Concatenated Convolutional Codes and Iterative Decoding

William E. Ryan
Department of Electrical and Computer Engineering
The University of Arizona
Box 210104
Tucson, AZ 85721
ryan@ece.arizona.edu

July 10, 2001

1. Introduction

Turbo codes, first presented to the coding community in 1993 [1], [2], represent one of the most important breakthroughs in coding since Ungerboeck introduced trellis codes in 1982 [3]. A turbo code encoder comprises a concatenation of two (or more) convolutional encoders and its decoder consists of two (or more) "soft" convolutional decoders which feed probabilistic information back and forth to each other in a manner that is reminiscent of a turbo engine. This chapter presents a tutorial exposition of parallel and serial concatenated convolutional codes (PCCCs and SCCCs), which we will also call parallel and serial turbo codes. Included here are a simple derivation for the performance of these codes and a straightforward presentation of their iterative decoding algorithms. The treatment is intended to be a launching point for further study in the field and to provide sufficient information for the design of computer simulations. This chapter borrows from some of the most prominent publications in the field [4]-[12].

The chapter is organized as follows. Section 2 describes details of the parallel and serial turbo code encoders. Section 3 derives a truncated union bound on the error rate of these codes under the assumption of maximum-likelihood decoding. This section explains the phenomenon of interleaver gain attained by these codes. Section 4 derives in detail the iterative (turbo) decoder for both PCCCs and SCCCs. Included in this section are the BCJR decoding algorithm for convolutional codes and soft-in/soft-out decoding modules for use in turbo decoding. The decoding algorithms are presented explicitly to facilitate the creation of computer programs. Section 5 contains a few concluding remarks.

2. Encoder Structures

Fig. 1 depicts a parallel turbo encoder. As seen in the figure, the encoder consists of two binary rate 1/2 convolutional encoders arranged in a so-called parallel concatenation, separated by a K-

bit pseudo-random interleaver or permuter. Also included is an optional puncturing mechanism to obtain high code rates [13]. Clearly, without the puncturer, the encoder is rate 1/3, mapping K data bits to $3K$ code bits. With the puncturer, the code rate $R = K/(K+P)$, where P is the number of parity bits remaining after puncturing. Observe that the constituent encoders are recursive systematic convolutional (RSC) codes. As will be seen in the sequel, recursive encoders are necessary to attain the exceptional performance (attributed to "interleaver gain") provided by turbo codes. Without any essential loss of generality, we assume that the constituent codes are identical.

Fig. 2 depicts a serial turbo encoder. As seen in the figure, the serially concatenated convolutional encoders are separated by an interleaver, and the inner encoder is required to be an RSC code, whereas the outer encoder need not be recursive [6]. However, RSC inner and outer encoders are often preferred since it is convenient to puncture only parity bits to obtain high code rates [14]. Further, an RSC outer code will facilitate our analysis below. The code rate for the serial turbo encoder is $R = R_o \cdot R_i$ where R_o and R_i are the code rates for the outer and inner codes, respectively.

For both parallel and serial turbo codes, the codeword length is $N = K/R$ bits, and we may consider both classes to be (N, K) block codes.

We now discuss in some detail the individual components of the turbo encoders.

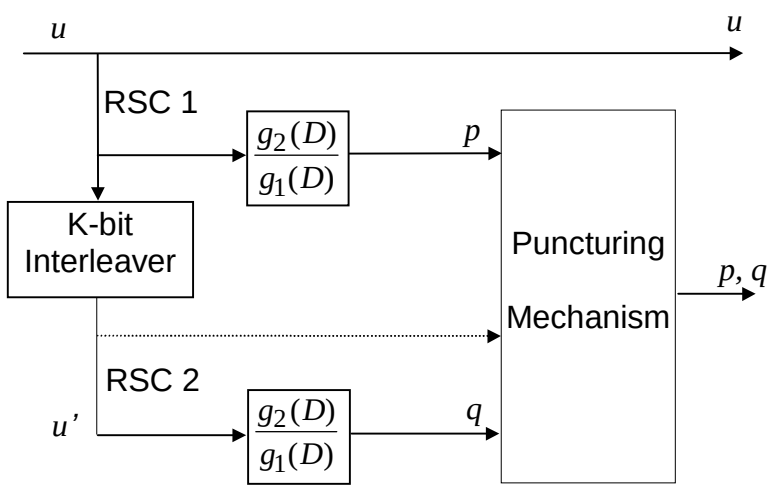


Fig. 1. PCCC encoder diagram.

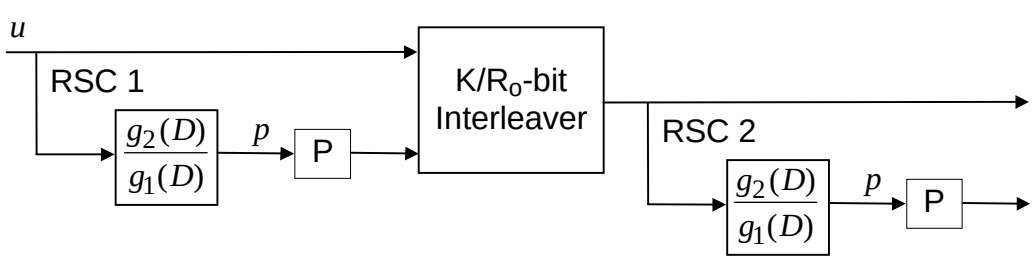


Fig. 2. SCCC encoder diagram with RSC component codes. "P" signifies possible puncturing of parity bits.

2.1. The Recursive Systematic Encoders

Whereas the generator matrix for a rate 1/2 non-recursive convolutional code has the form $G_{NR}(D) = [g_1(D) \quad g_2(D)]$, the equivalent recursive systematic encoder has the generator matrix

$$G_R(D) = \begin{bmatrix} 1 & \dfrac{g_2(D)}{g_1(D)} \end{bmatrix} .$$

Observe that the code sequence corresponding to the encoder input $u(D)$ for the former code is $u(D)G_{NR}(D) = [u(D)g_1(D) \quad u(D)g_2(D)]$, and that the identical code sequence is produced in the recursive code by the sequence $u'(D) = u(D)g_1(D)$, since in this case the code sequence is $u(D)g_1(D)G_R(D) = u(D)G_{NR}(D)$. Here, we loosely call the pair of polynomials $[u(D)g_1(D) \quad u(D)g_2(D)]$ a code sequence, although the actual code sequence is derived from this polynomial pair in the usual way.

Observe that, for the recursive encoder, the code sequence will be of finite weight if and only if the input sequence is divisible by $g_1(D)$. We have the following corollaries of this fact which we shall use later.

Fact 1. A weight-1 input will produce an infinite weight output for such an input is never divisible by a (non-trivial) polynomial $g_1(D)$. (In practice, "infinite" should be replaced by "large" since the input length is finite.)

Fact 2. For any non-trivial $g_1(D)$, there exists a family of weight-2 inputs of the form $D^j(1+D^p)$, $j \geq 0$, which produce finite weight outputs, i.e., which are divisible by $g_1(D)$. When $g_1(D)$ is a primitive polynomial of degree m, then $p = 2^m - 1$. More generally, p is the length of the pseudorandom sequence generated by $g_1(D)$.

Proof: Because the encoder is linear, its output due to a weight-2 input $D^j(1+D^t)$ is equal to the sum of its outputs due to D^j and D^jD^t. The output due to D^j will be periodic with period p since the encoder is a finite state machine (see Example 1 and Fig. 3): the state at time j must be reached again in a finite number of steps p, after which the state sequence is repeated indefinitely with period p. Now letting $t = p$, the output due to D^jD^p is just the output due to D^j shifted by p bits. Thus, the output due to $D^j(1+D^p)$ is the sum of the outputs due to D^j and D^jD^p which much be of finite length and weight since all but one period will cancel in the sum. □

In the context of the code's trellis, Fact 1 says that a weight-1 input will create a path that diverges from the all-zeros path, but never remerges. Fact 2 says that there will always exist a trellis path that diverges and remerges later which corresponds to a weight-2 data sequence.

Example 1. Consider the code with generator matrix

$$G_R(D) = \begin{bmatrix} 1 & \dfrac{1+D^2+D^3+D^4}{1+D+D^4} \end{bmatrix} .$$

Thus, $g_1(D) = 1+D+D^4$ and $g_2(D) = 1+D^2+D^3+D^4$ or, in octal form, $(g_1, g_2) = (31, 27)$. Observe that $g_1(D)$ is primitive so that, for example, $u(D) = 1+D^{15}$ produces the finite-length code sequence $(1+D^{15}, 1+D+D^2+D^3+D^5+D^7+D^8+D^{11})$. Of course, any delayed version of this input, say, $D^7(1+D^{15})$, will simply produce a delayed version of this code sequence. Fig. 3 gives one encoder realization for this code. We remark that, in addition to elaborating on Fact 2, this example serves to demonstrate the conventions generally used in the literature for specifying such encoders. □

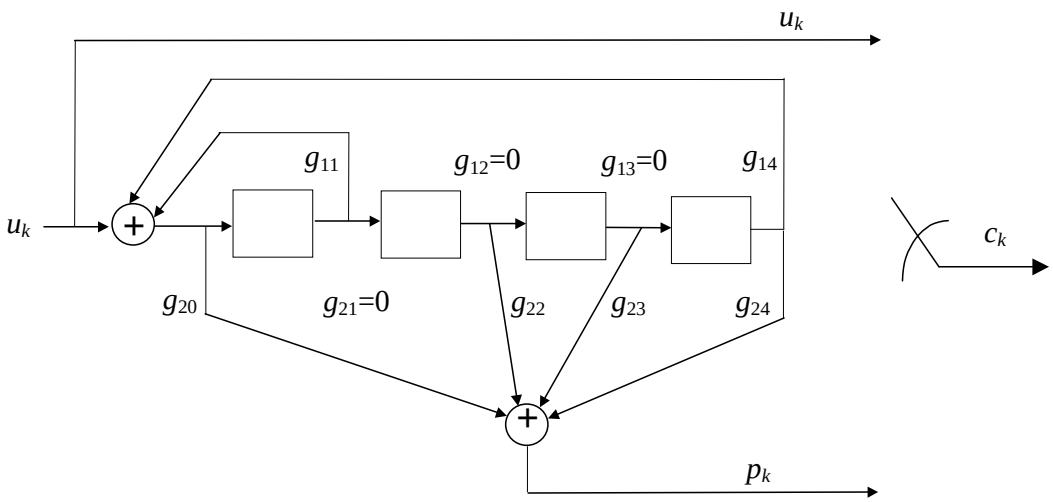


Fig. 3. RSC encoder with $(g_1, g_2) = (31, 27)$.

