Project BitSync

Mathematics Club, IIT Madras

Achintya Raghavan EE23B189

Atharva Moghe CS23B096

Pratyaksh Jain EE23B058

Aayush Lal CE24B033

Ananth Madhav V. EE24B088

Arjun Arunachalam ME24B088

Eppa Laxmi Narasimha Reddy CS24B074

Hemanth M. EE24B024

Himanshu Gupta ME24B109

Nithin G. EE24B046

Sai Ram Y. CE24B104

Smitali Bhandari CE24B119

Taarun A. EE24B069

Contents

1 Introduction

§1.1 Foundations of Probability Theory

§1.1.1 Random Variable

A **Random Variable** X is a function from a sample space Ω to the real numbers. It maps outcomes of a random experiment to numerical values.

Discrete random variable:

$$X : \Omega \to \{x_1, x_2, \ldots, x_n\}$$

with probability mass function (pmf) $P(X = x_i) = p_i$ such that:

$$\sum_i p_i = 1$$

§1.1.2 Entropy

For any event which occurs with a probability $P(x)$, The surprise is defined as

$$\log\left(\frac{1}{P(x)}\right)$$

Entropy is the expected surprise associated with a Random Variable.

$$H(X) = \sum_{x\in\mathcal{X}} P(x)\log\left(\frac{1}{P(x)}\right)$$

The **Entropy** of a random variable X measures the average uncertainty or information content:

$$H(X) = -\sum_{x\in\mathcal{X}} P(x)\log_2 P(x)$$

The 2 comes here just to represent the bits0,1. Here base of log is 2. Therefore $H(X)$ is usually represented in bits.

If all outcomes are equally likely, entropy is maximized:

$$H(X) = \log_2 |\mathcal{X}|$$

§1.1.3 Joint Entropy

For two random variables X and Y, the joint entropy is:

$$H(X,Y) = -\sum_{x,y} P(x,y)\log_2 P(x,y)$$

This quantifies the uncertainty of the pair (X, Y).

§1.1.4 Conditional Entropy

The **Conditional Entropy** $H(Y|X)$ measures the average uncertainty of Y given X is known:

$$H(Y|X) = -\sum_{x,y} P(x,y)\log_2 P(y|x)$$

§1.1.5 Mutual Information

The **Mutual Information** between X and Y quantifies how much knowing one of the variables reduces uncertainty of the other:

$$I(X;Y) = H(X) - H(X|Y) = H(Y) - H(Y|X)$$
$$= \sum_{x,y} P(x,y) \log_2 \left(\frac{P(x,y)}{P(x)P(y)} \right)$$

§1.1.6 Relative Entropy (KL Divergence)

The **Kullback–Leibler (KL) Divergence** measures how one distribution P diverges from a second Q:

$$D_{\mathrm{KL}}(P\|Q) = \sum_{x\in\mathcal{X}} P(x) \log_2 \left(\frac{P(x)}{Q(x)} \right)$$

It is always non-negative and equals 0 iff $P = Q$. It signifies how far two distributions are, but it is not a metric as it is not symmetric.

§1.1.7 Chain Rule of Entropy

Entropy satisfies the chain rule:

$$H(X,Y) = H(X) + H(Y|X)$$

§1.1.8 Key Inequalities

- $H(X) \geq 0$
- $H(Y|X) \leq H(Y)$
- $I(X;Y) \geq 0$
- $D_{\mathrm{KL}}(P\|Q) \geq 0$

§1.2 Bit Error Rate (BER) – A Brief Overview

§1.2.1 Definition

The **Bit Error Rate (BER)** is the probability that a bit is incorrectly received due to noise, interference, or other impairments in a communication channel.

$$\mathrm{BER} = \frac{N_e}{N_t}$$

where:

- N_e: Number of bit errors
- N_t: Total number of bits transmitted

§1.2.2 Theoretical BER

For a Binary Symmetric Channel (BSC) with crossover probability p, the theoretical BER is:

$$\mathrm{BER} = p$$

In AWGN channels using BPSK modulation:

$$\text{BER} = 1 - Q\left(\sqrt{\frac{E_b}{N_0}}\right)$$

where:

- $Q(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$
- $\frac{E_b}{N_0}$: Energy per bit to noise power spectral density ratio

§1.2.3 Practical Use

BER is a key performance metric in:

- Wireless communications
- Optical fiber systems
- Error-correcting code performance testing

§1.2.4 Empirical Estimation

If simulation is used:

$$\text{BER}_{\text{sim}} = \frac{\text{Number of incorrect bits}}{\text{Total transmitted bits}}$$

§1.3 Signal-to-Noise Ratio (SNR) – A Brief Overview

§1.3.1 Definition

The **Signal-to-Noise Ratio (SNR)** is a measure that compares the power of a signal to the power of background noise:

$$\text{SNR} = \frac{P_{\text{signal}}}{P_{\text{noise}}}$$

It is often expressed in decibels (dB):

$$\text{SNR}_{\text{dB}} = 10 \log_{10}\left(\frac{P_{\text{signal}}}{P_{\text{noise}}}\right)$$

§1.3.2 Importance

- Higher SNR means better signal quality and lower error rates.
- SNR is critical in evaluating communication system performance, such as BER and capacity.

§1.3.3 Related Quantity: E_b/N_0

For digital communications, the bit-level SNR is often represented as:

$$\frac{E_b}{N_0} = \frac{\text{Energy per bit}}{\text{Noise power spectral density}}$$

It is related to SNR through:

$$\frac{E_b}{N_0} = \frac{\text{SNR}}{R}$$

where R is the bit rate in bits per second per Hz (code rate or bandwidth efficiency).

§1.4 Mutual Information: A Detailed Overview

§1.4.1 Introduction

Mutual Information (MI) quantifies the amount of information obtained about one random variable through another. It measures the reduction in uncertainty about one variable given knowledge of the other.

§1.4.2 Definition

Let X and Y be two discrete random variables with joint distribution $P(x, y)$ and marginal distributions $P(x)$, $P(y)$. Then the mutual information between X and Y is:

$$I(X;Y) = \sum_{x\in\mathcal{X}} \sum_{y\in\mathcal{Y}} P(x,y) \log_2 \left(\frac{P(x,y)}{P(x)P(y)} \right)$$

Interpretation: MI is 0 if and only if X and Y are independent, i.e., $P(x, y) = P(x)P(y)$.

§1.4.3 Relationship to Entropy

Mutual information has several equivalent expressions using entropy:

$$\begin{aligned} I(X;Y) &= H(X) - H(X|Y) \\ &= H(Y) - H(Y|X) \\ &= H(X) + H(Y) - H(X,Y) \end{aligned}$$

where:

- $H(X)$: Entropy of X
- $H(Y)$: Entropy of Y
- $H(X,Y)$: Joint entropy
- $H(X|Y)$: Conditional entropy

§1.4.4 Intuition

- If X and Y are completely independent, knowing one tells nothing about the other $\rightarrow I(X;Y) = 0$. - If X determines Y exactly, then all uncertainty in Y is resolved by knowing X, and $I(X;Y) = H(Y)$.

§1.4.5 Example

Suppose X and Y have the following joint distribution:

$P(X,Y)$	$Y=0$	$Y=1$
$X=0$	0.25	0.25
$X=1$	0.25	0.25

Here, $P(x) = 0.5$, $P(y) = 0.5$, and $P(x,y) = P(x)P(y)$ for all x, y, so:

$$I(X;Y) = \sum_{x,y} 0.25 \cdot \log_2 \left(\frac{0.25}{0.5 \cdot 0.5} \right) = \sum_{x,y} 0.25 \cdot \log_2(1) = 0$$

This shows that X and Y are independent.

§1.4.6 Properties of Mutual Information

- $I(X;Y) \geq 0$
- Symmetry: $I(X;Y) = I(Y;X)$
- $I(X;Y) = 0$ iff $X \perp Y$
- Invariant to bijective transformations: $I(f(X);g(Y)) = I(X;Y)$

§1.4.7 Venn Diagram of Entropy and Mutual Information

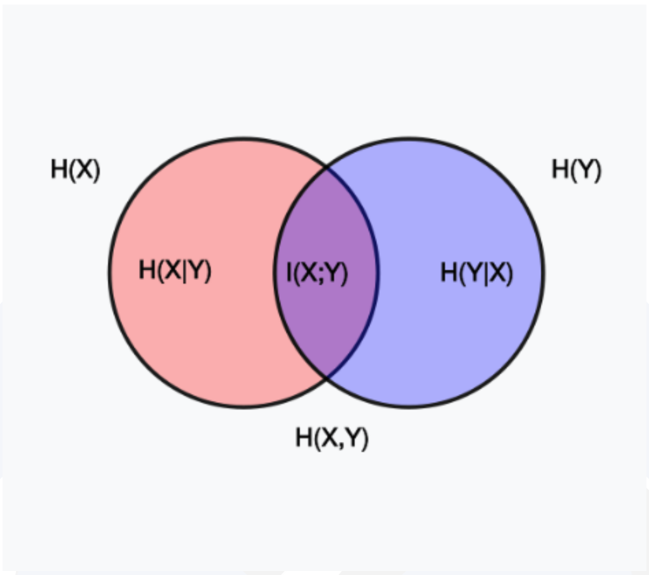


Figure 1.1: Venn Diagram

§1.4.8 Mutual Information in Communication

In information theory, mutual information quantifies how much information the output of a channel reveals about its input. The channel capacity is defined as the maximum mutual information achievable over all possible input distributions. The **Channel Capacity** C is:

$$C = \max_{P(x)} I(X;Y)$$

This is the maximum mutual information over all input distributions.

§1.5 Shannon's Channel Capacity

§1.5.1 Introduction

Shannon's channel capacity is a fundamental limit in information theory. It defines the **maximum reliable data rate** at which information can be transmitted over a communication channel with arbitrarily small error.

§1.5.2 Shannon's Noisy Channel Coding Theorem

Theorem: For a given communication channel with capacity C (in bits per second), any data rate $R < C$ is achievable with an arbitrarily small probability of error, using sufficiently long and well-designed codes.

$$\boxed{R < C \Rightarrow \text{Reliable Communication is Possible}}$$

§1.5.3 Shannon Capacity for AWGN Channel

For the Additive White Gaussian Noise (AWGN) channel with bandwidth B and signal-to-noise ratio $\frac{S}{N}$, the capacity is given by:

$$C = B \log_2 \left(1 + \frac{S}{N}\right) \quad \text{(bits per second)}$$

If $B = 1$ Hz (normalized bandwidth), then:

$$C = \log_2 \left(1 + \frac{S}{N}\right) \quad \text{(bits per channel use)}$$

§1.5.4 Interpretation

- Shannon capacity sets a theoretical limit—no code can perform better.
- Codes like Turbo, LDPC, and Polar Codes aim to approach this limit.
- The closer a code operates to capacity, the more efficient it is.

§1.5.5 Significance in Coding Theory

- Provides a benchmark to evaluate and compare error-correcting codes.
- Motivates the need for structured codes that can achieve near-capacity performance (e.g., LDPC, Polar).
- Guides system designers in choosing modulation, coding rates, and power levels.

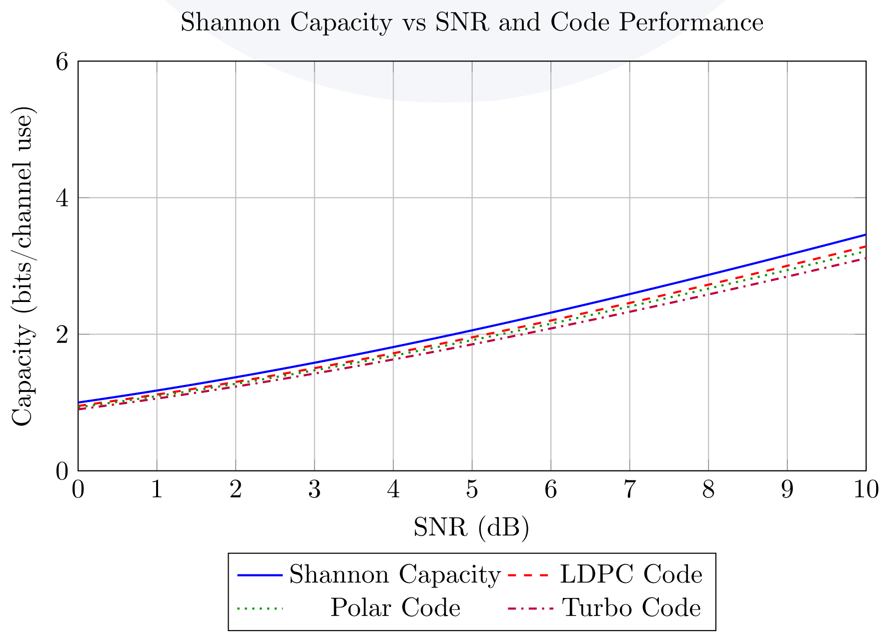
§1.5.6 Example

If a channel has $S/N = 15$ (i.e., 11.76 dB), then:

$$C = \log_2(1 + 15) \approx \log_2(16) = 4 \text{ bits/use}$$

This means we can reliably send 4 bits per symbol over this channel with appropriate coding.

§1.5.7 Channel Capacity vs SNR (in dB)



§1.5.8 Comparison of Coding Schemes vs Shannon Limit

Coding Scheme	**Closeness to Capacity**	**Complexity**	**Decoding Method**
LDPC	~95%	Moderate	Belief Propagation
Polar	~93%	Low–Moderate	Successive Cancellation (or list)
Turbo	~90%	High	Iterative Decoding
Shannon Limit	100% (Ideal)	Theoretical	–

Table 1.1: Comparison of Modern Codes vs Shannon Capacity

2 Matrices & Graphs

§2.1 Generator Matrix in Channel Coding

§2.1.1 Introduction

In linear block codes, the **generator matrix** G is a key component for encoding. It linearly maps a message vector u of length k to a codeword x of length n as:

$$x = uG$$

This process ensures that the resulting codeword belongs to a valid codebook, adhering to the code's constraints.

Why Do We Use Generator Matrices?

The generator matrix allows:

- Efficient and systematic encoding: Encoders can be implemented as simple matrix multipliers.
- Algebraic structure: Each row of G is a basis vector of the code space.
- Ensures codewords satisfy constraints defined by the code.

Different coding schemes use specially structured G matrices to optimize performance under noise, decoding complexity, and error detection/correction capability.

§2.1.2 LDPC Codes: Generator Matrix from Parity-Check Equations

LDPC (Low-Density Parity-Check) codes are defined primarily by their sparse parity-check matrix H, where each row represents a parity-check equation.

Purpose of Generator Matrix in LDPC

Here, multiplication and addition are done modulo 2, equivalently, we could also consider XOR's in place of addition. While H is used for decoding (e.g., belief propagation), G is required for encoding. G must generate only those codewords x that satisfy:

$$Hx^T = 0 \quad \text{or equivalently} \quad xH^T = 0, \quad \text{as} \quad x = uG \quad \implies uGH^T = 0$$

$$\text{as u could be any message vector} \implies GH^T = 0 \implies HG^T = 0$$

Hence, rows of G are vectors from null space of H.

Constructing G from H

Given H of size $(n-k) \times n$, we want a G of size $k \times n$ such that $GH^T = 0$.

