Introduction to Information Theory

Lecture 13

Lecturer: Haim Permuter *Scribe: Gilad Kliger*

Turbo Codes & Iterative Decoding

I. Introduction

Turbo codes were introduced by Berrou, Glavieux, and Thitimajshima in 1993. They were the first practical error-correcting codes that operated very close to the Shannon limit.

Before turbo codes, practical systems typically used convolutional codes, Viterbi decoding, Reed–Solomon codes, or concatenated coding schemes. Turbo codes introduced a new principle:

Use two simple codes and decode them iteratively by exchanging soft information.

This idea became known as the **turbo principle**.

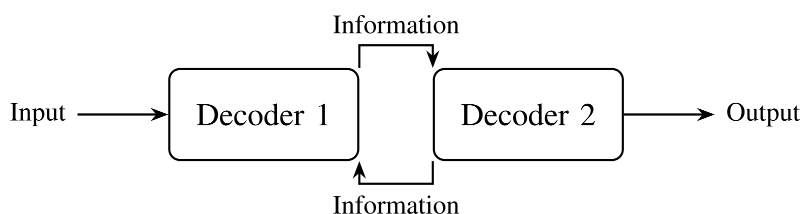

Fig. 1. The Iterative Decoding Principle.

A. The Turbo Principle

The name "turbo" is inspired by a turbo engine. In a turbo engine, energy is reused in order to improve performance. In a turbo decoder, information is reused iteratively:

$$\text{Decoder 1} \rightarrow \text{Decoder 2} \rightarrow \text{Decoder 1} \rightarrow \cdots \tag{1}$$

Each decoder produces new information that helps the other decoder.

II. Turbo Encoder Structure

A. Review: Convolutional Codes

A convolutional encoder has memory. At each time k, the encoder input is u_k, and the output depends on the current input and previous inputs.

For example, consider a rate-$1/2$ convolutional code with

$$g_1(D) = 1, \qquad g_2(D) = 1 + D. \tag{2}$$

Then the output is

$$v_{1,k} = u_k, \qquad v_{2,k} = u_k + u_{k-1}. \tag{3}$$

The first output is systematic because it equals the input bit.

B. Recursive Systematic Convolutional (RSC) Codes

Turbo codes usually use **recursive systematic convolutional** encoders, called **RSC encoders**. Before defining them formally, let us start with the simplest possible example.

1) A Very Simple Systematic Encoder: A systematic encoder sends the information bit itself as one of the outputs. For example, suppose

$$v_{s,k} = u_k. \tag{4}$$

Then the receiver directly observes a noisy version of the information bit. This is called the **systematic output**.

Now suppose we also generate a parity bit

$$v_{p,k} = u_k + u_{k-1}, \tag{5}$$

where addition is modulo 2. This is a simple non-recursive systematic convolutional encoder.

The generator matrix is

$$G(D) = [1,\ 1 + D]\,. \tag{6}$$

The first component, 1, produces the systematic bit. The second component, $1 + D$, produces the parity bit.

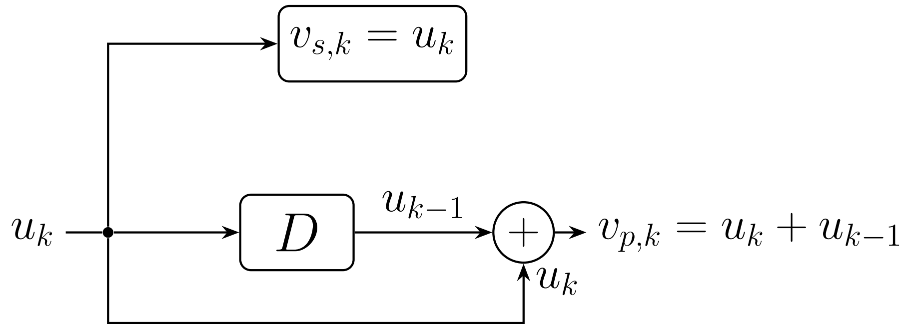

Fig. 2. Systematic non-recursive convolutional encoder ($G(D) = [1, 1 + D]$).

This encoder is systematic, but it is **not recursive**. There is no feedback.