2.2. The Interleaver

The function of the interleaver is to take each incoming block of bits and rearrange them in a pseudo-random fashion prior to encoding by the second encoder. For the PCCC, the interleaver permutes K bits and, for the SCCC, the interleaver permutes K/R_o bits. Unlike the classical interleaver (e.g., block or convolutional interleaver), which rearranges the bits in some systematic fashion, it is crucial that this interleaver sort the bits in a manner that lacks any apparent order, although it might be tailored in a certain way for weight-2 and weight-3 inputs as will be made clearer below. The $\mathcal{S}$-random interleaver [8] is quite effective in this regard. This particular interleaver ensures that any two inputs bits whose positions are within $\mathcal{S}$ of each other are separated by an amount greater than $\mathcal{S}$ at the interleaver output. $\mathcal{S}$ should be selected to be as large as possible for a given value of K. Also, as we shall see, performance increases with K, and so $K \geq 1000$ is typical.

2.3. The Puncturer

The role of the turbo code puncturer is identical to that of its convolutional code counterpart, that is, to delete selected bits to reduce coding overhead. We have found it most convenient to delete only parity bits, but there is no guarantee that this will maximize the minimum codeword distance. For example, to achieve a rate of 1/2, one might delete all even parity bits from the top encoder and all odd parity bits from the bottom one.

3. Performance with Maximum-Likelihood Decoding

As will be elaborated upon in the next section, a maximum-likelihood (ML) sequence decoder would be far too complex for a turbo code due to the presence of the permuter. However, the suboptimum iterative decoding algorithm to be described there offers near-ML performance. Hence, we shall now estimate the performance of an ML decoder on a binary-input AWGN channel with power spectral density $N_0/2$ (analysis of the iterative decoder is much more difficult).

Armed with the above descriptions of the components of the turbo encoders of Figs. 1 and 2, it is easy to conclude that it is linear since its components are linear. The constituent codes are certainly linear, and the interleaver is linear since it may be modeled by a permutation matrix. Further, the puncturer does not affect linearity since all the constituent codewords share the same puncture locations. As usual, the importance of linearity in evaluating the performance of a code is that one may choose the all-zeros sequence as a reference. Thus, we shall assume that the all-zeros codeword was transmitted. The development below holds for both parallel and serial turbo codes.

Now consider the all-zeros codeword (the 0th codeword) and the kth codeword, for some $k \in \{1, 2, ..., 2^K - 1\}$. The ML decoder will choose the kth codeword over the 0th codeword with probability $Q\left(\sqrt{2d_k RE_b/N_0}\right)$ where d_k is the weight of the kth codeword and E_b is the energy per information bit. The bit error rate for this two-codeword situation would then be

$$\begin{aligned} P_b(k \mid 0) &= w_k \text{ (bit errors/cw error)} \times \\ &\quad \frac{1}{K} \text{ (cw/ data bits)} \times \\ &\quad Q\left(\sqrt{2Rd_k E_b/N_0}\right) \text{ (cw errors/cw)} \\ &= \frac{w_k}{K} Q\left(\sqrt{\frac{2Rd_k E_b}{N_0}}\right) \text{ (bit errors/data bit)} \end{aligned}$$

where w_k is the weight of the kth data word. Now including all of the codewords and invoking the usual union bounding argument, we may write

$$\begin{aligned} P_b &= P_b(\text{choose any } k \in \{1, 2, ..., 2^K - 1\} \mid 0) \\ &\leq \sum_{k=1}^{2^K-1} P_b(k \mid 0) \\ &= \sum_{k=1}^{2^K-1} \frac{w_k}{K} Q\left(\sqrt{\frac{2Rd_k E_b}{N_0}}\right) . \end{aligned}$$

Note that every non-zero codeword is included in the above summation. Let us now reorganize the summation as

$$P_b \leq \sum_{w=1}^{K} \sum_{v=1}^{\binom{K}{w}} \frac{w}{K} Q\left(\sqrt{\frac{2Rd_{wv} E_b}{N_0}}\right) \tag{3.1}$$

where the first sum is over the weight-w inputs, the second sum is over the $\binom{K}{w}$ different weight-w inputs, and d_{wv} is the weight of the vth codeword produced by a weight-w input. We emphasize that (3.1) holds for any linear code.

Consider now the first few terms in the outer summation of (3.1) in the context of parallel and serial turbo codes. Analogous to the fact that the top encoder in the parallel scheme is recursive, we shall assume that the outer encoder in the serial scheme is also recursive. By doing so, our arguments below will hold for both configurations. Further, an RSC outer code facilitates the design of high rate serial turbo codes as mentioned above.

$w = 1$: From Fact 1 and associated discussion above, weight-1 inputs will produce only large weight codewords at both PCCC constituent encoder outputs since the trellis paths created never remerge with the all-zeros path. (We ignore the extreme case where the single 1 occurs at the end of the input words for both encoders for this is avoidable by proper interleaver design.) For the SCCC, the output of the outer encoder will have large weight due to Fact 1, and its inner encoder output will have large weight since its input has large weight. Thus, for both cases, each d_{1v} can be expected to be significantly greater than the minimum codeword weight so that the $w = 1$ terms in (3.1) will be negligible.

$w = 2$: Suppose that of the $\binom{K}{2}$ possible weight-2 encoder inputs, $u_*(D)$ is the one of least degree that yields the minimum turbo codeword weight, $d_{2,\min}$, for weight-2 inputs. Due to the presence of the pseudo-random interleaver, the encoder is not time-invariant, and only a small fraction of the inputs of the form $D^j u_*(D)$ (there are approximately K of them) will also produce turbo codewords of weight $d_{2,\min}$. (This phenomenon has been called *spectral thinning* [10].) Denoting by n_2 the number of weight-2 inputs which produce weight-$d_{2,\min}$ turbo codewords, we may conclude that $n_2 << K$. (For comparison, $n_2 \simeq K$ for a single RSC code as shifts of some worst-case input merely shifts the encoder output, thus maintaining a constant output weight.) Further, the overall minimum codeword weight, $d_{\min}$, is likely to be equal or close to $d_{2,\min}$ since low-degree, low-weight input words tend to produce low-weight codewords. (This is easiest to see in the parallel turbo code case which is systematic.)

$w \geq 3$: When w is small (e.g., $w = 3$ or 4), an argument similar to the $w = 2$ case may be made to conclude that the number of weight-w inputs, n_w, that yield the minimum turbo codeword weight for weight-w inputs, $d_{w,\min}$, is such that $n_w << K$. Further, we can expect $d_{w,\min}$ to be equal or close to $d_{\min}$. No such arguments can be made as w increases beyond about 5.

To summarize, by preserving only the dominant terms, the bound in (3.1) can be approximated as

$$P_b \simeq \sum_{w=2}^{3} \frac{w n_w}{K} Q\left(\sqrt{\frac{2R d_{w,\min} E_b}{N_0}}\right) \tag{3.2}$$

where $w \geq 4$ terms may be added in the event that they are not negligible (more likely to be necessary for SCCCs). We note that n_w and $d_{w,\min}$ are functions of the particular interleaver employed. Since $w = 2$ or 3 in (3.2) and $n_w << K$ with $K \geq 1000$, the coefficients out in front of the Q-function are much less than unity. (For comparison, the coefficient for a convolutional code can be much greater than unity, cf. [10]) This effect is called *interleaver gain* and demonstrates the necessity of large interleavers. Finally, we note that recursive encoders are crucial elements of a turbo code since, for non-recursive encoders, division by $g_1(D)$ (non-remergent trellis paths) would not be an issue and (3.2) would not hold (although (3.1) still would).

When $K \simeq 1000$, it is possible to exhaustively find via computer the weight spectra $\{d_{2v} : v = 1, ..., \binom{K}{2}\}$ and $\{d_{3v} : v = 1, ..., \binom{K}{3}\}$ corresponding to the weight-2 and -3 inputs. In this case, an improved estimate of P_b, given by a truncation of (3.1), is

$$P_b \simeq \sum_{w=2}^{3} \sum_{v=1}^{\binom{K}{w}} \frac{w}{K} Q\left(\sqrt{\frac{2R d_{wv} E_b}{N_0}}\right). \tag{3.3}$$

We remark that if codeword error rate, P_{cw}, is the preferred performance metric, then an

estimate of P_{cw} may be obtained from (3.2) or (3.3) by removing the factor w/K from these expressions. That this is so may be seen by following the derivation above for P_b.

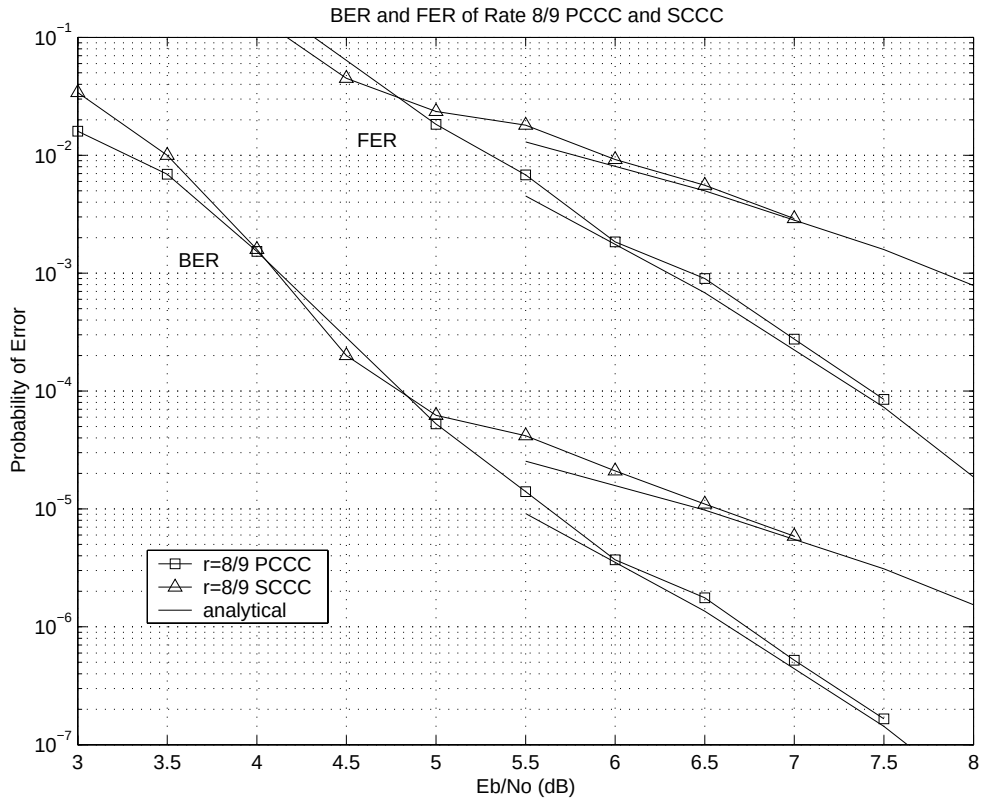


Fig. 4. PCCC and SCCC bit error rate (BER) and frame error rate (FER) simulation results together with analytical result in (3.3).

