1 - (2p - p^2)$.

Since $p^2 < p < 2p - p^2$ for $0 < p < 1$, it follows that:

$$I(W^-) \leq I(W) \leq I(W^+).$$

Thus, W^+ is more reliable (higher capacity) than W, while W^- is less reliable (lower capacity).

4.3.3 Step 3: Recursive Polarization for Larger N

For $N = 4$, define a four-input, four-output channel W_4:

$$W_4(y_1, y_2, y_3, y_4 \mid u_1, u_2, u_3, u_4) = W_2(y_1, y_2 \mid u_1 \oplus u_2, u_3 \oplus u_4) \cdot W_2(y_3, y_4 \mid u_2, u_4).$$

This corresponds to applying $\mathbf{G}_4 = \mathbf{G}_2^{\otimes 2}$, creating four virtual bit-channels. Some channels become more reliable, while others degrade, extending the polarization effect.

For general $N = 2^n$, recursively apply the polar transform, generating N bit-channels $W_1, W_2, \ldots, W_N$. Each iteration splits channels into better and worse subsets, amplifying polarization. As N grows, approximately half the channels become highly informative (nearly noiseless), and half become information-less (highly noisy), embodying the "Matthew Effect".

4.3.4 Step 4: Conservation of Information

Assume the input bits U_i are independent and identically distributed (i.i.d.) with a uniform distribution (Bernoulli $q = 0.5$). The total mutual information across all bit-channels is conserved due to the invertible nature of $\mathbf{G}_N$:

$$\sum_{i=1}^{N} I(W_i) = N \cdot I(W).$$

As $N \to \infty$, the bit-channels polarize such that:

$$\lim_{N\to\infty} \frac{1}{N} \left| \{i : I(W_N^{(i)}) \in (1-\delta, 1]\} \right| = I(W),$$

$$\lim_{N\to\infty} \frac{1}{N} \left| \{i : I(W_N^{(i)}) \in [0, \delta)\} \right| = 1 - I(W),$$

for any small $\delta > 0$. This confirms that:

- A fraction $I(W)$ of bit-channels (approximately $N \cdot I(W)$) have $I(W_i) \to 1$ (noiseless).
- A fraction $1 - I(W)$ (approximately $N \cdot (1 - I(W))$) have $I(W_i) \to 0$ (noisy).

4.3.5 Step 5: Code Construction and Rate

To construct the polar code, assign the $K = N \cdot I(W)$ message bits to the $N \cdot I(W)$ noiseless bit-channels (those with $I(W_i) \approx 1$). The remaining $N \cdot (1 - I(W))$ noisy channels are "frozen" to fixed values (e.g., 0). The generator matrix comprises the rows of $\mathbf{G}_N$ corresponding to these noiseless channels.

The rate of the code is:

$$\text{Rate} = \frac{K}{N} = \frac{N \cdot I(W)}{N} = I(W).$$

Polar codes efficiently achieve the Shannon capacity using a recursive structure and successive cancellation decoding, with complexity around $O(N \log N)$. As N grows, the error probability approaches zero, ensuring reliable transmission below the channel capacity.

4.3.6 Step 6: Conclusion of Proof

Since the rate $K/N \to I(W)$ and the error probability approaches 0 as $N \to \infty$, polar codes achieve Shannon's capacity for any B-DMC. This holds for specific channels like the BSC ($I(W) = 1 - H_2(p)$) and BEC ($I(W) = 1 - p$), as the polarization process is universal.

4.4 Bhattacharyya Parameters and Polarization

The Bhattacharyya parameter $Z(W)$ quantifies channel reliability. For a B-DMC, it is typically:

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

bounding the error probability of maximum-likelihood decoding. For a BEC, $Z(W)$ corresponds to the erasure probability p, and for a BSC, it relates to the error probability p.

4.4.1 Role in Polarization

Polarization transforms the original channel W into bit-channels with extreme reliability:

- **Good channels** (W^+): Lower $Z(W^+)$, indicating higher reliability. For a BEC, $Z(W^+) = p^2$.
- **Bad channels** (W^-): Higher $Z(W^-)$, indicating lower reliability. For a BEC, $Z(W^-) = p(2 - p)$.

The mutual information reflects this polarization:

- $I(W^+) = H_2(q)(1 - p^2) = 1 - p^2$ (BEC, $q = 0.5$), higher than $I(W) = 1 - p$.
- $I(W^-) = H_2(q)(1 - (2p - p^2)) = 1 - (2p - p^2)$, lower than $I(W)$.

Recursive application of the polar transform amplifies this effect. For $N = 2^n$, the bit-channels $W_1, W_2, \ldots, W_N$ polarize such that approximately $N \cdot I(W)$ channels have $Z(W_i) \to 0$ (noiseless), and $N \cdot (1 - I(W))$ channels have $Z(W_i) \to 1$ (noisy). Message bits are assigned to channels with low $Z(W_i)$, ensuring reliable transmission.

4.4.2 Quantitative Analysis

For a BEC, the erasure probability evolves as:

- W^+: p^2, quadratically smaller than p, enhancing reliability.
- W^-: $p(2 - p)$, larger than p, reducing reliability.

For example, if $p = 0.5$:

- $Z(W^+) = (0.5)^2 = 0.25$, $I(W^+) = 1 - 0.25 = 0.75$.
- $Z(W^-) = 0.5 \cdot (2 - 0.5 = 0.75$, $I(W^-) = 1 - 0.75 = 0.25$.
- Original: $Z(W) = 0.5$, $I(W) = 1 - 0.5 = 0.5$.

This demonstrates $I(W^-) = 0.25 < 0.5 = I(W) < 0.75 = I(W^+)$. Recursive polarization drives $Z(W_i)$ to 0 or 1, with the fraction of unpolarized channels (intermediate $Z(W_i)$) vanishing as N grows, ensuring capacity-achieving performance.

4.5 Comparison

To evaluate the performance of polar codes, we compare their error probability to a theoretical benchmark in the finite block length regime. The benchmark error probability, often referred to as the Normal Approximation (NA), is given by:

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

where $Q(\cdot)$ is the Q-function (tail probability of the standard normal distribution), N is the block length, V is the channel dispersion, C is the channel capacity, r is the coding rate, and $\frac{\log_2 N}{2N}$ accounts for finite block length effects. For an AWGN channel, the parameters are defined as:

$$C = \log_2(1 + P), \quad P = r \cdot \frac{E_b}{N_0}, \quad V = (\log_2 e)^2 \cdot \frac{P(P+2)}{2(P+1)^2},$$

where $\frac{E_b}{N_0}$ is the energy per bit to noise power spectral density ratio, a measure of SNR adjusted for the coding rate r.