Steps:

1. Use Gaussian elimination on H to bring it to systematic form:

$$H = [P \mid I_{n-k}]$$

2. Then, construct the generator matrix as:

$$G = [I_k \mid P^T]$$

This ensures all codewords satisfy the parity constraints.

Example:

Suppose we have parity-check equations:

$$\begin{aligned} x_1 \oplus x_3 \oplus x_4 = 0 \\ x_2 \oplus x_3 = 0 \end{aligned} \Rightarrow H = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Using row operations, convert H into:

$$H = [P \mid I] = \begin{bmatrix} 1 & 1 & \mid & 0 & 1 \\ 0 & 1 & \mid & 1 & 0 \end{bmatrix} \Rightarrow G = \left[I \mid P^T\right] = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

§2.1.3 Polar Codes: Generator Matrix via Binary Tree Construction

Polar codes are based on channel polarization. The generator matrix is constructed using the Kronecker power of a basic kernel.

Purpose of Generator Matrix in Polar Codes

The generator matrix causes channels to polarize—some become nearly perfect, others nearly useless. The matrix structure enables this polarization by recursively combining bits.

Recursive Construction of G_n

Start with:

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

Then,

$$G_n = F^{\otimes m}, \quad \text{where } n = 2^m$$

Example:

$$G_2 = F = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, \quad G_4 = F \otimes F = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Binary Tree View of Polar Encoding

Consider $n = 4$ input bits u_0, u_1, u_2, u_3 arranged in a binary tree:

$$\begin{tikzcd}[column sep=small] & & & u_0 \oplus u_1 \oplus u_2 \oplus u_3, u_1 \oplus u_3, u_2 \oplus u_3, u_3 \arrow[dll, no head] \arrow[drr, no head] & & & \\ & u_0 \oplus u_1, u_1 \arrow[dl, no head] \arrow[dr, no head] & & & & u_2 \oplus u_3, u_3 \arrow[dl, no head] \arrow[dr, no head] & \\ u_0 & & u_1 & & u_2 & & u_3 \end{tikzcd}$$

This structure mirrors the bit-combining logic in the matrix multiplication. At each level, bits are XORed and passed down recursively. This results in the encoded bits $x = uG_n$ being highly structured to enable successive cancellation decoding.

Final Generator Matrix for Information Set

Let $\mathcal{A} \subset \{0, 1, ..., n-1\}$ be the indices of reliable channels. Then:

$$G = \text{rows of } G_n \text{ corresponding to } \mathcal{A}$$

Example: If $\mathcal{A} = \{1, 3\}$:

$$G = \begin{bmatrix} G_n[1,:] \\ G_n[3,:] \end{bmatrix}$$

§2.2 Parity-Check Matrix in Channel Coding

§2.2.1 Introduction

The **parity-check matrix** H is used to define and validate codewords in a linear block code. A vector x is a valid codeword if and only if:

$$Hx^T = 0$$

§2.2.2 Why Do We Use a Parity-Check Matrix?

The parity-check matrix serves key purposes:

- It specifies the set of valid codewords (those that satisfy the parity equations).
- It is the backbone of most decoding algorithms, especially for LDPC codes (e.g., belief propagation).
- It enables syndrome decoding: for a received vector r, the syndrome $s = Hr^T$ detects and helps correct errors.

§2.2.3 Relation to Generator Matrix

For a linear (n, k) block code:

- G is a $k \times n$ generator matrix.
- H is an $(n-k) \times n$ parity-check matrix.
- They satisfy the orthogonality condition:
$$GH^T = 0$$

§2.2.4 Construction from Parity Equations

Each row of H represents a parity-check equation (i.e., a linear constraint on bits). For example:

$$x_1 \oplus x_3 \oplus x_4 = 0 \Rightarrow \text{row in } H = [1\ 0\ 1\ 1]$$

Multiple equations can be stacked row-wise to form H.

§2.2.5 LDPC Codes: Sparse H Matrix

In LDPC (Low-Density Parity-Check) codes:

- H is a sparse matrix with mostly 0s and few 1s.
- Each row corresponds to a low-weight parity-check equation.
- Each column corresponds to a code bit and appears in only a few parity checks.

Example:

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

This structure is chosen to allow iterative message-passing decoding on a bipartite (Tanner) graph.

§2.2.6 Polar Codes: Implicit Parity Checks

Polar codes do not use an explicit H matrix in practice. However:

- The frozen bits act as parity-check constraints (e.g., $u_i = 0$ for $i \notin \mathcal{A}$, where $\mathcal{A}$ is the set of good created channels that we would use for sending the message bits).
- These constraints, along with the generator matrix structure, ensure valid codewords.
- The decoding algorithm (successive cancellation) inherently checks consistency.

In summary, while LDPC uses an explicit H for decoding, Polar codes enforce parity through bit freezing and recursive encoding logic.

§2.2.7 Tanner Graph Representation of LDPC Codes

A **Tanner graph** is a bipartite graphical representation of an LDPC code that shows the relationship between code bits and parity-check equations.

Why Use a Tanner Graph?

- It provides a visual and structural way to understand LDPC codes.
- It is the foundation for iterative decoding algorithms such as belief propagation.
- Helps track message flow between bits and checks during decoding.

Structure

The Tanner graph is a bipartite graph with two types of nodes:

- **Variable nodes (bit nodes)**: Represent code bits $x_1, x_2, \ldots, x_n$.
- **Check nodes**: Represent parity-check constraints (rows of H).

An edge exists between a variable node x_j and a check node c_i if and only if $H_{i,j} = 1$.

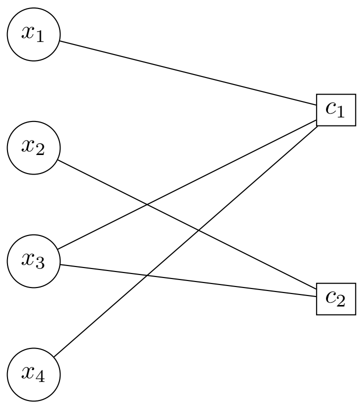
Example

Given an LDPC parity-check matrix:

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

The Tanner graph has:

- 4 variable nodes: x_1 to x_4
- 2 check nodes: c_1, c_2
- Edges:
  - x_1, x_3, and x_4 connect to c_1
  - x_2 and x_3 connect to c_2



Use in Decoding

During belief propagation:

- Variable nodes send messages (beliefs) to check nodes.
- Check nodes send back updated messages based on parity constraints.
- This process iterates to converge on likely bit values.

Tanner graphs enable low-complexity, parallelizable decoding even for long codes, making LDPC codes highly efficient for modern communications.

3 Channels

§3.1 Binary Phase Shift Keying (BPSK)

§3.1.1 Introduction

Binary Phase Shift Keying (BPSK) is the simplest form of Phase Shift Keying. It encodes **1 bit per symbol** by varying the **phase of a carrier signal**.

§3.1.2 Signal Representation

BPSK maps binary data as:

$$\text{Bit } 0 \rightarrow s_0(t) = \sqrt{2P}\cos(2\pi f_c t)$$

$$\text{Bit } 1 \rightarrow s_1(t) = -\sqrt{2P}\cos(2\pi f_c t)$$

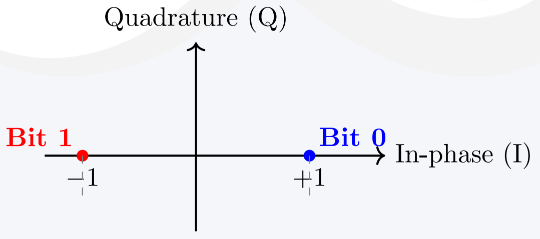
This is equivalent to a phase shift of π radians between the two signals:

$$s_1(t) = -s_0(t)$$

§3.1.3 Constellation Diagram

The BPSK constellation lies on the real axis:

$$0 \rightarrow +1, \quad 1 \rightarrow -1$$



§3.1.4 Bit Error Probability in AWGN

Under an AWGN channel with noise variance σ^2 and energy per bit E_b, the bit error rate (BER) for BPSK is:

$$P_b = 1 - Q\left(\sqrt{\frac{2E_b}{N_0}}\right)$$

where $Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^{\infty} e^{-t^2/2} dt$

§3.1.5 Features

- **Bandwidth Efficient:** Uses minimal bandwidth for 1 bit/symbol.
- **Power Efficient:** Performs well at low SNR.
- **Simple Implementation:** Easy to generate and decode.

§3.1.6 Applications

Used in:

- Satellite communication
- RFID and remote sensing
- 802.11 (Wi-Fi) legacy systems

§3.2 Additive White Gaussian Noise (AWGN)

§3.2.1 What is AWGN?

AWGN stands for **Additive White Gaussian Noise**. It is the most widely used noise model in communication systems and represents random noise affecting transmitted signals.

- **Additive:** The noise is added directly to the signal.
- **White:** The noise has equal power across all frequencies (flat power spectral density).
- **Gaussian:** The noise amplitude follows a Gaussian (normal) distribution.

§3.2.2 Mathematical Model

If $x(t)$ is the transmitted signal, the received signal $y(t)$ is modeled as:

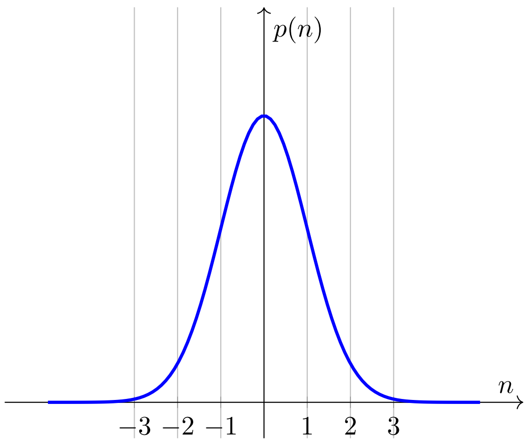
$$y(t) = x(t) + n(t)$$

where $n(t)$ is Gaussian noise with:

- Zero mean: $\mathbb{E}[n(t)] = 0$
- Variance: $\mathbb{E}[n^2(t)] = \sigma^2$
- Probability density:

$$p(n) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-n^2/(2\sigma^2)}$$

Gaussian PDF of AWGN ($\mu = 0$, $\sigma = 1$)



§3.2.3 Why Use the AWGN Model?

- It is a simple yet powerful model to test the performance of modulation and coding schemes.
- It approximates many real-world noise sources in electronic and wireless systems.
- It forms the basis for defining Shannon capacity:

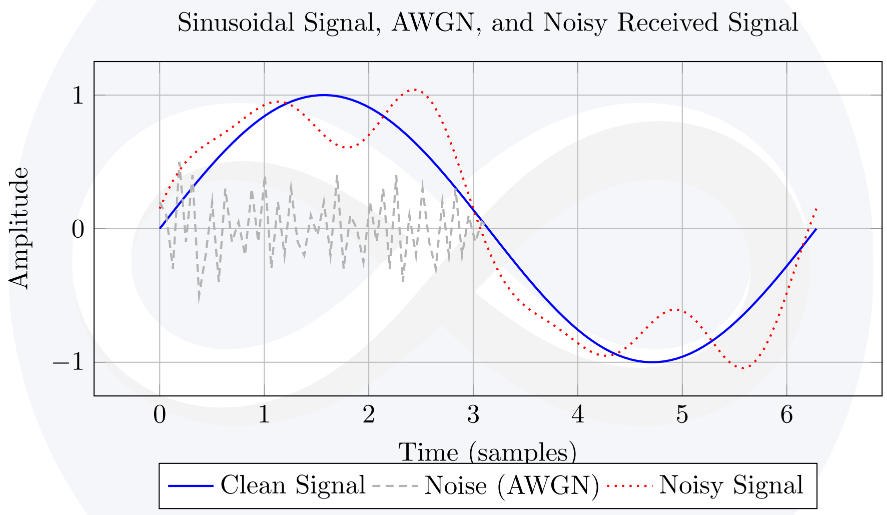
$$C = \log_2\left(1 + \frac{S}{N}\right)$$

§3.2.4 Application

AWGN is used to:

- Simulate realistic communication environments.
- Analyze and compare error-correcting codes (e.g., LDPC, Polar).
- Benchmark theoretical limits in channel capacity and SNR.

§3.2.5 Interpretation of AWGN on a signal



§3.3 Binary Erasure Channel (BEC)

§3.3.1 Introduction

The **Binary Erasure Channel (BEC)** is a type of discrete memoryless channel used to model communication systems where transmitted bits may be **erased** during transmission. In this model, each transmitted bit is either received correctly or erased.

§3.3.2 Channel Model

In the BEC, the transmitted bit can be:

$$\text{Bit } 0 \rightarrow \text{Received as } 0$$

$$\text{Bit } 1 \rightarrow \text{Received as } 1$$

$$\text{Bit } 0 \text{ or } 1 \rightarrow \text{Erased (denoted by } \epsilon)$$

The probability of erasure is denoted by p, while the probability of correctly receiving the bit is $1 - p$.

§3.3.3 Error and Erasure

Erasure occurs when the receiver does not receive any information about the transmitted bit. This can happen due to channel impairments or errors in the transmission. If the bit is received (not erased), it is either a 0 or a 1, depending on the transmission.

§3.3.4 Channel Capacity

The capacity of the BEC is given by:

$$C = 1 - p$$

Where p is the probability that the transmitted bit is erased. If $p = 0$, there are no erasures, and the channel capacity is 1. If $p = 1$, every transmitted bit is erased, and the channel capacity becomes 0.

§3.3.5 Use Case

BEC is useful for modeling channels with high reliability, where errors are rare, but information loss due to erasure (e.g., dropped packets in data transmission) is more significant. It is commonly used in:

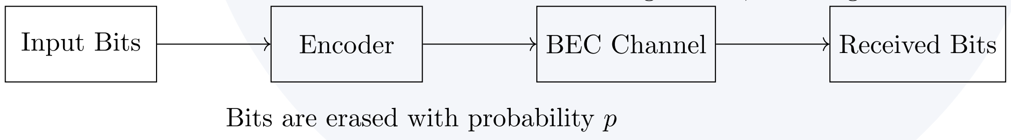
- Error correction codes like **LDPC** and **Turbo Codes**.
- Digital communication systems where packets may be dropped due to congestion or signal fading.
- Applications such as reliable data transmission, network protocols, and wireless communication.

§3.3.6 Error Probability

For a BEC with erasure probability p, the probability of error for a bit that is received is:

$$P_e = p$$

This means that the receiver has a chance of not receiving the bit, resulting in an erasure.



§3.4 Binary Symmetric Channel (BSC)

§3.4.1 Introduction

The Binary Symmetric Channel (BSC) is a fundamental model in information theory that is used to describe a noisy communication channel where bits are transmitted with a certain probability of error. In this model, each transmitted bit may be flipped (i.e., 0 to 1 or 1 to 0) with a fixed probability, but the error is symmetric (equal probability of flipping either bit).

§3.4.2 How Does BSC Arise?

The BSC arises when the communication channel introduces errors in a way that is symmetrical: for any given bit being transmitted, the probability of it being received incorrectly (flipped) is the same, regardless of whether the original bit was 0 or 1. This is particularly relevant when dealing with noisy communication channels, where interference, distortion, or other issues might cause bit errors.