2) Impulse Response of a Non-Recursive Encoder: Let the input be a single one followed by zeros:

$$u = (1, 0, 0, 0, 0, \ldots). \tag{7}$$

For the parity rule

$$v_{p,k} = u_k + u_{k-1}, \tag{8}$$

we obtain

$$v_p = (1, 1, 0, 0, 0, \ldots). \tag{9}$$

Thus, a single input 1 creates only two nonzero parity bits. The response dies out quickly.

$$\boxed{\text{Non-recursive encoder} \quad \Longrightarrow \quad \text{finite impulse response}} \tag{10}$$

3) Adding Feedback: The Simplest Recursive Encoder: Now consider a recursive encoder. Define the encoder state sequence s_k by

$$s_k = u_k + s_{k-1}, \tag{11}$$

where addition is modulo 2. The parity output is

$$v_{p,k} = s_k. \tag{12}$$

Equivalently,

$$s_k + s_{k-1} = u_k. \tag{13}$$

Using the delay operator D, this becomes

$$s(D)(1 + D) = u(D), \tag{14}$$

and therefore

$$s(D) = \frac{1}{1 + D} u(D). \tag{15}$$

Hence the generator matrix is

$$G(D) = \left[1, \frac{1}{1 + D}\right]. \tag{16}$$

The denominator $1 + D$ is the feedback polynomial. This denominator is what makes the encoder recursive.

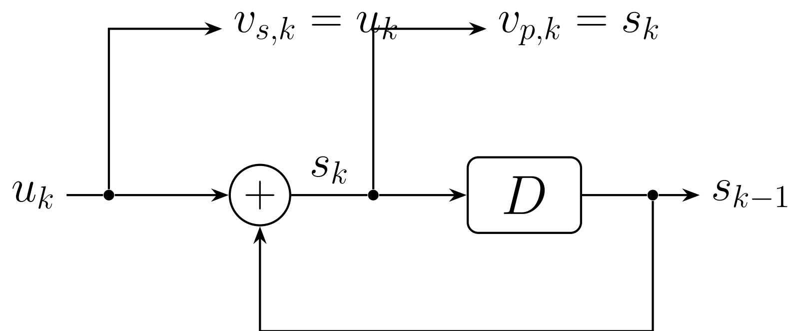

Fig. 3. Simplest Recursive Systematic Convolutional (RSC) encoder ($G(D) = [1, \frac{1}{1+D}]$).

The corresponding trellis has two states, given by the stored bit $s_{k-1} \in \{0, 1\}$. Each branch is labeled $u_k/(v_{s,k}v_{p,k})$.

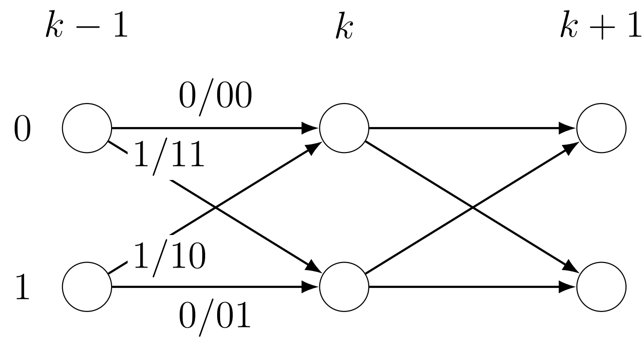

Fig. 4. Two-state trellis of the recursive encoder.

4) Impulse Response of the Recursive Encoder: Again take

$$u = (1, 0, 0, 0, 0, \ldots), \tag{17}$$

and assume

$$s_{-1} = 0. \tag{18}$$

Since $s_k = u_k + s_{k-1}$, we get

$$\begin{aligned} s_0 &= 1 + 0 = 1, \\ s_1 &= 0 + 1 = 1, \\ s_2 &= 0 + 1 = 1, \\ s_3 &= 0 + 1 = 1. \end{aligned} \tag{19}$$

Therefore,

$$v_p = (1, 1, 1, 1, 1, \ldots). \tag{20}$$

A single input 1 creates a long parity sequence.

$$\boxed{\text{Recursive encoder} \quad \Longrightarrow \quad \text{infinite impulse response}} \tag{21}$$

This is the first key reason recursive encoders are used in turbo codes.

5) Comparison: Non-Recursive vs Recursive:

Input u	Non-recursive parity $1 + D$	Recursive parity $\frac{1}{1+D}$
$100000\ldots$	$110000\ldots$	$111111\ldots$

(22)

The same low-weight input sequence produces a much longer parity sequence in the recursive encoder.

