Comparative Study of Turbo Decoding Techniques: An Overview

Jason P. Woodard and Lajos Hanzo

***Abstract*—In this contribution, we provide an overview of the novel class of channel codes referred to as turbo codes, which have been shown to be capable of performing close to the Shannon Limit. We commence with a brief discussion on turbo encoding, and then move on to describing the form of the iterative decoder most commonly used to decode turbo codes. We then elaborate on various decoding algorithms that can be used in an iterative decoder, and give an example of the operation of such a decoder using the so-called Soft Output Viterbi Algorithm (SOVA). Lastly, the effect of a range of system parameters is investigated in a systematic fashion, in order to gauge their performance ramifications.**

I. Introduction

TURBO coding was introduced in 1993 by Berrou, Glavieux, and Thitimajashima [1], [2], who reported extremely impressive results for a code with a long frame length. Since its recent invention, turbo coding has evolved at an unprecedented rate and has reached a state of maturity within just a few years due to the intensive research efforts of the turbo coding community. As a result, turbo coding has also found its way into standardized systems, such as the recently standardized third-generation (3G) mobile radio systems [3]. Even more impressive performance gains can be attained with the aid of turbo coding in the context of video broadcast systems [4], [5], where the associated system delay is less critical than in delay-sensitive interactive systems. Yet, surprisingly, in this area turbo codecs have not been used in standards at the time of writing. Motivated by these recent trends, in this contribution we endeavour to provide an accessible introduction to the field of turbo coding.

In their paper, Berrou *et al.* [1], [2] used a paralled concatenation of two Recursive Systematic Convolutional (RSC) codes, with an interleaver between the two encoders. The reason for using RSC codes will be augmented during our forthcoming in-depth discourse. Suffice to say at this stage that an iterative structure using a modified version of the classic minimum bit error rate (BER) Maximum Aposteriory Algorithm (MAP) due to Bahl *et al.* [6] was invoked, in order to decode the codes. Since then, a large body of work has been carried out in the area, aiming, for example, to reduce the decoder complexity, as suggested by Robertson *et al.* [7], Berrou *et al.* [9], as well as by Battail [8]. Le Goff *et al.* [10], Wachsmann and Huber [11], as well as Robertson and Worz [12] suggested to use the codes in conjunction with bandwidth efficient modulation schemes. Further advances in understanding the excellent preformance of the codes are due, for example, to Benedetto and Montorsi [13], [15], Perez *et al.* [14]. Hagenauer *et al.* [16], [17] extend the concept to use concatenated block codes. Jung and Naßhan [36], [34] characterized the coded performance under the constraints of short transmission frame length, which is characteristic of speech systems. In collaboration with Blanz, they also applied turbo codes to a CDMA system using joint detection and antenna diversity [39]. Barbulescu and Pietrobon addressed the issues of interleaver design [31]. Due to space limitations here we have to curtail listing the range of further contributors in the field, without whose advances this contribution could not have been written. In this paper, we build on a previous tutorial paper by Sklar [18] in describing the iterative decoder, and the component decoders used within it, that are employed to decode turbo codes. For more general information on turbo codes, the reader is referred to [18].

The paper is structured as follows. Section II is concerned with the basic iterative decoder scheme, leading on to a discussion on the MAP decoding algorithm and its underlying theory in Section III. Section IV justifies the advantages of iterative decoding, while Section V considers the simplification of the MAP algorithm, paving the way for introducing the Soft-Output Virebi Algorithm (SOVA), which is also augmented with examples in Section VII. Section VIII compares the various decoder principles, which are then comparatively studied in terms of their performance in Section IX over Gaussian channels, while in Section X over Rayleigh channels. We conclude in Section X.

II. Iterative Decoder Structure

Let us commence our discourse by considering the general structure of the iterative turbo decoder shown in Fig. 1. Two component decoders are linked by interleavers in a structure similar to that of the encoder. As seen in the figure, each decoder takes three inputs: 1) the systematically encoded channel output bits; 2) the parity bits transmitted from the associated component encoder; and 3) the information from the other component decoder about the likely values of the bits concerned. This information from the other decoder is referred to as *a-priori* information. The component decoders have to exploit both the inputs from the channel and this *a-priori* information. They must also provide what are known as soft outputs for the decoded bits. This means that as well as providing the decoded output bit sequence, the component decoders must also give the associated probabilities for each bit that it has been correctly decoded. Two

Manuscript received September 16, 1998; revised July 19, 2000. This work was supported by Motorola ECID, Swindon, U.K. and the European Commission in the framework of the First and Median projects.

J. P. Woodard is with the Department of Electrical and Computer Science, University of Southampton, SO17 1BJ, U.K. (e-mail: jpw@ecs.soton.ac.uk, http://www-mobile.ecs.soton.ac.uk).

L. Hanzo is with the Department of Electrical and Computer Science, University of Southampton, SO17 1BJ, U.K. (e -mail: lh@ecs.soton.ac.uk, http://www-mobile.ecs.soton.ac.uk).

Publisher Item Identifier S 0018-9545(00)10962-4.

0018–9545/00$10.00 © 2000 IEEE

suitable decoders are the so-called SOVA proposed by Hagenauer and Hoeher [19] and the MAP [6] algorithm of Bahl *et al.* which are described in Sections VI and III, respectively.

The soft outputs from the component decoders are typically represented in terms of the so-called Log Likelihood Ratios (LLRs), the magnitude of which gives the sign of the bit, and the amplitude the probability of a correct decision. The LLRs are simply, as their name implies, the logarithm of the ratio of two probabilities. For example, the LLR $L(u_k)$ for the value of a decoded bit u_k is given by

$$L(u_k) = \ln\left(\frac{P(u_k = +1)}{P(u_k = -1)}\right) \tag{1}$$

where $P(u_k = +1)$ is the probability that the bit $u_k = +1$, and similarly for $P(u_k = -1)$. Notice that the two possible values of the bit u_k are taken to be $+1$ and -1, rather than 1 and 0, as this simplifies the derivations that follow.

The decoder of Fig. 1 operates iteratively, and in the first iteration the first component decoder takes channel output values only, and produces a soft output as its estimate of the data bits. The soft output from the first encoder is then used as additional information for the second decoder, which uses this information along with the channel outputs to calculate its estimate of the data bits. Now the second iteration can begin, and the first decoder decodes the channel outputs again, but now with additional information about the value of the input bits provided by the output of the second decoder in the first iteration. This additional information allows the first decoder to obtain a more accurate set of soft outputs, which are then used by the second decoder as *a-priori* information. This cycle is repeated, and with every iteration the BER of the decoded bits tends to fall. However, the improvement in performance obtained with increasing numbers of iterations decreases as the number of iterations increases. Hence, for complexity reasons, usually only about eight iterations are used.

Due to the interleaving used at the encoder, care must be taken to properly interleave and de-interleave the LLRs which are used to represent the soft values of the bits, as seen in Fig. 1. Furthermore, because of the iterative nature of the decoding, care must be taken not to re-use the same information more than once at each decoding step. For this reason the concept of so-called extrinsic and intrinsic information was used in their seminal paper by Berrou *et al.* describing iterative decoding of turbo codes [1]. These concepts and the reason for the subtraction circles shown in Fig. 1 are described in Section IV. Having considered the basic decoder structure, let us now focus our attention on the MAP algorithm in the next section.

III. The Maximum A-Posteriori Algorithm

A. *Introduction and Mathematical Preliminaries*

In 1974 ,an algorithm, which has become known as the MAP algorithm, was proposed by Bahl *et al.* [6] in order to estimate the *a-posteriori* probabilities of the states and the transitions of a Markov source observed in memoryless noise. Bahl *et al.* showed how the algorithm could be used to decode both block and convolutional codes. When used to decode convolutional

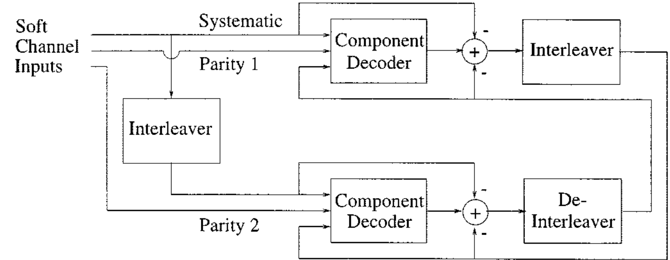

Fig. 1. Turbo decoder schematic.

codes, the algorithm is optimal in terms of minimizing the decoded BER, unlike the Viterbi algorithm [20], which minimizes the probability of an incorrect path through the trellis being selected by the decoder. Thus the Viterbi algorithm can be thought of as minimizing the number of *groups* of bits associated with these trellis paths, rather than the actual number of bits, which are decoded incorrectly. Nevertheless, as stated by Bahl *et al.* in [6], in most applications the performance of the two algorithms will be almost identical. However, the MAP algorithm examines every possible path through the convolutional decoder trellis and therefore initially seemed to be unfeasibly complex for application in most systems. Hence, it was not widely used before the discovery of turbo codes.

However, the MAP algorithm provides not only the estimated bit sequence, but also the probabilities for each bit that it has been decoded correctly. This is essential for the iterative decoding of turbo codes proposed by Berrou *et al.* [1], and so MAP decoding was used in this seminal paper. Since then research efforts have been invested in reducing the complexity of the MAP algorithm to a reasonable level. In this section we describe the theory behind the MAP algorithm as used for the soft output decoding of the component convolutional codes of turbo codes. It is assumed that binary codes are used.

The MAP algorithm gives, for each decoded bit u_k, the probability that this bit was $+1$ or -1, given the received symbol sequence $\underline{y}$. This is equivalent to finding the *a-posteriori* LLR $L(u_k \,|\, \underline{y})$, where

$$L(u_k \,|\, \underline{y}) = \ln\left(\frac{P(u_k = +1 \,|\, \underline{y})}{P(u_k = -1 \,|\, \underline{y})}\right). \tag{2}$$

If the previous state $S_{k-1} = \grave{s}$ and the present state $S_k = s$ are known in a trellis then the input bit u_k which caused the transition between these states will be known. This, along with Bayes' rule and the fact that the transitions between the previous S_{k-1} the present state S_k in a trellis are mutually exclusive (i.e., only one of them could have occured at the encoder), allow us to rewrite (2) as

$$L(u_k \,|\, \underline{y}) = \ln\left(\frac{\sum_{\substack{(\grave{s},s)\Rightarrow \\ u_k=+1}} P(S_{k-1} = \grave{s} \wedge S_k = s \wedge \underline{y})}{\sum_{\substack{(\grave{s},s)\Rightarrow \\ u_k=-1}} P(S_{k-1} = \grave{s} \wedge S_k = s \wedge \underline{y})}\right) \tag{3}$$

where $(\grave{s}, s) \Rightarrow u_k = +1$ is the set of transitions from the previous state $S_{k-1} = \grave{s}$ to the present state $S_k = s$ that can occur if the input bit $u_k = +1$, and similarly for $(\grave{s}, s) \Rightarrow u_k = -1$. For brevity we shall write $P(S_{k-1} = \grave{s} \wedge S_k = s \wedge \underline{y})$ as $P(\grave{s} \wedge s \wedge \underline{y})$.