Example 2. We consider in this example a PCCC and an SCCC code, both rate 8/9 with parameters $(N, K) = (1152, 1024)$. We use identical 4-state RSC encoders in the PCCC encoder whose generators polynomials are, in octal form, $(g_1, g_2) = (7, 5)$. To achieve a code rate of 8/9, only one bit is saved in every 16-bit block of parity bits at each encoder output. The outer constituent encoder in the SCCC encoder is this same 4-state RSC encoder, and the inner code is a rate-1 differential encoder with transfer function $\frac{1}{1 \oplus D}$. A rate of 8/9 is achieved in this case by saving one bit in every 8-bit block of parity bits. The PCCC interleaver is a 1024-bit pseudo-random interleaver with no constraints added (e.g., no $\mathcal{S}$-random constraint). The SCCC interleaver is a 1152-bit pseudo-random interleaver with no constraints added.

Fig. 4 presents performance results for these codes based on computer simulation using the iterative (i.e., non-ML) decoding algorithm of the next section. Simulation results for both bit error rate P_b (BER in the figure) and frame or codeword error rate P_{cw} (FER in the figure) are presented. Also included in the figure are analytic performance curves for ML decoding using the truncated union bound in (3.3). (P_{cw} is obtained by removing the factor w/K in (3.3) as indicated above.) We see the close agreement between the analytical and simulated results in this figure. □

In addition to illustrating the use of the estimate (3.3), this example helps explain the "flooring" effect of the error rate curves: it may be interpreted as the usual Q-function shape

for a signaling scheme with a modest $d_{\min}$, "pushed down" by the interleaver gain $w^* n_{w^*}/K$, where w^* is the value of w corresponding to the dominant term in (3.2) or (3.3).

We comment on the fact that the PCCC in Fig. 4 is substantially better than the SCCC whereas it is known that SCCCs generally have lower floors [6]. We attribute this to the fact that the outer RSC code in the SCCC has been punctured so severely that $d_{\min} = 2$ for this outer code (although $d_{\min}$ for the SCCC is a bit larger). The RSC encoders for the PCCC is punctured only half as much, and so $d_{\min} > 2$ for each of these encoders. We also attribute this to the fact that we have not used an optimized interleaver for this example. In support of these comments, we have also simulated rate 1/2 versions of this same code structure so that no puncturing occurs for the SCCC and much less occurs for the PCCC. In this case, $(N, K) = (2048, 1024)$ and $\mathcal{S}$-random interleavers were used ($\mathcal{S} = 16$ for PCCC and $\mathcal{S} = 20$ for SCCC). The results are presented in Fig. 5 where we observe that the SCCC has a much lower error rate floor, particularly for the FER curves. Finally, we remark that $w \geq 4$ terms in (3.2) are necessary for an accurate estimate of the floor level of the SCCC case in Fig. 5.

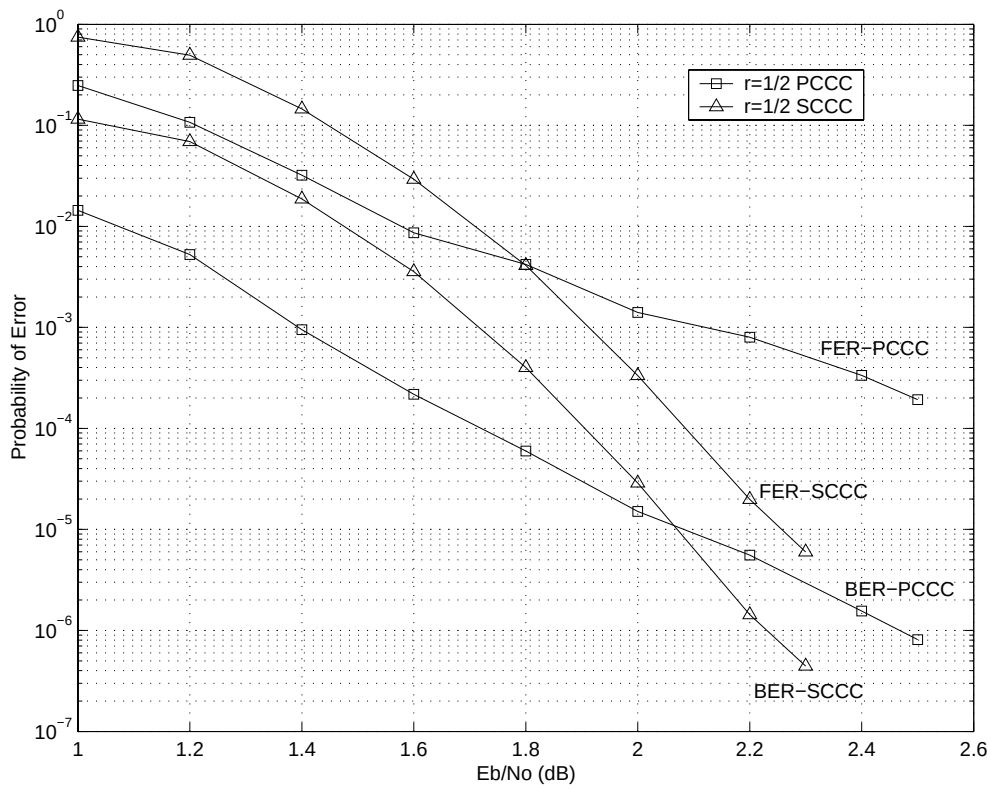


Fig. 5. Rate 1/2 PCCC and SCCC bit error rate (BER) and frame error rate (FER) simulation results.

4. The Iterative Decoders

4.1. Overview of the Iterative Decoder

Consider first an ML decoder for a rate 1/2 convolutional code (recursive or not), and assume a data word of length $K \geq 1000$. Ignoring the structure of the code, a naive ML decoder would

have to compare (correlate) 2^K code sequences to the noisy received sequence, choosing in favor of the codeword with the best correlation metric. Clearly, the complexity of such an algorithm is exorbitant. Fortunately, as we know, such a brute force approach is simplified greatly by Viterbi's algorithm which permits a systematic elimination of candidate code sequences.

Unfortunately, we have no such luck with turbo codes, for the presence of the interleaver immensely complicates the structure of a turbo code trellis. A near-optimal solution is an iterative decoder (also called a turbo decoder) involving two soft-in/soft-out (SISO) decoders which share probabilistic information cooperatively and iteratively. The goal of the iterative decoder is to iteratively estimate the *a posteriori* probabilities (APPs) $\Pr(u_k \mid \mathbf{y})$ where u_k is the kth data bit, $k = 1, 2, ..., K$, and $\mathbf{y}$ is the received codeword in noise, $\mathbf{y} = \mathbf{c} + \mathbf{n}$. In this equation, we assume the components of $\mathbf{c}$ take values in the set $\{\pm 1\}$ (and similarly for $\mathbf{u}$) and $\mathbf{n}$ is a noise word whose components are AWGN samples. Knowledge of the APPs allows for optimal decisions on the bits u_k via the maximum *a posteriori* (MAP) rule[1]

$$\frac{P(u_k = +1 \mid \mathbf{y})}{P(u_k = -1 \mid \mathbf{y})} \underset{-1}{\overset{+1}{\gtrless}} 1$$

or, more conveniently,

$$\hat{u}_k = \text{sign } [L(u_k)] \tag{4.1}$$

where $L(u_k)$ is the log *a posteriori* probability (log-APP) ratio defined as

$$L(u_k) \triangleq \log\left(\frac{P(u_k = +1 \mid \mathbf{y})}{P(u_k = -1 \mid \mathbf{y})}\right) .$$

We shall use the term log-likelihood ratio (LLR) in place of log-APP ratio for consistency with the literature.

The component SISO decoders which jointly estimate the LLRs $L(u_k)$ for parallel and serial turbo codes compute the LLRs for component code inputs (u_{ik}), component code outputs (c_{ik}), or both. Details on the SISO decoders will be presented below. For now, we simply introduce the convention that, for PCCCs, the top component encoder is encoder 1 (denoted E1) and the bottom component decoder is encoder 2 (denoted E2). For SCCCs, the outer encoder is encoder 1 (E1) and the inner encoder is encoder 2 (E2). The SISO component decoders matched to E1 and E2 will be denoted by D1 and D2, respectively. Because the SISO decoders D1 and D2 compute $L(u_{ik})$ and/or $L(c_{ik})$, $i = 1, 2$, we will temporarily use the notation $L(b_k)$ where b_k represents either u_{ik} or c_{ik}.

¿From Bayes' rule, the LLR for an arbitrary SISO decoder can be written as

$$L(b_k) = \log\left(\frac{P(\mathbf{y} \mid b_k = +1)}{P(\mathbf{y} \mid b_k = -1)}\right) + \log\left(\frac{P(b_k = +1)}{P(b_k = -1)}\right) \tag{4.2}$$

with the second term representing *a priori* information. Since $P(b_k = +1) = P(b_k = -1)$ typically, the *a priori* term is usually zero for conventional decoders. However, for *iterative* decoders, each component decoder receives *extrinsic* or *soft* information for each b_k from its companion

[1]It is well known that the MAP rule minimizes the probability of bit error. For comparison, the ML rule, which maximizes the likelihoods $P(\mathbf{y} \mid \mathbf{c})$ over the codewords $\mathbf{c}$, minimizes the probability of codeword error.

decoder which serves as *a priori* information. The idea behind extrinsic information is that D2 provides soft information to D1 for each b_k using only information not available to D1, and D1 does likewise for D2. For SCCCs, the iterative decoding proceeds as D2→D1→D2→D1→..., with the previous decoder passing soft information along to the next decoder at each half-iteration. For PCCCs, either decoder may initiate the chain of component decodings or, for hardware implementations, D1 and D2 may operate simultaneously.

This type of iterative algorithm is known to converge to the true value of the LLR $L(u_k)$ for the concatenated code provided that the graphical representation of this code contains no loops [15]-[17]. The graph of a turbo code does in fact contain loops [17], but the algorithm nevertheless provides near-optimal performance for virtually all turbo codes. That this is possible even in the presence of loops is not fully understood.

This section provided an overview of the turbo decoding algorithm in part to motivate the next section on SISO decoding of a single RSC code using the BCJR algorithm [18]. Following the description of the SISO decoder for a single RSC code will be sections that describe in full detail the iterative PCCC and SCCC decoders which utilize slightly modified SISO decoders.

4.2. The BCJR Algorithm and SISO Decoding

4.2.1. Probability Domain BCJR Algorithm for RSC Codes

Before we discuss the BCJR algorithm in the context of a turbo code, it is helpful to first consider the BCJR algorithm applied to a single rate 1/2 RSC code on an AWGN channel. Thus, as indicated in Fig. 3, the transmitted codeword $\mathbf{c}$ will have the form $\mathbf{c} = [c_1, c_2, ..., c_K] = [u_1, p_1, u_2, p_2, ..., u_K, p_K]$ where $c_k \triangleq [u_k, p_k]$. The received word $\mathbf{y} = \mathbf{c} + \mathbf{n}$ will have the form $\mathbf{y} = [y_1, y_2, ..., y_K] = [y_1^u, y_1^p, y_2^u, y_2^p, ..., y_K^u, y_K^p]$ where $y_k \triangleq [y_k^u, y_k^p]$, and similarly for $\mathbf{n}$. As above, we assume our binary variables take values from the set $\{\pm 1\}$.