6) A More Useful RSC Encoder: The very simple recursive encoder is useful for explanation, but practical turbo codes usually use encoders with more memory. A common example is

$$G(D) = \left[1, \frac{1 + D^2}{1 + D + D^2}\right]. \tag{23}$$

Here $g_f(D) = 1 + D^2$ is the feedforward polynomial, and $g_b(D) = 1 + D + D^2$ is the feedback polynomial. The encoder has memory $m = 2$, so it has $2^m = 4$ states.

7) Difference Equation: Let s_k be the convolutional encoder state. The feedback relation is

$$s_k + s_{k-1} + s_{k-2} = u_k. \tag{24}$$

Thus,

$$s_k = u_k + s_{k-1} + s_{k-2}. \tag{25}$$

The parity output is generated by the feedforward polynomial $1 + D^2$, so

$$v_{p,k} = s_k + s_{k-2}. \tag{26}$$

Therefore the encoder is described by

$$\boxed{\begin{aligned} s_k &= u_k + s_{k-1} + s_{k-2}, \\ v_{s,k} &= u_k, \\ v_{p,k} &= s_k + s_{k-2}. \end{aligned}} \tag{27}$$

All additions are modulo 2.

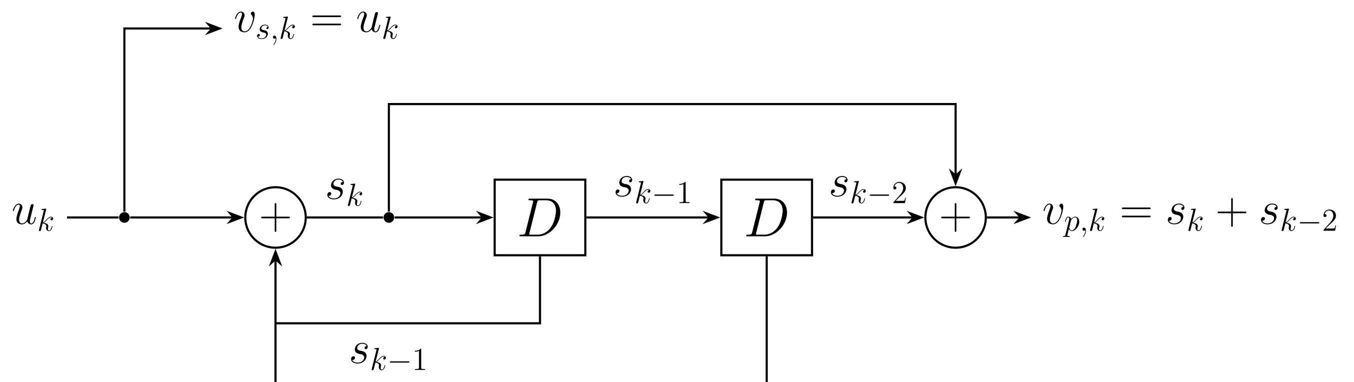

Fig. 5. Improved Recursive Systematic Convolutional (RSC) encoder with memory $m = 2$.

8) Improved RSC Encoder Diagram:

9) Example Calculation: Consider the input sequence $u = (1, 0, 0, 0, 0, 0)$, and assume $s_{-1} = s_{-2} = 0$. Using $s_k = u_k + s_{k-1} + s_{k-2}$, we get:

k	u_k	s_{k-1}	s_{k-2}	s_k
0	1	0	0	1
1	0	1	0	1
2	0	1	1	0
3	0	0	1	1
4	0	1	0	1
5	0	1	1	0

Now compute the parity $v_{p,k} = s_k + s_{k-2}$.

k	s_k	s_{k-2}	$v_{p,k}$
0	1	0	1
1	1	0	1
2	0	1	1
3	1	1	0
4	1	0	1
5	0	1	1

Thus, $v_p = (1, 1, 1, 0, 1, 1, \ldots)$. Again, a single input 1 creates a long parity response.

10) Why This Matters for Turbo Codes: Turbo codes are powerful because they combine recursive encoders, systematic transmission, interleaving, and iterative soft decoding. The recursive encoders make low-weight inputs produce long parity sequences. The interleaver makes sure that if a low-weight input pattern is bad for the first encoder, then its permuted version is usually not bad for the second encoder.