We now consider the individual probabilities $P(\grave{s} \wedge s \wedge \underline{y})$ from the numerator and denominator of (3). The received sequence $\underline{y}$ can be split up into three sections: the received codeword associated with the present transition $\underline{y}_k$, the received sequence prior to the present transition $\underline{y}_{j<k}$ and the received sequence after the present transition $\underline{y}_{j>k}$. We can thus write for the individual probabilities $P(\grave{s} \wedge s \wedge \underline{y})$

$$P(\grave{s} \wedge s \wedge \underline{y}) = P(\grave{s} \wedge s \wedge \underline{y}_{j<k} \wedge \underline{y}_k \wedge \underline{y}_{j>k}). \quad (4)$$

Fig. 2, which shows a section of a four state trellis for a $K = 3$ RSC code, shows this split of the received channel sequence. In this figure solid lines represent transitions as a result of a -1 input bit, and dashed lines represent transistion resulting from a $+1$ input bit. The $\alpha_{k-1}(\grave{s})$, $\gamma_k(\grave{s}, s)$, and $\beta_k(s)$ symbols shown represent values, which will be defined shortly, calculated by the MAP algorithm.

Using a derivation from Bayes' rule that $P(a \wedge b) = P(a \,|\, b)P(b)$ and the fact that if we assume that the channel is memoryless, then the future received sequence $\underline{y}_{j>k}$ will depend only on the present state s and not on the previous state $\grave{s}$ or the present and previous received channel sequences $\underline{y}_k$ and $\underline{y}_{j<k}$, we can write

$$\begin{aligned} P(\grave{s} \wedge s \wedge \underline{y}) &= P(\grave{s} \wedge s \wedge \underline{y}_{j<k} \wedge \underline{y}_k \wedge \underline{y}_{j>k}) \\ &= P(\underline{y}_{j>k} \,|\, s) \cdot P(\grave{s} \wedge s \wedge \underline{y}_{j<k} \wedge \underline{y}_k) \\ &= P(\underline{y}_{j>k} \,|\, s) \cdot P(\{\underline{y}_k \wedge s\} \,|\, \grave{s}) \\ &\quad \cdot P(\grave{s} \wedge \underline{y}_{j<k}) \\ &= \beta_k(s) \cdot \gamma_k(\grave{s}, s) \cdot \alpha_{k-1}(\grave{s}) \end{aligned} \quad (5)$$

where

$$\alpha_{k-1}(\grave{s}) = P(S_{k-1} = \grave{s} \wedge \underline{y}_{j<k}) \quad (6)$$

is the probability that the trellis is in state $\grave{s}$ at time $k-1$ and the received channel sequence up to this point is $\underline{y}_{j<k}$, as visualized in Fig. 2

$$\beta_k(s) = P(\underline{y}_{j>k} \,|\, S_k = s) \quad (7)$$

is the probability that given the trellis is in state s at time k the future received channel sequence will be $\underline{y}_{j>k}$, and lastly

$$\gamma_k(\grave{s}, s) = P(\{\underline{y}_k \wedge S_k = s\} \,|\, S_{k-1} = \grave{s}) \quad (8)$$

is the probability that given the trellis was in state $\grave{s}$ at time $k-1$, it moves to state s and the received channel sequence for this transition is $\underline{y}_k$.

Equation (5) shows that the probability $P(\grave{s} \wedge s \wedge \underline{y})$, that the encoder trellis took the transition from state $S_{k-1} = \grave{s}$ to state $S_k = s$ and the received sequence is $\underline{y}$, can be split into the product of three terms—$\alpha_{k-1}(\grave{s})$, $\gamma_k(\grave{s}, s)$ and $\beta_k(s)$. The meaning of these three probability terms is shown in Fig. 2, for the transition $S_{k-1} = \grave{s}$ to $S_k = s$ shown by the bold line in this

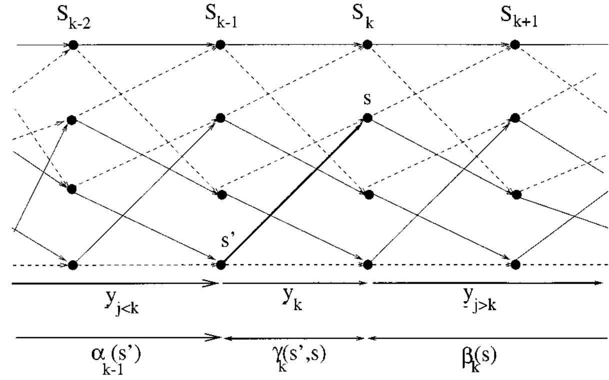

Fig. 2. MAP decoder trellis for $K = 3$ RSC code.

figure. From (3) and (5) we can write for the conditional LLR of u_k, given the received sequence $\underline{y}_k$

$$\begin{aligned} L(u_k \,|\, \underline{y}) &= \ln \left(\frac{\sum_{\substack{(\grave{s},s) \Rightarrow \\ u_k=+1}} P(S_{k-1} = \grave{s} \wedge S_k = s \wedge \underline{y})}{\sum_{\substack{(\grave{s},s) \Rightarrow \\ u_k=-1}} P(S_{k-1} = \grave{s} \wedge S_k = s \wedge \underline{y})} \right) \\ &= \ln \left(\frac{\sum_{\substack{(\grave{s},s) \Rightarrow \\ u_k=+1}} \alpha_{k-1}(\grave{s}) \cdot \gamma_k(\grave{s}, s) \cdot \beta_k(s)}{\sum_{\substack{(\grave{s},s) \Rightarrow \\ u_k=-1}} \alpha_{k-1}(\grave{s}) \cdot \gamma_k(\grave{s}, s) \cdot \beta_k(s)} \right). \end{aligned} \quad (9)$$

The MAP algorithm finds $\alpha_k(s)$ and $\beta_k(s)$ for all states s throughout the trellis, i.e., for $k = 0, 1 \ldots N-1$, and $\gamma_k(\grave{s}, s)$ for all possible transitions from state $S_{k-1} = \grave{s}$ to state $S_k = s$, again for $k = 0, 1 \ldots N-1$. These values are then used with (9) to give the conditional LLRs $L(u_k \,|\, \underline{y})$ that the MAP decoder delivers. These operations are summarized in the flowchart of Fig. 4. We now describe how the values $\alpha_k(s)$, $\beta_k(s)$, and $\gamma_k(\grave{s}, s)$ can be calculated.

B. The Forward Recursive Calculation of the $\alpha_k(s)$ Values

Consider first $\alpha_k(s)$. From the definition of $\alpha_{k-1}(\grave{s})$ in (6) we can write

$$\begin{aligned} \alpha_k(s) &= P(S_k = s \wedge \underline{y}_{j<k+1}) \\ &= P(s \wedge \underline{y}_{j<k} \wedge \underline{y}_k) \\ &= \sum_{\text{all } \grave{s}} P(s \wedge \grave{s} \wedge \underline{y}_{j<k} \wedge \underline{y}_k) \end{aligned} \quad (10)$$

where in the last line we split the probability $P(s \wedge \underline{y}_{y<k+1})$ into the sum of joint probabilities $P(s \wedge \grave{s} \wedge \underline{y}_{j<k+1})$ over all possible previous states $\grave{s}$. Using Bayes' rule and the assumption that the channel is memoryless again, we can proceed as follows:

$$\begin{aligned} \alpha_k(s) &= \sum_{\text{all } \grave{s}} P(s \wedge \grave{s} \wedge \underline{y}_{j<k} \wedge \underline{y}_k) \\ &= \sum_{\text{all } \grave{s}} P(\{s \wedge \underline{y}_k\} \,|\, \{\grave{s} \wedge \underline{y}_{j<k}\}) \cdot P(\grave{s} \wedge \underline{y}_{j<k}) \\ &= \sum_{\text{all } \grave{s}} P(\{s \wedge \underline{y}_k\} \,|\, \grave{s}) \cdot P(\grave{s} \wedge \underline{y}_{j<k}) \\ &= \sum_{\text{all } \grave{s}} \alpha_{k-1}(\grave{s}) \cdot \gamma_k(\grave{s}, s). \end{aligned} \quad (11)$$

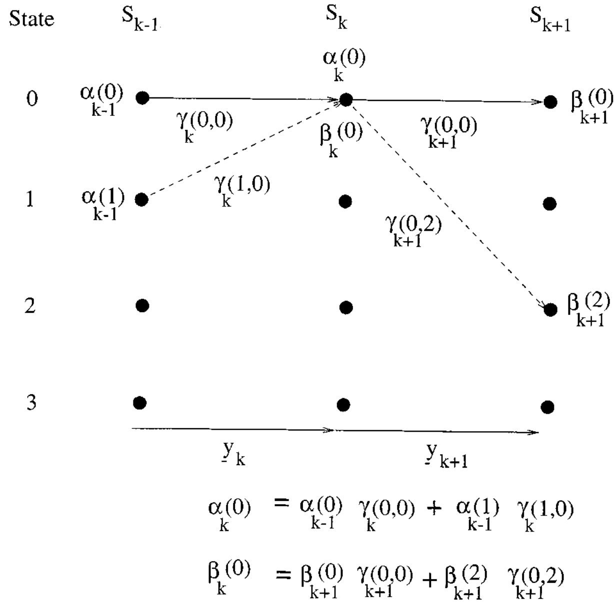

Fig. 3. Recursive calculation of $\alpha_k(0)$ and $\beta_k(0)$.

Thus, once the $\gamma_k(\grave{s}, s)$ values are known, the $\alpha_k(s)$ values can be calculated recursively. Assuming that the trellis has the initial state $S_0 = 0$, the initial conditions for this recursion are

$$\begin{aligned}\alpha_0(S_0 = 0) &= 1 \\ \alpha_0(S_0 = s) &= 0 \quad \text{for all } s \neq 0.\end{aligned} \tag{12}$$

Fig. 3 shows an example of how one $\alpha_k(s)$ value, for $s = 0$, is calculated recursively using values of $\alpha_{k-1}(\grave{s})$ and $\gamma_k(\grave{s}, s)$ for our example $K = 3$ RSC code. Notice that, as we are considering a binary trellis, only two previous states, $S_{k=1} = 0$ and $S_{k-1} = 1$, have paths to the state $S_k = 0$. Therefore, the summation in (11) is over only two terms.

C. The Backward Recursive Calculation of the $\beta_k(s)$ Values

The values of $\beta_k(s)$ can similarly be calculated recursively. Using a similar derivation to that for (11) it can be shown that

$$\begin{aligned}\beta_{k-1}(\grave{s}) &= P(\underline{y}_{j>k-1} \mid \grave{s}) \\ &= \sum_{\text{all } s} \beta_k(s) \cdot \gamma_k(\grave{s}, s).\end{aligned} \tag{13}$$

Thus, once the values $\gamma_k(\grave{s}, s)$ are known, a backward recursion can be used to calculate the values of $\beta_{k-1}(\grave{s})$ from the values of $\beta_k(s)$ using (13). Fig. 3 again shows an example of how the $\beta_k(0)$ value is calculated recursively using values of $\beta_{k+1}(s)$ and $\gamma_{k+1}(0, s)$ for our example $K = 3$ RSC code.