Our goal is the development of the BCJR algorithm [18] for computing the LLR

$$L(u_k) = \log\left(\frac{P(u_k = +1 \mid \mathbf{y})}{P(u_k = -1 \mid \mathbf{y})}\right)$$

given the received word $\mathbf{y}$. In order to incorporate the RSC code trellis into this computation, we rewrite $L(u_k)$ as

$$L(u_k) = \log \frac{\sum\limits_{U+} p\left(s_{k-1} = s', s_k = s, \mathbf{y}\right)}{\sum\limits_{U-} p\left(s_{k-1} = s', s_k = s, \mathbf{y}\right)} \tag{4.3}$$

where s_k is encoder state at time k, U^+ is set of pairs (s', s) for the state transitions $(s_{k-1} = s') \rightarrow (s_k = s)$ which correspond to the event $u_k = +1$, and U^- is similarly defined. To write (4.3) we used Bayes' rule, total probability, and then cancelled $1/p\left(\mathbf{y}\right)$ in the numerator and denominator. We see from (4.3) that we need only compute $p\left(s', s, \mathbf{y}\right) = p\left(s_{k-1} = s', s_k = s, \mathbf{y}\right)$ for all state transitions and then sum over the appropriate transitions in the numerator and denominator. We now provide the crucial results which facilitate the computation of $p\left(s', s, \mathbf{y}\right)$.

Result 1. The pdf $p\left(s', s, \mathbf{y}\right)$ may be factored as

$$p\left(s', s, \mathbf{y}\right) = \alpha_{k-1}\left(s'\right) \cdot \gamma_k\left(s', s\right) \cdot \beta_k\left(s\right) \tag{4.4}$$

where

$$\alpha_k(s) \triangleq p\left(s_k = s, \mathbf{y}_1^k\right)$$

$$\gamma_k(s', s) \triangleq p(s_k = s, y_k \mid s_{k-1} = s')$$

$$\beta_k(s) \triangleq p\left(\mathbf{y}_{k+1}^K \mid s_k = s\right)$$

and where $\mathbf{y}_a^b \triangleq [y_a, y_{a+1}, \cdots, y_b]$.

Proof: By several applications of Bayes' rule, we have

$$\begin{aligned}
p(s', s, \mathbf{y}) &= p\left(s', s, \mathbf{y}_1^{k-1}, y_k, \mathbf{y}_{k+1}^K\right) \\
&= p\left(\mathbf{y}_{k+1}^K \mid s', s, \mathbf{y}_1^{k-1}, y_k\right) p\left(s', s, \mathbf{y}_1^{k-1}, y_k\right) \\
&= p\left(\mathbf{y}_{k+1}^K \mid s', s, \mathbf{y}_1^{k-1}, y_k\right) p\left(s, y_k \mid s', \mathbf{y}_1^{k-1}\right) \cdot p\left(s', \mathbf{y}_1^{k-1}\right) \\
&= p\left(\mathbf{y}_{k+1}^K \mid s\right) \cdot p(s, y_k \mid s') \cdot p\left(s', \mathbf{y}_1^{k-1}\right) \\
&= \beta_k(s) \cdot \gamma_k(s', s) \cdot \alpha_{k-1}(s')
\end{aligned}$$

where the fourth line follows from the third because the variables omitted on the fourth line are conditionally independent. □

Result 2. The probability $\alpha_k(s)$ may be computed in a "forward recursion" via

$$\alpha_k(s) = \sum_{s'} \gamma_k(s', s)\, \alpha_{k-1}(s') \tag{4.5}$$

where the sum is over all possible encoder states.

Proof: By several applications of Bayes' rule and the theorem on total probability, we have

$$\begin{aligned}
\alpha_k(s) &\triangleq p\left(s, \mathbf{y}_1^k\right) \\
&= \sum_{s'} p\left(s', s, \mathbf{y}_1^k\right) \\
&= \sum_{s'} p\left(s, y_k \mid s', \mathbf{y}_1^{k-1}\right) p\left(s', \mathbf{y}_1^{k-1}\right) \\
&= \sum_{s'} p(s, y_k \mid s')\, p\left(s', \mathbf{y}_1^{k-1}\right) \\
&= \sum_{s'} \gamma_k(s', s)\, \alpha_{k-1}(s')
\end{aligned}$$

where the fourth line follows from the third due to conditional independence of $\mathbf{y}_1^{k-1}$. □

Result 3. The probability $\beta_k(s)$ may be computed in a "backward recursion" via

$$\beta_{k-1}(s') = \sum_{s} \beta_k(s)\, \gamma_k(s', s) \tag{4.6}$$

Proof: Applying Bayes' rule and the theorem on total probability, we have

$$\beta_{k-1}(s') \triangleq p\left(\mathbf{y}_k^K \mid s'\right)$$

$$
\begin{aligned}
&= \sum_s p\left(\mathbf{y}_k^K, s \mid s'\right) \\
&= \sum_s p\left(\mathbf{y}_{k+1}^K \mid s', s, y_k\right) p\left(s, y_k \mid s'\right) \\
&= \sum_s p\left(\mathbf{y}_{k+1}^K \mid s\right) p\left(s, y_k \mid s'\right) \\
&= \sum_s \beta_k(s)\, \gamma_k\left(s', s\right)
\end{aligned}
$$

where conditional independence led to the omission of variables on the fourth line. □

The recursion for the $\{\alpha_k(s)\}$ is initialized according to

$$
\alpha_0(s) = \begin{cases} 1, & s = 0 \\ 0, & s \neq 0 \end{cases}
$$

which makes the reasonable assumption that the convolutional encoder is initialized to the zero state. The recursion for the $\{\beta_k(s)\}$ is initialized according to

$$
\beta_K(s) = \begin{cases} 1, & s = 0 \\ 0, & s \neq 0 \end{cases}
$$

which assumes that "termination bits" have been appended at the end of the data word so that the convolutional encoder is again in state zero at time K.

All that remains at this point is the computation of $\gamma_k(s', s) = p(s, y_k \mid s')$. Observe that $\gamma_k(s', s)$ may be written as

$$
\begin{aligned}
\gamma_k(s', s) &= \frac{P(s', s)}{P(s')} \cdot \frac{p(s', s, y_k)}{P(s', s)} \\
&= P(s \mid s')\, p(y_k \mid s', s) \\
&= P(u_k)\, p(y_k \mid u_k) \qquad (4.7)
\end{aligned}
$$

where the event u_k corresponds to the event $s' \to s$. Note $P(s \mid s') = P(s' \to s) = 0$ if s is not a valid state from state s' and $P(s' \to s) = 1/2$ otherwise (since we assume binary-input encoders with equiprobable inputs). Hence, $\gamma_k(s', s) = 0$ if $s' \to s$ is not valid and, otherwise,

$$
\begin{aligned}
\gamma_k(s', s) &= \frac{P(u_k)}{\sqrt{2\pi}\sigma} \exp\left[-\frac{\|y_k - c_k\|^2}{2\sigma^2}\right] \qquad (4.8) \\
&= \frac{1}{2\sqrt{2\pi}\sigma} \exp\left[-\frac{(y_k^u - u_k)^2 + (y_k^p - p_k)^2}{2\sigma^2}\right] \qquad (4.9)
\end{aligned}
$$

where $\sigma^2 = N_0/2$.

In summary, we may compute $L(u_k)$ via (4.3) using (4.4), (4.5), (4.6), and (4.8). This "probability domain" version of the BCJR algorithm is numerically unstable for long and even moderate codeword lengths, and so we now present the stable "log domain" version of it.

4.2.2. Log Domain BCJR Algorithm for RSC Codes

In the log-BCJR algorithm, $\alpha_k(s)$ is replaced by the *forward metric*

$$\begin{aligned}\tilde{\alpha}_k(s) &\triangleq \log(\alpha_k(s)) \\ &= \log\left(\sum_{s'} \alpha_{k-1}(s')\,\gamma_k(s',s)\right) \\ &= \log\left(\sum_{s'} \exp\left(\tilde{\alpha}_{k-1}(s') + \tilde{\gamma}_k(s',s)\right)\right) \end{aligned} \tag{4.10}$$

where the *branch metric* $\tilde{\gamma}_k(s',s)$ is given by

$$\begin{aligned}\tilde{\gamma}_k(s',s) &= \log \gamma_k(s',s) \\ &= -\log\left(2\sqrt{2\pi}\sigma\right) - \frac{\|y_k - c_k\|^2}{2\sigma^2}. \end{aligned} \tag{4.11}$$

We will see that the first term in (4.11) may be dropped. Note that (4.10) not only defines $\tilde{\alpha}_k(s)$, but it gives its recursion. These log-domain forward metrics are initialized as

$$\tilde{\alpha}_0(s) = \begin{cases} 0, & s = 0 \\ -\infty, & s \neq 0 \end{cases} \tag{4.12}$$

The probability $\beta_{k-1}(s')$ is replaced by the *backward metric*

$$\begin{aligned}\tilde{\beta}_{k-1}(s') &\triangleq \log(\beta_{k-1}(s')) \\ &= \log\left(\sum_{s} \exp \tilde{\beta}_k(s) + \tilde{\gamma}_k(s',s)\right) \end{aligned} \tag{4.13}$$

with initial conditions

$$\tilde{\beta}_K(s) = \begin{cases} 0, & s = 0 \\ -\infty, & s \neq 0 \end{cases} \tag{4.14}$$

under the assumption that the encoder has been terminated.

As before, the $L(u_k)$ is computed as

$$\begin{aligned} L(u_k) &= \log \frac{\sum_{U^+} \alpha_{k-1}(s')\,\gamma_k(s',s)\,\beta_k(s)}{\sum_{U^-} \alpha_{k-1}(s')\,\gamma_k(s',s)\,\beta_k(s)} \\ &= \log\left[\sum_{U^+} \exp\left(\tilde{\alpha}_{k-1}(s') + \tilde{\gamma}_k(s',s) + \tilde{\beta}_k(s)\right)\right] \\ &\quad - \log\left[\sum_{U^-} \exp\left(\tilde{\alpha}_{k-1}(s') + \tilde{\gamma}_k(s',s) + \tilde{\beta}_k(s)\right)\right]. \end{aligned} \tag{4.15}$$

It is evident from (4.15) that the constant term in (4.11) may be ignored since it may be factored all the way out of both summations. At first glance, equations (4.10)-(4.15) do not look any

simpler than the probability domain algorithm, but we use the following results to attain the simplification.

Result 4

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

Proof: Without loss of generality, when $x > y$, the right-hand side equals x. $\square$

Now define

$$\max^*(x, y) \triangleq \log\left(e^x + e^y\right) \tag{4.16}$$

so that from Result 4,

$$\max^*(x, y) = \max(x, y) + \log\left(1 + e^{-|x-y|}\right). \tag{4.17}$$

This may be extended to more than two variables. For example,

$$\max^*(x, y, z) \triangleq \log\left(e^x + e^y + e^z\right)$$

which may be computed pairwise according to the following result.