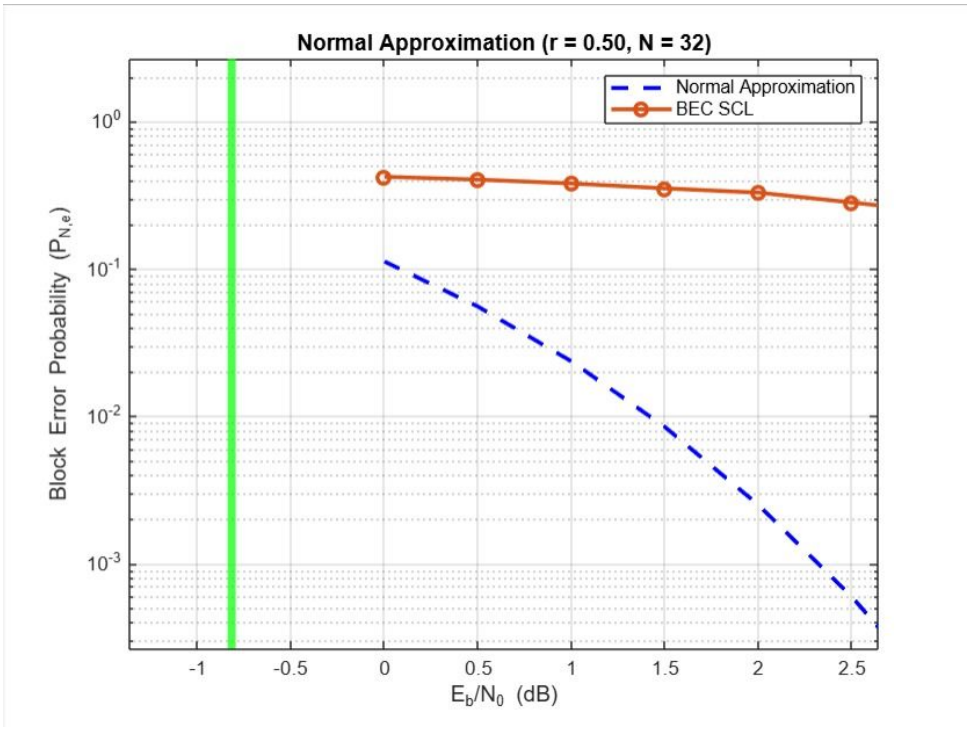
For polar codes under successive cancellation (SC) decoding, the error probability scales exponentially with the block length:

$$P_e \sim 2^{-\sqrt{N}}.$$

This exponential decay is faster than the NA's polynomial decay, as $Q(x) \sim \frac{e^{-x^2/2}}{x\sqrt{2\pi}}$ for large x. However, in the finite block length regime, polar codes with SC decoding often exhibit higher error probabilities than the NA due to suboptimal decoding.

To validate the performance of a Polar/LDPC decoder, we compare its empirical block error probability with the NA. For a BEC-SCL (Binary Erasure Channel with Successive Cancellation List) decoder with $r = 0.5$ and $N = 32$, the empirical block error probability is plotted alongside the NA curve, as shown in Figure 1. The SCL decoding enhances performance over standard SC decoding, reducing the gap between the empirical error probability and the theoretical NA, especially at small block lengths like $N = 32$. This demonstrates that while polar codes theoretically outperform the NA in terms of scaling,

practical implementations benefit significantly from advanced decoding techniques like SCL, making them more competitive for real-world applications.



Graph: BEC-SCL Decoder Performance vs. Normal Approximation

Figure 1: Empirical block error probability of a BEC-SCL decoder ($r = 0.5$, $N = 32$) compared to the Normal Approximation (NA).

The graph in Figure 1 illustrates the empirical performance of the BEC-SCL decoder against the NA, highlighting the effectiveness of SCL decoding in approaching the theoretical limit, even at finite block lengths.

4.6 Conclusion

Polar codes achieve Shannon's capacity through channel polarization, segregating bit-channels into $N \cdot I(W)$ noiseless and $N \cdot (1 - I(W))$ noisy channels, as evidenced by the polarization limits. The step-by-step proof establishes that the rate $K/N \to I(W)$ with vanishing error probability. This applies to B-DMCs like the BSC ($I(W) = 1 - H_2(p)$) and BEC ($I(W) = 1 - p$), as well as AWGN channels where $C = B \cdot \log_2\left(1 + \frac{S}{N}\right)$. Bhattacharyya parameters, approximated by erasure probabilities in BECs, quantify channel reliability, driving the polarization process. As shown in the comparison, polar codes exhibit a favorable error probability scaling compared to theoretical benchmarks like the Normal Approximation, and practical implementations using SCL decoding further bridge the gap to theoretical limits, positioning polar codes as a foundational technology for near-optimal communication systems across diverse channel models.

5 Decoding

5.1 Successive Cancellation Decoder (SC)

Because of the recursive structure of channel polarization, we can use a simple decoding scheme called **Successive Cancellation (SC)** for decoding.

As we have seen how to polarize a channel in the previous section, the decoding problem reduces to decoding a **single parity check code (SPC)** to estimate u_0. After deciding u_0, we can use it to decode u_1, which turns out to be a **repetition code**.

Since we transmit bits over an **AWGN channel with BPSK modulation**, we assume we receive **log-likelihood ratios (LLRs)** of the transmitted bits.

Define LLRs:

$$a = \log \frac{P(y_1 \mid u_1 = 0)}{P(y_1 \mid u_1 = 1)}$$
$$b = \log \frac{P(y_2 \mid u_2 = 0)}{P(y_2 \mid u_2 = 1)}$$

We want to decode the first bit u_0 in a 2-bit polar code, where:

$$x_1 = u_0 \oplus u_1, \quad x_2 = u_1$$

We receive y_1, y_2 from the channel. To calculate:

$$L(u_0) = \log \frac{P(y_1, y_2 \mid u_0 = 1)}{P(y_1, y_2 \mid u_0 = 0)}$$

Since $x_1 = u_0 \oplus u_1$ and $x_2 = u_1$, we must marginalize over $u_1 \in \{0, 1\}$:

$$L(u_0) = \log \left(\frac{\sum_{u_1 \in \{0,1\}} P(y_1 \mid x_1 = u_0 \oplus u_1) \cdot P(y_2 \mid x_2 = u_1)}{\sum_{u_1 \in \{0,1\}} P(y_1 \mid x_1 = \bar{u}_0 \oplus u_1) \cdot P(y_2 \mid x_2 = u_1)} \right)$$

Define:

$$\lambda_1(u) = P(y_1 \mid x_1 = u)$$
$$\lambda_2(u) = P(y_2 \mid x_2 = u)$$

Then:

$$L(u_0) = \log \left(\frac{\lambda_1(0)\lambda_2(0) + \lambda_1(1)\lambda_2(1)}{\lambda_1(1)\lambda_2(0) + \lambda_1(0)\lambda_2(1)} \right)$$

Express in terms of LLRs:

Let:

$$a = \log \frac{\lambda_1(0)}{\lambda_1(1)}$$
$$b = \log \frac{\lambda_2(0)}{\lambda_2(1)}$$

Then:

$$\lambda_1(0) = e^a \cdot \lambda_1(1)$$
$$\lambda_2(0) = e^b \cdot \lambda_2(1)$$

Substitute into the LLR expression:

$$L(u_0) = \log \left(\frac{e^{a+b} + 1}{e^a + e^b} \right)$$

Alternatively, using the tanh form:

$$L(u_0) = 2 \tanh^{-1} \left(\tanh \left(\frac{a}{2} \right) \cdot \tanh \left(\frac{b}{2} \right) \right)$$

Min-sum approximation (to reduce computational cost):

$$f(a, b) \approx \text{sign}(a) \cdot \text{sign}(b) \cdot \min(|a|, |b|)$$

Thus:

$$L(u_0) = f(a, b)$$

Estimate u_0:

$$\hat{u}_0 = \begin{cases} 0, & \text{if } L(u_0) \geq 0 \\ 1, & \text{if } L(u_0) < 0 \end{cases}$$