D. Calculation of the $\gamma_k(\grave{s}, s)$ Values

We now consider how the transition probability values $\gamma_k(\grave{s}, s)$ in (5) can be calculated from the received channel sequence and any *a-priori* information that is available. Using the definition of $\gamma_k(\grave{s}, s)$ from (8) and the derivation from Bayes' rule we have

$$\begin{aligned}\gamma_k(\grave{s}, s) &= P(\{\underline{y}_k \wedge s\} \mid \grave{s}) \\ &= P(\underline{y}_k \mid \{\grave{s} \wedge s\}) \cdot P(s \mid \grave{s}) \\ &= P(\underline{y}_k \mid \{\grave{s} \wedge s\}) \cdot P(u_k) \\ &= P(\underline{y}_k \mid \underline{x}_k) \cdot P(u_k)\end{aligned} \tag{14}$$

where

- u_k input bit necessary to cause the transition from state $S_{k-1} = \grave{s}$ to state $S_k = s$;
- $P(u_k)$ *a-priori* probability of this bit;
- $\underline{x}_k$ transmitted codeword associated with this transition.

Hence, the transition probability $\gamma_k(\grave{s}, s)$ is given by the product of the *a-priori* probability of the input bit u_k necessary for the transisiton, and the probability that given the codeword $\underline{x}_k$ associated with the transition was transmitted we received the channel sequence $\underline{y}_k$. The *a-priori* probability $P(u_k)$ is derived in an iterative decoder from the output of the previous component decoder, and the conditional received sequence probability $P(\underline{y}_k \mid \underline{x}_k)$ is given, assuming a memoryless Gaussian channel with BPSK modulation, as

$$\begin{aligned}P(\underline{y}_k \mid \underline{x}_k) &= \prod_{l=1}^{n} P(y_{kl} \mid x_{kl}) \\ &= \prod_{l=1}^{n} \frac{1}{\sqrt{2\pi}\sigma} e^{\left(-\frac{E_b R}{2\sigma^2}(y_{kl} - a x_{kl})^2\right)}\end{aligned} \tag{15}$$

where

- x_{kl} and y_{kl} individual bits within the transmitted and received codewords $\underline{y}_k$ and $\underline{x}_k$;
- n number of these bits in each codeword;
- E_b transmitted energy per bit;
- σ^2 noise variance;
- a fading amplitude (we have $a = 1$ for nonfading AWGN channels).

E. Summary of the MAP Algorithm

From the description given above, we see that the MAP decoding of a received sequence $\underline{y}$ to give the *a-posteriori* LLR $L(u_k \mid \underline{y})$ can be carried out as follows. As the channel values y_{kl} are received, they and the *a-priori* LLRs $L(u_k)$ (which are provided in an iterative turbo decoder by the other component decoder—see Section IV) are used to calculate $\gamma_k(\grave{s}, s)$ according to (14) and (15). As the channel values y_{kl} are received, and the $\gamma_k(\grave{s}, s)$ values are calculated, the forward recursion from (11) can be used to calculate $\alpha_k(\grave{s}, s)$. Once all the channel values have been received, and $\gamma_k(\grave{s}, s)$ has been calculated for all $k = 1, 2 \ldots N$, the backward recursion from (13) can be used to calculate the $\beta_k(\grave{s}, s)$ values. Finally, all the calculated values of $\alpha_k(\grave{s}, s)$, $\beta_k(\grave{s}, s)$ and $\gamma_k(\grave{s}, s)$ are used in (9) to calculate the values of $L(u_k \mid \underline{y})$. These operations are summarized in the flowchart of Fig. 4. Care must be taken to avoid numerical underflow problems in the recursive calculation of $\alpha_k(\grave{s}, s)$ and $\beta_k(\grave{s}, s)$, but such problems can be avoided by careful normalization of these values. Such normalization cancels out in the ratio in (9) and so causes no change in the LLRs produced by the algorithm.

The MAP algorithm is, in the form described in this section, extremely complex due to the multiplications needed in (11) and (13) for the recursive calculation of $\alpha_k(\grave{s}, s)$ and $\beta_k(\grave{s}, s)$, the multiplications and exponential operations required to calculate $\gamma_k(\grave{s}, s)$ using (15), and the multiplication and natural logarithm

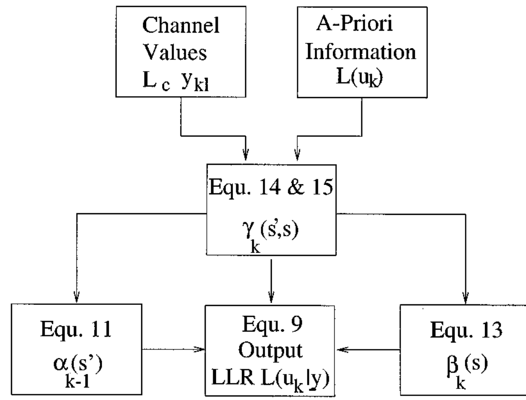

Fig. 4. Summary of the key operations in the MAP algorithm.

operations required to calculate $L(u_k \,|\, \underline{y})$ using (9). However, much work has been done to reduce this complexity, and the Log-MAP algorithm [7], which will be described in Section V, gives the same performance as the MAP algorithm, but at a significantly reduced complexity and without the numerical problems described above. In the next section we will first describe the principles behind the iterative decoding of turbo codes, and how the MAP algorithm described in this section can be used in such a scheme, before detailing the Log-MAP algorithm.

IV. Iterative Turbo Decoding Principles

A. Turbo Decoding Mathematical Preliminaries

In this section, we explain the concepts of extrinsic and intrinsic information as used by Berrou *et al.* [1], and highlight how the MAP algorithm described in the previous section, and other soft-in soft-out decoders, can be used in the iterative decoding of turbo codes.

It can be shown [1] that, for a systematic code such as a RSC code, the output from the MAP decoder, given by (9), can be re-written as

$$L(u_k \,|\, \underline{y}) = \ln\left(\frac{\sum_{\substack{(\grave{s},s)\Rightarrow \\ u_k=+1}} \alpha_{k-1}(\grave{s}) \cdot \gamma_k(\grave{s},s) \cdot \beta_k(s)}{\sum_{\substack{(\grave{s},s)\Rightarrow \\ u_k=-1}} \alpha_{k-1}(\grave{s}) \gamma_k(\grave{s},s) \cdot \beta_k(s)}\right) = L(u_k) + L_c y_{ks} + L_e(u_k) \quad (16)$$

where

$$L_e(u_k) = \ln\left(\frac{\sum_{\substack{(\grave{s},s)\Rightarrow \\ u_k=+1}} \alpha_{k-1}(\grave{s}) \cdot \chi_k(\grave{s},s) \cdot \beta_k(s)}{\sum_{\substack{(\grave{s},s)\Rightarrow \\ u_k=-1}} \alpha_{k-1}(\grave{s}) \cdot \chi_k(\grave{s},s) \cdot \beta_k(s)}\right). \quad (17)$$

Here, $L(u_k)$ is the *a-priori* LLR given by (1), and L_c is called the channel reliability measure and is given by

$$L_c = \frac{4a}{2\sigma^2} \quad (18)$$

y_{ks} is the received version of the transmitted systematic bit $x_{ks} = u_k$ and

$$\chi_k(\grave{s},s) = \exp\left(\frac{L_c}{2} \sum_{l=2}^{n} y_{kl} x_{yl}\right). \quad (19)$$

Thus, we can see that the *a-posteriori* LLR $L(u_k \,|\, \underline{y})$ calculated with the MAP algorithm can be thought of as comprising of three terms—$L(u_k)$, $L_c y_{ks}$, and $L_e(u_k)$. The *a-priori* LLR term $L(u_k)$ comes from $P(u_k)$ in the expression for the branch transition probability $\gamma_k(\grave{s},s)$ in (14). This probability should come from an independent source. In most cases we will have no independent or *a-priori* knowledge of the likely value of the bit u_k, and so the *a-priori* LLR $L(u_k)$ will be zero, corresponding to an *a-priori* probability $P(u_k) = 0.5$. However, in the case of an iterative turbo decoder, each component decoder can provide the other decoder with an estimate of the *a-priori* LLR $L(u_k)$, as described later.

The second term $L_c y_{ks}$ in (16) is the soft output of the channel for the systematic bit u_k, which was directly transmitted across the channel and received as y_{ks}. When the channel signal-to-noise (SNR) is high, the channel reliability value L_c of (18) will be high and this systematic bit will have a large influence on the *a-posteriori* LLR $L(u_k \,|\, \underline{y})$. Conversely, when the channel is poor and L_c is low, the soft output of the channel for the received systematic bit y_{ks} will have less impact on the *a-posteriori* LLR delivered by the MAP algorithm.

The final term in (16), $L_e(u_k)$, is derived, using the constraints imposed by the code used, from the *a-priori* information sequence $L(u_n)$ and the received channel information sequence $\underline{y}$, *excluding* the received systematic bit y_{ks} and the *a-priori* information $L(u_k)$ for the bit u_k. Hence, it is called the *extrinsic* LLR for the bit u_k. Equation (16) shows that the extrinsic information from a MAP decoder can be obtained by subtracting the *a-priori* information $L(u_k)$ and the received systematic channel input $L_c y_{ks}$ from the soft output $L(u_k \,|\, \underline{y})$ of the decoder. This is the reason for the subtraction paths shown in Fig. 1. Equations similar to (16) can be derived for the other component decoders which are used in iterative turbo decoding.

We summarize below what is meant by the terms *a-priori*, *a-posteriori*, and *extrinsic information* which are central concepts behind the iterative decoding of turbo codes use throughout this treatise.

a-priori The *a-priori* information about a bit is information known before decoding starts, from a source other than the received sequence or the code constraints. It is also sometimes referred to as intrinsic information to contrast with the extrinsic information described next.

extrinsic The extrinsic information about a bit u_k is the information provided by a decoder based on the received sequence and on *a-priori* information *excluding* the received systematic bit y_{ks} and the *a-priori* information $L(u_k)$ for the bit u_k. Typically, the component decoder provides this information using the constraints imposed on the transmitted sequence by the code used. It processes the received bits and *a-priori* information surrounding the systematic bit u_k, and uses this information and the code constraints to provide information about the value of u_k.

a-posteriori The *a-posteriori* information about a bit is the information that the decoder gives taking into

account *all* available sources of information about u_k. It is the *a-posteriori* LLR, i.e., $L(u_k \mid \underline{y})$, that the MAP algorithm gives as its output.