§3.4.3 Mathematical Representation of BSC

The BSC is characterized by the following parameters:

- The input to the channel is a sequence of binary bits $x_1, x_2, \ldots, x_n$ where $x_i \in \{0, 1\}$.
- Each bit x_i has a probability p of being flipped. This means that:

$$P(y_i = 1 \mid x_i = 0) = p, \quad P(y_i = 0 \mid x_i = 1) = p$$

  where y_i is the received bit and p is the bit-flip probability.
- The probability of receiving the correct bit is $1 - p$.

The output bit sequence $y_1, y_2, \ldots, y_n$ is the result of applying the BSC to the input bit sequence.

§3.4.4 Use Case

The BSC model is widely used in coding theory and error correction. It helps in understanding how noisy channels impact data transmission and how to design error-correcting codes, such as Hamming codes or LDPC codes, to recover the original data even when some bits are flipped due to noise.

§3.4.5 Error Probability

The error probability for a BSC is the probability that a transmitted bit is received incorrectly. This is denoted by the parameter p, where:

$$P(\text{error}) = p$$

The higher the value of p, the more likely it is that errors will occur during transmission.

§3.4.6 Channel Capacity

The channel capacity C of a BSC is the maximum rate at which information can be transmitted over the channel with a negligible probability of error. The capacity of a BSC with error probability p is given by:

$$C = 1 - H(p)$$

where $H(p)$ is the binary entropy function defined as:

$$H(p) = -p \log_2 p - (1 - p) \log_2 (1 - p)$$

The channel capacity tells us the maximum achievable data rate, given the noise level.

§3.4.7 Diagrammatic Representation of BSC

The diagram of a Binary Symmetric Channel is shown below:

$$\begin{tikzcd}\boxed{\text{Input Bits}} \arrow[r] & \boxed{\text{BSC Channel}} \arrow[r] & \boxed{\text{Received Bits}}\end{tikzcd}$$

With error probability p

4 Decoders

§4.1 SISO Decoder for Single Parity Check (SPC) Codes

§4.1.1 Introduction to Single Parity Check (SPC) Codes

A Single Parity Check (SPC) code is a simple error-detecting code used to ensure that the total number of 1's in a codeword is even. The SPC code adds a parity bit to the data, allowing detection of errors that change the parity.

For a given data vector $\mathbf{d} = [d_1, d_2, \ldots, d_k]$, the parity bit p is computed as:

$$p = \bigoplus_{i=1}^{k} d_i$$

where $\oplus$ represents the XOR operation. The encoded codeword $\mathbf{c}$ is then:

$$\mathbf{c} = [d_1, d_2, \ldots, d_k, p]$$

where the last bit p is the parity bit.

§4.1.2 SISO Decoding for SPC Codes

SISO decoding aims to estimate the transmitted bits from the received data using soft information. Soft information refers to likelihoods (probabilities) rather than binary decisions. In SPC codes, we aim to decide the most likely transmitted bits while also considering the parity check.

§4.1.3 Mathematical Framework for SISO Decoding

Consider the transmission of a codeword $\mathbf{c}$ over a noisy channel. The received vector is $\mathbf{r} = [r_1, r_2, \ldots, r_n]$, where $n = k + 1$, and the received bits may be affected by noise.

The probability of receiving r_i given the transmitted bit c_i is modeled as:

$$P(r_i \mid c_i) = \begin{cases} 1 - p, & \text{if } r_i = c_i \\ p, & \text{if } r_i \neq c_i \end{cases}$$

where p is the error probability (e.g., from a BSC).

The goal of SISO decoding is to compute the posterior probabilities of the transmitted bits, given the received vector $\mathbf{r}$. The likelihood for each bit c_i is given by:

$$P(c_i \mid \mathbf{r}) \propto P(\mathbf{r} \mid c_i) P(c_i)$$

where $P(c_i)$ is the prior probability of c_i, and $P(\mathbf{r} \mid c_i)$ is the likelihood of receiving $\mathbf{r}$ given c_i.

The a priori probability $P(c_i)$ is typically set based on the likelihood that a bit is 0 or 1. For simplicity, assume:

$$P(c_i = 0) = P(c_i = 1) = 0.5$$

The likelihood function $P(\mathbf{r} \mid c_i)$ is derived from the channel model. Assuming a BSC with error probability p, the likelihood for each received bit is given by:

$$P(r_i \mid c_i) = \begin{cases} 1-p, & \text{if } r_i = c_i \\ p, & \text{if } r_i \neq c_i \end{cases}$$

§4.1.4 Parity Check Constraint

The most important aspect of SISO decoding for SPC codes is the parity check. For a valid codeword, the following parity check equation must hold:

$$\bigoplus_{i=1}^{k+1} c_i = 0$$

This means that the sum of the codeword bits (including the parity bit) must be even. This constraint is used during the decoding process.

§4.2 Log-Likelihood Ratio (LLR) and its Application to SPC Codes

§4.2.1 Introduction to Log-Likelihood Ratio (LLR)

The Log-Likelihood Ratio (LLR) is a crucial concept in soft decision decoding. It represents the logarithmic ratio of the probabilities of a transmitted bit being 1 versus 0, based on received data. LLR is a vital tool used in various error correction algorithms, such as Turbo Codes, LDPC codes, and SPC codes.

The LLR for a bit c_i (where $c_i \in \{0,1\}$) is defined as:

$$\text{LLR}(c_i) = \log\left(\frac{P(c_i = 1 \mid \mathbf{r})}{P(c_i = 0 \mid \mathbf{r})}\right)$$

Where: - $P(c_i = 1 \mid \mathbf{r})$ is the posterior probability that the bit c_i is 1 given the received vector $\mathbf{r}$. - $P(c_i = 0 \mid \mathbf{r})$ is the posterior probability that the bit c_i is 0 given the received vector $\mathbf{r}$.

In terms of likelihoods, the LLR can be rewritten as:

$$\left(\frac{P(c_i = 1 \mid \mathbf{r})}{P(c_i = 0 \mid \mathbf{r})}\right) = \left(\frac{P(\mathbf{r} \mid c_i = 1)}{P(\mathbf{r} \mid c_i = 0)}\right)\left(\frac{P(c_i = 1)}{P(c_i = 0)}\right)$$

$$\text{LLR}(c_i) = \log\left(\frac{P(\mathbf{r} \mid c_i = 1)}{P(\mathbf{r} \mid c_i = 0)}\right) + \log\left(\frac{P(c_i = 1)}{P(c_i = 0)}\right)$$

For simplicity, if we assume that $P(c_i = 1) = P(c_i = 0) = 0.5$ (i.e., the a priori probabilities are equal), the LLR simplifies to:

$$\text{LLR}(c_i) = \log\left(\frac{P(\mathbf{r} \mid c_i = 1)}{P(\mathbf{r} \mid c_i = 0)}\right)$$

§4.2.2 Derivation Using Hyperbolic Tangent

In some cases, the LLR is related to the hyperbolic tangent function due to its connection with the logistic function. The likelihood ratio can be rewritten as:

$$e^{\text{LLR}(c_i)} = \left(\frac{P(\mathbf{r} \mid c_i = 1)}{P(\mathbf{r} \mid c_i = 0)}\right)$$

$$\left(\frac{(e^{\text{LLR}(c_i)} - 1)}{(e^{\text{LLR}(c_i)} + 1)}\right) = \left(\frac{P(\mathbf{r} \mid c_i = 1) - P(\mathbf{r} \mid c_i = 0)}{P(\mathbf{r} \mid c_i = 1) + P(\mathbf{r} \mid c_i = 0)}\right)$$

$$\tanh\left(\frac{\text{LLR}}{2}\right) = \left(\frac{P(\mathbf{r} \mid c_i = 1) - P(\mathbf{r} \mid c_i = 0)}{P(\mathbf{r} \mid c_i = 1) + P(\mathbf{r} \mid c_i = 0)}\right)$$

$$\text{LLR}(c_i) = 2\tanh^{-1}\left(\frac{P(\mathbf{r} \mid c_i = 1) - P(\mathbf{r} \mid c_i = 0)}{P(\mathbf{r} \mid c_i = 1) + P(\mathbf{r} \mid c_i = 0)}\right)$$

This formulation leads to the following relationship between the LLR and the log hyperbolic tangent:

$$\text{LLR}(c_i) = 2 \cdot \text{atanh}\left(\frac{P(\mathbf{r} \mid c_i = 1) - P(\mathbf{r} \mid c_i = 0)}{P(\mathbf{r} \mid c_i = 1) + P(\mathbf{r} \mid c_i = 0)}\right)$$

Where $\text{atanh}(x)$ is the inverse of the hyperbolic tangent function.

§4.2.3 Intrinsic and Extrinsic Factors of LLR

LLRs are calculated using both intrinsic and extrinsic information. The distinction between these two is significant in iterative decoding algorithms, such as Turbo Codes or LDPC Codes.

1. Intrinsic Information: This is the soft information derived directly from the channel and represents the likelihood of a bit being 0 or 1 based on the received signal. In the context of a BSC (Binary Symmetric Channel), intrinsic information is simply the probability of a bit being flipped due to noise.

Mathematically, the intrinsic LLR for bit c_i can be written as:

$$\text{LLR}_{\text{intrinsic}}(c_i) = \log\left(\frac{P(c_i = 1 \mid \mathbf{r})}{P(c_i = 0 \mid \mathbf{r})}\right)$$

2. Extrinsic Information: This represents information about the transmitted bits that comes from the decoder's previous iteration or feedback from other parts of the decoder. Extrinsic information is used to refine the decoding process in successive iterations.

The extrinsic LLR is typically updated during iterative decoding and is related to the updated information from previous decodings.

Mathematically, the extrinsic LLR can be written as:

$$\text{LLR}_{\text{extrinsic}}(c_i) = \log\left(\frac{P(\mathbf{r} \mid c_i = 1)}{P(\mathbf{r} \mid c_i = 0)}\right) - \text{LLR}_{\text{extrinsic, previous}}(c_i)$$

§4.2.4 Min-Sum Approximation Method

The Min-Sum approximation is a simplification technique often used in decoding algorithms such as LDPC decoding. In the Min-Sum algorithm, the LLRs are approximated by replacing the logarithmic functions with simpler operations.

For instance, in the case of a simple BSC, the Min-Sum approximation replaces the likelihood ratio with the minimum of the two likelihoods:

$$\text{LLR}_{\text{min-sum}}(c_i) = \min\left(P(\mathbf{r} \mid c_i = 1), P(\mathbf{r} \mid c_i = 0)\right)$$

This method is computationally simpler, but it may lead to performance degradation compared to the full MAP algorithm. It is often used in hardware implementations where computational complexity is a concern.

Example: SPC Code and Finding LLR

Consider a simple Single Parity Check (SPC) code with two data bits, d_1 and d_2, and one parity bit p, where the parity check equation is:

$$p = d_1 \oplus d_2$$

The encoded codeword is $\mathbf{c} = [d_1, d_2, p]$.

Let us assume the received vector $\mathbf{r} = [r_1, r_2, r_3]$ has some noise, and we want to compute the LLR for each bit.

1. Calculate the Likelihoods:

For each received bit, the likelihood $P(r_i \mid c_i)$ is computed based on the error probability of the channel. Assume the error probability is $p = 0.1$ (for simplicity, using a BSC model).

2. Compute the LLR for d_1:

For bit d_1, the likelihood ratio is:

$$\mathrm{LLR}(d_1) = \log\left(\frac{P(r_1 \mid d_1 = 1)}{P(r_1 \mid d_1 = 0)}\right)$$

Using the channel model, we substitute the values for $P(r_1 \mid d_1 = 1)$ and $P(r_1 \mid d_1 = 0)$ based on the noise level.

3. Compute the LLR for d_2:

Similarly, for bit d_2, the LLR is computed as:

$$\mathrm{LLR}(d_2) = \log\left(\frac{P(r_2 \mid d_2 = 1)}{P(r_2 \mid d_2 = 0)}\right)$$

4. Compute the LLR for p:

For the parity bit p, the LLR is computed based on the parity check constraint. The LLR for the parity bit is influenced by both d_1 and d_2, and the LLR can be modified by considering the parity constraint.

From this:

$$\tanh\left(\frac{|\mathrm{LLR}(p)|}{2}\right) = \tanh\left(\frac{|\mathrm{LLR}(r_1)|}{2}\right)\tanh\left(\frac{|\mathrm{LLR}(r_2)|}{2}\right)$$

This can be rethought as

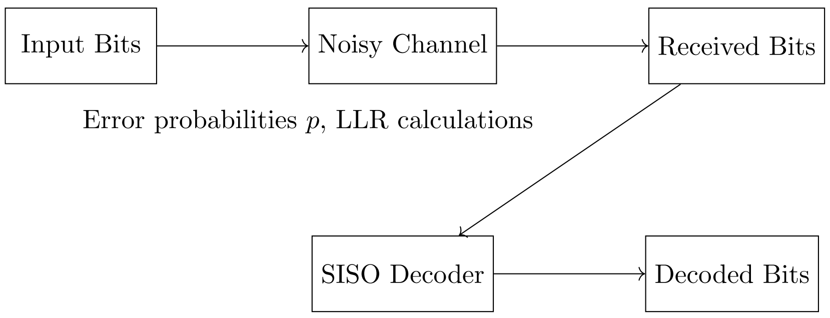
$$\mathrm{LLR}(p) = (\mathrm{sgn}(r_1)\mathrm{sgn}(r_2)\min(\mathrm{abs}(r_1), \mathrm{abs}(r_2))$$

Why does this happen so? Visualize the approximation in the graph and get to know it:

The LLR is an essential tool for soft decision decoding in error correction. By using intrinsic and extrinsic information, LLR helps improve decoding performance, especially in noisy channels. Through mathematical techniques like the Min-Sum approximation, we can simplify the decoding process in hardware, albeit at the cost of performance. Understanding the interplay of these factors is key to designing efficient error-correcting codes.

§4.2.5 Diagrammatic Representation of SISO Decoding for SPC Codes

The following diagram shows the general flow of a SISO decoder for SPC codes.



This diagram shows the flow from the transmission of bits through a noisy channel, the receipt of noisy data, and the SISO decoding process, which outputs the decoded bits.

§4.3 SISO Decoder for Repetition Codes

§4.3.1 Introduction to Repetition Codes

A Repetition Code is one of the simplest error-correcting codes. It encodes a single bit by repeating it n times. For example, if we want to send bit $b \in \{0, 1\}$, the codeword is:

$$\mathbf{c} = [b, b, \ldots, b] \in \{0, 1\}^n$$

The decoder receives a noisy version of this codeword, denoted as:

$$\mathbf{r} = [r_1, r_2, \ldots, r_n]$$

A SISO decoder provides a soft output (LLR) based on the soft input (received values), without making hard decisions prematurely.

§4.3.2 Soft Input: Log-Likelihood Ratio (LLR)

Each received symbol r_i corresponds to a probability of having transmitted $b = 0$ or $b = 1$. The soft input for bit b from received symbol r_i is the LLR:

$$L_i = \log\left(\frac{P(r_i \mid b = 1)}{P(r_i \mid b = 0)}\right)$$

For example, in an AWGN channel with BPSK modulation:

$$b = 0 \Rightarrow x = +1, \quad b = 1 \Rightarrow x = -1$$

Then the received value is:

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

Using this model, the conditional PDF is:

$$P(r_i \mid b) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(r_i - x_b)^2}{2\sigma^2}\right)$$

where $x_b = +1$ if $b = 0$ and $x_b = -1$ if $b = 1$.