Result 5

$$\max^*(x, y, z) = \max^*\left[\max^*(x, y), z\right]$$

Proof:

$$\begin{aligned} RHS &= \log\left[e^{\max^*(x,y)} + e^z\right] \\ &= \log\left[e^{\log(e^x+e^y)} + e^z\right] \\ &= \log\left[e^x + e^y + e^z\right] \\ &= LHS \quad \square \end{aligned}$$

Given the function $\max^*(\cdot)$, we may now rewrite (4.10), (4.13), and (4.15) as

$$\tilde{\alpha}_k(s) = \max_{s'}{}^* \left[\tilde{\alpha}_{k-1}(s') + \tilde{\gamma}_k(s', s)\right], \tag{4.18}$$

$$\tilde{\beta}_{k-1}(s') = \max_{s}{}^* \left[\tilde{\beta}_k(s) + \tilde{\gamma}_k(s', s)\right], \tag{4.19}$$

and

$$\begin{aligned} L(u_k) &= \max_{U^+}{}^* \left[\tilde{\alpha}_{k-1}(s') + \tilde{\gamma}_k(s', s) + \tilde{\beta}_k(s)\right] \\ &\quad - \max_{U^-}{}^* \left[\tilde{\alpha}_{k-1}(s') + \tilde{\gamma}_k(s', s) + \tilde{\beta}_k(s)\right]. \end{aligned} \tag{4.20}$$

Fig. 6 illustrates pictorially the trellis-based computations that these last three equations represent.

We see from (4.18), (4.19), and (4.20) how the log domain computation of $L(u_k)$ is vastly simplified relative to the probability domain computation. From (4.17), implementation of the $\max^*(\cdot)$ function involves only a two-input $\max(\cdot)$ function plus a lookup table for the "correction term" $\log\left(1 + e^{-|x-y|}\right)$. Robertson *et al.* [11] have shown that a table size of eight is usually sufficient.

Note that the correction term is bounded as

$$0 < \log\left(1 + e^{-|x-y|}\right) \leq \log(2) \simeq 0.693$$

so that $\max^*(x, y) \simeq \max(x, y)$ when $|\max(x, y)| \geq 7$. When $\max^*(x, y)$ is replaced by $\max(\cdot)$ in (4.18) and (4.19), these recursions become forward and reverse Viterbi algorithms, respectively. The performance loss associated with this approximation in turbo decoding depends on the specific turbo code, but a loss of about 0.5 dB is typical [11].

Finally, observe the sense in which the BCJR decoder is a SISO decoder: the decoder input is the "soft-decision" (unquantized) word $\mathbf{y} \in \mathbb{R}^{2K}$ and its outputs are the soft outputs $L(u_k) \in \mathbb{R}$ on which final hard-decisions may be made according to (4.1). Alternatively, in a concatenated code context, these soft outputs may be passed to a companion decoder.

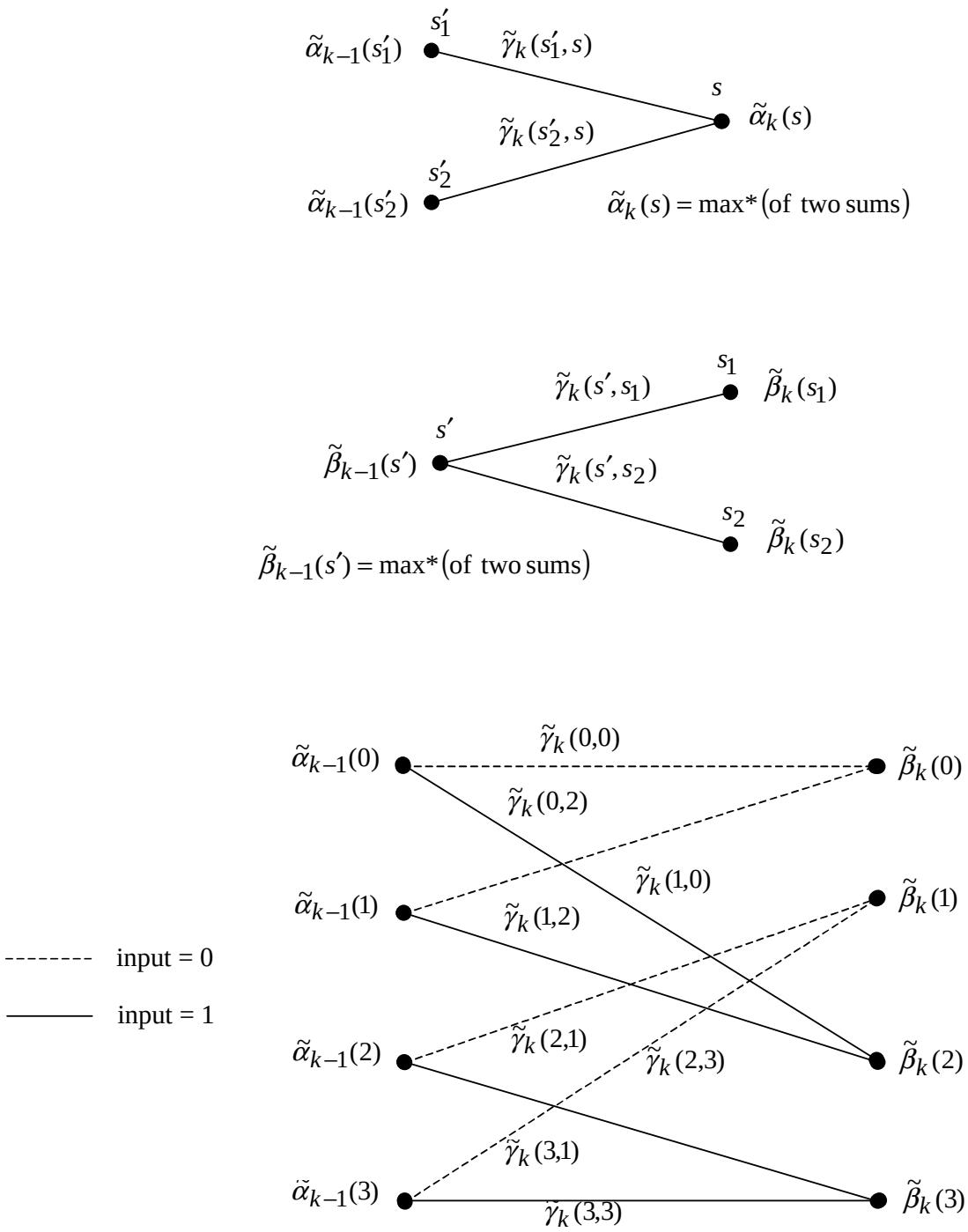


Fig. 6. The top diagram depicts the forward recursion in (4.18), the middle diagram depicts the backward recursion in (4.19), and the bottom diagram depicts the computation of $L(u_k)$ via (4.20).

Summary of the Log-Domain BCJR Algorithm We assume as above a rate 1/2 RSC encoder, a data block $\mathbf{u}$ of length K, and that encoder starts and terminates in the zero state (the last m bits of $\mathbf{u}$ are so selected, where m is the encoder memory size). In practice, the value $-\infty$ used in initialization is simply some large-magnitude negative number.

Initialize $\tilde{\alpha}_0(s)$ and $\tilde{\beta}_K(s)$ according to (4.12) and (4.14).

for $k = 1 : K$

- get $y_k = [y_k^u, y_k^p]$
- compute $\tilde{\gamma}_k(s', s) = -\left\|y_k - c_k\right\|^2 / 2\sigma^2$ for all allowable state transitions $s' \to s$ (note $c_k = c_k(s', s)$ here)[2]
- compute $\tilde{\alpha}_k(s)$ for all s using the recursion (4.18)

end

for $k = K : -1 : 2$

- compute $\tilde{\beta}_{k-1}(s')$ for all s' using (4.19)

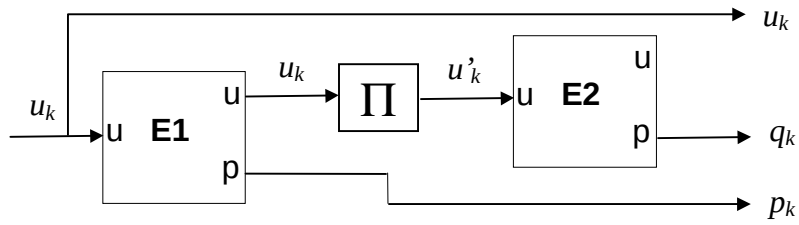
end

for $k = 1 : K$

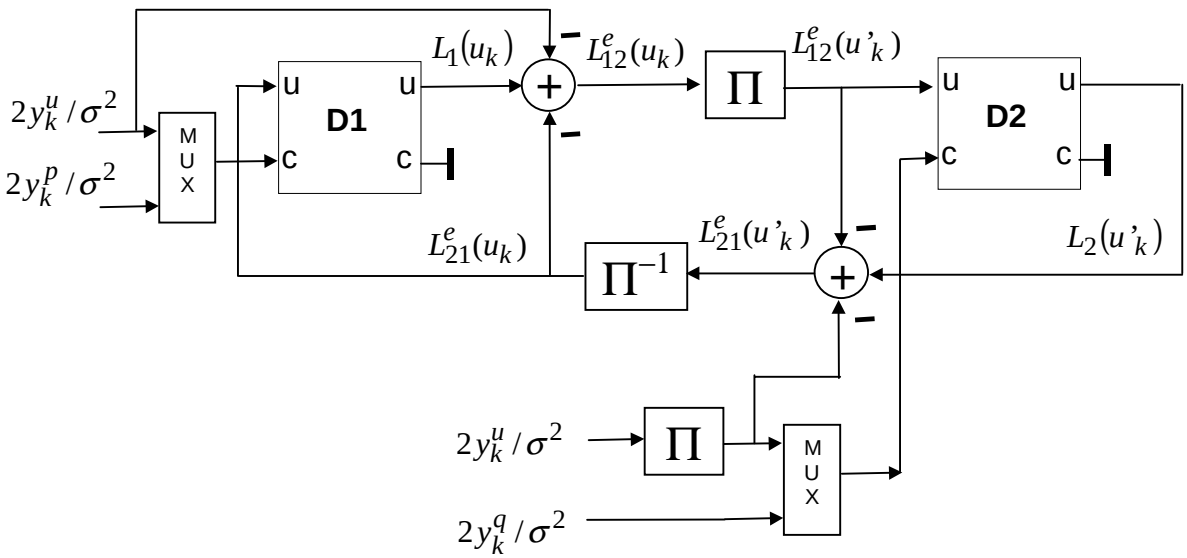
- compute $L(u_k)$ using (4.20)
- compute hard decisions via $\hat{u}_k = \text{sign}\left[L(u_k)\right]$

end

[2] We may alternatively use $\tilde{\gamma}_k(s', s) = u_k y_k^u / \sigma^2 + p_k y_k^p / \sigma^2$. (See next section.)



(a) PCCC encoder



(b) PCCC iterative decoder

Fig. 7. Block diagrams for the PCCC encoder and iterative decoder.