B. Iterative Turbo Decoding

We now describe how the iterative decoding of turbo codes is carried out. Consider initially the first component decoder in the first iteration. This decoder receives the channel sequence $L_c\underline{y}^{(1)}$ containing the received versions of the transmitted systematic bits $L_c y_{ks}$, and the parity bits $L_c y_{kl}$, from the first encoder. Usually, to obtain a half rate code, half of these parity bits will have been punctured at the transmitter, and so the turbo decoder must insert zeros in the soft channel output $L_c y_{kl}$ for these punctured bits. The first component decoder can then process the soft channel inputs and produce its estimate $L_{11}(u_k \mid \underline{y})$ of the conditional LLRs of the data bits u_k, $k = 1, 2 \ldots N$. In this notation, the subscript 11 in $L_{11}(u_k \mid \underline{y})$ indicates that this is the *a-posteriori* LLR in the first iteration from the first component decoder. Note that in this first iteration the first component decoder will have no *a-priori* information about the bits, and hence $P(u_k)$ in (14) giving $\gamma_k(\grave{s}, s)$ will be 0.5.

Next, the second component decoder comes into operation. It receives the channel sequence $\underline{y}^{(2)}$ containing the *interleaved* version of the received systematic bits, and the parity bits from the second encoder. Again the turbo-decoder will need to insert zeroes into this sequence if the parity bits generated by the encoder are punctured before transmission. However, now, in addition to the received channel sequence $\underline{y}^{(2)}$, the decoder can use the conditional LLR $L_{11}(u_k \mid \underline{y})$ provided by the first component decoder to generate *a-priori* LLRs $L(u_k)$ to be used by the second component decoder. Metaphorically speaking, these *a-priori* LLRs $L(u_k)$—which are related to bit u_k—would be provided by an "independent conduit of information, in order to have two independent channel-impaired opinions" concerning bit u_k. This would provide a "second channel-impaired opinion" as regards to bit u_k. In an iterative turbo decoder, the extrinsic information $L_e(u_k)$ from the other component decoder is used as the *a-priori* LLRs, after being interleaved to arrange the decoded data bits $\underline{u}$ in the same order as they were encoded by the second encoder. The second component decoder thus uses the received channel sequence $\underline{y}^{(2)}$ and the *a-priori* LLRs $L(u_k)$ (derived by interleaving the extrinsic LLRs $L_e(u_k)$ of the first component decoder) to produce its *a-posteriori* LLRs $L_{12}(u_k \mid \underline{y})$. This is then the end of the first iteration.

For the second iteration the first component encoder again processes its received channel sequence $\underline{y}^{(1)}$, but now it also has *a-priori* LLRs $L(u_k)$ provided by the extrinsic portion $L_e(u_k)$ of the *a-posteriori* LLRs $L_{12}(u_k \mid \underline{y})$ calculated by the second component encoder, and hence it can produce an improved *a-posteriori* LLR $L_{21}(u_k \mid \underline{y})$. The second iteration then continues with the second component decoder using the improved *a-posteriori* LLRs $L_{21}(u_k \mid \underline{y})$ from the first encoder to derive, through (16), improved *a-priori* LLRs $L(u_k)$ which it uses in conjunction with its received channel sequence $\underline{y}^{(2)}$ to calculate $L_{22}(u_k \mid \underline{y})$.

This iterative process continues, and with each iteration on average the BER of the decoded bits will fall. However, in [18, Fig. 9], the improvement in performance for each additional iteration carried out falls as the number of iterations increases. Hence, for complexity reasons usually only around six to eight iterations are carried out, as no significant improvement in performance is obtained with a higher number of iterations.

Fig. 5 shows how the *a-posteriori* LLRs $L(u_k \mid \underline{y})$ output from the component decoders in an iterative decoder vary with the number of iterations used. The output from the second component decoder is shown after one, two, four, and eight iterations. The input sequence consisted entirely of -1 values, hence negative *a-posteriori* LLR $L(u_k \mid \underline{y})$ values correspond to a correct hard decision, and positive values to an incorrect hard decision. The encoded bits were transmitted over an AWGN channel at a channel SNR of -1 dB, and then decoded using an iterative turbo decoder using the MAP algorithm. It can be seen that as the number of iterations used increases, the number of positive *a-posteriori* LLR $L(u_k \mid \underline{y})$ values, and hence the BER, decreases until after eight iterations there are no incorrectly decoded values. Furthermore, as the number of iterations increases, the decoders become more certain about the value of the bits and hence the magnitudes of the LLRs gradually become larger. The erroneous decisions in the figure appear in bursts, since deviating from the error-free path trellis path typically inflicts several bit errors.

When the series of iterations finishes the output from the turbo decoder is given by the de-interleaved *a-posteriori* LLRs of the second component decoder, $L_{i2}(u_k \mid \underline{y})$, where i is the number of iterations used. The sign of these *a-posteriori* LLRs gives the hard decision output, i.e., whether the decoder believes that the transmitted data bit u_k was $+1$ or -1, and in some applications the magnitude of these LLRs, which gives the confidence the decoder has in its decision, may also be useful.

Ideally, for the iterative decoding of turbo codes, the *a-priori* information used by a component decoder in the context of bit u_k should be based on a "conduit of information" independent from the channel outputs used by that decoder. More explicitly, for example, an original systematic information bit and the parity-related information smeared across a number of bits by the encoder due to the code constraints imposed are affected by the channel differently. Hence, even if the original systematic information bit was badly corrupted by the channel, the surrounding parity information may assist the decoder in obtaining a high-confidence estimate concerning the channel-impaired original systematic bit. However, in turbo decoders the extrinsic LLR $L_e(u_k)$ for the bit u_k, as explained above, uses all the available received parity bits and all the received systematic bits except the received value y_{ks} associated with the bit u_k. The same received systematic bits are also used by the other component decoder, which uses the interleaved or de-interleaved version of $L_e(u_k)$ as its *a-priori* LLRs. Hence, the *a-priori* LLRs $L(u_k)$ are not truly independent from the channel outputs $\underline{y}$ used by the component decoders. However, due to the fact that the component convolutional codes have a short memory, usually of only 4 b or less, the extrinsic LLR $L_e(u_k)$ is only significantly affected by the received systematic bits relatively close

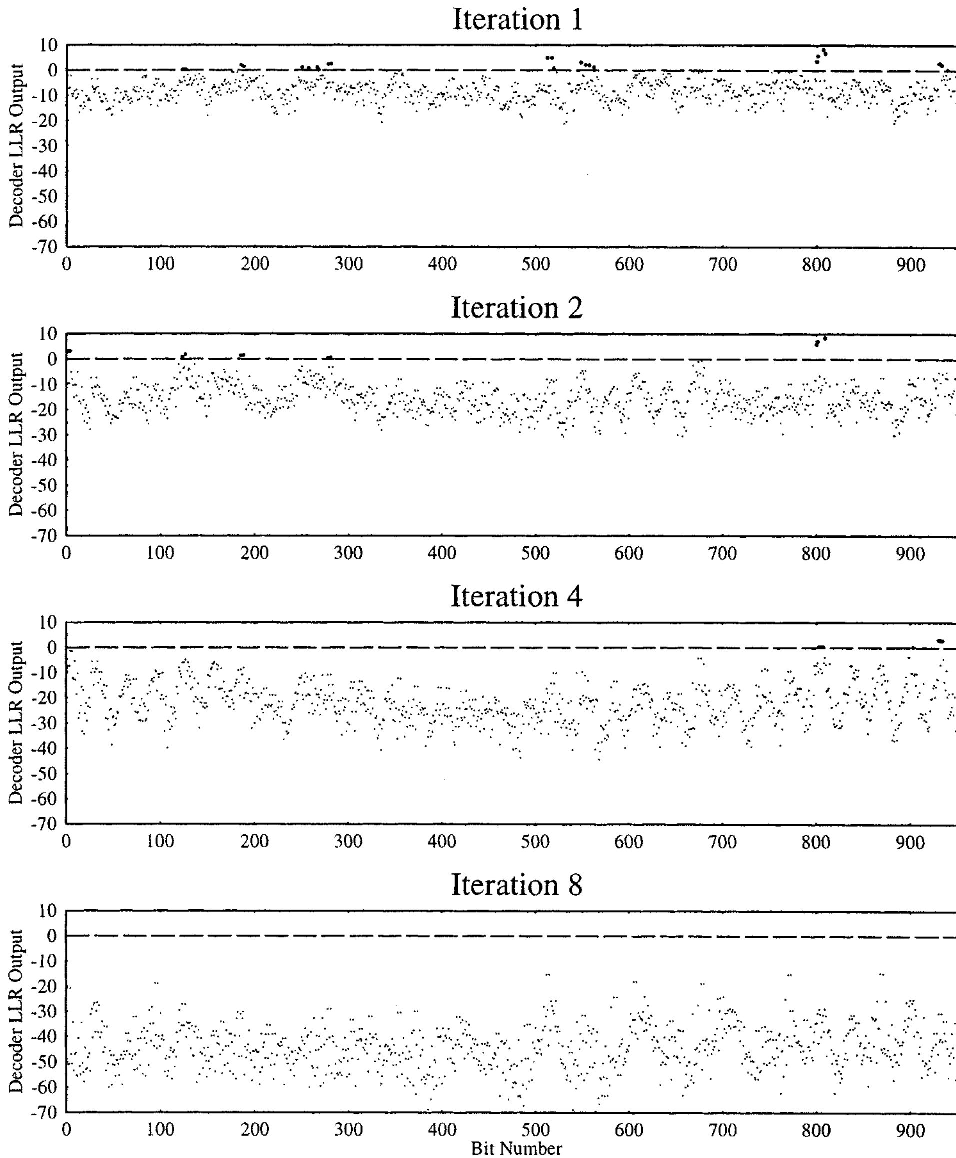

Fig. 5. Soft outputs from the MAP decoder in an iterative turbo decoder for a transmitted stream of all -1.

to the bit u_k. When this extrinsic LLR $L_e(u_k)$ is used as the *a-priori* LLR $L(u_k)$ by the other component decoder, because of the interleaving used, the bit u_k and its neighbors will probably have been well separated. Henc, the dependence of the *a-priori* LLRs $L(u_k)$ on the received systematic channel values $L_c y_{ks}$ which are also used by the other component decoder will have relatively little effect, and the iterative decoding provides good results.

Another justification for using the iterative arrangement described above is how well it has been found to work. In the limited experiments that have been carried out with optimal decoding of turbo codes [21]–[23] it has been found that optimal decoding performs only a fraction of a decibel (around 0.35–0.5 dB) better than iterative decoding with the MAP algorithm. Furthermore, various turbo coding schemes have been found [23], [24], that approach the Shannonian limit, which gives the best performance theoretically available, to a similar fraction of a decibel. Therefore, it seems that, for a variety of codes, the iterative decoding of turbo codes gives an almost optimal performance. Hence, it is this iterative decoding structure, which is almost exclusively used with turbo codes.

Having described how the MAP algorithm can be used in the iterative decoding of turbo codes, we now proceed to describe other soft-in soft-out decoders, which are less complex and can be used instead of the MAP algorithm. In the forthcoming section, we first describe two related algorithms, the Max-Log-MAP [25], [26] and the Log-MAP [7], which are derived from the MAP algorithm, and then another, referred to as the SOVA [8], [9], [19], derived from the Viterbi algorithm.