Then, the LLR becomes:

$$L_i = \log\left(\frac{\exp\left(-\frac{(r_i+1)^2}{2\sigma^2}\right)}{\exp\left(-\frac{(r_i-1)^2}{2\sigma^2}\right)}\right) = \frac{(r_i - 1)^2 - (r_i + 1)^2}{2\sigma^2} = \frac{2r_i}{\sigma^2}$$

§4.3.3 Total LLR for Repetition Code (Soft Output)

Since all bits in a repetition code are the same, the posterior LLR for the bit b is the sum of all individual LLRs:

$$L_{\text{posterior}} = \sum_{i=1}^{n} L_i = \sum_{i=1}^{n} \frac{2r_i}{\sigma^2} = \frac{2}{\sigma^2} \sum_{i=1}^{n} r_i$$

Thus, the posterior probability is:

$$P(b = 1 \mid \mathbf{r}) = \frac{1}{1 + \exp(-L_{\text{posterior}})}$$

$$P(b = 0 \mid \mathbf{r}) = \frac{1}{1 + \exp(L_{\text{posterior}})}$$

This LLR is then passed as soft output to the next stage in a turbo decoder, or used directly for final decision.

§4.3.4 Extrinsic Information for Iterative Decoding

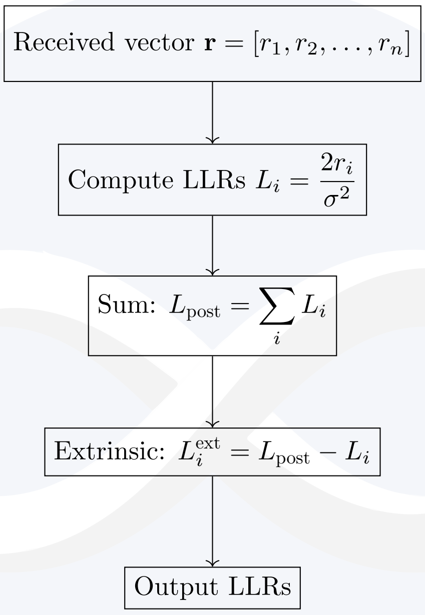
In SISO decoders, we compute the extrinsic LLR to pass to the next module in iterative decoding (e.g., in turbo decoding).

Extrinsic LLR from each i-th position is:

$$L_i^{(\text{ext})} = L_{\text{posterior}} - L_i$$

The extrinsic output does not include the information already contained in the current position, so it avoids positive feedback loops.

§4.3.5 Diagram: SISO Decoder for Repetition Code


Received vector $\mathbf{r} = [r_1, r_2, \dots, r_n]$
Compute LLRs $L_i = \frac{2r_i}{\sigma^2}$
Sum: $L_{\text{post}} = \sum_i L_i$
Extrinsic: $L_i^{\text{ext}} = L_{\text{post}} - L_i$
Output LLRs

§4.3.6 Example

Let us take a simple example with $n = 3$, and the received values from the channel are:

$$\mathbf{r} = [0.8, -0.1, 1.0], \quad \sigma^2 = 1$$

Then:

$$L_1 = \frac{2 \cdot 0.8}{1} = 1.6, \quad L_2 = \frac{2 \cdot (-0.1)}{1} = -0.2, \quad L_3 = \frac{2 \cdot 1.0}{1} = 2.0$$

$$L_{\text{posterior}} = 1.6 - 0.2 + 2.0 = 3.4$$

So:

$$P(b = 1 \mid \mathbf{r}) = \frac{1}{1 + e^{-3.4}} \approx 0.967704535, \quad P(b = 0 \mid \mathbf{r}) = 1 - P(b = 1) \approx 0.032295465$$

Extrinsic outputs:

$$L_1^{\text{ext}} = 3.4 - 1.6 = 1.8, \quad L_2^{\text{ext}} = 3.4 + 0.2 = 3.6, \quad L_3^{\text{ext}} = 3.4 - 2.0 = 1.4$$

The SISO decoder for repetition codes operates by summing the soft information (LLRs) of all received bits, exploiting the repetition structure. It returns both a posterior soft decision and extrinsic values for iterative use.

§4.4 Successive Cancellation (SC) Decoding for Polar Codes

§4.4.1 Introduction to SC Decoding

Successive Cancellation (SC) decoding is an efficient soft-input decoder for Polar Codes, leveraging the recursive structure of the encoding. Polar Codes split the input into "frozen" and "information" bits. SC decoding estimates bits one by one, conditioning on previous decisions.

§4.4.2 Setup and Notation

Let the codeword length be $N = 2^n$, and the input bits be $\mathbf{u} = [u_0, u_1, \ldots, u_{N-1}]$, where $u_i \in \{0, 1\}$. The codeword is given by:

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

where $G_N = B_N F^{\otimes n}$ is the Polar transform, $F = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$, and B_N is the bit-reversal matrix.

§4.4.3 SC Decoding Goal

Given the channel outputs $\mathbf{y} = [y_0, y_1, \ldots, y_{N-1}]$, estimate $\hat{\mathbf{u}} = [\hat{u}_0, \hat{u}_1, \ldots, \hat{u}_{N-1}]$ sequentially using soft information.

We compute the LLR (Log Likelihood Ratio) for each bit u_i, denoted:

$$L_N^{(i)}(y_0^{N-1}, \hat{u}_0^{i-1}) = \log\left(\frac{P(y_0^{N-1} \mid u_i = 0, \hat{u}_0^{i-1})}{P(y_0^{N-1} \mid u_i = 1, \hat{u}_0^{i-1})}\right)$$

Then:

$$\hat{u}_i = \begin{cases} 0 & \text{if } L_N^{(i)} > 0 \\ 1 & \text{if } L_N^{(i)} < 0 \\ 0 & \text{(frozen)} \end{cases}$$

§4.4.4 Recursive LLR Computation

Define LLR recursion for $i \in [0, N/2)$ and λ indicating the current level:

- Left child node LLR:

$$L^{(2i)} = f(L_1^{(i)}, L_2^{(i)}) = \text{sign}(L_1) \cdot \text{sign}(L_2) \cdot \min(|L_1|, |L_2|)$$

(Using min-sum approximation)

- Right child node LLR (conditioned on previous estimate $\hat{u}_{2i}$):

$$L^{(2i+1)} = g(L_1, L_2, \hat{u}_{2i}) = \begin{cases} L_2 + L_1 & \text{if } \hat{u}_{2i} = 0 \\ L_2 - L_1 & \text{if } \hat{u}_{2i} = 1 \end{cases}$$

§4.4.5 Min-Sum Approximation (Simplified LLR Update)

The exact function $f(L_1, L_2)$ involves:

$$f(L_1, L_2) = \log\left(\frac{1 + e^{-(L_1+L_2)}}{1 + e^{-(L_1-L_2)}}\right)$$

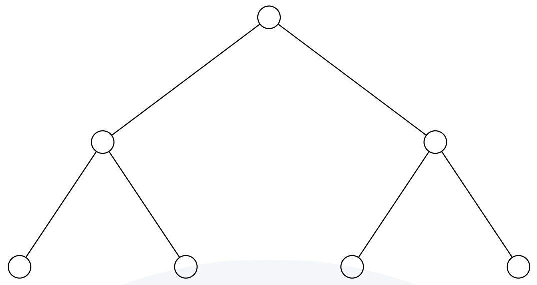
This is computationally intensive.

Min-sum approximation simplifies:

$$f(L_1, L_2) \approx \text{sign}(L_1) \cdot \text{sign}(L_2) \cdot \min(|L_1|, |L_2|)$$

$$g(L_1, L_2, \hat{u}) = L_2 + (-1)^{\hat{u}} L_1$$

§4.4.6 LLR Tree Diagram (Example: $N = 4$)



Each internal node performs either f or g, depending on whether it's estimating an even or odd index.

§4.4.7 The entire decoding algorithm

1) Each node receives LLR's from parent nodes, divides into 2 equal halves, does f of the LLR's in respective positions, and sends it to the left child node.

2) Then, whenever an LLR reaches to the leaf (the nodes with no child nodes), it takes a hard decision and sends it to the parent node.

3) The parent node does g operation and then sends the LLR's to the right child node.

4) When it receives the hard decisions from the right child too, it does Bitwise XOR of the two hard output arrays, appends right child outputs to this array and sends it above for further decoding.

§4.4.8 Example: $N = 4$ SC Decoding

Suppose we have 4 received LLRs: $[L_0, L_1, L_2, L_3] = [2.0, -1.0, 1.5, 0.5]$

We want to decode $\hat{u}_0, \hat{u}_1, \hat{u}_2, \hat{u}_3$

Step 1: Compute LLRs for u_0 (uses all inputs):

$$L_0^{(0)} = f(f(L_0, L_1), f(L_2, L_3)) = f(f(2.0, -1.0), f(1.5, 0.5)) = f(-1.0, 0.5) = -0.5 \Rightarrow \hat{u}_0 = 1$$

Step 2: Compute u_1:

$$L_0^{(1)} = g(f(2.0, -1.0), f(1.5, 0.5), \hat{u}_0 = 1) = f(2.0, -1.0) = -1.0, f(1.5, 0.5) = 0.5 \Rightarrow g(-1.0, 0.5, 1) = 0.5 - (-1.0) = 1.5 \Rightarrow \hat{u}_1$$

Step 3, 4: Continue similarly for $\hat{u}_2, \hat{u}_3$

§4.5 BCJR Algorithm for Soft-Decision Decoding of Reed–Solomon Codes

§4.5.1 Introduction

The BCJR (Bahl–Cocke–Jelinek–Raviv) algorithm performs **symbol-by-symbol MAP decoding** using soft information from the channel. BCJR algorithm is used for Convolutional Codes by leveraging a **trellis-based representation** of the code.

§4.5.2 BCJR Prerequisites

BCJR operates over a trellis. To apply it to convolution codes:

- Construct a **symbol-level trellis** over $\mathbb{F}_q$ (Field of possible states/symbols, formed by the bits stored in memory in the convolutional encoder).
- Channel observations must provide **a posteriori probabilities** (APPs) for each symbol.

Let:

$$y = (y_0, y_1, \ldots, y_{n+k-1}) \in \mathbb{C}^n \quad \text{be the received soft values}$$

k = number of memory bits in the convolutional encoder = length of list being convoluted with the message - 1. While sending a n bit message through a convolutional encoder, we added k extra bits to the message, to end the Trellis in a desired final state. So, the length of the codeword becomes n+k-1.

§4.5.3 Trellis Construction for convolutional Codes

Each convolutional codeword can be described via a trellis where:
- Each path through the trellis encodes a valid convolutional codeword
- Transitions correspond to symbol decisions $c_i \in \mathbb{F}_q$

§4.5.4 The BCJR Algorithm in a Trellis

Define:
- $\alpha_i(s)$: Forward state probability
- $\beta_i(s)$: Backward state probability
- $\gamma_i(s', s, c_i) = P(c_i) \cdot P(y, c_i | s \to s')$

Then:

$$\alpha_0(s) = \begin{cases} 1 & s = s_0 \text{ (initial state)} \\ 0 & \text{otherwise} \end{cases}$$

$$\alpha_{i+1}(s) = \sum_{s'} \sum_{c_i} \alpha_i(s') \cdot \gamma_i(s', s, c_i)$$

$$\beta_n(s) = \begin{cases} 1 & s = s_n \text{ (final state)} \\ 0 & \text{otherwise} \end{cases}$$

$$\beta_i(s) = \sum_{s'} \sum_{c_i} \gamma_i(s, s', c_i) \cdot \beta_{i+1}(s')$$

§4.5.5 APP and Soft Decision

The a posteriori probability for symbol $c_i = a$ is:

$$P(c_i = a | y) = \frac{1}{Z} \sum_{(s',s): f(s',a)=s} \alpha_i(s') \cdot \gamma_i(s', s, a) \cdot \beta_{i+1}(s)$$

where Z is a normalization constant.

§4.5.6 Significance:

- Soft decoding improves performance in AWGN/fading channels
- BCJR decoding with a feedback loop((Turbo Codes) of recursive systematic codes with is useful in deep-space and magnetic recording

§4.5.7 Introduction

Belief Propagation (BP), also known as the Sum-Product Algorithm, is an inference algorithm that operates on factor graphs or Tanner graphs. It is widely used for decoding LDPC codes and Turbo codes, performing inference in probabilistic graphical models.

It allows computing marginal probabilities:

$$P(x_i|y) \quad \text{for each variable } x_i$$

where y is the observation (e.g., received signal from a noisy channel).

§4.5.8 Factor Graph Representation

A joint distribution over a set of variables $\{x_1, x_2, \ldots, x_n\}$ is factored as:

$$P(x_1, \ldots, x_n) = \frac{1}{Z} \prod_{a \in \mathcal{F}} f_a(x_{N(a)})$$

where: - $\mathcal{F}$: Set of factor nodes - $N(a)$: The neighborhood of factor a - Z: Normalization constant

Example: In LDPC decoding, each factor node corresponds to a parity-check equation:

$$f_a(x_{i_1}, \ldots, x_{i_d}) = \mathbf{1}_{x_{i_1} \oplus \cdots \oplus x_{i_d} = 0}$$

§4.5.9 Messages in BP

BP involves passing messages between variable nodes and factor nodes. Let:

- $m_{i \to a}(x_i)$: message from variable node x_i to factor node a - $m_{a \to i}(x_i)$: message from factor node a to variable node x_i

Initialization: For each variable node x_i, initialize:

$$m_{i \to a}^{(0)}(x_i) = P(y_i|x_i)$$

if channel observations y_i are available.

§4.5.10 Message Update Rules

At iteration t, the message updates are:

Variable to Factor:

$$m_{i \to a}^{(t)}(x_i) = \prod_{b \in N(i) \setminus \{a\}} m_{b \to i}^{(t-1)}(x_i)$$

Factor to Variable:

$$m_{a \to i}^{(t)}(x_i) = \sum_{\mathbf{x}_{N(a) \setminus i}} f_a(x_{N(a)}) \prod_{j \in N(a) \setminus \{i\}} m_{j \to a}^{(t)}(x_j)$$

The **marginal belief** for x_i is estimated as:

$$b_i(x_i) \propto \prod_{a \in N(i)} m_{a \to i}^{(t)}(x_i)$$

§4.5.11 Log-Domain Version (Log-Sum-Exp)

To avoid numerical underflow, we work in log-domain:

$$L_{i\to a} = \log \frac{m_{i\to a}(0)}{m_{i\to a}(1)}$$

The factor-to-variable update becomes:

$$L_{a\to i} = 2\tanh^{-1}\left(\prod_{j\in N(a)\setminus\{i\}} \tanh\left(\frac{L_{j\to a}}{2}\right)\right)$$

This is known as the **tanh rule** in BP for binary symmetric channels.