Decoding u_1 using estimate $\hat{u}_0$:
If $\hat{u}_0 = 0$, then $x_1 = u_1$ and $x_2 = u_1$.
If $\hat{u}_0 = 1$, then $x_1 = \bar{u}_1$ and $x_2 = u_1$.
So, decoding u_1 becomes decoding a repetition code. We compute:

$$L(u_1 \mid \hat{u}_0) = \log \left(\frac{P(y_1, y_2 \mid u_1 = 0, \hat{u}_0)}{P(y_1, y_2 \mid u_1 = 1, \hat{u}_0)} \right)$$

This becomes:

$$L(u_1 \mid \hat{u}_0) = \log \frac{\lambda_1(\hat{u}_0) \cdot \lambda_2(0)}{\lambda_1(\hat{u}_0 \oplus 1) \cdot \lambda_2(1)}$$

Split as:

$$L = \log \frac{\lambda_2(0)}{\lambda_2(1)} + \log \frac{\lambda_1(\hat{u}_0)}{\lambda_1(\hat{u}_0 \oplus 1)}$$

$$L = b + (1 - 2\hat{u}_0) \cdot a$$

Define:

$$g(a, b, \hat{u}_0) = b + (1 - 2\hat{u}_0) \cdot a$$

$$L(u_1) = g(a, b, \hat{u}_0)$$

Estimate u_1:

$$\hat{u}_1 = \begin{cases} 0, & \text{if } L(u_1) \geq 0 \\ 1, & \text{if } L(u_1) < 0 \end{cases}$$

6 SC Decoder MATLAB Code

6.1 Function: getFrozenAndInfoBits

```
function [infoBits, F] = getFrozenAndInfoBits(Q, N, K)
    Q1 = Q(Q <= N);
    F = Q1(1:N-K);                  % Frozen bits
    infoBits = Q1(N-K+1:end);       % Info bits
end
```

6.2 Function: addAWGN

```
function r = addAWGN(cword, EbNo, Rate)
    N = length(cword);
    sigma = sqrt(1 / (2 * Rate * EbNo));
    s = 1 - 2 * cword;              % BPSK modulation
    r = s + sigma * randn(1, N); % AWGN noise
end
```

6.3 Function: polarEncoder

```
function cword = polarEncoder(msg, infoBits, N)
    n = log2(N);
    u = zeros(1, N);
    u(infoBits) = msg;

    m = 1;
    for d = n - 1:-1:0
        for i = 1:2*m:N
            a = u(i:i+m-1);
            b = u(i+m:i+2*m-1);
            u(i:i+2*m-1) = [mod(a + b, 2), b];
        end
        m = m * 2;
    end
    cword = u;
end
```

6.4 Function: SCDecoder

```
function msg_cap = SCDecoder(r, F, infoBits)
    N = length(r);
    n = log2(N);
    L = zeros(n + 1, N);
    ucap = zeros(n + 1, N);
    ns = zeros(1, 2 * N - 1);

    f = @(a, b) (1 - 2 * (a < 0)) .* (1 - 2 * (b < 0)) .* min(abs(
        ↪ a), abs(b));
    g = @(a, b, c) b + (1 - 2 * c) .* a;

    node = 0; depth = 0; done = 0;
    L(1, :) = r;

    while ~done
        if depth == n
            if ismember(node + 1, F)
                ucap(n + 1, node + 1) = 0;
            else
                ucap(n + 1, node + 1) = L(n + 1, node + 1) < 0;
            end
            if node == N - 1
                done = 1;
            else
                node = floor(node / 2);
                depth = depth - 1;
            end
        else
            npos = 2^depth - 1 + node + 1;
            if ns(npos) == 0
                temp = 2^(n - depth);
                Ln = L(depth + 1, temp * node + 1 : temp * (node +
                    ↪  1));
                a = Ln(1:temp/2); b = Ln(temp/2+1:end);
                node = node * 2;
                depth = depth + 1;
                temp = temp / 2;
                L(depth + 1, temp * node + 1 : temp * (node + 1))
                    ↪ = f(a, b);
                ns(npos) = 1;
            elseif ns(npos) == 1
                temp = 2^(n - depth);
```

```
                Ln = L(depth + 1, temp * node + 1 : temp * (node +
                    ↪  1));
                lnode = 2 * node;
                ldepth = depth + 1;
                ltemp = temp / 2;
                ucapn = ucap(ldepth + 1, ltemp * lnode + 1 : ltemp
                    ↪  * (lnode + 1));
                a = Ln(1:temp/2);
                b = Ln(temp/2+1:end);
                node = node * 2 + 1;
                depth = depth + 1;
                temp = temp / 2;
                L(depth + 1, temp * node + 1 : temp * (node + 1))
                    ↪ = g(a, b, ucapn);
                ns(npos) = 2;
            else
                temp = 2^(n - depth);
                lnode = 2 * node;
                rnode = 2 * node + 1;
                cdepth = depth + 1;
                ctemp = temp / 2;
                ucapl = ucap(cdepth + 1, ctemp * lnode + 1 : ctemp
                    ↪  * (lnode + 1));
                ucapr = ucap(cdepth + 1, ctemp * rnode + 1 : ctemp
                    ↪  * (rnode + 1));
                ucap(depth + 1, temp * node + 1 : temp * (node +
                    ↪ 1)) = ...
                     [mod(ucapl + ucapr, 2), ucapr];
                node = floor(node / 2);
                depth = depth - 1;
            end
        end
    end

    msg_cap = ucap(n + 1, infoBits);
end
```

6.5 Function: runSCSimulation

```
function [BER_sim, BLER_sim] = runSCSimulation(Q, N, K,
    ↪ EbNodB_range, Nblocks)
    Rate = K / N;
```

```
    [infoBits, F] = getFrozenAndInfoBits(Q, N, K);
    BER_sim = zeros(size(EbNodB_range));
    BLER_sim = zeros(size(EbNodB_range));

    for idx = 1:length(EbNodB_range)
        EbNodB = EbNodB_range(idx);
        EbNo = 10^(EbNodB / 10);
        Nbiterrs = 0;
        Nblockerrs = 0;

        for blk = 1:Nblocks
            msg = randi([0 1], 1, K);
            cword = polarEncoder(msg, infoBits, N);
            r = addAWGN(cword, EbNo, Rate);
            msg_cap = SCDecoder(r, F, infoBits);

            bit_errors = sum(msg ~= msg_cap);
            Nbiterrs = Nbiterrs + bit_errors;
            if bit_errors > 0
                Nblockerrs = Nblockerrs + 1;
            end
        end

        BER_sim(idx) = Nbiterrs / (K * Nblocks);
        BLER_sim(idx) = Nblockerrs / Nblocks;
        fprintf('Eb/N0 = %.1f dB, BER = %.5f\n', EbNodB, BER_sim(
            ↪ idx));
    end
    BER_SC=BER_sim;
    assignin('base', 'BER_SC', BER_SC);
    BLER_SC=BLER_sim;
    assignin('base', 'BLER_SC', BER_SC);
end
```