11) Key Takeaway:

$$\boxed{\text{Turbo codes use RSC encoders because recursion spreads information over time.}} \tag{28}$$

A non-recursive encoder forgets a single input quickly. A recursive encoder remembers it through feedback. This memory is exactly what makes iterative turbo decoding effective.

C. Overall Turbo Encoder Output

A turbo encoder consists of:

- an information sequence u,
- a first RSC encoder,
- an interleaver Π,
- a second RSC encoder,
- systematic bits and two parity streams.

The output is usually $x_k = (u_k, p_{1,k}, p_{2,k})$, so the mother code has rate $R = \frac{1}{3}$. Higher rates are obtained by puncturing. In Figure 6 we present the turbo encoder diagram.

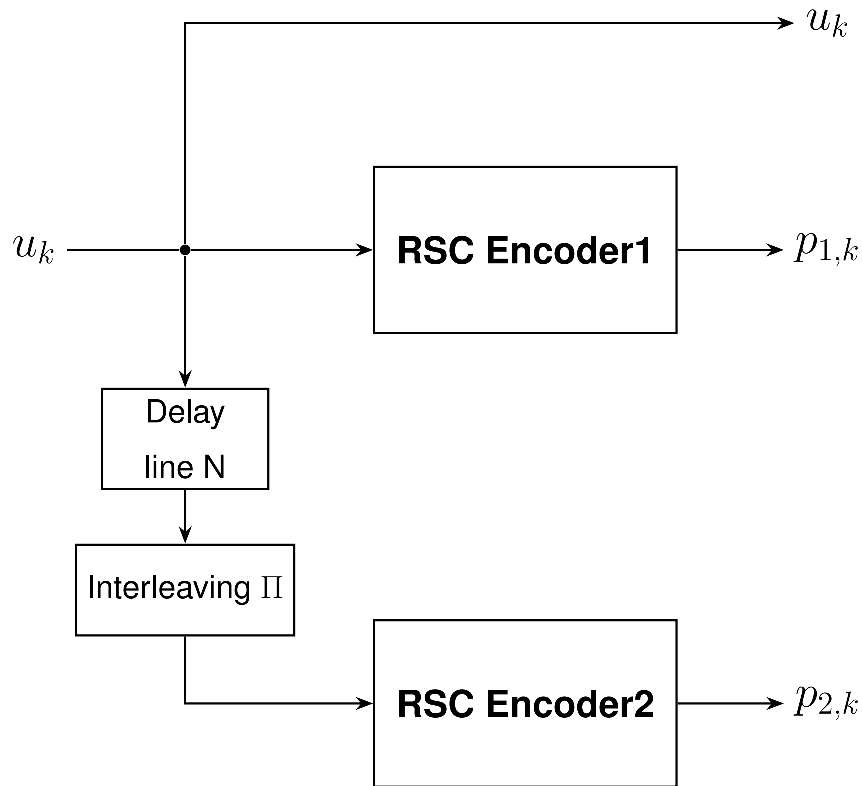

Fig. 6. Parallel concatenated turbo encoder with two RSC constituent encoders separated by an interleaver.

D. The Interleaver (Π*)*

The interleaver permutes the input sequence. If $u = (u_1, u_2, \ldots, u_N)$, then $u^\Pi = (u_{\Pi(1)}, u_{\Pi(2)}, \ldots, u_{\Pi(N)})$. The purpose is that the two encoders see the same information bits in different orders. Thus, a low-weight pattern that is bad for the first encoder will usually not be bad for the second encoder.

III. Iterative Decoding Principle

A. Channel Model

Assume BPSK modulation: $0 \mapsto +1,\ 1 \mapsto -1$. The received symbol is $y_k = x_k + n_k$, where $n_k \sim \mathcal{N}(0, \sigma^2)$.

For BPSK over AWGN, the channel log-likelihood ratio is

$$L_c(y_k) = \log \frac{p(y_k|u_k = 1)}{p(y_k|u_k = 0)}. \tag{29}$$

For the convention used in these notes ($0 \mapsto +1$ and $1 \mapsto -1$), the sign is reversed:

$$L_c(y_k) = -\frac{2y_k}{\sigma^2}. \tag{30}$$

Positive y_k supports bit 0, and negative y_k supports bit 1. A larger $|L_c(y_k)|$ means greater confidence.