V. Modifications of the MAP Algorithm

A. Introduction

The MAP algorithm as described in Section III is much more complex than the Viterbi algorithm and with hard decision outputs performs almost identically to it. Therefore, for almost 20 years it was largely ignored. However, its application in turbo codes renewed interest in the algorithm, and it was realized that

its complexity can be dramatically reduced without affecting its performance. Initially the Max-Log-MAP algorithm was proposed by Koch and Baier [25] and Erfanian *et al.* [26]. This technique simplified the MAP algorithm by transferring the recursions into the log domain and invoking an approximation to dramatically reduce the complexity. Because of this approximation its performance is sub-optimal compared to that of the MAP algorithm. However, Robertson *et al.* [7] in 1995 proposed the Log-MAP algorithm, which corrected the approximation used in the Max-Log-MAP algorithm and hence gave a performance identical to that of the MAP algorithm, but at a fraction of its complexity. These two algorithms are described in this section.

B. Mathematical Description of the Max-Log-MAP Algorithm

The MAP algorithm calculates the *a-posteriori* LLRs $L(u_k \mid \underline{y})$ using (9). To do this requires the following values:

1) The $\alpha_{k-1}(\grave{s})$ values, which are calculated in a forward recursive manner using (11);
2) the $\beta_k(s)$ values, which are calculated in a backward recursion using (13),; and
3) the branch transition probabilities $\gamma_k(\grave{s}, s)$, which are calculated using (14).

The Max-Log-MAP algorithm simplifies this by transferring these equations into the log arithmetic domain and then using the approximation

$$\ln\left(\sum_i e^{x_i}\right) \approx \max_i(x_i) \tag{20}$$

where $\max_i(x_i)$ means the maximum value of x_i. Then, with $A_k(s)$, $B_k(s)$ and $\Gamma_k(\grave{s}, s)$ defined as follows:

$$A_k(s) \stackrel{\triangle}{=} \ln(\alpha_k(s)) \tag{21}$$

$$B_k(s) \stackrel{\triangle}{=} \ln(\beta_k(s)) \tag{22}$$

and

$$\Gamma_k(\grave{s}, s) \stackrel{\triangle}{=} \ln(\gamma_k(\grave{s}, s)) \tag{23}$$

we can rewrite (11) as

$$\begin{aligned} A_k(s) &\stackrel{\triangle}{=} \ln(\alpha_k(s)) \\ &= \ln\left(\sum_{\text{all } \grave{s}} \alpha_{k-1}(\grave{s})\gamma_k(\grave{s}, s)\right) \\ &= \ln\left(\sum_{\text{all } \grave{s}} \exp[A_{k-1}(\grave{s}) + \Gamma_k(\grave{s}, s)]\right) \\ &\approx \max_{\grave{s}}(A_{k-1}(\grave{s}) + \Gamma_k(\grave{s}, s)). \end{aligned} \tag{24}$$

Equation (24) implies that for each path in Fig. 2 from the previous stage in the trellis to the state $S_k = s$ at the present stage, the algorithm adds a branch metric term $\Gamma_k(\grave{s}, s)$ to the previous value $A_{k-1}(\grave{s})$ to find a new value $\tilde{A}_k(s)$ for that path. The new value of $A_k(s)$ according to (24) is then the maximum of the $\tilde{A}_k(s)$ values of the various paths reaching the state $S_k = s$. This can be thought of as selecting one path as the "survivor" and discarding any other paths reaching the state. The value of $A_k(s)$ should give the natural logarithm of the probability that the trellis is in state $S_k = s$ at stage k, given that the received channel sequence up to this point has been $\underline{y}_{j<k}$. However, because of the approximation of (20) used to derive (24), only the Maximum Likelihood (ML) path through the state $S_k = s$ is considered when calculating this probability. Thus, the value of A_k in the Max-Log-MAP algorithm actually gives the probability of the most likely path through the trellis to the state $S_k = s$, rather than the probability of *any* path through the trellis to state $S_k = s$. This approximation is one of the reasons for the sub-optimal performance of the Max-Log-MAP algorithm compared to the MAP algorithm.

We see from (24) that in the Max-Log-MAP algorithm the forward recursion used to calculate $A_k(s)$ is exactly the same as the forward recursion in the Viterbi algorithm—for each pair of merging paths the survivor is found using two additions and one comparison. Notice that for binary trellises the summation, and maximization, over all previous states $S_{k-1} = \grave{s}$ in (24) will in fact be over only two states, because there will be only two previous states $S_{k-1} = \grave{s}$ with paths to the present state $S_k = s$. For all other values of $\grave{s}$ we will have $\gamma_k(\grave{s}, s) = 0$.

Similarly to (24) for the forward recursion used to calculate the $A_k(s)$, we can rewrite (13) as

$$\begin{aligned} B_{k-1}(\grave{s}) &\stackrel{\triangle}{=} \ln(\beta_{k-1}(\grave{s})) \\ &\approx \max_s(B_k(s) + \Gamma_k(\grave{s}, s)) \end{aligned} \tag{25}$$

giving the backward recursion used to calculate the $B_{k-1}(\grave{s})$ values. Again, this is equivalent to the recursion used in the Viterbi algorithm except it proceed backward rather than forwaards through the trellis.

Using (14) and (15), we can write the branch metrics $\Gamma_k(\grave{s}, s)$ in the above recursive equations for $A_k(s)$ and $B_{k-1}(\grave{s})$ as

$$\begin{aligned} \Gamma_k(\grave{s}, s) &\stackrel{\triangle}{=} \ln(\gamma_k(\grave{s}, s)) \\ &= C + \frac{1}{2}u_k L(u_k) + \frac{L_c}{2}\sum_{l=1}^{n} y_{kl}x_{kl} \end{aligned} \tag{26}$$

where C does not depend on u_k or on the transmitted codeword $\underline{x}_k$ and so can be considered a constant and omitted. Hence, the branch metric is equivalent to that used in the Viterbi algorithm, with the addition of the *a-priori* LLR term $u_k L(u_k)$. Furthermore, the correlation term $\sum_{l=1}^{n} y_{kl}x_{kl}$ is weighted by the channel reliability value L_c of (18).

Finally, from (9), we can write for the *a-posteriori* LLRs $L(u_k \mid \underline{y})$ which the Max-Log-MAP algorithm calculates

$$\begin{aligned} L(u_k \mid \underline{y}) &= \ln\left(\frac{\sum_{\substack{(\grave{s},s)\Rightarrow \\ u_k=+1}} \alpha_{k-1}(\grave{s}) \cdot \gamma_k(\grave{s}, s) \cdot \beta_k(s)}{\sum_{\substack{(\grave{s},s)\Rightarrow \\ u_k=-1}} \alpha_{k-1}(\grave{s}) \cdot \gamma_k(\grave{s}, s) \cdot \beta_k(s)}\right) \\ &\approx \max_{\substack{(\grave{s},s)\Rightarrow \\ u_k=+1}} (A_{k-1}(\grave{s}) + \Gamma_k(\grave{s}, s) + B_k(s)) \\ &\quad - \max_{\substack{(\grave{s},s)\Rightarrow \\ u_k=-1}} (A_{k-1}(\grave{s}) + \Gamma_k(\grave{s}, s) + B_k(s)). \end{aligned} \tag{27}$$

This means that in the Max-Log-MAP algorithm for each bit u_k the *a-posteriori* LLR $L(u_k \mid \underline{y})$ is calculated by considering every transition from the trellis stage S_{k-1} to the stage S_k. These transitions are grouped into those that might have occured if $u_k = +1$, and those that might have occured if $u_k = -1$. For

both of these groups the transition giving the maximum value of $A_{k-1}(\grave{s}) + \Gamma(\grave{s}, s) + B_k(s)$ is found, and the *a-posteriori* LLR is calculated based on only these two "best" transitions.

The Max-Log-MAP algorithm can be summarized as follows. Forward and backward recursions, both similar to the forward recursion used in the Viterbi algorithm, are invoked to calculate $A_k(s)$ using (24) and $B_k(s)$ employing (25). The branch metric $\Gamma_k(\grave{s}, s)$ is given by (26), where the constant term C can be omitted. Once both the forward and backward recursions have been carried out, the *a-posteriori* LLRs can be calculated using (27). Thus the complexity of the Max-Log-MAP algorithm is not significantly higher than that of the Viterbi algorithm—instead of one recursion two are carried out, the branch metric of (26) has the additional *a-priori* term $u_k L(u_k)$ term added to it, and for each bit (27) must be used to give the *a-posteriori* LLRs. Viterbi states [27] that the complexity of the Log-MAP-Max algorithm is no greater than three times that of a Viterbi decoder. Unfortunately the storage requirements are much greater due to the need to store both the forward and backward recursively calculated metrics $A_k(s)$ and $B_k(s)$ before the $L(u_k \mid \underline{y})$ values can be calculated. However, Viterbi also states [27], [28] that by increasing the computational load slightly, to four times that of the Viterbi algorithm, the memory requirements can be dramatically reduced to become essentially equal to those of the Viterbi decoder.

C. Correcting the Approximation—The Log-MAP Algorithm

The Max-Log-MAP algorithm gives a slight degradation in performance compared to the MAP algorithm due to the approximation of (20). When used for the iterative decoding of turbo codes, Robertson *et al.* [7] found this degradation to result in a drop in performance of about 0.35 dB. However, the approximation of (20) can be made exact by using the Jacobian logarithm

$$\begin{aligned} \ln(e^{x_1} + e^{x_2}) &= \max(x_1, x_2) + \ln(1 + e^{-|x_1 - x_2|}) \\ &= \max(x_1, x_2) + f_c(|x_1 - x_2|) \\ &= g(x_1, x_2) \end{aligned} \tag{28}$$

where $f_c(x)$ can be thought of as a correction term. This is then the basis of the Log-MAP algorithm proposed by Robertson *et al.* [7]. Similarly to the Max-Log-MAP algorithm, values for $A_k(s) \triangleq \ln(\alpha_k(s))$ and $B_k(s) \triangleq \ln(\beta_k(s))$ are calculated using a forward and a backward recursion. However, the maximization in (24) and (25) is complemented by the correction term in (28). This means that the exact rather than approximate values of $A_k(s)$ and $B_k(s)$ are calculated. The correction term $f_c(\delta)$ need not be computed for every value of δ, but instead can be stored in a look-up table. Robertson *et al.* [7] found that such a look-up table need contain only eight values for δ, ranging between 0 and 5. This means that the Log-MAP algorithm is only slightly more complex than the Max-Log-MAP algorithm, but it gives exactly the same performance as the MAP algorithm. Therefore, it is a very attractive algorithm to use in the component decoders of an iterative turbo decoder.

Having described two techniques based on the MAP algorithm, which exhibited reduced complexity, in the next section we highlight the principles of an alternative soft-in soft-out decoder based on the Viterbi algorithm.