6.6 Function: plotSCLResults

```
function plotSCLResults(EbNodB_range, BER_sim, BLER_sim, N, K,
    ↪ Rate)
    % Plot SCL decoder results
    figure;
    semilogy(EbNodB_range, BER_sim, 'o-', 'LineWidth', 2);
    grid on;
```

```
    xlabel('Eb/N0 (dB)');
    ylabel('Bit Error Rate (BER)');
    title(sprintf('Polar Code SCL Decoder: N = %d, K = %d, Rate =
        ↪ %.2f', N, K, Rate));

    figure;
    semilogy(EbNodB_range, 1 - BLER_sim, 's-', 'LineWidth', 2);
    grid on;
    xlabel('Eb/N0 (dB)');
    ylabel('Probability of Successful Decoding');
    title(sprintf('Success Probability: N = %d, K = %d, Rate = %.2
        ↪ f', N, K, Rate));
end
```

6.7 Function: mail_SC

```
% Main script to run SC simulation
N = 256;
K = 128;
EbNodB_range = 0:0.5:10;
Nblocks = 10000;
Q=[0 1 2 4 8 16 32 3 5 64 9 6 17 10 18 128 12 33 65 20 256 34 24
    ↪ 36 7 129 66 512 11 40 68 130 ...
    19 13 48 14 72 257 21 132 35 258 26 513 80 37 25 22 136 260
        ↪ 264 38 514 96 67 41 144 28 69 42 ...
    516 49 74 272 160 520 288 528 192 544 70 44 131 81 50 73 15
        ↪ 320 133 52 23 134 384 76 137 82 56 27 ...
    97 39 259 84 138 145 261 29 43 98 515 88 140 30 146 71 262 265
        ↪  161 576 45 100 640 51 148 46 75 266 273 517 104 162 ...
    53 193 152 77 164 768 268 274 518 54 83 57 521 112 135 78 289
        ↪ 194 85 276 522 58 168 139 99 86 60 280 89 290 529 524
        ↪ ...
    196 141 101 147 176 142 530 321 31 200 90 545 292 322 532 263
        ↪ 149 102 105 304 296 163 92 47 267 385 546 324 208 386
        ↪ 150 153 ...
    165 106 55 328 536 577 548 113 154 79 269 108 578 224 166 519
        ↪ 552 195 270 641 523 275 580 291 59 169 560 114 277 156
        ↪ 87 197 ...
    116 170 61 531 525 642 281 278 526 177 293 388 91 584 769 198
        ↪ 172 120 201 336 62 282 143 103 178 294 93 644 202 592
        ↪ 323 392 ...
```

```
297 770 107 180 151 209 284 648 94 204 298 400 608 352 325 533
   ↪  155 210 305 547 300 109 184 534 537 115 167 225 326 306
   ↪  772 157 ...
656 329 110 117 212 171 776 330 226 549 538 387 308 216 416
   ↪ 271 279 158 337 550 672 118 332 579 540 389 173 121 553
   ↪ 199 784 179 ...
228 338 312 704 390 174 554 581 393 283 122 448 353 561 203 63
   ↪  340 394 527 582 556 181 295 285 232 124 205 182 643 562
   ↪  286 585 ...
299 354 211 401 185 396 344 586 645 593 535 240 206 95 327 564
   ↪  800 402 356 307 301 417 213 568 832 588 186 646 404 227
   ↪  896 594 ...
418 302 649 771 360 539 111 331 214 309 188 449 217 408 609
   ↪ 596 551 650 229 159 420 310 541 773 610 657 333 119 600
   ↪ 339 218 368 ...
652 230 391 313 450 542 334 233 555 774 175 123 658 612 341
   ↪ 777 220 314 424 395 673 583 355 287 183 234 125 557 660
   ↪ 616 342 316 ...
241 778 563 345 452 397 403 207 674 558 785 432 357 187 236
   ↪ 664 624 587 780 705 126 242 565 398 346 456 358 405 303
   ↪ 569 244 595 ...
189 566 676 361 706 589 215 786 647 348 419 406 464 680 801
   ↪ 362 590 409 570 788 597 572 219 311 708 598 601 651 421
   ↪ 792 802 611 ...
602 410 231 688 653 248 369 190 364 654 659 335 480 315 221
   ↪ 370 613 422 425 451 614 543 235 412 343 372 775 317 222
   ↪ 426 453 237 ...
559 833 804 712 834 661 808 779 617 604 433 720 816 836 347
   ↪ 897 243 662 454 318 675 618 898 781 376 428 665 736 567
   ↪ 840 625 238 ...
359 457 399 787 591 678 434 677 349 245 458 666 620 363 127
   ↪ 191 782 407 436 626 571 465 681 246 707 350 599 668 790
   ↪ 460 249 682 ...
573 411 803 789 709 365 440 628 689 374 423 466 793 250 371
   ↪ 481 574 413 603 366 468 655 900 805 615 684 710 429 794
   ↪ 252 373 605 ...
848 690 713 632 482 806 427 904 414 223 663 692 835 619 472
   ↪ 455 796 809 714 721 837 716 864 810 606 912 722 696 377
   ↪ 435 817 319 ...
621 812 484 430 838 667 488 239 378 459 622 627 437 380 818
   ↪ 461 496 669 679 724 841 629 351 467 438 737 251 462 442
   ↪ 441 469 247 ...
```

```
    683 842 738 899 670 783 849 820 728 928 791 367 901 630 685
       ↪ 844 633 711 253 691 824 902 686 740 850 375 444 470 483
       ↪ 415 485 905 ...
    795 473 634 744 852 960 865 693 797 906 715 807 474 636 694
       ↪ 254 717 575 913 798 811 379 697 431 607 489 866 723 486
       ↪ 908 718 813 ...
    476 856 839 725 698 914 752 868 819 814 439 929 490 623 671
       ↪ 739 916 463 843 381 497 930 821 726 961 872 492 631 729
       ↪ 700 443 741 ...
    845 920 382 822 851 730 498 880 742 445 471 635 932 687 903
       ↪ 825 500 846 745 826 732 446 962 936 475 853 867 637 907
       ↪ 487 695 746 ...
    828 753 854 857 504 799 255 964 909 719 477 915 638 748 944
       ↪ 869 491 699 754 858 478 968 383 910 815 976 870 917 727
       ↪ 493 873 701 ...
    931 756 860 499 731 823 922 874 918 502 933 743 760 881 494
       ↪ 702 921 501 876 847 992 447 733 827 934 882 937 963 747
       ↪ 505 855 924 ...
    734 829 965 938 884 506 749 945 966 755 859 940 830 911 871
       ↪ 639 888 479 946 750 969 508 861 757 970 919 875 862 758
       ↪ 948 977 923 ...
    972 761 877 952 495 703 935 978 883 762 503 925 878 735 993
       ↪ 885 939 994 980 926 764 941 967 886 831 947 507 889 984
       ↪ 751 942 996 ...
    971 890 509 949 973 1000 892 950 863 759 1008 510 979 953 763
       ↪ 974 954 879 981 982 927 995 765 956 887 985 997 986 943
       ↪ 891 998 766 ...
    511 988 1001 951 1002 893 975 894 1009 955 1004 1010 957 983
       ↪ 958 987 1012 999 1016 767 989 1003 990 1005 959 1011
       ↪ 1013 895 1006 1014 1017 1018 ...
    991 1020 1007 1015 1019 1021 1022 1023]+1; % Reliability
       ↪ sequence

[BER_sim, BLER_sim] = runSCSimulation(Q, N, K, EbNodB_range,
   ↪ Nblocks);
plotSCLResults(EbNodB_range, BER_sim, BLER_sim, N, K);
```