B. Soft Information and APP Decoding

Turbo decoding is based on soft information. For a bit u_k, define the log-likelihood ratio

$$L(u_k) = \log \frac{P(u_k = 1|Y)}{P(u_k = 0|Y)}. \tag{31}$$

If $L(u_k) > 0$, then bit 1 is more likely. If $L(u_k) < 0$, then bit 0 is more likely. The magnitude $|L(u_k)|$ measures the reliability.

APP stands for **a posteriori probability**. An APP decoder computes $P(u_k = b|Y)$, and therefore computes the APP LLR. The BCJR algorithm is an APP decoding algorithm for convolutional codes.

C. Trellis Representation

A convolutional code can be represented by a trellis. Let s_k be the encoder state at time k. A branch $s_{k-1} \to s_k$ corresponds to an input bit u_k and output symbols.

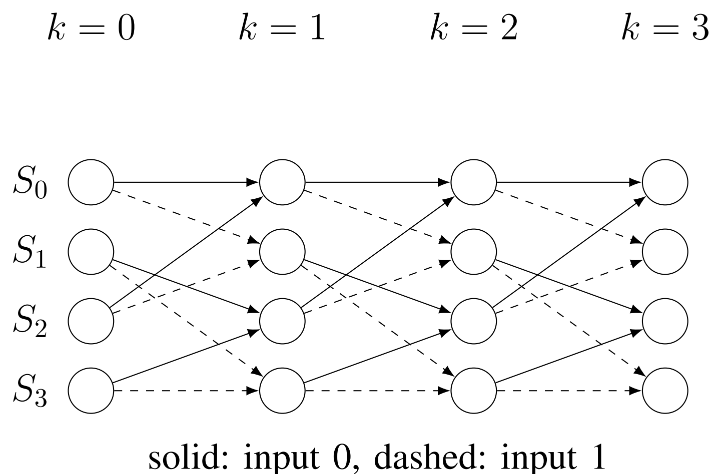

Fig. 7. Four-state convolutional-code trellis over four time instants.

D. The BCJR Algorithm

The BCJR algorithm computes the APP probability of each input bit by summing over all trellis paths that are consistent with that bit.

1) Forward Metric: Define

$$\alpha_k(s) = P(s_k = s, Y_1^k). \tag{32}$$

The recursion is

$$\alpha_k(s) = \sum_{s'} \alpha_{k-1}(s')\gamma_k(s', s). \tag{33}$$

2) Backward Metric: Define

$$\beta_k(s) = P(Y_{k+1}^N | s_k = s). \tag{34}$$

The recursion is

$$\beta_{k-1}(s') = \sum_{s} \gamma_k(s', s)\beta_k(s). \tag{35}$$

3) Branch Metric: The branch metric is

$$\gamma_k(s', s) = P(y_k, s_k = s | s_{k-1} = s') = P(u_k)P(y_k | x_k(s', s)). \tag{36}$$

In turbo decoding, $P(u_k)$ is determined by the a priori information coming from the other decoder.

4) APP Probability: The probability that $u_k = b$ is obtained by summing over all branches labeled by $u_k = b$. Therefore,

$$L(u_k) = \log \frac{\sum\limits_{(s',s):u_k=1} \alpha_{k-1}(s')\gamma_k(s', s)\beta_k(s)}{\sum\limits_{(s',s):u_k=0} \alpha_{k-1}(s')\gamma_k(s', s)\beta_k(s)}. \tag{37}$$

IV. Decoder Implementation: Log-Domain Algorithms

A. Why Log-Domain BCJR?

The quantities $\alpha_k(s)$, $\beta_k(s)$, $\gamma_k(s', s)$ can become extremely small. For long blocks, $\alpha_k(s) \approx 10^{-100}$ or smaller. This creates numerical underflow. The solution is to work in the logarithmic domain.

B. Log-Domain Metrics and Jacobian Logarithm

Define

$$A_k(s) = \log \alpha_k(s), \quad B_k(s) = \log \beta_k(s), \quad \Gamma_k(s', s) = \log \gamma_k(s', s). \tag{38}$$

The main operation in Log-BCJR is $\log(e^x + e^y)$. Define

$$\max{}^\star(x, y) = \log(e^x + e^y) = \max(x, y) + \log\left(1 + e^{-|x-y|}\right). \tag{39}$$

The term $\log\left(1 + e^{-|x-y|}\right)$ is called the correction term.