§4.5.12 Application: LDPC Decoding on BSC

Suppose a (3,6) LDPC code over BSC with crossover probability p. The likelihood for each bit is:

$$L_i^{\text{channel}} = \log\left(\frac{1-p}{p}\right)\cdot(1-2y_i)$$

Each bit node uses BP to combine channel input with check node feedback.

§4.5.13 Loop-Free Case (Tree)

If the graph is a tree (no cycles), BP converges to exact marginals in finite iterations equal to the depth of the tree.

§4.5.14 Loopy BP

When the factor graph has cycles (as in most LDPC codes), BP is no longer exact but often performs well in practice — this is called **Loopy BP**.

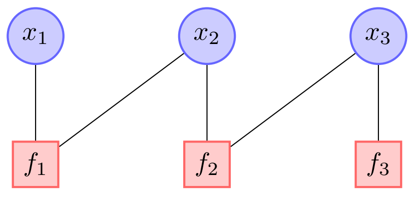
§4.5.15 Convergence and Damping

BP may not always converge in loopy graphs. Damping helps:

$$m^{(t)} = (1-\lambda)m_{\text{new}}^{(t)} + \lambda m^{(t-1)}$$

with $\lambda \in (0,1)$.

§4.5.16 Visual Representation



Each edge corresponds to message-passing. BP iterates over this graph.

§4.5.17 Complexity

If each factor node has degree d, and variable nodes are binary: - Each check node update: $O(2^{d-1})$ - In practice, optimized forms reduce cost (e.g., min-sum, normalized min-sum)

§4.5.18 Approximations: Min-Sum Algorithm

The Min-Sum approximation replaces the tanh rule with:

$$L_{a\to i} = \min_{j\in N(a)\setminus i} |L_{j\to a}| \cdot \prod_j \text{sign}(L_{j\to a})$$

Often scaled to reduce overestimation:

$$L_{a\to i} = \alpha \cdot \text{min-sum} \quad \text{where } 0 < \alpha < 1$$

§4.6 Syndrome Decoding: Theory and Derivations

§4.6.1 Introduction

Syndrome decoding is a powerful technique for decoding **linear block codes**. It leverages the structure of linear codes to determine whether an error occurred, and if so, to identify the most likely error pattern.

§4.6.2 Preliminaries

Let $\mathcal{C} \subset \mathbb{F}_2^n$ be a linear block code of dimension k and length n. Then:

- Generator matrix $G \in \mathbb{F}_2^{k\times n}$
- Parity-check matrix $H \in \mathbb{F}_2^{(n-k)\times n}$

A codeword $c \in \mathcal{C}$ satisfies:

$$Hc^T = 0$$

§4.6.3 Received Vector and Syndrome

Let the codeword c be transmitted and $r = c + e$ be received, where e is the error vector.

We compute the **syndrome**:

$$s = Hr^T = H(c+e)^T = Hc^T + He^T = He^T$$

because $Hc^T = 0$.

So the syndrome depends only on the error pattern e, not on the transmitted codeword.

§4.6.4 Syndrome Table Method

To decode, we:

1. Compute the syndrome $s = Hr^T$
2. Find the error vector e corresponding to s
3. Recover $c = r - e$

This process is based on a **syndrome decoding table** which maps all possible syndromes to likely error patterns (usually of minimum weight).

§4.6.5 Matrix View

Given $H \in \mathbb{F}_2^{(n-k)\times n}$, and $r \in \mathbb{F}_2^n$, we compute:

$$s = Hr^T = \begin{bmatrix} h_1^T \\ h_2^T \\ \vdots \\ h_{n-k}^T \end{bmatrix}^T \cdot r^T = \begin{bmatrix} h_1^T r^T \\ h_2^T r^T \\ \vdots \\ h_{n-k}^T r^T \end{bmatrix}^T$$

Each entry of s is a parity check equation.

§4.6.6 Example: Hamming(7,4) Code

Generator matrix:

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

Parity-check matrix:

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

Suppose received vector:

$$r = [1\ 0\ 1\ 0\ 0\ 1\ 0]$$

Compute:

$$s = Hr^T$$

Find syndrome:

$$s = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \Rightarrow \text{Lookup syndrome table} \Rightarrow e = [0\ 0\ 0\ 1\ 0\ 0\ 0]$$

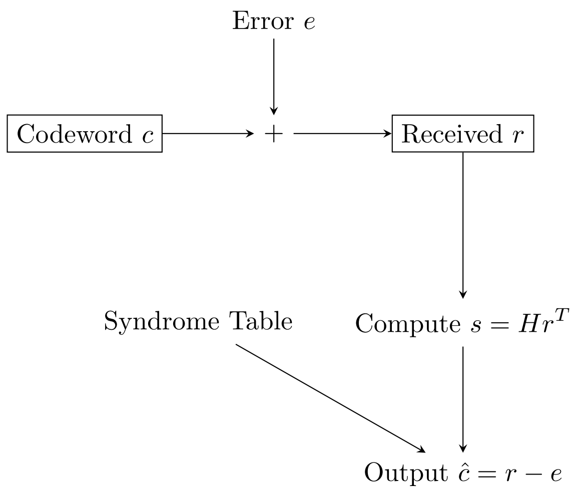
So correct codeword:

$$c = r - e = [1\ 0\ 1\ 1\ 0\ 1\ 0]$$

§4.6.7 Decoding Algorithm Summary

1. Compute $s = Hr^T$
2. Use syndrome table or logic to find most probable e
3. Output: $\hat{c} = r - e$

§4.6.8 Visualization



§4.7 Lagrange Interpolation Decoder for Reed-Solomon Codes

§4.7.1 Overview

Lagrange interpolation is used in Reed-Solomon decoding when errors are known in location (e.g., erasure decoding). It reconstructs the original polynomial $f(x)$ from known (correct) evaluations.

§4.7.2 Reed-Solomon Encoding

Construct a polynomial $f(x)$ of degree $< k$ with coefficients as the message. The codeword is formed by evaluating $f(x)$ at n distinct points:

$$c = (f(\alpha_1), f(\alpha_2), \ldots, f(\alpha_n))$$

§4.7.3 Lagrange Interpolation Formula

If we have k correct (or uncorrupted) symbol positions:

$$(\alpha_1, y_1), (\alpha_2, y_2), \ldots, (\alpha_k, y_k)$$

then the message polynomial can be recovered using:

$$f(x) = \sum_{j=1}^{k} y_j \cdot \ell_j(x)$$

where

$$\ell_j(x) = \prod_{\substack{1 \le m \le k \\ m \ne j}} \frac{x - \alpha_m}{\alpha_j - \alpha_m}$$

§4.7.4 When is Lagrange Interpolation Used?

- In **erasure decoding** (when error locations are known).
- In **secret sharing** and polynomial commitment schemes.
- When fewer than $n - k$ erasures occur in an RS code.

§4.7.5 Example

Suppose an RS code over $GF(2^3)$, with message $f(x)$ of degree < 3. Received (correct) points:

$$f(1) = 2, \quad f(3) = 5, \quad f(4) = 1$$

Then the interpolation polynomial is:

$$f(x) = 2 \cdot \ell_1(x) + 5 \cdot \ell_2(x) + 1 \cdot \ell_3(x)$$

with each $\ell_j(x)$ computed using the formula above.

5 Source Encoders

§5.1 Huffman Source Encoding: Theory and Construction

§5.1.1 Introduction

Huffman coding is a lossless data compression algorithm that produces optimal prefix codes for a discrete memoryless source. It minimizes the average codeword length, making it more efficient than Shannon–Fano when it comes to compression close to the entropy limit.

§5.1.2 Source Model and Entropy

Let the source alphabet be $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ with associated symbol probabilities $\{p_1, p_2, \ldots, p_n\}$, where:

$$\sum_{i=1}^{n} p_i = 1, \quad p_i > 0.$$

The entropy of the source is:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

This represents the lower bound on the average number of bits per symbol required for lossless encoding.

§5.1.3 Huffman Coding Algorithm

Step-by-Step Construction:

1. List all source symbols in ascending order of probabilities.
2. Combine the two least probable symbols to form a new symbol with probability equal to their sum.
3. Repeat until there is only one combined symbol left (the root of the tree).
4. Assign binary digits: '0' to left branches, '1' to right branches (or vice versa).
5. Traverse the tree to generate codes for original symbols.

§5.1.4 Mathematical Guarantee of Optimality

Let $\{l_i\}$ be the lengths of the resulting codewords. Then the average codeword length is:

$$L_{\text{avg}} = \sum_{i=1}^{n} p_i l_i$$

Huffman codes minimize L_{avg} under the prefix condition. We have:

$$H(X) \leq L_{\text{avg}} < H(X) + 1$$

§5.1.5 Kraft–McMillan Inequality

The code lengths from Huffman coding satisfy:

$$\sum_{i=1}^{n} 2^{-l_i} \leq 1$$

with equality if and only if the code is complete.

§5.1.6 Example: Huffman Tree Construction

Let the symbol probabilities be:

Symbol	Probability
A	0.4
B	0.2
C	0.2
D	0.1
E	0.1

Table 5.1: Original symbol probabilities

Tree Construction:

- Combine D(0.1) + E(0.1) → Node1(0.2)
- Combine Node1(0.2) + B(0.2) → Node2(0.4)
- Combine C(0.2) + Node2(0.4) → Node3(0.6)
- Combine A(0.4) + Node3(0.6) → Root(1.0)

Final Codewords (one possible assignment):

Symbol	Codeword	Length	Contribution to L_{avg}
A	0	1	0.4
C	10	2	0.4
B	111	3	0.6
D	1100	4	0.4
E	1101	4	0.4

Table 5.2: Generated Huffman codes

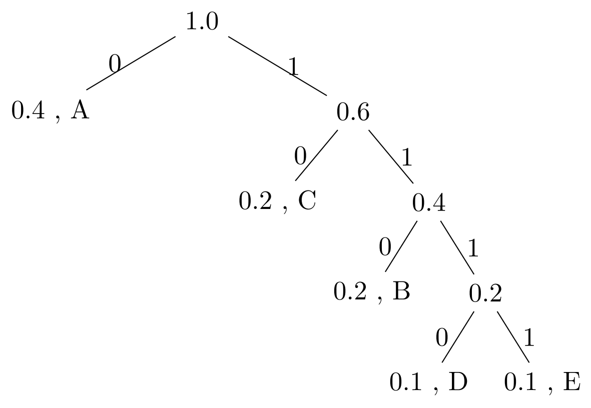
Average Codeword Length:

$$L_{\text{avg}} = 0.4 \cdot 1 + 0.2 \cdot 3 + 0.2 \cdot 2 + 0.1 \cdot 4 + 0.1 \cdot 4 = 2.2 \text{ bits/symbol}$$

Entropy:

$$H(X) = -\left(0.4 \log_2 0.4 + 0.2 \log_2 0.2 + 0.2 \log_2 0.2 + 0.1 \log_2 0.1 + 0.1 \log_2 0.1\right) \approx 2.12$$

§5.1.7 Huffman Tree Diagram



§5.1.8 Huffman Coding vs Shannon–Fano Coding

- Huffman coding always produces optimal prefix codes.
- Shannon–Fano may not be optimal but is conceptually simpler.
- Huffman uses bottom-up tree building, while Shannon–Fano uses top-down recursive partitioning.

§5.1.9 subsApplications

- Data compression (ZIP, JPEG)
- Multimedia encoding (MP3, MPEG)
- Network protocols

§5.1.10 Conclusion

Huffman coding is an optimal, greedy algorithm for symbol-by-symbol source compression. It achieves a nearly minimal average codeword length, bounded closely by the entropy of the source, and produces instantaneous prefix codes suitable for real-time transmission and decoding.

§5.2 Shannon–Fano Source Encoding: Theory and Construction

§5.2.1 Introduction

Shannon–Fano coding is an entropy-based source coding method designed to generate prefix codes (instantaneous codes) for a discrete memoryless source. The key idea is to assign shorter binary codes to more probable symbols and longer codes to less probable ones, thus reducing the average codeword length.

§5.2.2 Preliminaries

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be the source alphabet with symbol probabilities $\{p_1, p_2, \ldots, p_n\}$, where:

$$\sum_{i=1}^{n} p_i = 1, \quad \text{and } p_i > 0.$$

Entropy of the Source:

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

The entropy represents the minimum average number of bits required to represent each symbol from the source.

§5.2.3 Shannon–Fano Coding: Algorithm

Step-by-step Construction:

1. List all source symbols x_i in order of decreasing probability p_i.
2. Divide the list into two parts such that the total probability of each part is as close as possible.
3. Assign binary digits: '0' to symbols in the upper half, '1' to the lower half.
4. Recurse the process for each subset until each symbol has a unique binary code.

§5.2.4 Mathematical Interpretation of the Step 2 Split

Given a probability list $P = \{p_1, p_2, \ldots, p_n\}$, find index $k \in [1, n-1]$ such that:

$$\left| \sum_{i=1}^{k} p_i - \sum_{i=k+1}^{n} p_i \right| \text{ is minimized.}$$

This minimizes the imbalance in probability mass between the two partitions.

§5.2.5 Prefix Property

Shannon–Fano codes are prefix-free: no codeword is a prefix of another. This ensures instantaneous decoding, no ambiguity during transmission

§5.2.6 Average Codeword Length

Let l_i be the length of the codeword for symbol x_i. Then,

$$L_{\text{avg}} = \sum_{i=1}^{n} p_i \cdot l_i$$

Shannon's Source Coding Theorem bounds this, for the most optimal coding algorithm:

$$H(X) \leq L_{\text{avg}} < H(X) + 1$$

Shannon-Fano Coding is not optimal, but it has been proved that it satisfies this property.

$$H(X) \leq L_{\text{avg}} < H(X) + 2$$

§5.2.7 Example: Encoding 5 Symbols

Let the source symbols and probabilities be:

Symbol	Probability	Code	Length
A	0.4	00	2
B	0.2	11	2
C	0.2	100	3
D	0.1	01	2
E	0.1	101	3

Table 5.3: Shannon–Fano coding table

Average length:

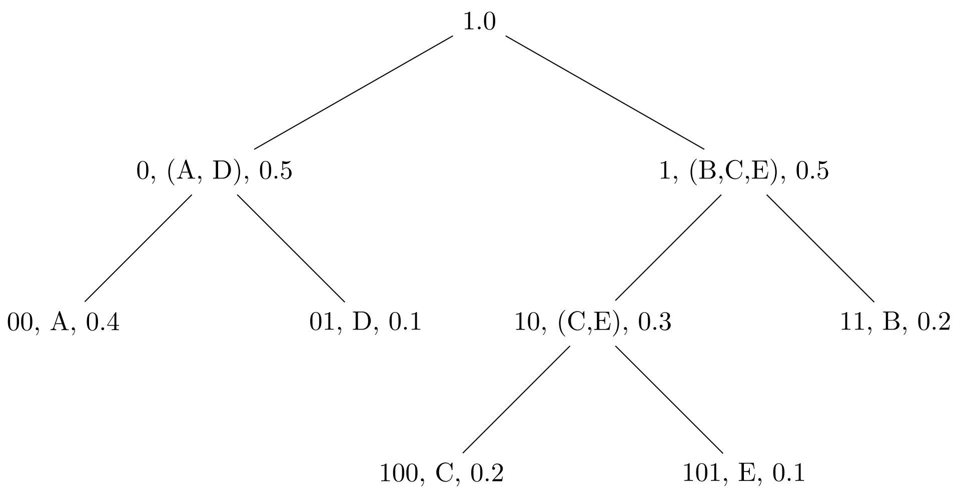
$$L_{\text{avg}} = 0.4 \cdot 1 + 0.2 \cdot 2 + 0.2 \cdot 3 + 0.1 \cdot 2 + 0.1 \cdot 3 = 2.3 \text{ bits/symbol}$$

Entropy:

$$H(X) = -\left(0.4 \log_2 0.4 + 0.2 \log_2 0.2 + 0.2 \log_2 0.2 + 0.1 \log_2 0.1 + 0.1 \log_2 0.1\right) \approx 2.12$$

Here, the average length is less than $H(X) + 2$, satisfying the bounds.