7 SCL Decoder MATLAB Code

7.1 Function: getFrozenAndInfoBits

```
function [infoBits, F] = getFrozenAndInfoBits(Q, N, K)
```

```
    Q1 = Q(Q <= N);
    F = Q1(1:N-K);                  % Frozen bits
    infoBits = Q1(N-K+1:end);       % Info bits
end
```

7.2 Function: addNoiseAndQuantize

```
function rq = addNoiseAndQuantize(cword, sigma, rmax, maxqr)
    % Add AWGN noise and quantize
    s = 1 - 2 * cword; % BPSK modulation
    r = s + sigma * randn(1, length(cword)); % AWGN channel
    r = min(max(r, -rmax), rmax); % Saturate
    rq = round(r/rmax*maxqr); % Quantize
end
```

7.3 Function: generateMessageWithCRC

```
function [msg, msgcrc] = generateMessageWithCRC(A, crcL, crcg)
    % Generate random message and append CRC
    msg = randi([0 1], 1, A);
    [quot, rem] = gfdeconv([zeros(1,crcL) fliplr(msg)], crcg);
    msgcrc = [msg fliplr([rem zeros(1,crcL-length(rem))])];
end
```

7.4 Function: polarEncode

```
function cword = polarEncode(msgcrc, Q1, N, infoBits)
    % Polar encoding
    u = zeros(1, N);
    u(infoBits) = msgcrc;

    n = log2(N);
    m = 1;
    for d = n-1:-1:0
        for i = 1:2*m:N
            a = u(i:i+m-1);
            b = u(i+m:i+2*m-1);
            u(i:i+2*m-1) = [mod(a+b,2) b];
        end
        m = m * 2;
    end
```

```
    cword = u;
end
```

7.5 Function: processLeftChild

```
function [node, depth, LLR, ns] = processLeftChild(node, depth, n,
    ↪ LLR, ns, npos, f)
    % Process left child node
    temp = 2^(n-depth);
    Ln = squeeze(LLR(:,depth+1,temp*node+1:temp*(node+1)));
    a = Ln(:,1:temp/2); b = Ln(:,temp/2+1:end);

    node = node * 2;
    depth = depth + 1;
    temp = temp / 2;

    LLR(:,depth+1,temp*node+1:temp*(node+1)) = f(a,b);
    ns(npos) = 1;
end
```

7.6 Function: processRightChild

```
function [node, depth, LLR, ns] = processRightChild(node, depth, n
    ↪ , LLR, ucap, ns, npos, g, maxqr)
    % Process right child node
    temp = 2^(n-depth);
    Ln = squeeze(LLR(:,depth+1,temp*node+1:temp*(node+1)));
    a = Ln(:,1:temp/2); b = Ln(:,temp/2+1:end);

    lnode = 2*node;
    ldepth = depth + 1;
    ltemp = temp/2;
    ucapn = squeeze(ucap(:,ldepth+1,ltemp*lnode+1:ltemp*(lnode+1))
        ↪ );

    node = node * 2 + 1;
    depth = depth + 1;
    temp = temp / 2;

    LLR(:,depth+1,temp*node+1:temp*(node+1)) = g(a,b,ucapn);
    ns(npos) = 2;
```

```
end
```

7.7 Function: processParentNode

```
function [node, depth, ucap] = processParentNode(node, depth, n,
    ↪ ucap)
    % Process parent node (combine decisions)
    temp = 2^(n-depth);
    lnode = 2*node;
    rnode = 2*node + 1;
    cdepth = depth + 1;
    ctemp = temp/2;

    ucapl = squeeze(ucap(:,cdepth+1,ctemp*lnode+1:ctemp*(lnode+1))
        ↪ );
    ucapr = squeeze(ucap(:,cdepth+1,ctemp*rnode+1:ctemp*(rnode+1))
        ↪ );

    ucap(:,depth+1,temp*node+1:temp*(node+1)) = [mod(ucapl+ucapr
        ↪ ,2) ucapr];
    node = floor(node/2);
    depth = depth - 1;
end
```

7.8 Function: sclDecode

```
function msg_cap = sclDecode(rq, F, infoBits, nL, crcg, N, rmax,
    ↪ maxqr,A)
    % SCL decoder implementation
    n = log2(N);

    % Initialize data structures
    LLR = zeros(nL, n+1, N);     % Beliefs
    ucap = zeros(nL, n+1, N);    % Decisions
    PML = Inf * ones(nL, 1);     % Path metrics
    PML(1) = 0;
    ns = zeros(1, 2*N-1);        % Node state vector

    % Define functions
    satx = @(x,th) min(max(x,-th),th);
    f = @(a,b) (1-2*(a<0)).*(1-2*(b<0)).*min(abs(a),abs(b)); %
        ↪ minsum
```

```
    g = @(a,b,c) satx(b+(1-2*c).*a,maxqr); % g function

    % Initialize root node
    LLR(:,1,:) = repmat(rq, nL, 1, 1);

    node = 0; depth = 0; done = 0;
    while ~done
        if depth == n
            % Leaf node processing
            DM = squeeze(LLR(:,n+1,node+1));

            if any(F == (node+1))
                % Frozen bit - set to 0
                ucap(:,n+1,node+1) = 0;
                PML = PML + abs(DM).*(DM < 0);
            else
                % Information bit - expand paths
                dec = DM < 0;
                PM2 = [PML; PML + abs(DM)];
                [PML, pos] = mink(PM2, nL);

                pos1 = pos > nL;
                pos(pos1) = pos(pos1) - nL;
                dec = dec(pos);
                dec(pos1) = 1 - dec(pos1);

                % Rearrange decoder states
                LLR = LLR(pos,:,:);
                ucap = ucap(pos,:,:);
                ucap(:,n+1,node+1) = dec;
            end

            if node == N-1
                done = 1;
            else
                node = floor(node/2);
                depth = depth - 1;
            end
        else
            % Non-leaf node processing
            npos = 2^depth - 1 + node + 1;

            if ns(npos) == 0
```

```
                % Step L - process left child
                [node, depth, LLR, ns] = processLeftChild(node,
                    ↪ depth, n, LLR, ns, npos, f);

            elseif ns(npos) == 1
                % Step R - process right child
                [node, depth, LLR, ns] = processRightChild(node,
                    ↪ depth, n, LLR, ucap, ns, npos, g, maxqr);

            else
                % Step U - combine and go to parent
                [node, depth, ucap] = processParentNode(node,
                    ↪ depth, n, ucap);
            end
        end
    end

    % Final decision with CRC check
    msg_capl = squeeze(ucap(:,n+1,infoBits));
    msg_cap = performCRCCheck(msg_capl, crcg, A);
end
```

7.9 Function: performCRCCheck

```
function msg_cap = performCRCCheck(msg_capl, crcg, A)
    % Perform CRC check on candidate messages
    if isempty(crcg)
        msg_cap = msg_capl(1, 1:A); % Just use the best path if no
            ↪ CRC
        return;
    end

    cout = 1; % Default to best path if no CRC passes
    for c1 = 1:size(msg_capl, 1)
        [~, rem] = gfdeconv(fliplr(msg_capl(c1,:)), crcg);
        if isequal(rem, 0)
            cout = c1;
            break;
        end
    end
    msg_cap = msg_capl(cout, 1:A); % Extract message bits (without
        ↪ CRC)
```

```
end
```