4.3. The PCCC Iterative Decoder

We present in this section the iterative decoder for a PCCC consisting of two component rate 1/2 RSC encoders concatenated in parallel. We assume no puncturing so that the overall code rate is 1/3. Block diagrams of the PCCC encoder and its iterative decoder with component SISO decoders are presented in Fig. 7. As indicated in Fig. 7(a), the transmitted codeword $\mathbf{c}$ will have the form $\mathbf{c} = [c_1, c_2, ..., c_K] = [u_1, p_1, q_1, ..., u_K, p_K, q_K]$ where $c_k \triangleq [u_k, p_k, q_k]$. The received word $\mathbf{y} = \mathbf{c} + \mathbf{n}$ will have the form $\mathbf{y} = [y_1, y_2, ..., y_K] = [y_1^u, y_1^p, y_1^q, ..., y_K^u, y_K^p, y_K^q]$ where $y_k \triangleq [y_k^u, y_k^p, y_k^q]$, and similarly for $\mathbf{n}$. We denote the codewords produced by E1 and E2 by, respectively, $\mathbf{c}_1 = [c_1^1, c_2^1, ..., c_K^1]$ where $c_k^1 \triangleq [u_k, p_k]$ and $\mathbf{c}_2 = [c_1^2, c_2^2, ..., c_K^2]$ where $c_k^2 \triangleq [u'_k, q_k]$. Note that $\{u'_k\}$ is a permuted version of $\{u_k\}$ and is not actually transmitted (see Fig. 7(a)). We define the noisy received versions of $\mathbf{c}_1$ and $\mathbf{c}_2$ to be $\mathbf{y}_1$ and $\mathbf{y}_2$, respectively, having components $y_k^1 \triangleq [y_k^u, y_k^p]$ and $y_k^2 \triangleq [y_k^{u'}, y_k^q]$, respectively. Note that $\mathbf{y}_1$ and $\mathbf{y}_2$ can be assembled from $\mathbf{y}$ in an obvious fashion (using an interleaver to obtain $\left\{y_k^{u'}\right\}$ from $\{y_k^u\}$). By doing so, the component decoder inputs are the two vectors $\mathbf{y}_1$ and $\mathbf{y}_2$ as indicated in the Fig. 7(b).

In contrast to the BCJR decoder of the previous sections whose input was $\mathbf{y} = \mathbf{c} + \mathbf{n}$ and

whose output was $\{L(u_k)\}$ (or $\{\hat{u}_k\}$), the SISO decoders in Fig. 7(b) possess two inputs and two outputs. The SISO decoders are essentially the BCJR decoders discussed above, except the SISO decoders have the ability to accept from a companion decoder "extrinsic information" about its encoder's input (SISO input label 'u') and/or about its encoder's output (SISO input label 'c'). The SISO decoders also have the ability to produce likelihood information about its encoder's input (SISO output label 'u') and/or about its encoder's output (SISO output label 'c'). Note that the SISO decoder is to be interpreted as a decoding module not all of whose inputs or outputs need be used [7]. (Note the RSC encoders in Fig. 7(a) have also been treated as modules.) As we will see, the SISO modules are connected in a slightly different fashion for the SCCC case.

Note from Fig. 7(b) that the extrinsic information to be passed from D1 to D2 about bit u_k, denoted $L^e_{12}(u_k)$, is equal to the LLR $L_1(u_k)$ produced by D1 minus the channel likelihood $2y^u_k/\sigma^2$ and the extrinsic information $L^e_{21}(u_k)$ that D1 had just received from D2. The idea is that $L^e_{12}(u_k)$ should indeed be extrinsic (and uncorrelated with) the probabilistic information already possessed by D2. As we will see, $L^e_{12}(u_k)$ is strictly a function of received E1 parity $\{y^p_k\}$ which is not directly sent to D2. Observe that $\{L^e_{12}(u_k)\}$ must be interleaved prior to being sent to D2 since E2 and D2 operate on the interleaved data bits u'_k. Symmetrical comments may be made about the extrinsic information to be passed from D2 to D1, $L^e_{21}(u_k)$ (e.g., it is a function of E2 parity and de-interleaving is necessary).

We already know from the previous section how the SISO decoders process the samples from the channel, $\mathbf{y}_i$ $(i = 1, 2)$, to obtain LLR's about the decoder inputs. We need now to discuss how the SISO decoders include the extrinsic information in their computations. As indicated earlier, the extrinsic information takes the role of *a priori* information in the iterative decoding algorithm (cf. (4.2) and surrounding discussion),

$$L^e(u_k) \triangleq \log\left(\frac{P(u_k = +1)}{P(u_k = -1)}\right). \tag{4.21}$$

The *a priori* term $P(u_k)$ shows up in (4.8) in an expression for $\gamma_k(s', s)$. In the log domain, (4.8) becomes[3]

$$\tilde{\gamma}_k(s', s) = \log P(u_k) - \log\left(\sqrt{2\pi}\sigma\right) - \frac{\|y_k - c_k\|^2}{2\sigma^2}. \tag{4.22}$$

Now observe that we may write

$$\begin{aligned} P(u_k) &= \left(\frac{\exp[-L^e(u_k)/2]}{1 + \exp[-L^e(u_k)]}\right) \cdot \exp[u_k L^e(u_k)/2] \\ &= A_k \exp[u_k L^e(u_k)/2] \end{aligned} \tag{4.23}$$

where the first equality follows since it equals

$$\begin{aligned} \left(\frac{\sqrt{P_-/P_+}}{1 + P_-/P_+}\right)\sqrt{P_+/P_-} &= P_+ \text{ when } u_k = +1 \text{ and} \\ \left(\frac{\sqrt{P_-/P_+}}{1 + P_-/P_+}\right)\sqrt{P_-/P_+} &= P_- \text{ when } u_k = -1, \end{aligned}$$

[3]For the time being, we will discuss a generic SISO decoder so that we may avoid using cumbersome superscripts until it is necessary to do so.

where we have defined $P_+ \triangleq P(u_k = +1)$ and $P_- \triangleq P(u_k = -1)$ for convenience. Substitution of (4.23) into (4.22) yields

$$\tilde{\gamma}_k(s', s) = \log\left(A_k/\sqrt{2\pi}\sigma\right) + u_k L^e(u_k)/2 - \frac{\|y_k - c_k\|^2}{2\sigma^2} \tag{4.24}$$

where we will see that the first term may be ignored.

Thus, the extrinsic information received from a companion decoder is included in the computation through the branch metric $\tilde{\gamma}_k(s', s)$. The rest of the BCJR/SISO algorithm proceeds as before using equations (4.18), (4.19), and (4.20).

Upon substitution of (4.24) into (4.20), we have

$$\begin{aligned} L(u_k) &= L^e(u_k) + \max_{U^+}{}^* \left[\tilde{\alpha}_{k-1}(s') + u_k y_k^u/\sigma^2 + p_k y_k^p/\sigma^2 + \tilde{\beta}_k(s)\right] \\ &\quad - \max_{U^-}{}^* \left[\tilde{\alpha}_{k-1}(s') + u_k y_k^u/\sigma^2 + p_k y_k^p/\sigma^2 + \tilde{\beta}_k(s)\right] \end{aligned} \tag{4.25}$$

where we have use the fact that

$$\begin{aligned} \|y_k - c_k\|^2 &= (y_k^u - u_k)^2 + (y_k^p - p_k)^2 \\ &= (y_k^u)^2 - 2u_k y_k^u + u_k^2 + (y_k^p)^2 - 2p_k y_k^p + p_k^2 \end{aligned}$$

and only the terms dependent on U^+ or U^-, $u_k y_k^u/\sigma^2$ and $p_k y_k^p/\sigma^2$, survive after the subtraction. Now note that $u_k y_k^u/\sigma^2 = y_k^u/\sigma^2$ under the first $\max^*(\cdot)$ operation in (4.25) (U^+ is the set of state transitions for which $u_k = +1$) and $u_k y_k^u/\sigma^2 = -y_k^u/\sigma^2$ under the second $\max^*(\cdot)$ operation. Using the definition for $\max^*(\cdot)$, it is easy to see that these terms may be isolated out so that

$$\begin{aligned} L(u_k) &= 2y_k^u/\sigma^2 + L^e(u_k) + \max_{U^+}{}^* \left[\tilde{\alpha}_{k-1}(s') + p_k y_k^p/\sigma^2 + \tilde{\beta}_k(s)\right] \\ &\quad - \max_{U^-}{}^* \left[\tilde{\alpha}_{k-1}(s') + p_k y_k^p/\sigma^2 + \tilde{\beta}_k(s)\right]. \end{aligned} \tag{4.26}$$

The interpretation of this new expression for $L(u_k)$ is that the first term is likelihood information received directly from the channel, the second term is extrinsic likelihood information received from a companion decoder, and the third "term" ($\max_{U^+}{}^* - \max_{U^-}{}^*$) is extrinsic likelihood information to be passed to a companion decoder. Note that this third term is likelihood information gleaned from received parity not available to the companion decoder. Thus, specializing to decoder D1, for example, on any given iteration, D1 computes

$$L_1(u_k) = 2y_k^u/\sigma^2 + L_{21}^e(u_k) + L_{12}^e(u_k)$$

where $L_{21}^e(u_k)$ is extrinsic information received from D2, and $L_{12}^e(u_k)$ is the third term in (4.26) which is to be used as extrinsic information from D1 to D2.

4.3.1. Summary of the PCCC Iterative Decoder

The algorithm given below for the iterative decoding of a parallel turbo code follows directly from the development above. The constituent decoder order is D1, D2, D1, D2, etc. Implicit is the fact that each decoder must have full knowledge of the trellis of the constituent encoders. For example, each decoder must have a table (array) containing the input bits and parity bits for all possible state transitions $s' \rightarrow s$. Also required are interleaver and de-interleaver functions (arrays) since D1 and D2 will be sharing reliability information about each u_k, but D2's information is permuted relative to D1. We denote these arrays by $P[\cdot]$ and $Pinv[\cdot]$, respectively. For example, the permuted word $\mathbf{u}'$ is obtained from the original word $\mathbf{u}$ via the pseudo-code statement: for $k = 1 : K$, $u'_k = u_{P[k]}$, end.

We next point out that knowledge of the noise variance $\sigma^2 = N_0/2$ by each SISO decoder is necessary. Also, a simple way to obtain higher code rates via (simulated) puncturing is, in the computation of $\gamma_k(s', s)$, to set to zero the received parity samples, y_k^p or y_k^q, corresponding to the punctured parity bits, p_k or q_k. (This will set to zero the term in the branch metric corresponding to the punctured bit.) Thus, puncturing need not be performed at the encoder for computer simulations. We mention also that termination of encoder E2 to the zero state can be problematic due to the presence of the interleaver (for one solution, see [19]). Fortunately, there is only a small performance loss when E2 is not terminated. In this case, $\beta_K(s)$ for D2 may be set to $\alpha_K(s)$ for all s, or it may be set to a nonzero constant (e.g., $1/S_2$, where S_2 is the number of E2 states).

Finally, we remark that some sort of iteration stopping criterion is necessary. The most straightforward criterion is to set a maximum number of iterations. However, this can be inefficient since the correct codeword is often found after only two or three iterations. Another straightforward technique is to utilize a carefully chosen outer error detection code. After each iteration, a parity check is made and the iterations stop whenever no error is detected. Other stopping criteria are presented in [9] and elsewhere in the literature.