§5.2.8 Shannon–Fano Tree Diagram



§5.2.9 Comparison with Huffman Coding

Shannon–Fano does not always produce the optimal prefix code (i.e., the shortest average length), while Huffman coding guarantees it.

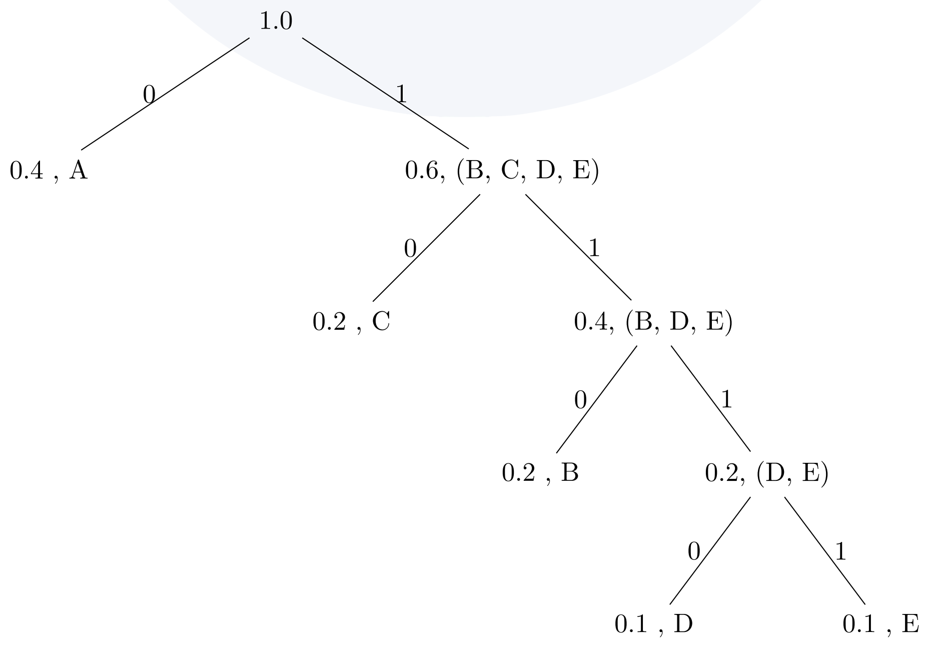
However, Shannon–Fano is conceptually simpler and historically important.

§5.2.10 Conclusion

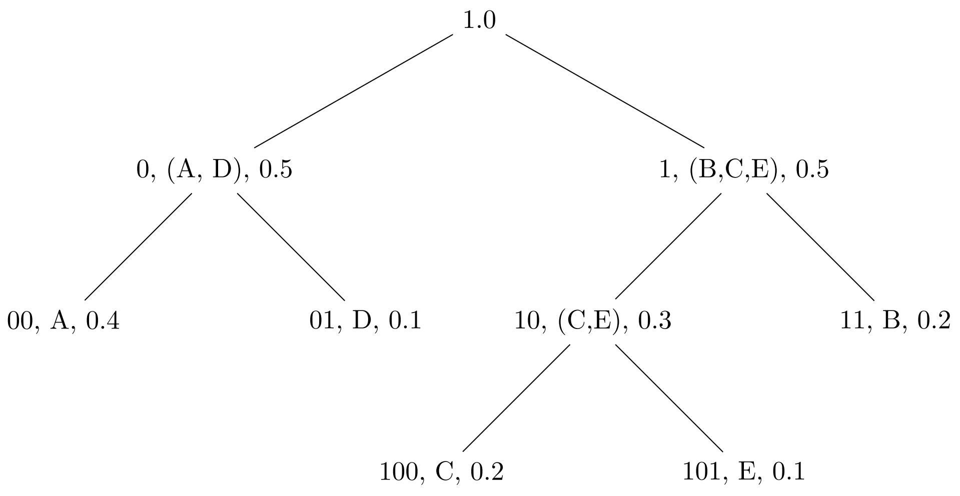
Shannon–Fano coding is a foundational technique in lossless data compression. Its average codeword length is close to entropy, and it guarantees prefix-free binary codes through a probabilistic partitioning process. Although not optimal like Huffman coding, it offers an intuitive understanding of how probability influences compression.

§5.3 Visual Comparison: Huffman vs Shannon-Fano Coding

§5.3.1 Huffman Tree Diagram



§5.3.2 Shannon–Fano Tree Diagram



Legend:

A(0.4), B(0.2), C(0.2), D(0.1), E(0.1) are symbols sorted by decreasing probability.

Huffman builds tree bottom-up via pairing least probable nodes.

Shannon-Fano splits symbol list recursively to minimize difference of summed probabilities.

Average Length in Huffman Coding : 2.2 bits/symbol

Average Length in Shannon-Fano Coding : 2.3 bits/symbol

6 Channel Encoders

§6.1 LDPC Codes

§6.1.1 Introduction

LDPC (Low-Density-Parity-Check) Codes are a kind of error correcting codes which are inspired by the limitations in Hamming Codes. They are used since they provide efficient error correction and reduce computional load.They are specified by the sparse parity-check matrix (H) which helps in encoding of the message and the generator matrix(G) which helps in the decoding of the received codeword.

§6.1.2 Code Implementation

LDPC_Simulation over BEC (Binary Erasure Channel).py

- Requirements

```
import numpy as np
from itertools import combinations
import random
```

- This function helps in Constructing the generator matrix from all the non-empty subsets of the message bits and arranging the matrix G in the form of $[P|I\_k]$. The parity-check matrix H is derived from the relation in the form of $[I\_n - k|P^T]$.

```
def constructor_generator_and_parity_check(self):
```

- The message is converted to the codeword by using $codeword = message \times Gmod2$

```
def encode(self,message):
```

- The codeword is transmitted through the channel and each information bit is erased or transmitted successfully with the erasure probability ϵ.If the bits are erased they are replaced with the symbol 'e'.

```
def transmit(self,codeword)
```

- The received message is decoded by iterating through each row of the matrix H to get several parity-check equations. If a parity-check equation has one unknown bit, it can be solved. The iterations for decoding take place upto a certain limit or until no further updates have been in the decoding message.

- The decoded message is returned in the end

```
def decode(self,received)
```

- Checking if both the original encoded codeword and decoded corrected codeword are correct

```
def compare(original,decoded)
```

§6.2 Low-Density Parity-Check (LDPC) Codes: A Mathematical Insight(Explanation)

§6.2.1 Mathematical Structure of LDPC Codes

An LDPC code is defined by a sparse binary matrix $\mathbf{H} \in \{0,1\}^{m \times n}$. A codeword $\mathbf{c} \in \{0,1\}^n$ satisfies the parity-check equation:

$$\mathbf{H}\mathbf{c}^T = \mathbf{0} \quad \text{over} \quad \mathbb{F}_2.$$

The sparsity of **H** ensures that the number of ones is small compared to the total number of entries, leading to efficient decoding.

Tanner Graph Representation

LDPC codes are often visualized via a bipartite graph called the **Tanner graph**. It consists of:

- Variable nodes (one for each code bit)
- Check nodes (one for each parity constraint)

An edge exists between a variable node v_j and a check node c_i if $H_{ij} = 1$.

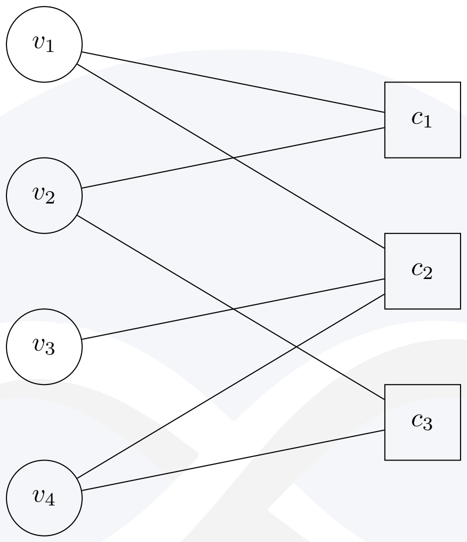


Figure 6.1: An example Tanner graph for a simple LDPC code.

§6.2.2 Encoding and Decoding

While encoding LDPC codes can be non-trivial due to irregular structure, decoding is highly efficient via iterative algorithms.

Belief Propagation (BP) Algorithm

The Sum-Product Algorithm (SPA) decodes by passing messages iteratively between variable and check nodes. At each iteration:

- Variable nodes send estimates of their value to check nodes.
- Check nodes send back consistency information.

The BP algorithm exploits the sparsity to reduce complexity to $\mathcal{O}(n)$ per iteration.

Mathematically, messages $m_{v\to c}$ and $m_{c\to v}$ are updated using:

$$m_{c\to v} = 2\tanh^{-1}\left(\prod_{v'\in N(c)\setminus v} \tanh\left(\frac{m_{v'\to c}}{2}\right)\right),$$

where $N(c)$ denotes the neighboring variable nodes of check node c.

§6.2.3 Applications of LDPC Codes

LDPC codes are employed in various modern communication standards:

- **5G NR**: For high-reliability data channels.
- **Wi-Fi (IEEE 802.11n/ac/ax)**: Enhances wireless data throughput.
- **Satellite Communication**: DVB-S2, DVB-S2X standards.
- **Data Storage**: Hard disks, solid-state drives.

Their close-to-capacity performance, scalability, and adaptability make LDPC codes ubiquitous.

Typically, LDPC codes exhibit **waterfall behavior**: a sharp decrease in BER beyond a threshold SNR.

§6.2.4 Conclusion

LDPC codes have revolutionized coding theory, offering near-optimal error correction with manageable computational resources. Their graphical structure, efficient iterative decoding, and application versatility continue to make them an essential tool in modern digital communication systems.

§6.3 Reed Solomon Codes

§6.3.1 Introduction

Reed-Solomon codes were developed by Iriving S Reed and Gustave Solomon when working at MIT Labs in 1960. They use an alphabet set of length $q = p^m$, q being a prime power (i.e. the encoding set is a galois field) and encode the input at n points; later interpolating it to a polynomial. Using this, errors are detected or corrected and **binary erasures** can be taken care of.

§6.3.2 Documentation

`SimulateChannel.py`

- Requirements

  ```
  import numpy as np
  import seaborn as sns
  import matplotlib.pyplot as plt
  ```

- Function to encode the text. Returns a list of encoded values. In this case, the encoding is just the number assigned to ASCII characters in Python.

  ```
  def encode_alphabet_naive(text):
  ```

- A custom encoding based on values modulo (3).

  ```
  def encode_alphabet_custom(text):
  ```

- Function to decode. Here, decoding is based on the original function; i.e. Python assigned ASCII value to character.

  ```
      def decode_alphabet(encoded_values):
  ```

- Returns a list of random values from the list $[1, n]$, with n being the length of the pdist. The values are chosen based on corresponding probabilities in the pdist.

  ```
  def random_out(pdist):
      return np.random.choice(np.arange(1, len(pdist) + 1), p=pdist)
  ```

Example: If pdist = [0.1, 0.1, 0.8], output could be [1, 3, 3] as 3 has higher probability.

- Plot the channel matrix as a heatmap.

```
def plot_channel_matrix(probab_matrx):
```

- Mean squared error function. Inputs actual, predicted are lists. Converted to numpy array to support index by index operations directly (actual - predicted = array of differences in values).

```
def mean_squared_error(actual, predicted):
    actual = np.array(actual)
    predicted = np.array(predicted)
    return np.mean((actual - predicted) ** 2)
```

- Normalize rows and columns of the probability matrix.

```
probab_matrx = probab_matrx / probab_matrx.sum(axis=1, keepdims=True)
probab_matrx = probab_matrx / probab_matrx.sum(axis=0, keepdims=True)
```

- Show results and error of custom encoding and ASCII encoding. To decide optimal encoding.

`Hamming_Code_Encoder_Decoder.py`

- Requirements. Numpy
- Define Generator matrix G. (4, 7) and parity check matrix H (3, 7).
- Encoder: Converts 4 bit to 7 bit. Accepts a 4 bit encoding as input. Conversion is done by taking dot product of encoding with a generator matrix.

```
def encode_batch(data_bits):
```

- Decoder: Takes in 7-bit input and converts it into the original 4-bit message. Procedure as follows.
  - Get syndrome for error to bit mapping. Take dot product of parity check matrix and transpose of received code.
  - Obtain error bit based on positions obtained from syndrome tuple.
  - Copy the bit to a temporary variable, corrected.
  - If the bit is errored, flip corrected.
  - Append corrected to decoded array.

```
def decode_batch(received_batch):
```

- Simulate a channel with gaussian noise. Initialise standard deviation of noise distribution. Array of length codewords.shape (length of code) picked from the distribution using np.random.normal(). Add noise to the codeword array and return the clipped array (round off values to integral/binary bits) using np.clip and np.round.

```
def gaussian_channel(codewords, noise_std=0.5):
```

- Simulate a channel with uniform noise; i.e. bit flipping with probability p. Induce flips with probability p by choosing random integers in the range [1, codewords.shape], with probability flip_prob. Return the error induced array by adding and taking mod 2 for binary. Can be extended to n-nary.

```
def uniform_channel(codewords, flip_prob=0.1):
```

- Return the bit error rate (number of errored bits/total number of bits).

```
def bit_error_rate(original, decoded):
```

`Reed_Solomon_Encoder_Decoder.py`

- Requirements. `numpy, galois.`
- Simulate binary erasure channel with erasure_prob. An array of random indices is chosen with probability erasure_prob using np.random.rand. Original codeword is copied to an array and chosen indices are replaced with '?' to indicate erasures. Returns the array.

  ```
  def binary_erasure_channel(codeword, erasure_prob):
  ```

- Class for Reed Solomon Error Correction. Initialised variables and functions.
  - m = Degree of field.
  - GF = Galois Field of order m for the prime $p = 2$ in this case.
  - n = Length of the codeword
  - k = Total length of the message.
  - `def rs_encode(self, message)` - First check for length of message. Fit the message into a polynomial in the GF by using `galois.Poly()` and store its range in a variable xs. The codeword will be the set of values at the points $\{x_i\}$; stored in the variable codeword.
  - `def rs_decode(self, received)` - Decode the message by computing the value of the interpolated polynomial at the points $\{x_i\}$. Use Lagrange Interpolation inbuilt in galois library. xs $= \{x_i\}$ is known and input is the receieved code. Add zeroes in the message to account for erasures and to maintain length of decoded message.
  - `def compute_rate(self)` - Returns rate $= k/n$. (Number of message bits/Total number of bits).
- `main()` to call functions and display results.