C. Log-MAP Forward and Backward Recursions

1) Forward Recursion: Using the Jacobian logarithm,

$$A_k(s) = \max{}^\star \left\{A_{k-1}(s') + \Gamma_k(s', s) : s' \to s\right\}. \tag{40}$$

2) Backward Recursion: Similarly,

$$B_{k-1}(s') = \max{}^\star_s \left[\Gamma_k(s', s) + B_k(s)\right]. \tag{41}$$

D. Log-MAP APP LLR

Define $M_k(s', s) = A_{k-1}(s') + \Gamma_k(s', s) + B_k(s)$. Then,

$$L(u_k) = \max{}^\star_{(s',s):u_k=1} M_k(s', s) - \max{}^\star_{(s',s):u_k=0} M_k(s', s). \tag{42}$$

This is the Log-MAP form of BCJR.

E. The Max-Log-MAP Approximation

The Max-Log-MAP approximation ignores the correction term:

$$\max{}^\star(x, y) \approx \max(x, y). \tag{43}$$

The APP LLR becomes

$$L(u_k) \approx \max_{(s',s):u_k=1} M_k(s', s) - \max_{(s',s):u_k=0} M_k(s', s). \tag{44}$$

Max-Log-MAP is simpler but slightly less accurate than Log-MAP.

V. The Turbo Decoder

A. Extrinsic Information Exchange

The APP LLR produced by a decoder can be decomposed as

$$L(u_k) = L_c(u_k) + L_a(u_k) + L_e(u_k), \tag{45}$$

where $L_c(u_k)$ is channel info, $L_a(u_k)$ is a priori info, and $L_e(u_k)$ is extrinsic info. Therefore,

$$L_e(u_k) = L(u_k) - L_c(u_k) - L_a(u_k). \tag{46}$$

The extrinsic information is the new information learned by the decoder. Only $L_e(u_k)$ is passed to the other decoder.

B. Turbo Decoder Architecture

The turbo decoder contains two soft-input soft-output decoders. Each decoder is usually a Log-MAP BCJR decoder.

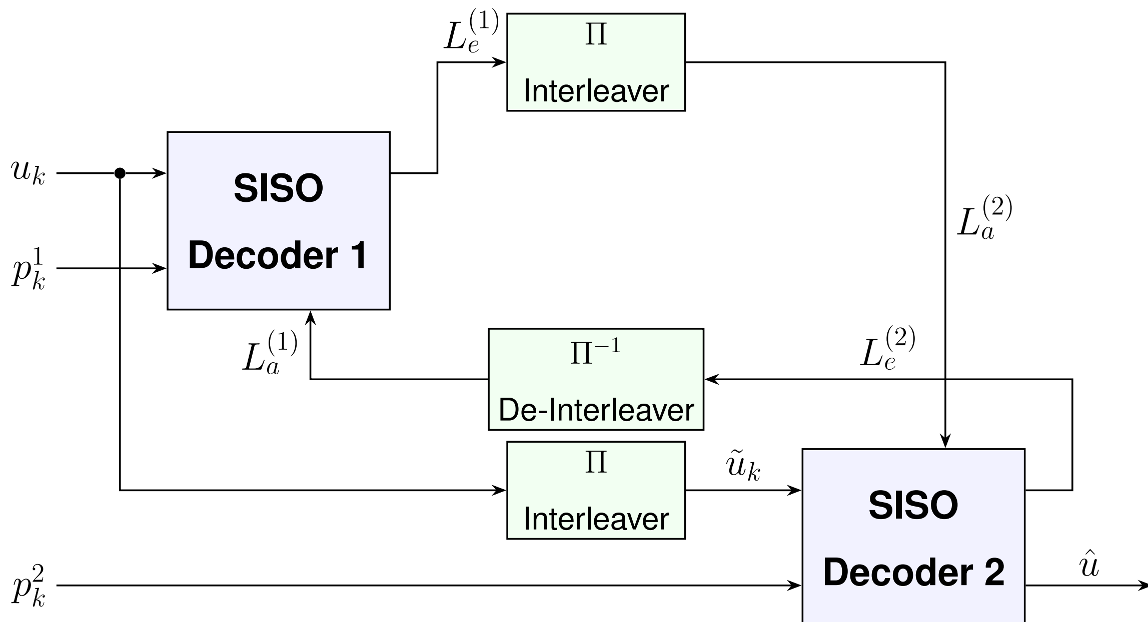

Fig. 8. Iterative turbo decoder in which two Log-MAP decoders exchange interleaved and deinterleaved extrinsic information.