VI. The SOVA Algorithm

A. Mathematical Description of the SOVA Algorithm

In this section, we describe a variation of the Viterbi algorithm, referred to as the SOVA [9], [19]. This algorithm has two modifications over the classical Viterbi algorithm which allow it to be used as a component decoder for turbo codes. Firstly the path metrics used are modified to take account of *a-priori* information when selecting the ML path through the trellis. Secondly, the algorithm is modified so that it provides a soft output in the form of the *a-posteriori* LLR $L(u_k \mid \underline{y})$ for each decoded bit.

The first modification is easily accomplished. Consider the state sequence $\underline{s}_k^s$ which gives the states along the surviving path at state $S_k = s$ at stage k in the trellis. The probability that this is the correct path through the trellis is given by

$$p\left(\underline{s}_k^s \mid \underline{y}_{j \le k}\right) = \frac{p\left(\underline{s}_k^s \wedge \underline{y}_{j \le k}\right)}{p(\underline{y}_{j \le k})}. \tag{29}$$

As the probability of the received sequence $\underline{y}_{j \le k}$ for transitions up to and including the kth transition is constant for all paths $\underline{s}_k$ through the trellis to stage k, the probability that the path $\underline{s}_k^s$ is the correct one is proportional to $p(\underline{s}_k^s \wedge \underline{y}_{j \le k})$. Therefore, our metric should be defined so that maximizing the metric will maximize $p(\underline{s}_k^s \wedge \underline{y}_{j \le k})$. The metric should also be easily computable in a recursive manner as we go from the $(k-1)$th stage in the trellis to the kth stage. If the path $\underline{s}_k^s$ at the kth stage has the path $\underline{s}_{k-1}^{\grave{s}}$ for its first $k-1$ transitions then, assuming a memoryless channel and using the definition of $\gamma_k(\grave{s}, s)$ from (8), we will have

$$\begin{aligned} p\left(\underline{s}_k^s \wedge \underline{y}_{j \le k}\right) &= p\left(\underline{s}_{k-1}^{\grave{s}} \wedge \underline{y}_{j \le k-1}\right) \cdot p(s \wedge \underline{y}_k \mid \grave{s}) \\ &= p\left(\underline{s}_{k-1}^{\grave{s}} \wedge \underline{y}_{j \le k-1}\right) \cdot \gamma_k(\grave{s}, s). \end{aligned} \tag{30}$$

A suitable metric for the path $\underline{s}_k^s$ is therefore $M(\underline{s}_k^s)$, where

$$\begin{aligned} M\left(\underline{s}_k^s\right) &\triangleq \ln\Big(p\left(\underline{s}_k^s \wedge \underline{y}_{j \le k}\right)\Big) \\ &= M\left(\underline{s}_{k-1}^{\grave{s}}\right) + \ln(\gamma_k(\grave{s}, s)). \end{aligned} \tag{31}$$

Using (26) and omitting the constant term C we then have

$$M\left(\underline{s}_k^s\right) = M\left(\underline{s}_{k-1}^{\grave{s}}\right) + \frac{1}{2} u_k L(u_k) + \frac{L_c}{2} \sum_{l=1}^{n} y_{kl} x_{kl}. \tag{32}$$

Hence, our metric in the SOVA algorithm is updated as in the Viterbi algorithm, with the additional $u_k L(u_k)$ term included so that the *a-priori* information available is taken into account. Notice that this is equivalent to the forward recursion in (24) used to calculate $A_k(s)$ in the Max-Log-MAP algorithm.

Let us now discuss the second modification of the algorithm required, i.e., to give soft outputs. In a binary trellis there will be two paths reaching state $S_k = s$ at stage k in the trellis. The modified Viterbi algorithm, which takes account of the *a-priori* information $u_k L(u_k)$, calculates the metric from (32) for both merging paths, and discards the path with the lower metric. If the two paths $\underline{s}_k^s$ and $\underline{\hat{s}}_k^s$ reaching state $S_k = s$ have metrics $M(\underline{s}_k^s)$ and $M(\underline{\hat{s}}_k^s)$, and the path $\underline{s}_k^s$ is selected as the survivor because

its metric is higher, then we can define the metric difference Δ_k^s as

$$\Delta_k^s = M\left(\underline{s}_k^s\right) - M\left(\hat{\underline{s}}_k^s\right) \geq 0. \tag{33}$$

The probability that we have made the correct decision when we selected path $\underline{s}_k^s$ as the survivor and discarded path $\hat{\underline{s}}_k^s$, is then

$$P(\text{correct decision at } S_k = s) = \frac{P\left(\underline{s}_k^s\right)}{P\left(\underline{s}_k^s\right) + P\left(\hat{\underline{s}}_k^s\right)}. \tag{34}$$

Upon taking into account our metric definition in (31) we have

$$P(\text{correct decision at } S_k = s) = \frac{e^{M(\underline{s}_k^s)}}{e^{M(\underline{s}_k^s)} + e^{M(\hat{\underline{s}}_k^s)}} = \frac{e^{\Delta_k^s}}{1 + e^{\Delta_k^s}} \tag{35}$$

and the LLR that this is the correct decision is simply given by Δ_k^s.

Fig. 6 shows a simplified section of the trellis of the $K = 3$ RSC code, with the metric differences Δ_k^s marked at various points in the trellis.

When we reach the end of the trellis and have identified the ML path through the trellis, we need to find the LLRs giving the reliability of the bit decisions along the ML path. Observations of the Viterbi algorithm have shown that all the surviving paths at a stage l in the trellis will normally have come from the same path at some point before l in the trellis. This point is taken to be at most δ transitions before l, where usually δ is set to be five times the constraint length of the convolutional code. Therefore, the value of the bit u_k associated with the transition from state $S_{k-1} = \grave{s}$ to state $S_k = s$ on the ML path may have been different if, instead of the ML path, the Viterbi algorithm had selected one of the paths which merged with the ML path up to δ transitions later, i.e., up to the trellis stage $k + \sigma$. By the arguments above if the algorithm had selected any of the paths which merged with the ML path after this point the value of u_k would not be affected, because such paths will have diverged from the ML path after the transition from $S_{k-1} = \grave{s}$ to $S_k = s$. Thus, when calculating the LLR of the bit u_k, the soft output Viterbi algorithm (SOVA) must take account of the probability that the paths merging with the ML path from stage k to stage $k + \delta$ in the trellis were incorrectly discarded. This is done by considering the values of the metric difference $\Delta_i^{s_i}$ for all states s_i along the ML path from trellis stage $i = k$ to $i = k + \delta$. It is shown by Hagenauer in [29] that this LLR can be approximated by

$$L(u_k \,|\, \underline{y}) \approx u_k \min_{\substack{i=k \ldots k+\delta \\ u_k \neq u_k^i}} \Delta_i^{s_i} \tag{36}$$

where u_k is the value of the bit given by the ML path, and u_k^i is the value of this bit for the path which merged with the ML path and was discarded at trellis stage i. Thus the minimization in (36) is carried out only for those paths merging with the ML path which would have given a different value for the bit u_k if they had been selected as the survivor. The paths which merge with the ML path, but would have given the same value for u_k as the ML path, obviously do not affect the reliability of the decision of u_k.

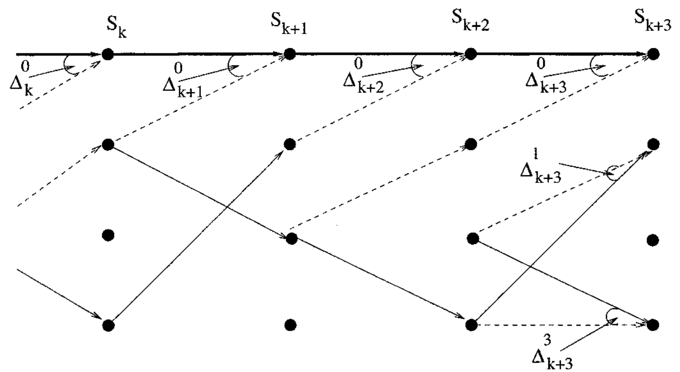

Fig. 6. Simplified section of the trellis for our $K = 3$ RSC code with SOVA decoding.

For clarification of these operations refer again to Fig. 6 showing a simplified section of the trellis for the $K = 3$ RSC code. In this figure, as before, solid lines represent transitions taken when the input bit is a -1, and dashed lines represent transitions taken when the input bit is a $+1$. We assume that the all-zero path is identified as the ML path, and this path is shown as a bold line. Also shown are the paths which merge with this ML path. It can be seen from the figure that the ML path gives a value of -1 for u_k, but the paths merging with the ML path at trellis stages S_k, S_{k+1}, and S_{k+3} all give a value of $+1$ for the bit u_k. Hence, if we assume for simplicity that $\sigma = 3$, from (36) the LLR $L(u_k \,|\, \underline{y})$ will be given by -1 multiplied by the minimum of the metric differences Δ_k^0, Δ_{k+1}^0, and Δ_{k+3}^0.

B. Implementation of the SOVA Algorithm

The SOVA algorithm is implemented as follows. For each state at each stage in the trellis the metric $M(\underline{s}_k^s)$ is calculated for both of the two paths merging into the state using (32). The path with the highest metric is selected as the survivor, and for this state at this stage in the trellis a pointer to the previous state along the surviving path is stored, just as in the classical Viterbi algorithm. However, in order to allow the reliability of the decoded bits to be calculated, the information used in (36) to give $L(u_k \,|\, \underline{y})$ is also stored. Thus the difference Δ_k^s between the metrics of the surviving and the discarded paths is stored, together with a binary vector containing $\delta + 1$ bits, which indicate whether or not the discarded path would have given the same series of bits u_l for $l = k$ back to $l = k - \delta$ as the surviving path does. This series of bits is called the update sequence in [29], and as noted by Hagenauer it is given by the result of a modulo two addition (i.e., an exclusive-or operation) between the previous $\delta + 1$ decoded bits along the surviving and discarded paths. When the SOVA has identified the ML path, the stored update sequences and metric differences along this path are used in (36) to calculate the values of $L(u_k \,|\, \underline{y})$.

The SOVA algorithm described in this section is the least complex of all the soft-in soft-out decoders discussed in this chapter. In [7] it is shown by Robertson *et al.* that the SOVA algorithm is about half as complex as the Max-Log-MAP algorithm. However, the SOVA algorithm is also the least accurate of the algorithms we have described in this chapter and, when used in an iterative turbo decoder, performs about 0.6 dB worse [7] than a decoder using the MAP algorithm.

Let us now augment our understanding of iterative turbo decoding by considering a specific example in the next section.

VII. Turbo Decoding Example

In this section, we discuss an example of turbo decoding using the SOVA algorithm [9], [19] detailed in Section VI. This example serves to illustrate the details of the SOVA algorithm and the iterative decoding of turbo codes discussed in Section IV.