7.10 Function: processSNRPoint

```
function [BER, BLER] = processSNRPoint(Q1, N, K, crcL, F, infoBits
    ↪ , EbNodB, Rate, Nblocks, rmax, maxqr, nL, crcg)
    % Process a single SNR point
    EbNo = 10^(EbNodB / 10);
    sigma = sqrt(1 / (2 * Rate * EbNo));
    crcL = length(crcg) - 1;  % Degree of CRC polynomial
    A = K - crcL;
    Nbiterrs = 0;
    Nblockerrs = 0;

    parfor blk = 1:Nblocks
        [msg, msgcrc] = generateMessageWithCRC(A, crcL, crcg);
        cword = polarEncode(msgcrc, Q1, N, infoBits);
        rq = addNoiseAndQuantize(cword, sigma, rmax, maxqr);
        msg_cap = sclDecode(rq, F, infoBits, nL, crcg, N, rmax,
            ↪ maxqr,A);

        % Count errors
        bit_errors = sum(msg ~= msg_cap);
        Nbiterrs = Nbiterrs + bit_errors;
        if bit_errors > 0
            Nblockerrs = Nblockerrs + 1;
        end
    end

    BER = Nbiterrs / (A * Nblocks);
    BLER = Nblockerrs / Nblocks;
end
```

7.11 Function: runSCLSimulation

```
function [BER_sim, BLER_sim] = runSCLSimulation(Q, N, A, crcL,
    ↪ EbNodB_range, Nblocks, rmax, maxqr, nL)
    % Main simulation function for SCL decoder
    crcg = fliplr([1 1 1 0 0 0 1 0 0 0 0 1]); % CRC polynomial
    K = A + crcL;
    n = log2(N);
```

```
    Rate = A/N;

    Q1 = Q(Q<=N); % Reliability sequence for N
    F = Q1(1:N-K); % Frozen positions
    infoBits = Q1(N-K+1:end); % Information bits

    % Initialize results
    BER_sim = zeros(size(EbNodB_range));
    BLER_sim = zeros(size(EbNodB_range));

    % Simulation loop
    for idx = 1:length(EbNodB_range)
        EbNodB = EbNodB_range(idx);
        [BER_sim(idx), BLER_sim(idx)] = processSNRPoint(Q1, N, K,
            ↪ crcL, F, infoBits, EbNodB, Rate, Nblocks, rmax,
            ↪ maxqr, nL, crcg);
        disp([EbNodB BER_sim(idx)]);
    end
    BER_SCL=BER_sim;
    BLER_SCL=BLER_sim;
    assignin('base', 'BER_SCL', BER_SCL);
    assignin('base', 'BLER_SCL', BER_SCL);
    % Plot results
    plotSCLResults(EbNodB_range, BER_sim, BLER_sim, N, K, Rate);
end
```

7.12 Function: plotSCLResults

```
function plotSCLResults(EbNodB_range, BER_sim, BLER_sim, N, K,
    ↪ Rate)
    % Plot SCL decoder results
    figure;
    semilogy(EbNodB_range, BER_sim, 'o-', 'LineWidth', 2);
    grid on;
    xlabel('Eb/N0 (dB)');
    ylabel('Bit Error Rate (BER)');
    title(sprintf('Polar Code SCL Decoder: N = %d, K = %d, Rate =
        ↪ %.2f', N, K, Rate));

    figure;
    semilogy(EbNodB_range, 1 - BLER_sim, 's-', 'LineWidth', 2);
    grid on;
```

```
    xlabel('Eb/N0 (dB)');
    ylabel('Probability of Successful Decoding');
    title(sprintf('Success Probability: N = %d, K = %d, Rate = %.2
        ↪ f', N, K, Rate));
end
```

7.13 Function: main_SCL

```
Q=[0 1 2 4 8 16 32 3 5 64 9 6 17 10 18 128 12 33 65 20 256 34 24
    ↪ 36 7 129 66 512 11 40 68 130 ...
    19 13 48 14 72 257 21 132 35 258 26 513 80 37 25 22 136 260
        ↪ 264 38 514 96 67 41 144 28 69 42 ...
    516 49 74 272 160 520 288 528 192 544 70 44 131 81 50 73 15
        ↪ 320 133 52 23 134 384 76 137 82 56 27 ...
    97 39 259 84 138 145 261 29 43 98 515 88 140 30 146 71 262 265
        ↪  161 576 45 100 640 51 148 46 75 266 273 517 104 162 ...
    53 193 152 77 164 768 268 274 518 54 83 57 521 112 135 78 289
        ↪ 194 85 276 522 58 168 139 99 86 60 280 89 290 529 524
        ↪ ...
    196 141 101 147 176 142 530 321 31 200 90 545 292 322 532 263
        ↪ 149 102 105 304 296 163 92 47 267 385 546 324 208 386
        ↪ 150 153 ...
    165 106 55 328 536 577 548 113 154 79 269 108 578 224 166 519
        ↪ 552 195 270 641 523 275 580 291 59 169 560 114 277 156
        ↪ 87 197 ...
    116 170 61 531 525 642 281 278 526 177 293 388 91 584 769 198
        ↪ 172 120 201 336 62 282 143 103 178 294 93 644 202 592
        ↪ 323 392 ...
    297 770 107 180 151 209 284 648 94 204 298 400 608 352 325 533
        ↪  155 210 305 547 300 109 184 534 537 115 167 225 326 306
        ↪  772 157 ...
    656 329 110 117 212 171 776 330 226 549 538 387 308 216 416
        ↪ 271 279 158 337 550 672 118 332 579 540 389 173 121 553
        ↪ 199 784 179 ...
    228 338 312 704 390 174 554 581 393 283 122 448 353 561 203 63
        ↪  340 394 527 582 556 181 295 285 232 124 205 182 643 562
        ↪  286 585 ...
    299 354 211 401 185 396 344 586 645 593 535 240 206 95 327 564
        ↪  800 402 356 307 301 417 213 568 832 588 186 646 404 227
        ↪  896 594 ...
    418 302 649 771 360 539 111 331 214 309 188 449 217 408 609
        ↪ 596 551 650 229 159 420 310 541 773 610 657 333 119 600
```

```
    ↪ 339 218 368 ...
652 230 391 313 450 542 334 233 555 774 175 123 658 612 341
    ↪ 777 220 314 424 395 673 583 355 287 183 234 125 557 660
    ↪ 616 342 316 ...
241 778 563 345 452 397 403 207 674 558 785 432 357 187 236
    ↪ 664 624 587 780 705 126 242 565 398 346 456 358 405 303
    ↪ 569 244 595 ...
189 566 676 361 706 589 215 786 647 348 419 406 464 680 801
    ↪ 362 590 409 570 788 597 572 219 311 708 598 601 651 421
    ↪ 792 802 611 ...
602 410 231 688 653 248 369 190 364 654 659 335 480 315 221
    ↪ 370 613 422 425 451 614 543 235 412 343 372 775 317 222
    ↪ 426 453 237 ...
559 833 804 712 834 661 808 779 617 604 433 720 816 836 347
    ↪ 897 243 662 454 318 675 618 898 781 376 428 665 736 567
    ↪ 840 625 238 ...
359 457 399 787 591 678 434 677 349 245 458 666 620 363 127
    ↪ 191 782 407 436 626 571 465 681 246 707 350 599 668 790
    ↪ 460 249 682 ...
573 411 803 789 709 365 440 628 689 374 423 466 793 250 371
    ↪ 481 574 413 603 366 468 655 900 805 615 684 710 429 794
    ↪ 252 373 605 ...
848 690 713 632 482 806 427 904 414 223 663 692 835 619 472
    ↪ 455 796 809 714 721 837 716 864 810 606 912 722 696 377
    ↪ 435 817 319 ...
621 812 484 430 838 667 488 239 378 459 622 627 437 380 818
    ↪ 461 496 669 679 724 841 629 351 467 438 737 251 462 442
    ↪ 441 469 247 ...
683 842 738 899 670 783 849 820 728 928 791 367 901 630 685
    ↪ 844 633 711 253 691 824 902 686 740 850 375 444 470 483
    ↪ 415 485 905 ...
795 473 634 744 852 960 865 693 797 906 715 807 474 636 694
    ↪ 254 717 575 913 798 811 379 697 431 607 489 866 723 486
    ↪ 908 718 813 ...
476 856 839 725 698 914 752 868 819 814 439 929 490 623 671
    ↪ 739 916 463 843 381 497 930 821 726 961 872 492 631 729
    ↪ 700 443 741 ...
845 920 382 822 851 730 498 880 742 445 471 635 932 687 903
    ↪ 825 500 846 745 826 732 446 962 936 475 853 867 637 907
    ↪ 487 695 746 ...
828 753 854 857 504 799 255 964 909 719 477 915 638 748 944
    ↪ 869 491 699 754 858 478 968 383 910 815 976 870 917 727
    ↪ 493 873 701 ...
```

```
    931 756 860 499 731 823 922 874 918 502 933 743 760 881 494
      ↪ 702 921 501 876 847 992 447 733 827 934 882 937 963 747
      ↪ 505 855 924 ...
    734 829 965 938 884 506 749 945 966 755 859 940 830 911 871
      ↪ 639 888 479 946 750 969 508 861 757 970 919 875 862 758
      ↪ 948 977 923 ...
    972 761 877 952 495 703 935 978 883 762 503 925 878 735 993
      ↪ 885 939 994 980 926 764 941 967 886 831 947 507 889 984
      ↪ 751 942 996 ...
    971 890 509 949 973 1000 892 950 863 759 1008 510 979 953 763
      ↪ 974 954 879 981 982 927 995 765 956 887 985 997 986 943
      ↪ 891 998 766 ...
    511 988 1001 951 1002 893 975 894 1009 955 1004 1010 957 983
      ↪ 958 987 1012 999 1016 767 989 1003 990 1005 959 1011
      ↪ 1013 895 1006 1014 1017 1018 ...
    991 1020 1007 1015 1019 1021 1022 1023]+1;
N = 32;
A = 5; crcL = 11;
EbNodB_range = 0:0.5:10;
Nblocks = 10000;
rmax = 3; % max received value
maxqr = 31; % max integer received value
nL = 4; % list size
K=A+crcL;
[BER_sim, BLER_sim] = runSCLSimulation(Q, N, A, crcL, EbNodB_range
  ↪ , Nblocks, rmax, maxqr, nL);
```