§6.4 Reed–Solomon (RS) Codes: Structure and Lagrange Decoding(Explanation)

§6.4.1 Mathematical Structure of RS Codes

An (n, k) RS code over a finite field $\mathbb{F}_q$ (q a prime power) encodes a message of k symbols into a codeword of n symbols, where $n \leq q - 1$.

Messages are viewed as polynomials:

$$m(x) = m_0 + m_1 x + \cdots + m_{k-1} x^{k-1}.$$

Encoding is performed by evaluating $m(x)$ at n distinct nonzero points $\{\alpha_1, \alpha_2, \ldots, \alpha_n\} \subset \mathbb{F}_q$.

Thus, the codeword is:

$$\mathbf{c} = (m(\alpha_1), m(\alpha_2), \ldots, m(\alpha_n)).$$

§6.4.2 Encoding: Evaluation Map

The encoding map $E : \mathbb{F}_q^k \to \mathbb{F}_q^n$ is:

$$E(m_0, \ldots, m_{k-1}) = (m(\alpha_1), \ldots, m(\alpha_n)).$$

Due to properties of polynomials over finite fields, $m(x)$ is uniquely determined by its evaluations at any k distinct points.

§6.4.3 Decoding via Lagrange Interpolation

If no errors occur, full decoding simply reconstructs the original polynomial $m(x)$ via Lagrange interpolation:

$$m(x) = \sum_{i=1}^{k} y_i \prod_{\substack{1 \le j \le k \\ j \ne i}} \frac{x - \alpha_j}{\alpha_i - \alpha_j},$$

where $y_i = m(\alpha_i)$ are the received (error-free) symbols.

In the presence of errors, standard Lagrange interpolation fails. One may attempt:

- **Erasure Decoding**: If error locations are known, only interpolate on correct points.
- **Error Correction**: Using additional redundancy, solve for the true polynomial even if some points are corrupted.

This leads to solving a *polynomial interpolation problem with errors*, related to finding two polynomials $E(x), Q(x)$ such that:

$$E(x)r(x) = Q(x)$$

with $\deg(E) = t$, $\deg(Q) \le k + t - 1$, and t being the number of errors.

§6.4.4 Applications of Reed–Solomon Codes

RS codes are crucial in several technologies:

- **QR Codes**: Correction of damage/smudges.
- **Compact Discs (CDs)**: Handle scratches.
- **Satellite and Space Communication**: High error probability channels.
- **Data Storage Systems**: RAID arrays for disk failure recovery.

§6.4.5 Figures and Plots

§6.4.6 Encoding and Interpolation View

Generally, RS codes can correct up to $\left\lfloor \frac{n-k}{2} \right\rfloor$ symbol errors.

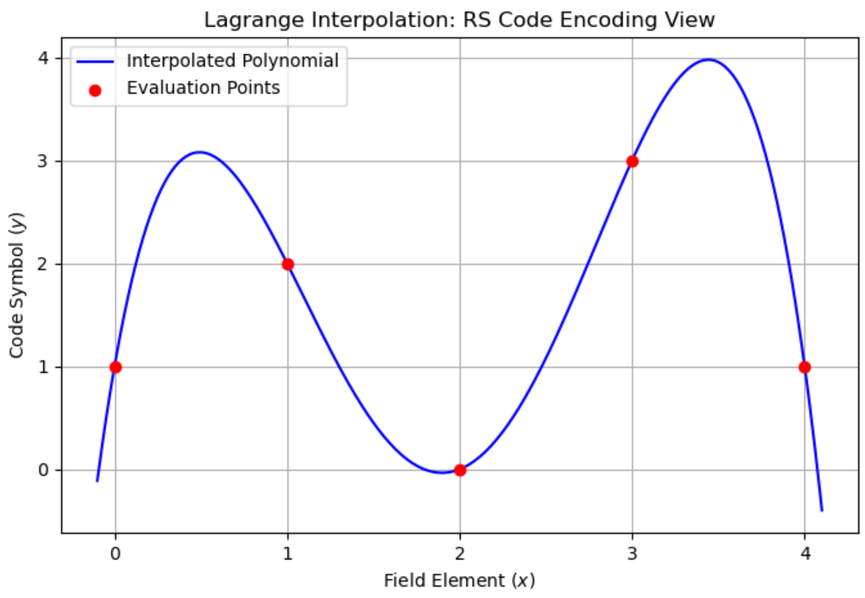


Figure 6.2: Correct encoding and interpolation of RS code: Lagrange polynomial through evaluation points.

§6.4.7 Conclusion

Reed–Solomon codes exemplify the power of algebraic methods in communication theory. Lagrange interpolation provides a clean and intuitive decoding strategy, particularly valuable when the number of errors is small. Their robustness against burst errors ensures their continuing relevance in practical systems.

§6.5 Turbo Codes: Structure and Implementation

§6.5.1 Introduction

Turbo codes are a groundbreaking class of forward error correction (FEC) codes that revolutionized digital communication by approaching the Shannon limit, the theoretical maximum efficiency for reliable data transmission over noisy channels. Introduced in 1993 by Claude Berrou, Alain Glavieux, and Punya Thitimajshima, these codes combine parallel concatenated convolutional codes (separated by an interleaver) with iterative decoding to achieve exceptional error-correction performance.

§6.5.2 Interleavers

Interleavers play a crucial role in turbo codes by dispersing burst errors across multiple code blocks. This process makes the errors appear more random, which enhances the performance of error correction algorithms. These algorithms are generally more effective when handling random errors as opposed to clustered ones. Pseudorandom interleavers allow for the use of deinterleavers during the decoding phase, ensuring that the original data order is restored for accurate correction.

In this document, we discuss three basic types of pseudorandom interleavers:

1. **Modular Interleaver**

   The Modular Interleaver leverages the mathematical property that when `m` and `n` are coprimes, the multiples of `m` (i.e., `m, 2m, 3m, 4m, ..., n × m`) produce distinct remainders when divided by `n`. These remainders create a permutation of values ranging from 0 to `n - 1`.

Inputs: Two integers m and n
Purpose: Creates a simple permutation that spreads out input values across a range, ensuring diversity in their distribution.

Modular Interleaver Code

```c
#include <stdio.h>
#include <stdlib.h>

int* generatePermutation(int m, int n) {
    int* permut = (int*) malloc(n * sizeof(int));
    if (permut == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    for (int i = 0; i < n; ++i) {
        permut[i] = (m * (i + 1)) % n;
    }
    return permut;
}

void printPermutation(int* permut, int n) {
    for (int i = 0; i < n; ++i) {
        printf("%d ", permut[i]);
    }
    printf("\n");
}

int main() {
    int m, n;
    printf("Enter values for m and n: ");
    scanf("%d %d", &m, &n);
    int* p = generatePermutation(m, n);
    printPermutation(p, n);
    free(p);
    return 0;
}
```

2. **Block Interleaver**

A Block Interleaver organizes data into a 2D matrix, filling the matrix row by row, and then reads it out column by column to form the interleaved array.

Logic:

- Fill a matrix row-wise with sequential data.
- Read the matrix column-wise to form the interleaved array.

Inputs: Number of rows r and columns c
Purpose: This method is particularly useful for managing burst errors by spreading data across multiple blocks, providing a structured form of interleaving.

Block Interleaver Code

```c
#include <stdio.h>
#include <stdlib.h>

void Print(int* p, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", p[i]);
    }
    printf("\n");
}

int* Block_Interleaver(int r, int c) {
    int* p = (int*) malloc(r * c * sizeof(int));
    if (p == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    int* arr = (int*) malloc(r * sizeof(int*));
    for (int i = 0; i < r; i++) {
        arr[i] = (int*) malloc(c * sizeof(int));
    }
    int k = 1;
    for (int i = 0; i < r; ++i) {
        for (int j = 0; j < c; ++j) {
            arr[i][j] = k++;
        }
    }
    int idx = 0;
    for (int i = 0; i < c; ++i) {
        for (int j = 0; j < r; ++j) {
            p[idx++] = arr[j][i];
        }
    }
    for (int i = 0; i < r; i++) {
        free(arr[i]);
    }
    free(arr);
    return p;
}

int main() {
    int r, c;
    printf("Enter the number of rows and columns for the interleaver matrix (r, c): ");
    scanf("%d %d", &r, &c);
    int* p = Block_Interleaver(r, c);
    int size = r * c;
    printf("Interleaved Data: ");
    Print(p, size);
    free(p);
    return 0;
}
```

3. **Random Interleaver**

A Random Interleaver rearranges the input data by shuffling the elements using a random number generator. The shuffling helps to mitigate the effects of burst errors, which tend to affect consecutive bits in data.

How It Works: The program uses a user-provided seed to initialize the random number generator. This seed ensures that the shuffling process is repeatable. The data is then reordered according to a randomly generated sequence of indices.

Purpose: By randomizing the order of data, this interleaver reduces the likelihood of burst errors affecting consecutive bits, making error correction more effective.

Random Interleaver Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int* interleaver(int* data, int size, int key) {
    int* p = (int*) malloc(size * sizeof(int));
    int* indices = (int*) malloc(size * sizeof(int));
    srand(key);
    for (int i = 0; i < size; i++) {
        indices[i] = i;
    }
    for (int i = 0; i < size; i++) {
        int j = rand() % size;
        int temp = indices[i];
        indices[i] = indices[j];
        indices[j] = temp;
    }
    for (int i = 0; i < size; i++) {
        p[i] = data[indices[i]];
    }
    free(indices);
    return p;
}

void printArray(int* arr, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int size, key;
    printf("Enter the number of elements (size): ");
    scanf("%d", &size);
    printf("Enter the key (seed): ");
```

```
        scanf("%d", &key);
        int* data = (int*) malloc(size * sizeof(int));
        for (int i = 0; i < size; i++) {
            data[i] = i + 1;
        }
        int* interleavedData = interleaver(data, size, key);
        printf("Interleaved data: ");
        printArray(interleavedData, size);
        free(data);
        free(interleavedData);
        return 0;
    }
```

§6.5.3 Non-Systematic Convolutional (NSC) Code

Overview

1. **Encoder Structure and Operation**
   A non-systematic convolutional encoder uses k memory registers and n modulo-2 adders to produce n output bits per input bit. For a rate-$1/2$ encoder with constraint length $K = 3$, the generator polynomials are:

$$\begin{aligned} G_1 = (1,1,1) \quad &\Rightarrow \quad n_1 = m_1 \oplus m_0 \oplus m_{-1} \\ G_2 = (0,1,1) \quad &\Rightarrow \quad n_2 = m_0 \oplus m_{-1} \end{aligned} \tag{6.1}$$

   Output bits are linear combinations of current and past inputs, with $\oplus$ denoting modulo-2 addition.

2. **Key Properties**
   - Generator Polynomials Defined through shift-register connections, represented as binary vectors or polynomial expressions in delay operator D:

$$\mathbf{G} = [G^{(1)}(D), G^{(2)}(D), ..., G^{(n)}(D)] \tag{6.2}$$

   - Free Distance The minimum Hamming distance between codewords determines error correction capability:

$$t = \left\lfloor \frac{d_{\text{free}} - 1}{2} \right\rfloor \tag{6.3}$$

     Non-systematic codes typically achieve higher d_{free} than systematic counterparts.
   - Linearity and Group Property The code forms a linear subspace under polynomial addition, satisfying:

$$\mathcal{C}(a(D) + b(D)) = \mathcal{C}(a(D)) + \mathcal{C}(b(D)) \tag{6.4}$$

3. **Decoding**
   The Viterbi algorithm uses these key steps:
   a) Branch metric calculation: Hamming distance between received bits and possible outputs
   b) Path metric accumulation: $M_t(s_i) = \min[M_{t-1}(s_j) + BM(s_j \to s_i)]$
   c) Survivor path selection through trellis traceback

Dependencies

```
import numpy as np
from typing import Dict, List
```

Function Descriptions

1. `build_trellis(generator_polys: List[int], constraint_length: int) -> Dict[tuple, tuple]`
   - Purpose: Constructs trellis diagram for convolutional encoding.
   - Methodology:
     - Generates 2^{K-1} states where K = constraint length.
     - For each state and input bit (0/1)
       a) Computes next state using bit shift: next_state = (state $\ll$ 1) & (n_states − 1).
       b) Calculates output bits via polynomial XOR operations:

$$\text{output\_bits}[j] = \bigoplus_{i=0}^{K-1} (g_j^{(i)} \cdot \text{state\_bit}[i])$$

       where g_j is the j-th generator polynomial
   - Key Parameters:
     - `generator_polys`: Octal representations of generator polynomials
     - `constraint_length`: Determines state space size and memory depth

2. `viterbi_decode(received: List[int], trellis: Dict[tuple, tuple], n_states: int, n_outputs: int) -> List[int]`
   - Purpose: Implements maximum likelihood sequence estimation using Viterbi algorithm.
   - Methodology:
     a) Initializes path metrics matrix with ∞ except start state (0)
     b) Forward pass:
       - For each timestep, calculates Hamming distance between received bits and expected outputs
       - Updates path metrics using:

$$\text{path\_metric}[t, s] = \min(\text{path\_metric}[t-1, s'] + \text{branch\_metric})$$

       - Stores survivor paths
     c) Traceback:
       - Starts from minimum metric state at final timestep
       - Reconstructs input bits by following survivor paths backward

Example Execution

```
Generator polynomials: decimal 7,5 → octal 111,101
Constraint length: 3
Received sequence: 11 01 11 01 10
Decoded message: [1, 0, 1, 1, 0]
```

Notes

- Polynomial handling: Decimal inputs converted to binary coefficients
- State management: Bitmasking used for efficient state transitions
- Received sequence processing: Assumes hard-decision inputs
- Traceback: Implements register-exchange method with reverse traversal

Parameter	Value
Constraint Length (K)	3
Memory Elements	2
States	4
Code Rate	1/2

§6.5.4 Product Code Implementation

Overview

This script implements encoding and decoding for a Product Code using simulated transmission over an AWGN BPSK channel. Parity bits are computed using per-pair XORs along both rows and columns, and the decoder uses a min-sum based message-passing approach to iteratively decode the received bits.

Let, $L(d) = log(\frac{P(d=+1}{P(d=-1)})$ is the apriori LLR for input bit d. The output LLR

$$L(\hat{d}) = L_c(x) + L(d) + L_e(\hat{d})$$

where $L_c(x)$ is the LLR of channel measurement at receiver, calculated using MAP- Maximum a posteriori rule, $L_e(\hat{d})$ is the extrinsic LLR value.

$$L_c(x_k) = \log_e \left[\frac{p(x_k \mid d_k = +1)}{p(x_k \mid d_k = -1)}\right]$$

which simplifies to

$$L_c(x_k) = \frac{2}{\sigma^2} x_k$$

. For the product code, the iterative decoding algorithm proceeds as follows:

1. Set the a priori LLR, $L(d)$
2. Decode horizontally, and using $L(\hat{d}) = L_c(x) + L(d) + L_e(\hat{d})$ obtain the horizontal extrinsic LLR as shown below:
$$L_{eh}(\hat{d}) = L(\hat{d}) - L_c(x) - L(d)$$
3. Set $L(d) = L_{eh}(\hat{d})$ for the vertical decoding of step 4.
4. Similarly, decode vertically and obtain the vertical extrinsic LLR as shown below:
$$L_{ev}(\hat{d}) = L(\hat{d}) - L_c(x) - L(d)$$
5. Set $L(d) = L_{ev}(\hat{d})$ and repeat steps 2 through 5.
6. After enough iterations (that is, repetitions of steps 2 through 5) to yield a reliable decision, go to step 7.
7. The soft output is
$$L(\hat{d}) = L_c(x) + L_{eh}(\hat{d}) + L_{ev}(\hat{d})$$

Dependencies