We consider a simple half-rate turbo code using the $K = 3$ RSC code. The reason for using RSC codes instead of conventional nonsystematic, nonrecursive codes is two-fold, which we attempt to make plausible at this stage. Firstly, it would be rather wasteful in terms of both transmitted signal energy and bit rate to transmit the information and the parity bits of both component encoders twice. This would namely erode the performance benefits of turbo coding. If, however, a systematic component encoder is used, it is straightforward to puncture or obliterate one of the original systematic information bits from the transmitted bitstream. Furthermore, systematic codes impose less constraints on the encoded bitstream, than their nonsystematic counterparts, since in systematic encoders the original information bits are directly copied to the encoder's output. Hence, the systematic codes exhibit a slightly better BER performance, than the nonsystematic codes, since the latter codes are overwhelmed by the plethora of channel errors and hence precipitate more errors upon attempting to correct errors, when the channel BER is high. Since turbo codes are of most interest at high-channel BERs, systematic codes are preferred.

Secondly, the importance of the recursive nature of the RSC encoder can be made plausible as follows. For a nonrecursive convolutional code the trellis path corresponding to an input sequence of $\mathbf{u}_1 = (\ldots; +1; -1; +1; +1; +1; \ldots)$ containing a single -1 emerges from and merges back into the all-zero trellis path within a finite number of trellis transitions, depending on the minimum distance of the code. For recursive codes however, the input sequence $\mathbf{u}_1$ would result in a -1 eternally cycling through the encoder's shift register stages, such that the corresponding trellis path never remerges into the all-zero path. This would result in an output sequence containing an infinite number of "-1"s. Since their associated output is quite different, the closest neighbor transmitted sequences of $\mathbf{u}_0 = (\ldots; +1; +1; +1; +1; +1; \ldots)$ (the all-"+1" dataword) and $\mathbf{u}_1$ above would rarely be confused with each other in the decoder in the case of recursive component codes. The path corresponding to $\mathbf{u}_1$ is hence a very unlikely deviation from the all-zero path during the decoding process in the case of a recursive code, whereas it is the most likely deviation for a nonrecursive code.

Following the above brief justification for using RSC codes the generator polynomials are expressed in octal form as 7 and 5, as shown in [18, Fig. 6]. Two such codes are combined, as shown in [18, Fig. 7], with a 3×3 block interleaver to give a simple turbo code. The parity bits from both the component codes are punctured, so that alternate parity bits from the first and the second component encoder are transmitted. Thus the first, third, fifth, seventh, and ninth parity bits from the first component encoder are transmitted, and the second, fourth, sixth, and eighth parity bits from the second component encoder are transmitted. The first component encoder is terminated using two bits chosen to take this encoder back to the all zero state. The transmitted sequence will therefore contain nine systematic and nine parity bits. Of the systematic bits, seven will be the input bits, and two will be the bits chosen to terminate the first trellis. Of the nine parity bits, five will come from the first encoder, and four from the second encoder.

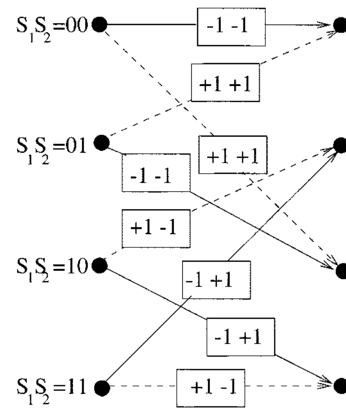

Fig. 7. State transition diagram for the $(2, 1, 3)$ RSC component codes.

The state transition diagram for the component RSC codes is shown in Fig. 7. As in all our diagrams in this section, a solid line denotes a transition resulting from a -1 input bit, and a dashed lines represents an input bit of $+1$. The figures within the boxes along the transition lines give the output bits associated with that transition—the first bit is the systematic bit, which is the same as the input bit, and the second is the parity bit.

For the sake of simplicity we assume that an all -1 input sequence is used. Thus there will be seven input bits which are -1, and the encoder trellis will remain in the $S_1S_2 = 00$ state. The two bits necessary to terminate the trellis will be -1 in this case and, as can be seen from Fig. 7, the resulting parity bits will also be -1. Thus, all 18 of the transmitted bits will be -1 for an all -1 input sequence. Assuming that BPSK modulation is used with the transmitted symbols being -1 or $+1$, the transmitted sequence will be a series of 18 -1's. The received channel output sequence for the example-together with the input and the parity bits detailed above are shown in Table V. Notice that approximately half the parity bits from each component encoder are puncturedthis is represented by a dash in Table V. Also note that the received channel sequence values shown in Table V are the matched filter outputs, which were denoted by y_{kl} in previous sections. If hard decision demodulation were used then negative values would be decoded as -1's, and positive values as $+1$'s. It can be seen that from the 18 coded bits which were transmitted, all of which were -1, three would be decoded as $+1$ if hard decision demodulation were used.

In order to illustrate the difference between iterative turbo decoding and the decoding of convolutional codes, we initially consider how the received sequence shown in Table V would be decoded by a convolutional decoder using the Viterbi algorithm. Imagine the half-rate $K = 3$ RSC code detailed above used as an ordinary convolutional code to encode an input sequence of seven -1's. If trellis termination was used then two

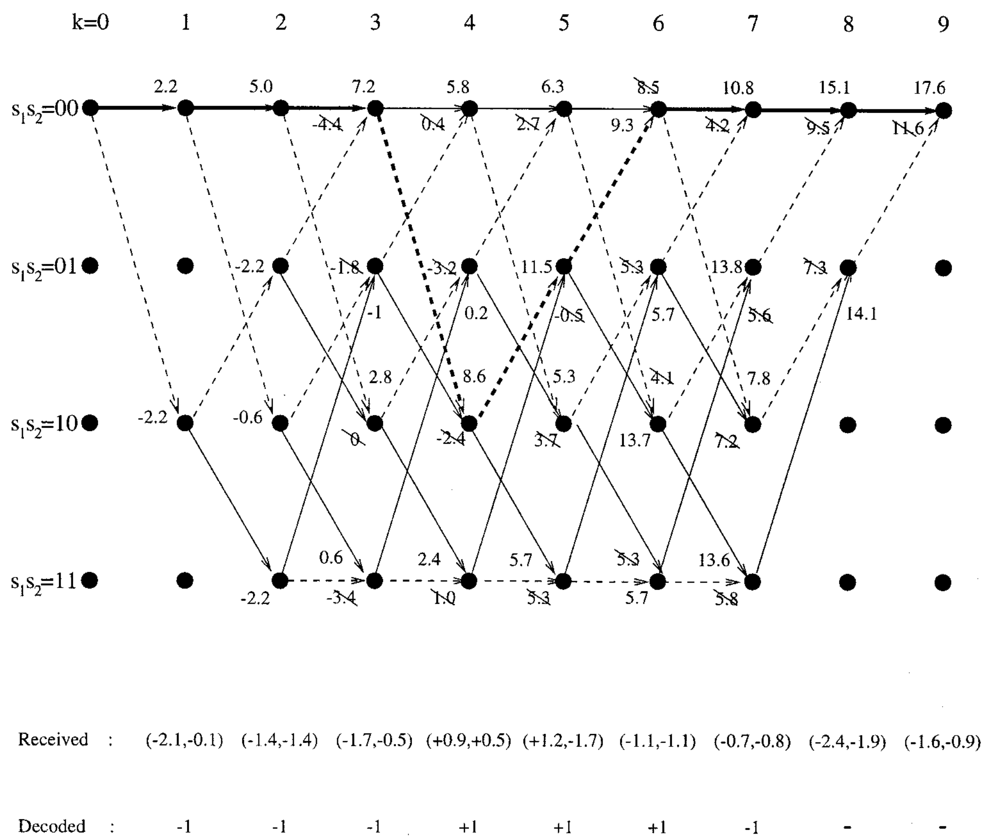

Fig. 8. Trellis diagram for the Viterbi decoding of the received sequence shown in Table V.

-1's would be employed to terminate the trellis, and the transmitted sequence would consist of 18 -1's, just as for our turbo coding example. If the received sequence was as shown in Table V, then the Viterbi algorithm decoding this sequence would have the trellis diagram shown in Fig. 8. The metrics shown in this figure are given by the cross correlation of the received and expected channel sequences for a given path, and the Viterbi algorithm maximizes this metric to find the ML path, which is shown by the bold line in Fig. 8. Notice that at each state in the trellis where two paths merge, the path with the lower metric is discarded and its metric is shown crossed out in the figure. As can be seen from Fig. 8, the Viterbi algorithm makes an incorrect decision at stage $k=6$ in the trellis and selects a path other than the all zero path as the survivor. This results in three of the seven bits being decoded incorrectly as $+1$'s.

Having seen how Viterbi decoding of a RSC convolutional code would fail and produce three errors given the received sequence, we now proceed to detail the operation of an iterative turbo decoder for the same channel sequence. Consider first the operation of the first component decoder in the first iteration. The component decoder uses the SOVA algorithm to decide upon not only the most likely input bits, but also the LLRs of these bits, as described in Section VI.

The metric for the SOVA algorithm is given by (32), which is repeated here for convenience

$$M\left(\underline{s}_k^s\right) = M\left(\underline{s}_{k-1}^{\hat{s}}\right) + \frac{1}{2}u_k L(u_k) + \frac{L_c}{2}\sum_{l=1}^{Q} y_{kl}x_{kl}. \quad (37)$$

As initially we are considering the operation of the first decoder in the first iteration there is no *a-priori* information and hence we have $L(u_k)=0$ for all k, which corresponds to an *a-priori* probability of 0.5. The received sequence given in Table V was derived from the transmitted channel sequence (which has $E_b = 1$) by adding AWGN with variance $\sigma = 1$. Hence, as the fading amplitude is $a=1$, from (18) we have for the channel reliability measure $L_c = 2$.

Fig. 9 shows the trellis for this first component decoder in the first iteration. Due to the puncturing of the parity bits used at the encoder, the second, fourth, sixth, and eighth parity bits have been received as zeros. The *a-priori* and channel values shown in Fig. 9 are given as $L(u_k)/2$ and $L_c y_{kl}/2$ so that the metric values, given by (37), can be calculated by simple addition and subtraction of the values shown. As we have $L(u_k)=0$ and $L_c = 2$, these metrics are again given by the cross correlation of the expected and received channel sequences. Notice however, that because of the puncturing used the metric values shown in Fig. 9 are not the same as those in Fig. 8. Despite this the ML path, shown by the bold line in Fig. 9, is the same as the one that was chosen by the Viterbi algorithm shown in Fig. 8, with three of the input bits being decoded as $+1$'s rather than -1's.