8 Simulation Result

8.1 SC Simulation Result(r=0.25)

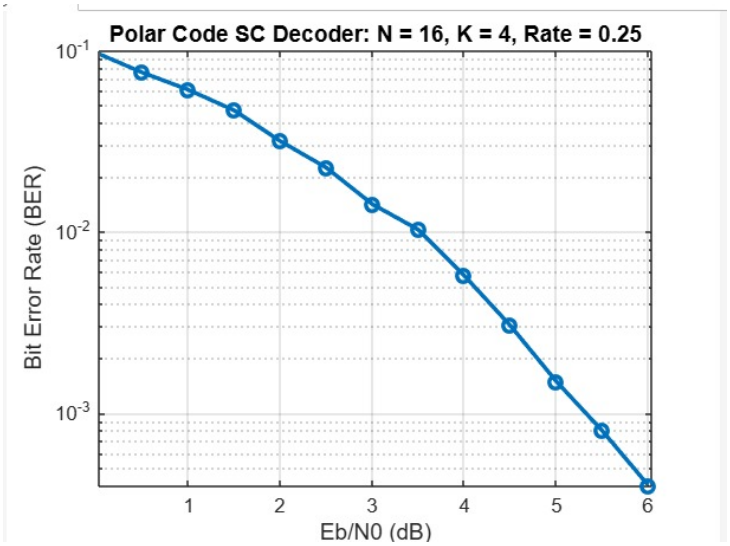


Figure 2: BER of SC Decoder for rate=0.25

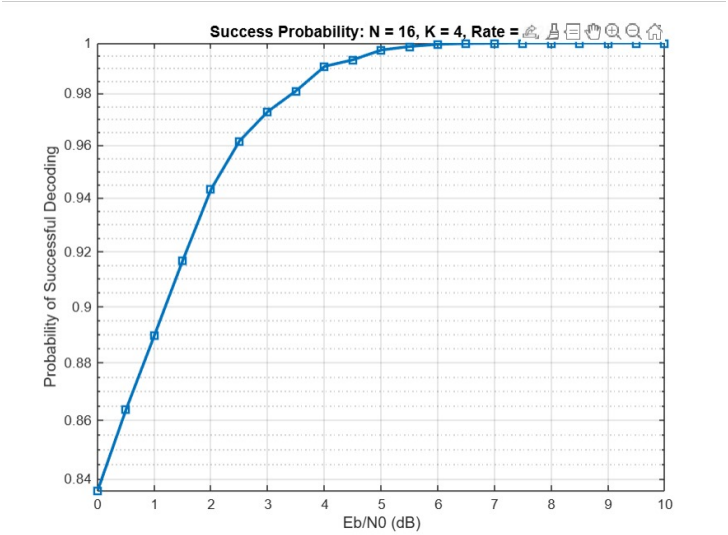


Figure 3: Performance of SC decoder for rate=0.25

8.2 SCL Simulation Result(r=0.5)

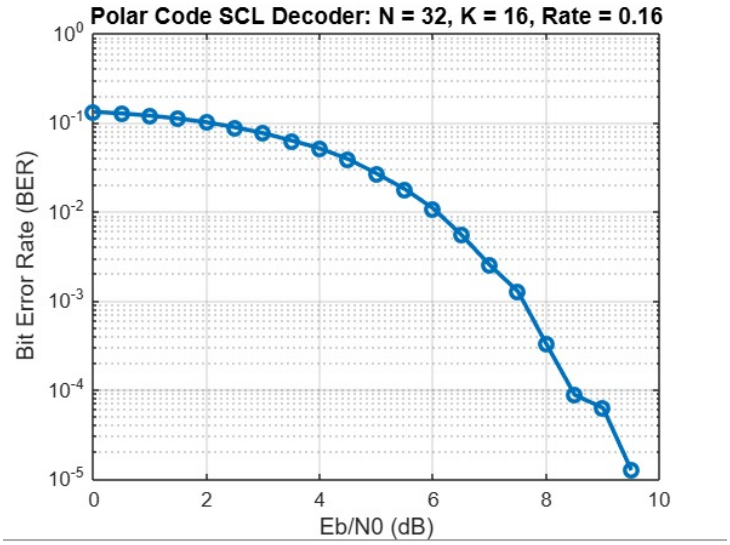


Figure 4: BER of SCL decoder for rate=0.5

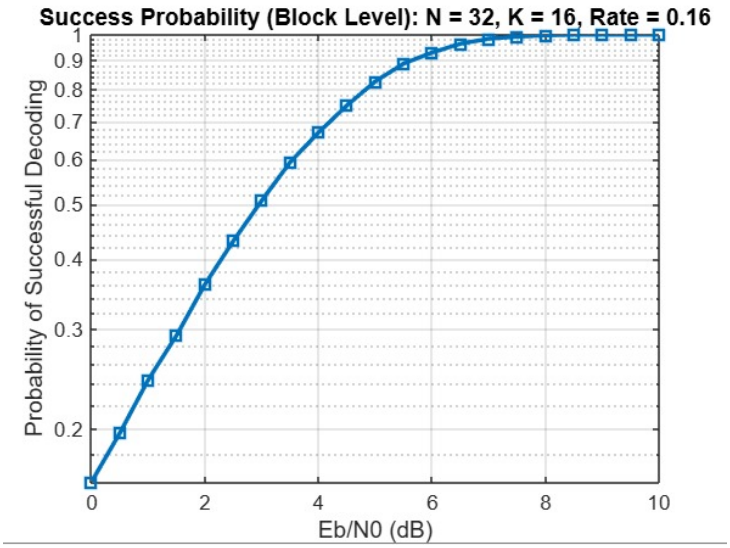


Figure 5: Performance of SCL decoder for rate=0.5

8.3 SCL Simulation Result(r=0.625)

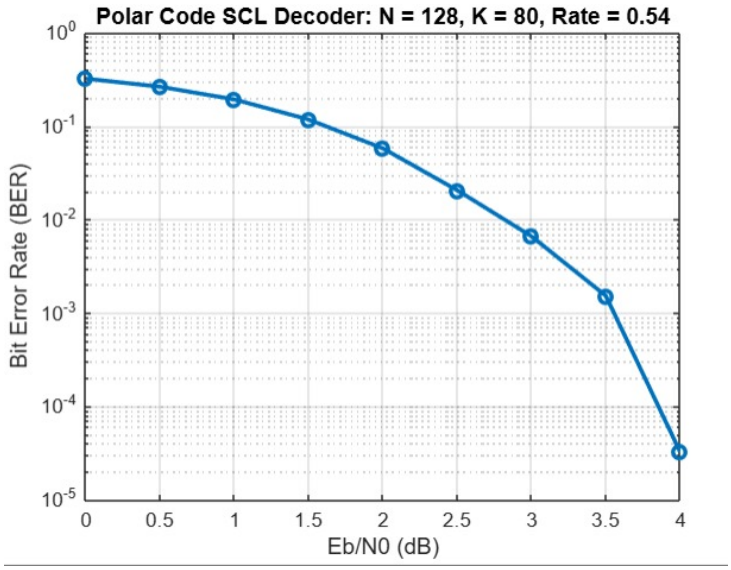


Figure 6: BER of SCL decoder for rate=0.625

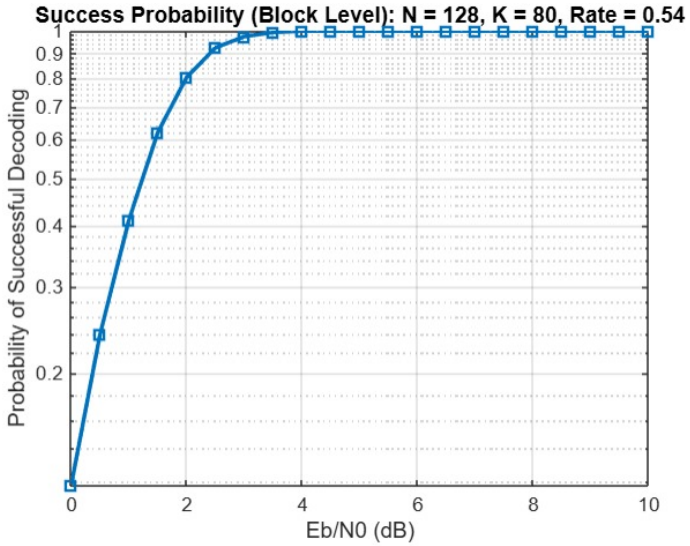