Initialization
D1:

$$\tilde{\alpha}_0^{(1)}(s) = 0 \text{ for } s = 0$$
$$= -\infty \text{ for } s \neq 0$$

$$\tilde{\beta}_K^{(1)}(s) = 0 \text{ for } s = 0$$
$$= -\infty \text{ for } s \neq 0$$

$$L_{21}^e(u_k) = 0 \text{ for } k = 1, 2, ..., K$$

D2:

$$\tilde{\alpha}_0^{(2)}(s) = 0 \text{ for } s = 0$$
$$= -\infty \text{ for } s \neq 0$$

$\tilde{\beta}_K^{(2)}(s) = \tilde{\alpha}_K^{(2)}(s)$ for all s (set once after computation of $\{\tilde{\alpha}_K^{(2)}(s)\}$ in the *first* iteration)

$L_{12}^e(u_k)$ is to be determined from D1 after the first half-iteration and so need not be initialized

The nth Iteration
D1:

for $k = 1 : K$

- get $y_k^1 = [y_k^u, y_k^p]$
- compute $\tilde{\gamma}_k(s', s)$ for all allowable state transitions $s' \to s$ from (4.24) which simplifies to (see discussion following (4.24))

$$\tilde{\gamma}_k(s', s) = u_k L_{21}^e(u_{Pinv[k]})/2 + u_k y_k^u/\sigma^2 + p_k y_k^p/\sigma^2$$

  [u_k (p_k) in this expression is set to the value of the encoder input (output) corresponding to the transition $s' \to s$]
- compute $\tilde{\alpha}_k^{(1)}(s)$ for all s using (4.18)

end

for $k = K : -1 : 2$

- compute $\tilde{\beta}_{k-1}^{(1)}(s)$ for all s using (4.19)

end

for $k = 1 : K$

- compute $L_{12}^e(u_k)$ using[4]

$$\begin{aligned} L_{12}^e(u_k) = & \max_{U^+}{}^* \left[\tilde{\alpha}_{k-1}^{(1)}(s') + p_k y_k^p/\sigma^2 + \tilde{\beta}_k^{(1)}(s)\right] \\ & - \max_{U^-}{}^* \left[\tilde{\alpha}_{k-1}^{(1)}(s') + p_k y_k^p/\sigma^2 + \tilde{\beta}_k^{(1)}(s)\right] \end{aligned}$$

end

D2:

for $k = 1 : K$

- get $y_k^2 = [y_{P[k]}^u, y_k^q]$

[4]Note here we are computing $L_{12}^e(u_k)$ directly rather than computing $L_1(u_k)$ and then subtracting $2y_k^u/\sigma^2 + L_{21}^e(u_k)$ from it to obtain $L_{12}^e(u_k)$ as in Fig. 5(b). We will do likewise in the analogous step for D2.

- compute $\tilde{\gamma}_k(s', s)$ for all allowable state transitions $s' \to s$ from

$$\tilde{\gamma}_k(s', s) = u_k L_{12}^e(u_{P[k]})/2 + u_k y_{P[k]}^u/\sigma^2 + q_k y_k^q/\sigma^2$$

  [u_k (q_k) in this expression is set to the value of the encoder input (output) corresponding to the transition $s' \to s$]

- compute $\tilde{\alpha}_k^{(2)}(s)$ for all s using (4.18)

end

for $k = K : -1 : 2$

- compute $\tilde{\beta}_{k-1}^{(2)}(s)$ for all s using (4.19)

end

for $k = 1 : K$

- compute $L_{21}^e(u_k)$ using

$$\begin{aligned} L_{21}^e(u_k) = & \max_{U^+}{}^* \left[\tilde{\alpha}_{k-1}^{(2)}(s') + q_k y_k^q/\sigma^2 + \tilde{\beta}_k^{(2)}(s)\right] \\ & - \max_{U^-}{}^* \left[\tilde{\alpha}_{k-1}^{(2)}(s') + q_k y_k^q/\sigma^2 + \tilde{\beta}_k^{(2)}(s)\right] \end{aligned}$$

end

After the Last Iteration

for $k = 1 : K$

- compute

$$L_1(u_k) = 2y_k^u/\sigma^2 + L_{21}^e(u_{Pinv[k]}) + L_{12}^e(u_k)$$

- $\hat{u}_k = \text{sign}\ [L(u_k)]$

end

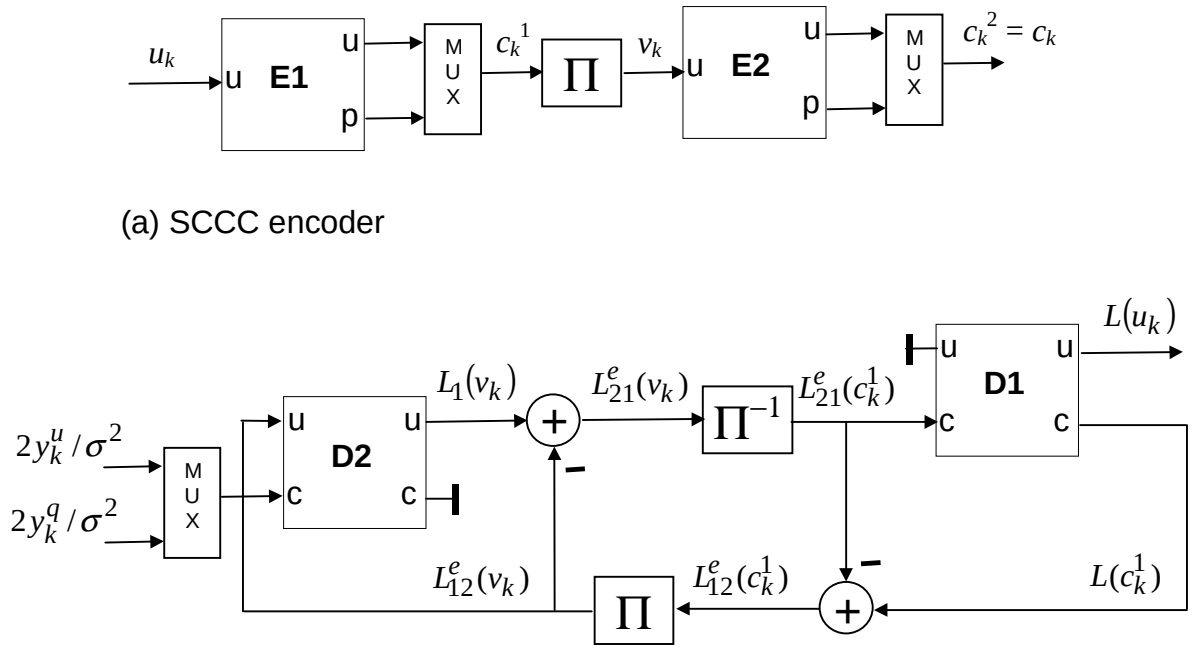


Fig. 8. Block diagrams for the SCCC encoder and iterative decoder.

4.4. The SCCC Iterative Decoder

We present in this section the iterative decoder for an SCCC consisting of two component rate 1/2 RSC encoders concatenated in series. We assume no puncturing so that the overall code rate is 1/4. Higher code rates are achievable via puncturing and/or by replacing the inner encoder with a rate 1 differential encoder with transfer function $\frac{1}{1\oplus D}$. It is straightforward to derive the iterative decoding algorithm for other SCCC codes from the special case that we consider here.

Block diagrams of the SCCC encoder and its iterative decoder with component SISO decoders are presented in Fig. 8. We denote by $\mathbf{c}_1 = [c_1^1, c_2^1, ..., c_{2K}^1] = [u_1, p_1, u_2, p_2, ..., u_K, p_K]$ the codeword produced by E1 whose input is $\mathbf{u} = [u_1, u_2, ..., u_K]$. We denote by $\mathbf{c}_2 = [c_1^2, c_2^2, ..., c_{2K}^2] = [v_1, q_1, v_2, q_2, ..., v_{2K}, q_{2K}]$ (with $c_k^2 \triangleq [v_k, q_k]$) the codeword produced by E2 whose input $\mathbf{v} = [v_1, v_2, ..., v_{2K}]$ is the interleaved version of $\mathbf{c}_1$, that is, $\mathbf{v} = \mathbf{c}_1'$. As indicated in Fig.8(a), the transmitted codeword $\mathbf{c}$ is the codeword $\mathbf{c}_2$. The received word $\mathbf{y} = \mathbf{c} + \mathbf{n}$ will have the form $\mathbf{y} = [y_1, y_2, ..., y_{2K}] = [y_1^v, y_1^q, ..., y_{2K}^v, y_{2K}^q]$ where $y_k \triangleq [y_k^v, y_k^q]$, and similarly for $\mathbf{n}$.

The iterative SCCC decoder in Fig. 8(b) employs two SISO decoding modules (described in the previous section). Note that unlike the PCCC case, these SISO decoders share extrinsic information on the code bits $\{c_k^1\}$ (equivalently, on the input bits $\{v_k\}$) in accordance with the fact that these are the bits known to both encoders. A consequence of this is that D1 must provide likelihood information on E1 *output* bits whereas D2 produces likelihood information on E2 *input* bits as indicated in Fig. 8(b). However, because LLRs must be obtained on the original data bits u_k so that final decisions may be made, D1 must also compute likelihood information on E1 input bits. Note also that, because E1 feeds no bits directly to the channel,

D1 receives no samples directly from the channel. Instead, the only input to D1 is the extrinsic information it receives from D2.

Thus, the SISO module D1 requires two features that we have not discussed in any detail to this point. The first is providing likelihood information on the encoder's input *and* output. However, since we assume the component codes are systematic, we need only compute LLRs on the encoder's output bits $[u_1, p_1, u_2, p_2, ..., u_K, p_K]$. Doing this is a simple matter of modifying the summation indices in (4.3) to those relevant to the output bit of interest. For example, the LLR $L(p_k)$ for the E1 parity bit p_k is obtained via

$$L(p_k) = \log \frac{\sum\limits_{P^+} p\left(s_{k-1} = s', s_k = s, \mathbf{y}\right)}{\sum\limits_{P^-} p\left(s_{k-1} = s', s_k = s, \mathbf{y}\right)} \tag{4.27}$$

where P^+ is set of state transition pairs (s', s) corresponding to the event $p_k = +1$, and P^- is similarly defined. (A trellis-based BCJR/SISO decoder is generally capable of decoding either the encoder's input or its output, whether or not the code is systematic. This is evident since the trellis branches are labeled by both inputs and outputs.)

The second feature is required by D1 is decoding with only extrinsic information as input. In this case the branch metric is simply modified as (cf. (4.24))

$$\tilde{\gamma}_k(s', s) = u_k L^e_{21}(u_k)/2 + p_k L^e_{21}(p_k)/2. \tag{4.28}$$

Other than these modifications, the iterative SCCC decoder proceeds much like the PCCC iterative decoder and as indicated in Fig. 8(b).

4.4.1. Summary of the SCCC Iterative Decoder

Essentially all of the comments mentioned for the PCCC decoder hold here as well and so we do not repeat them. The only difference is that the decoding order is D2, D1, D2, D1, etc.