```
import numpy as np
from itertools import combinations
```

Function Descriptions

1. `simulate_awgn_bpsk_channel`

   Purpose: Simulates BPSK transmission over an AWGN channel and returns the Log-Likelihood Ratio (LLR) values.
   Inputs: The bits/ bit matrix which is to be passed through the channel and signal to noise ratio in decibel.
   Returns: `llr` Log likelihood ratio matrix of the input matrix after noise addition.

2. `encoder`

   Purpose: Encodes the $d \times d$ input data matrix, generates per-pair parity values along rows and columns, and simulates their transmission over AWGN.
   For calculating the parity dictionary, we XOR the respective data_matrix values.

$$d_i \oplus d_j = p_{ij}$$

   Returns:

   - `Lc_matrix` – LLR for each data bit.
   - `parity_h` – Dictionary of row-wise XOR parities with keys `(i,j1,j2)` for row `i` and pair`(j1,j2)`
   - `parity_v` – Dictionary of column-wise XOR parities with keys `(i1,i2,j)` with column `j` and pair `(i1,i2)`

3. `min_sum_xor`

   Purpose: Implements a min-sum approximation for XOR constraints in decoding.

4. `decoder`

   Purpose: Decodes the data matrix using iterative message passing. Each iteration consists of row-wise and column-wise extrinsic LLR updates using min-sum decoding.
   We use the information obtained of d_i from the d_js using the XORed parity bits.

$$d_i = d_j \oplus p_{ij} \qquad i, j = 1, 2$$

   For example: If the data matrix were 2x2, the Extrinsic LLR of d_1 will be sum of these two:

$$L_{eh}(\hat{d}_1) = \left[L_c(x_2) + L(\hat{d}_2) \right] \boxplus L_c(x_{12})$$

$$L_{ev}(\hat{d}_1) = \left[L_c(x_3) + L(\hat{d}_3) \right] \boxplus L_c(x_{13})$$

   For Horizontal Extrinsic value updates,

```
    L_horizontal = L.copy()
        for (i, j1, j2), parity in parity_h.items():
            ext_j1 = min_sum_xor(L[i, j2] + Lc_matrix[i, j2], parity)
            ext_j2 = min_sum_xor(L[i, j1] + Lc_matrix[i, j1], parity)
            L_horizontal[i, j1] += ext_j1
            L_horizontal[i, j2] += ext_j2
```

Returns:

- `L` – Final LLR matrix after decoding.
- `decoded_bits` – Final decision bits from LLRs.

Example Execution

```
Lc_matrix, parity_h, parity_v = encoder([[1,0],[0,1]], 10*np.log10(0.5))
print(Lc_matrix)
L, decoded_bits = decoder(Lc_matrix, [[0,0],[0,0]], parity_h, parity_v, 2)
print(L)
print(decoded_bits)
```

```
[[-1.86797365 -2.34015361]
 [-2.56058726  0.60310293]]
[[ 6 -5]
 [-4  8]]
[[1 0]
 [0 1]]
```

Notes

- This implementation is designed for square matrices ($d \times d$).
- XOR parity is taken over all possible pairs in each row and column.
- LLRs are updated iteratively using a hard min-sum rule.
- The decoder supports use of non-zero a priori input LLRs.

§6.5.5 Turbo Code Implementation with Libraries

Overview

1. **Encoding** Turbo coding starts with data encoding using two RSC encoders. The original data and its interleaved version are each fed into one encoder. Each encoder produces parity bits. Together with the original bits, they form the transmitted signal.
2. **Interleaving** The interleaver scrambles the order of the input bits before the second encoding step, distributing errors to enhance error correction during decoding.
3. **Puncturing (Not Implemented)** This optional step omits some parity bits to adjust the code rate. It helps in achieving higher data throughput at the cost of slightly reduced error performance.

Dependencies

This implementation uses Python, TensorFlow, and the Sionna library. TensorFlow enables tensor operations and Sionna provides optimized communication primitives including the BCJR decoder.

Function Descriptions

1. `gaussian_channel_simulator`
   Purpose: Adds Gaussian noise to the modulated signal.
   Inputs: Encoded data list, standard deviation of the Gaussian noise.
   Output: Noisy encoded signal, simulating a realistic channel.

2. `zero_padding`
   Purpose: Ensures the message length is a multiple of the block size by appending zeros.

3. `rs_encoder`
   Purpose: Applies (7,5) Recursive Systematic Convolutional encoding.
   Inputs: Message list, number of information bits in a block, option to enforce final state (0,0).
   Output: Encoded message with parity appended.

4. `interleaver`
   Purpose: Applies a permutation based on modular multiplication to scatter bits. But, for this interleaver to work properly, number of information bits in a block must be of the form of a prime - 2, the seed also has some constraints
   Inputs: seed(a number), number of information bits in a block + 2, list/array to be permuted.
   Output: Interleaved list/array.

5. `turbo_encoder_without_puncturing`
   Purpose: Combines two RSC encoders and an interleaver to generate systematic and redundant bits.

6. `mix_sys_parity`
   Purpose: Alternates between systematic and parity data for combined output.

7. `turbo_decode`
   Purpose: Iteratively decodes received sequences using the BCJR algorithm imported from sionna library. Here, the seed and its inverse should be such that seed*inverse must give remainder 1 when divided by number of information bits per block+2
   Inputs: Systematic, parity1, parity2 sequences; interleaver parameters(seed and its inverse, block length); number of iterations.
   Output: Final log-likelihood ratios after iterations.

8. `hard_message_output`
   Purpose: Converts LLRs into binary hard decisions. Removes final two bits added for (0,0) state.

9. `no_of_bit_errors_finder`
   Purpose: Computes the number of bit mismatches between original and decoded sequences.

Example Execution

```
Encoded message: [1, 0, 1, 0, ..., 0]
Noisy channel output: [1, 0, 1, ..., 0]
Padded original message: [1, 0, 1, 0, ..., 0]
Decoded message: [1, 0, 1, 0, ..., 0]
Bit errors: 6
Bit errors without coding: 173
```

§6.6 Polar Codes Python Implementation Documentation

§6.6.1 Introduction

This document provides a detailed explanation of the Python implementation of Polar Codes as provided in the file. Polar codes are a class of capacity-achieving error-correcting codes. This document explains each function used in the implementation.

§6.6.2 Function Documentation

`gaussian_channel_simulator`

Purpose: Simulates the transmission of a signal over an additive white Gaussian noise (AWGN) channel.

```python
def gaussian_channel_simulator(encoded_message_arr, std_dev_of_error):
    modulated_arr = (2 * encoded_message_arr) - 1
    return np.random.normal(0, std_dev_of_error, modulated_arr.shape) + modulated_arr
```

This modulates bits opposite to normal BPSK format, i.e, 0 is mapped to -1 and 1 to mapped to 1, then adds Gaussian noise of given standard deviation.

`worst_to_best_channel_indices_arr_reducer`

Purpose: Reduces the worst to best channel indices array to the required number of channels.

```python
def worst_to_best_channel_indices_arr_reducer(no_of_channels, worst_to_best_channel_indices_arr):
    ...
```

Ensures the returned array has indices only within the range [0, no_of_channels-1].

`channel_inputs_organiser`

Purpose: Arranges the input bits and frozen bits before encoding.

```python
def channel_inputs_organiser(depth_of_tree, worst_to_best_channel_indices_arr, message_list):
    ...
```

Pads the message with zeros (frozen bits) and reorders them so that the frozen bits go to the least reliable channels.

`polar_operation`

Purpose: Combines two lists using bitwise XOR followed by second list as it is.

```python
def polar_operation(list1, list2):
    ...
```

Returns the XOR result followed by the second list.

`polar_encoder`

Purpose: Encodes a message using the Polar encoding scheme.

```python
def polar_encoder(depth_of_tree, ...):
    ...
```

Constructs the encoded message by recursively applying polar operation.

`Node` Class

Purpose: Defines a node in the successive cancellation decoding tree. Each node processes log-likelihood ratios (LLRs), and forwards decisions. LLRs have not been exactly used, the received values have been directly used. Includes:

- `minsum_calculater`: Min-sum approximation for the log sum exp, this is the llr's for the left child.
- `llr_for_right_child_calculator`: Computes LLRs for right child, using the hard output from the left node
- `create_and_send_info_to_left/right_child`: Creates a new node and computes their respective LLRs.
- `send_info_above`: Propagates bits upward for further decoding.

`frozen_bits_remover`

Purpose: Removes padded zeros(frozen bits) from the decoded message.

```
def frozen_bits_remover(decoded_message_list, ...):
    ...
```

Returns the actual decoded message by excluding frozen bits.

`polar_decoder`

Purpose: Decodes a single chunk of received bits.

```
def polar_decoder(received_list, ...):
    ...
```

Implements Successive Cancellation decoding using the Node class.

`complete_polar_encoder` and `complete_polar_decoder`

Purpose: Handle full-length messages by dividing them into chunks. The polar encoder and decoder used before could only handle small message lengths, length of message must be less than number of channels there. Here, it breaks the big message into chunks and overcomes that issue.

```
def complete_polar_encoder(...):
    ...

def complete_polar_decoder(...):
    ...
```

Breaks a long message into multiple parts and processes each chunk separately.

`no_of_bit_errors_finder`

Purpose: Calculates bit error count.

```
def no_of_bit_errors_finder(message_list, decoded_message_list):
    ...
```

Compares the decoded message with the original and returns the number of bit errors.

§6.7 Polar Codes: Structure, Encoding, and Decoding(Explanation)

§6.7.1 Channel Polarization

The core idea is to *polarize* a set of identical channels into a set of very reliable (good) and very unreliable (bad) channels.

This is achieved by recursive combining and splitting of channels.

Given a binary-input channel W, define:

$$W^-(y_1, y_2|u_1) = \sum_{u_2} \frac{1}{2} W(y_1|u_1 \oplus u_2) W(y_2|u_2),$$

$$W^+(y_1, y_2, u_1|u_2) = \frac{1}{2} W(y_1|u_1 \oplus u_2) W(y_2|u_2),$$

where $u_1, u_2 \in \{0, 1\}$ and $\oplus$ denotes modulo-2 addition.

As recursion depth increases, some synthetic channels become noiseless (W^+), and some become completely noisy (W^-).

§6.7.2 Encoding

For blocklength $N = 2^n$, the generator matrix is:

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

where:

- B_N is the bit-reversal permutation matrix,
- $F = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ is the basic kernel,
- $F^{\otimes n}$ denotes the n-fold Kronecker product.

Encoding involves:

1. Insert known *frozen bits* (typically zeros) at bad channels.
2. Insert *information bits* at good channels.
3. Apply the generator matrix G_N to the input vector.

§6.7.3 Decoding: Successive Cancellation

Decoding proceeds bit-by-bit, using the already decoded bits.

For bit u_i, the decoder computes likelihoods:

$$L(u_i) = \log \frac{P(y|u_0^{i-1}, u_i = 0)}{P(y|u_0^{i-1}, u_i = 1)},$$

and makes a decision:

$$\hat{u}_i = \begin{cases} 0, & \text{if } L(u_i) > 0, \\ 1, & \text{otherwise.} \end{cases}$$

Frozen bits are set to known values during decoding.

§6.7.4 Applications of Polar Codes

Polar codes have entered practical use, including:

- **5G NR standard**: Polar codes for control channels.
- **Low-latency communications**: Due to their fast decodability.
- **Ultra-reliable low-latency communication (URLLC)**: Future wireless technologies.

§6.7.5 Figures and Plots

Channel Polarization Diagram

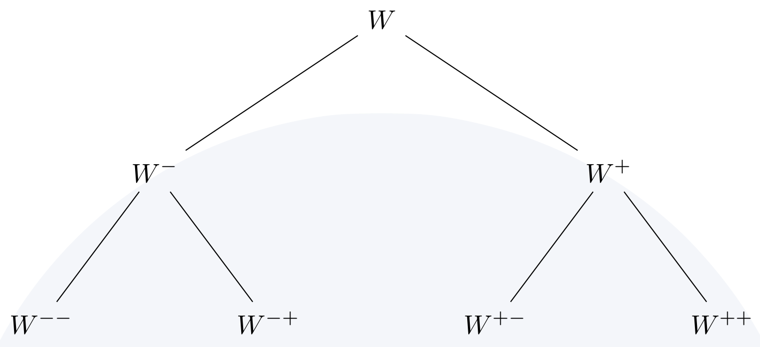


Figure 6.3: Recursive channel polarization: each step doubles the number of synthetic channels.

§6.7.6 Error Probability vs Blocklength

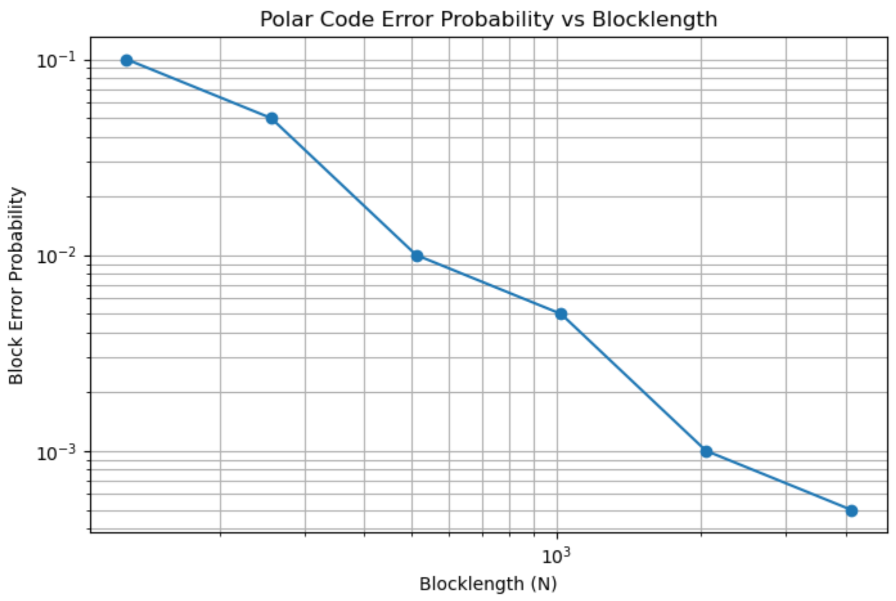


Figure 6.4: Polar code block error rate vs blocklength under successive cancellation decoding.

As blocklength N increases, the fraction of good channels approaches the channel capacity C.

§6.8 Conclusion

Polar codes achieve channel capacity with low complexity through the remarkable phenomenon of channel polarization. Their structured recursive construction and efficient successive cancellation decoding make them highly attractive for modern communication systems.