Figure 7: Performance of SCL decoder for rate=0.625

8.4 Comparison Simulation result(r=0,5)

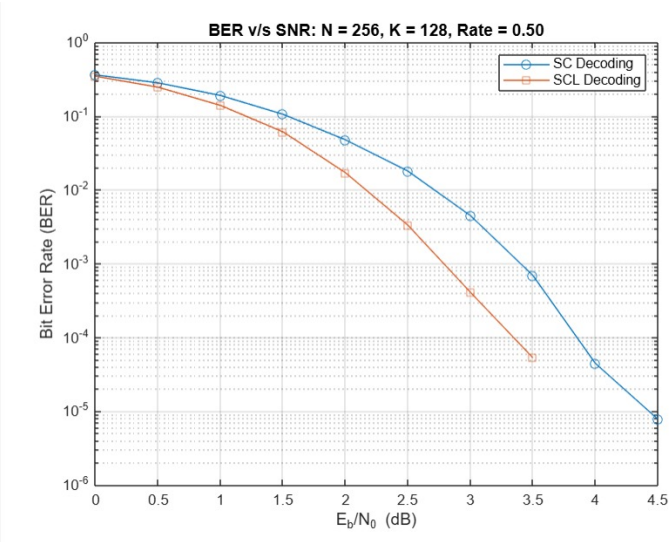


Figure 8: BER Comparison graph for rate=0,5

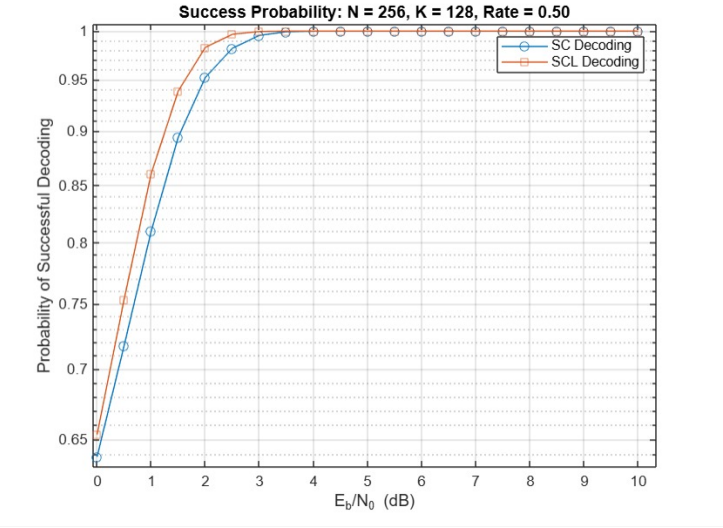


Figure 9: Performance Comparison graph for rate=0,5

References

[1] J. Smith and A. Kumar. Fundamentals of Channel Coding: Polar Codes and Beyond. University Press, 2023.

[2] L. Jones. Advances in Polar Coding: Achieving Shannon's Capacity. Journal of Communication Theory, 45(3):112–130, 2022.

[3] E. Arıkan, *The Flesh of Polar Codes by Emre Telatar*,
https://youtu.be/VhyoZSB9g0w?si=mIbTlIK27XiL7tko