Initialization
D1

$$\begin{aligned}\tilde{\alpha}^{(1)}_0(s) &= 0 \text{ for } s = 0\\ &= -\infty \text{ for } s \neq 0\end{aligned}$$

$$\begin{aligned}\tilde{\beta}^{(1)}_K(s) &= 0 \text{ for } s = 0\\ &= -\infty \text{ for } s \neq 0\end{aligned}$$

$L^e_{21}(c^1_k)$ is to be determined from D2 after the first half-iteration and so need not be initialized

D2:

$$\begin{aligned}\tilde{\alpha}^{(2)}_0(s) &= 0 \text{ for } s = 0\\ &= -\infty \text{ for } s \neq 0\end{aligned}$$

$\tilde{\beta}^{(2)}_{2K}(s) = \tilde{\alpha}^{(2)}_{2K}(s)$ for all s (set after computation of $\{\tilde{\alpha}^{(2)}_{2K}(s)\}$ in the *first* iteration)

$L_{12}^{e}(v_k) = 0$ for $k = 1, 2, ..., 2K$

The nth Iteration
D2:

for $k = 1 : 2K$

- get $y_k = [y_k^v, y_k^q]$
- compute $\tilde{\gamma}_k(s', s)$ for all allowable state transitions $s' \to s$ from

$$\tilde{\gamma}_k(s', s) = v_k L_{12}^{e}(v_k)/2 + v_k y_k^v/\sigma^2 + q_k y_k^q/\sigma^2$$

  [v_k (q_k) in the above expression is set to the value of the encoder input (output) corresponding to the transition $s' \to s$; $L_{12}^{e}(v_k)$ is $L_{12}^{e}(c_{P[k]}^{1})$, the interleaved extrinsic information from the previous D1 iteration.]

- compute $\tilde{\alpha}_k^{(2)}(s)$ for all s using (4.18)

end

for $k = 2K : -1 : 2$

- compute $\tilde{\beta}_{k-1}^{(2)}(s)$ for all s using (4.19)

end

for $k = 1 : 2K$

- compute $L_{21}^{e}(v_k)$ using

$$\begin{aligned} L_{21}^{e}(v_k) &= \max_{V^+}{}^{*} \left[\tilde{\alpha}_{k-1}^{(2)}(s') + \tilde{\gamma}_k(s', s) + \tilde{\beta}_k^{(2)}(s)\right] \\ &\quad - \max_{V^-}{}^{*} \left[\tilde{\alpha}_{k-1}^{(2)}(s') + \tilde{\gamma}_k(s', s) + \tilde{\beta}_k^{(2)}(s)\right] - L_{12}^{e}(v_k) \\ &= \max_{V^+}{}^{*} \left[\tilde{\alpha}_{k-1}^{(2)}(s') + v_k y_k^v/\sigma^2 + q_k y_k^q/\sigma^2 + \tilde{\beta}_k^{(2)}(s)\right] \\ &\quad - \max_{V^-}{}^{*} \left[\tilde{\alpha}_{k-1}^{(2)}(s') + v_k y_k^v/\sigma^2 + q_k y_k^q/\sigma^2 + \tilde{\beta}_k^{(2)}(s)\right] \end{aligned}$$

  where V^+ is set of state transition pairs (s', s) corresponding to the event $v_k = +1$, and V^- is similarly defined.

end

D1:

for $k = 1 : K$

– for all allowable state transitions $s' \to s$ set $\tilde{\gamma}_k(s', s)$ via

$$\begin{aligned}\tilde{\gamma}_k(s', s) &= u_k L_{21}^e(u_k)/2 + p_k L_{21}^e(p_k)/2 \\ &= u_k L_{21}^e(c_{2k-1}^1)/2 + p_k L_{21}^e(c_{2k}^1)/2\end{aligned}$$

[u_k (p_k) in the above expression is set to the value of the encoder input (output) corresponding to the transition $s' \to s$; $L_{21}^e(c_{2k-1}^1)$ is $L_{21}^e(v_{Pinv[2k-1]})$, the de-interleaved extrinsic information from the previous D2 iteration, and similarly for $L_{21}^e(c_{2k}^1)$.]

– compute $\tilde{\alpha}_k^{(1)}(s)$ for all s using (4.18)

end

for $k = K : -1 : 2$

– compute $\tilde{\beta}_{k-1}^{(1)}(s)$ for all s using (4.19)

end

for $k = 1 : K$

– compute $L_{12}^e(u_k) = L_{12}^e(c_{2k-1}^1)$ using

$$\begin{aligned}L_{12}^e(u_k) &= \max_{U^+}{}^* \left[\tilde{\alpha}_{k-1}^{(1)}(s') + \tilde{\gamma}_k(s', s) + \tilde{\beta}_k^{(1)}(s)\right] \\ &- \max_{U^-}{}^* \left[\tilde{\alpha}_{k-1}^{(1)}(s') + \tilde{\gamma}_k(s', s) + \tilde{\beta}_k^{(1)}(s)\right] - L_{21}^e(c_{2k-1}^1) \\ &= \max_{U^+}{}^* \left[\tilde{\alpha}_{k-1}^{(1)}(s') + p_k L_{21}^e(p_k)/2 + \tilde{\beta}_k^{(1)}(s)\right] \\ &- \max_{U^-}{}^* \left[\tilde{\alpha}_{k-1}^{(1)}(s') + p_k L_{21}^e(p_k)/2 + \tilde{\beta}_k^{(1)}(s)\right]\end{aligned}$$

– compute $L_{12}^e(p_k) = L_{12}^e(c_{2k}^1)$ using

$$\begin{aligned}L_{12}^e(p_k) &= \max_{P^+}{}^* \left[\tilde{\alpha}_{k-1}^{(1)}(s') + \tilde{\gamma}_k(s', s) + \tilde{\beta}_k^{(1)}(s)\right] \\ &- \max_{P^-}{}^* \left[\tilde{\alpha}_{k-1}^{(1)}(s') + \tilde{\gamma}_k(s', s) + \tilde{\beta}_k^{(1)}(s)\right] - L_{21}^e(c_{2k}^1) \\ &= \max_{P^+}{}^* \left[\tilde{\alpha}_{k-1}^{(1)}(s') + u_k L_{21}^e(u_k)/2 + \tilde{\beta}_k^{(1)}(s)\right] \\ &- \max_{P^-}{}^* \left[\tilde{\alpha}_{k-1}^{(1)}(s') + u_k L_{21}^e(u_k)/2 + \tilde{\beta}_k^{(1)}(s)\right]\end{aligned}$$

end

After the Last Iteration

for $k = 1 : K$

– for all allowable state transitions $s' \to s$ set $\tilde{\gamma}_k(s', s)$ via

$$\tilde{\gamma}_k(s', s) = u_k L_{21}^e(c_{2k-1}^1)/2 + p_k L_{21}^e(c_{2k}^1)/2$$

– compute $L(u_k)$ using

$$L(u_k) = \max_{U^+}{}^* \left[\tilde{\alpha}^{(1)}_{k-1}(s') + \tilde{\gamma}_k(s', s) + \tilde{\beta}^{(1)}_k(s)\right] - \max_{U^-}{}^* \left[\tilde{\alpha}^{(1)}_{k-1}(s') + \tilde{\gamma}_k(s', s) + \tilde{\beta}^{(1)}_k(s)\right]$$

– $\hat{u}_k = \text{sign}\,[L(u_k)]$

end

5. Conclusion

We have seen in this chapter the how and why of both parallel and serial turbo codes. That is, we have seen how to decode these codes using an iterative decoder, and why they should be expected to perform so well. The decoding algorithm summaries should be sufficient to decode any binary parallel and serial turbo codes, and can in fact be easily extended to the iterative decoding of any binary hybrid schemes. It is not much more work to figure out how to decode any of the turbo trellis-coded modulation (turbo-TCM) schemes that appear in the literature. In any case, this chapter should serve well as a starting point for the study of concatenated codes (and perhaps graph-based codes) and their iterative decoders.

Acknowledgment

The author would like to thank Rajeev Ramamurthy and Bo Xia for producing Figs. 4 and 5, and Steve Wilson, Masoud Salehi, and John Proakis for helpful comments. He would also like to thank Cheryl Drier for typing the first draft.

References

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error- correcting coding and decoding: Turbo codes," *Proc. 1993 Int. Conf. Comm.*, pp. 1064- 1070.

[2] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding: turbo-codes," *IEEE Trans. Comm.*, pp. 1261-1271, Oct. 1996.

[3] G. Ungerboeck, "Channel coding with multilevel/phase signals," *IEEE Trans. Inf. Theory*, pp. 55-67, Jan. 1982.

[4] S. Benedetto and G. Montorsi, "Unveiling turbo codes: Some results on parallel concatenated coding schemes," *IEEE Trans. Inf. Theory*, pp. 409-428, Mar. 1996.

[5] S. Benedetto and G. Montorsi, "Design of parallel concatenated codes," *IEEE Trans. Comm.*, pp. 591-600, May 1996.

[6] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Serial concatenation of interleaved codes: Performance analysis, design, and iterative decoding," *IEEE Trans. Inf. Theory*, pp. 909-926, May 1998.

[7] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "A soft-input soft-output maximum a posteriori (MAP) module to decode parallel and serial concatenated codes," TDA Progress Report 42-127, Nov. 15, 1996.

[8] D. Divsalar and F. Pollara, "Multiple turbo codes for deep-space communications," JPL TDA Progress Report, 42-121, May 15, 1995.

[9] J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Trans. Inf. Theory*, pp. 429-445, Mar. 1996.

[10] L. Perez, J. Seghers, and D. Costello, "A distance spectrum interpretation of turbo codes," *IEEE Trans. Inf. Theory*, pp. 1698-1709, Nov. 1996.

[11] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and suboptimal MAP decoding algorithms operating in the log domain," *Proc. 1995 Int. Conf. on Comm.*, pp. 1009-1013.

[12] A. Viterbi, "An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes," *IEEE JSAC*, pp. 260-264, Feb. 1998.

[13] O. Acikel and W. Ryan, "Punctured turbo codes for BPSK/QPSK channels," *IEEE Trans. Comm.*, pp. 1315-1323, Sept. 1999.

[14] O. Acikel and W. Ryan, "Punctured high rate SCCCs for BPSK/QPSK channels," *Proc. 2000 IEEE International Conference on Communications*, vol. 1, pp. 434-439.

[15] J. Pearl, *Probabilistic Reasoning in Intelligent Systems*, San Mateo, CA: Morgan Kaufmann, 1988.

[16] B. Frey, *Graphical Models for Machine Learning and Digital Communication*, MIT Press, 1998.

[17] N. Wiberg, *Codes and Decoding on General Graphs*, Ph.D. dissertation, U. Linköping, Sweden, 1996.

[18] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inf. Theory*, pp. 284-287, Mar. 1974.

[19] D. Divsalar and F. Pollara, "Turbo codes for PCS applications," *Proc. 1995 Int. Conf. Comm.*, pp. 54-59.