C. Iterative Decoding Procedure

The turbo decoding algorithm works as follows:

Step 1: Initialize all a priori LLRs to zero: $L_a^{(1)}(u_k) = 0$.

Step 2: Decoder 1 uses y_s, y_{p1}, and $L_a^{(1)}$ to compute $L^{(1)}(u_k)$.

Step 3: Compute extrinsic information: $L_e^{(1)}(u_k) = L^{(1)}(u_k) - L_c(u_k) - L_a^{(1)}(u_k)$.

Step 4: Interleave $L_e^{(1)}$ and pass it to Decoder 2 as a priori information.

Step 5: Decoder 2 uses y_s^{Π}, y_{p2}, and $L_a^{(2)}$ to compute $L^{(2)}(u_k^{\Pi})$.

Step 6: Compute $L_e^{(2)}(u_k^{\Pi}) = L^{(2)}(u_k^{\Pi}) - L_c(u_k^{\Pi}) - L_a^{(2)}(u_k^{\Pi})$.

Step 7: Deinterleave $L_e^{(2)}$ and pass it back to Decoder 1.

Step 8: Repeat for several iterations.

Step 9: At the final iteration, make hard decisions:

$$\hat{u}_k = \begin{cases} 1, & L(u_k) > 0, \\ 0, & L(u_k) < 0. \end{cases} \tag{47}$$

D. Complete Log-MAP Algorithm

For each decoder:

1) Compute all branch metrics: $\Gamma_k(s', s)$.
2) Forward recursion: $A_k(s) = \max^\star_{s'} [A_{k-1}(s') + \Gamma_k(s', s)]$.
3) Backward recursion: $B_{k-1}(s') = \max^\star_{s} [\Gamma_k(s', s) + B_k(s)]$.
4) APP LLR:

$$L(u_k) = \max\nolimits^\star_{(s',s):u_k=1}\Big[M_k(s', s)\Big] - \max\nolimits^\star_{(s',s):u_k=0}\Big[M_k(s', s)\Big]. \tag{48}$$

5) Extrinsic information: $L_e(u_k) = L(u_k) - L_c(u_k) - L_a(u_k)$.

E. Why Turbo Decoding Works

Turbo decoding works because each decoder sees different parity constraints. Decoder 1 uses (y_s, y_{p1}), while Decoder 2 uses (y_s^{Π}, y_{p2}). The interleaver makes the two decoding problems sufficiently different. Each decoder extracts information that is not obvious to the other decoder.

F. Important Conceptual Point

A turbo decoder should not simply pass its full APP LLR to the other decoder. If it did, the same information would be reused again and again, creating positive feedback. Instead, it passes only $L_e = L - L_c - L_a$. This prevents the decoder from feeding back information that came from the other decoder in the previous iteration.

VI. Performance and Summary

A. Performance

Turbo codes achieve excellent performance because they combine:

- recursive convolutional encoders,
- large interleavers,
- soft-input soft-output decoding,
- iterative exchange of extrinsic information,
- Log-MAP or Max-Log-MAP decoding.

In practice, a small number of iterations, often between 5 and 10, is enough.

B. Summary

Turbo codes are built from two simple convolutional codes connected by an interleaver. The encoder output is (u, p_1, p_2). The decoder uses two BCJR decoders that exchange extrinsic information.

The BCJR algorithm computes APP probabilities using α, β, γ. The Log-MAP algorithm implements BCJR in the logarithmic domain using $\max^\star(x, y) = \max(x, y) + \log(1 + e^{-|x-y|})$. The Max-Log-MAP approximation replaces $\max^\star$ by $\max$.

The key turbo idea is iterative improvement:

$$\text{soft information} \rightarrow \text{extrinsic information} \rightarrow \text{better soft information.} \qquad (49)$$

References

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, “Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes,” *Proceedings of IEEE ICC*, 1993.

[2] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, “Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate,” *IEEE Transactions on Information Theory*, 1974.

[3] S. Benedetto and G. Montorsi, “Unveiling Turbo Codes: Some Results on Parallel Concatenated Coding Schemes,” *IEEE Transactions on Information Theory*, 1996.