We now discuss how, having determined the ML path, the SOVA algorithm finds the LLRs for the decoded bits. Fig. 10 is a simplified version of the trellis from Fig. 9, which shows only the ML path and the paths that merge with this ML path and are discarded. Also shown are the metric differences, denoted by Δ_k^s in Section VI, between the ML and the discarded paths. These metric differences, together with the previously de-

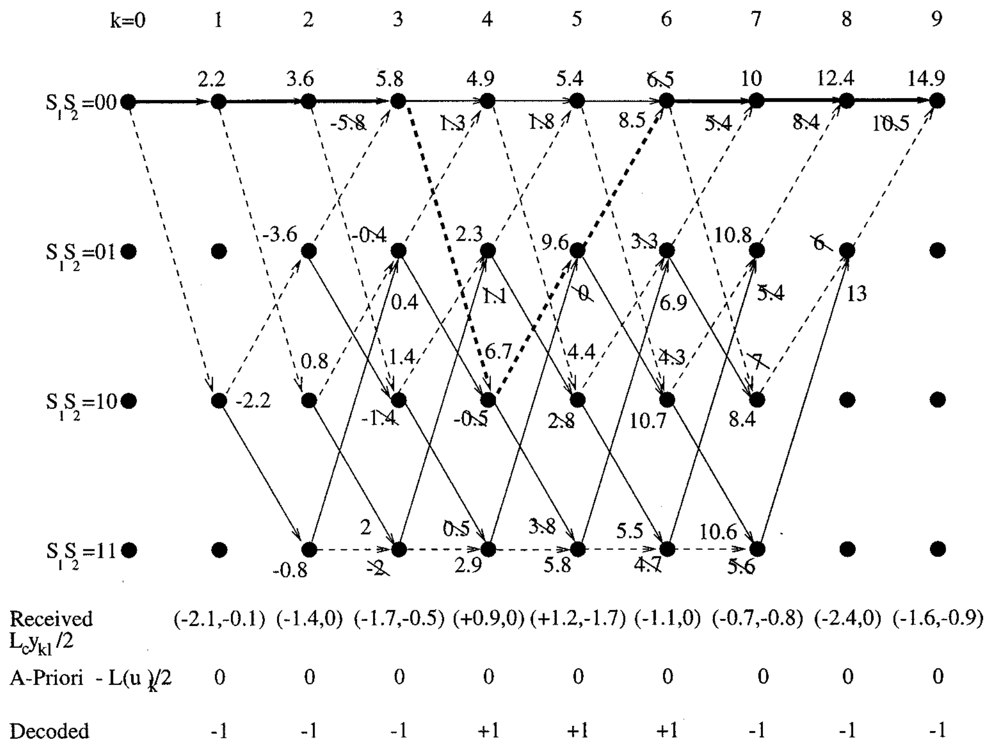

Fig. 9. Trellis diagram for the SOVA decoding in the first iteration of the first decoder.

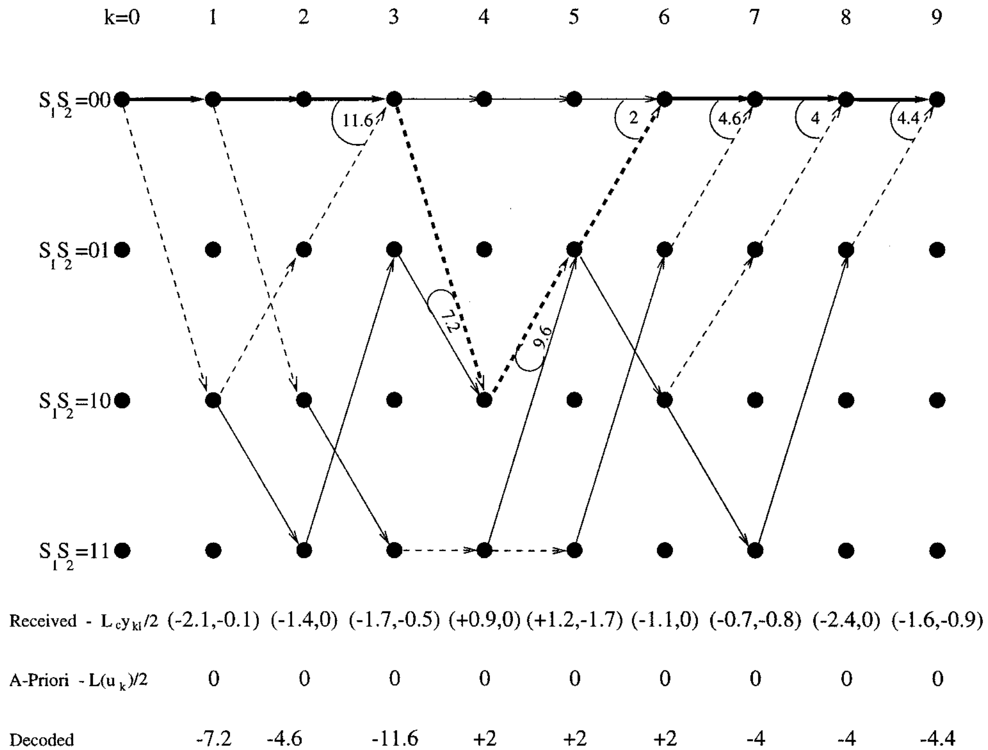

Fig. 10. Simplified trellis diagram for the SOVA decoding in the first iteration of the first decoder.

fined update sequences that indicate for which of the bits the survivor and discarded paths would have given different values, are stored by the SOVA algorithm for each node at each stage in the trellis. When the ML path has been identified, the algorithm uses these stored values along the ML path to find the LLR for each decoded bit. Table I shows these stored values for the example trellis shown in Figs. 9 and 10. The calculation of the decoded LLRs shown in this table is detailed below.

Notice in Table I that at trellis stages $k = 1$ and $k = 2$ there is no metric difference or update sequence stored because, as can

TABLE I
SOVA OUTPUT FOR THE FIRST ITERATION OF THE FIRST DECODER

Trellis Stage k	Decoded Bit u_k	Metric Diff Δ_k^s	Update Sequence	Decoded LLR
1	-1	–	–	-7.2
2	-1	–	–	-4.6
3	-1	11.6	111	-11.6
4	+1	7.2	1001	+2
5	+1	9.6	10010	+2
6	+1	2	111000	+2
7	-1	4.6	1100010	-4
8	-1	4	11100000	-4
9	-1	4.4	100100000	-4.4

be seen from Figs. 9 and 10, there are no paths merging with the ML path at these stages. For all subsequent stages there is a merging path, and values of the metric differences and update sequences are stored. For the update sequence a "1" indicates that the ML and the discarded merging path would have given different values for a particular bit. At stage k in the trellis we have taken the Most Significant Bit (MSB), on the left-hand side, to represent u_k, the next bit to represent u_{k-1}, etc. until the Least Significant Bit (LSB), which represents u_1. For the RSC code any two paths merging at trellis stage k give different values for the bit u_k, and so the MSB in the update sequences in Table I is always 1. Notice furthermore that although in the example the update sequences are all of different lengths, this is only because of the very short frame length we have used. More generally, as explained in Section VI, all the stored update sequences will be $\delta + 1$ bits long, where δ is usually set to be five times the constraint length of the convolutional code.

We now explain how the SOVA algorithm can use the stored update sequences and metric differences along the ML path to calculate the LLRs for the decoded bits. Equation (36) shows that the decoded *a-posteriori* LLR $L(u_k \,|\, \underline{y})$ for a bit u_k is given by the minimum metric difference of merging paths along the ML path. This minimum is taken only over the metric differences for stages $i = k, k+1, \ldots k+\delta$ where the value u_k^i of the bit u_k given by the path merging with the ML path at stage i is different from the value given for this bit by the ML path. Whether or not the condition $u_k = u_k^i$ is met is determined using the stored update sequences. Denoting the update sequence stored at stage l along the ML path as $\underline{e}_l$, for each bit u_k the SOVA algorithm examines the MSB of $\underline{e}_k$, the second MSB of $\underline{e}_{k+1}$, etc. up to the $(\delta+1)$th bit (which will be the LSB) of $\underline{e}_{k+\delta}$. For our example this examination of the update sequences is limited because of our short frame length, but the same principles are used. Taking the fourth bit u_4 as an example, to determine the decoded LLR $L(u_4 \,|\, \underline{y})$ for this bit the algorithm examines the MSB of $\underline{e}_4$ in row four of Table I, the second MSB of $\underline{e}_5$ in row five, etc. up to the sixth MSB of $\underline{e}_9$ in row nine. It can be seen, from the corresponding rows in Table I, that only the paths merging at stages $k = 4$ and $k = 6$ of the trellis give values different from the ML path for the bit u_4. Hence, the decoded LLR $L(u_4 \,|\, \underline{y})$ from the SOVA algorithm for this bit is calculated using (36) as the value of the bit given by the ML path $(+1)$ times the minimum of the metric differences stored at Stages 4 and 6 of the trellis (7.2 and 2), yeilding $L(u_4 \,|\, \underline{y}) = +2$.

The remaining decoded LLR values in Table I are computed following a similar procedure. However, it is worth noting explicitly that the low value (2) of the metric difference for the merging path at Stage 6 in the trellis, which is where the incorrect path is chosen as the survivor, gives the LLR for the bits where this path and the ML path give different values. Hence, the LLRs for the three incorrectly decoded bits, i.e., u_4, u_5 and u_6, have the lowest magnitudes of any of the decoded bits.

We now move on to describing the operation of the second component decoder in the first iteration. This decoder uses the extrinsic information from the first decoder as *a-priori* information to assist its operation, and therefore should be able to provide a better estimate of the encoded sequence than the first decoder was. Equation (16) from Section IV give the extrinisic information $L_e(u_k)$ from a component decoder as the soft output $L(u_k \,|\, \underline{y})$ from the decoder with the *a-priori* information $L(u_k)$ (if any was available) and the received systematic channel information $L_c y_{ks}$ subtracted. This is equivalent to [18, eq. (13)]. Table II shows the extrinsic information calculated from (16) from the first decoder, which is then interleaved by a 3×3 block interleaver and used as the *a-priori* information for the second component decoder. The second component decoder also uses the interleaved received systematic channel values, and the received parity bits from the second encoder which were not punctured (i.e., the second, fourth, sixth, and eighth bits).

Fig. 11 shows the trellis for the SOVA decoding of the second decoder in the first iteration. The extrinsic information values from Table II are shown after being interleavered and divided by two as $L(u_k)/2$. Also shown is the channel information $L_c y_{ks}/2$ used by this decoder. Notice that as the trellis is not terminated for the second component encoder, paths terminating in all four possible states of the trellis are considered at the decoder. However, the metric for the $S_1 S_2 = 00$ state is the maximum of the four final metrics, and hence this all zero state is used as the final state of the trellis.

The ML path chosen by the second component decoder is shown by a bold line in Fig. 11, together with the LLR values output by the decoder. These are calculated, using update sequences and minumum metric differences, in the same way as was explained for the first decoder using Fig. 10 and Table I. It can be seen that the decoder makes an incorrect decision at stage $k = 5$ in the trellis and selects a path other than the all zero path as the survivor. However, the incorrectly chosen path gives decoded bits of $+1$ for only two transitions, and hence only two, rather than three, decoding errors are made. Furthermore, the difference in the metrics between the correct and the chosen path at trellis stage $k = 5$ is only 2.2, and so the magnitude of the decoded LLRs $L(u_k \,|\, \underline{y})$ for the two incorrectly decoded bits, u_2 and u_5, is only 2.2. This is significantly lower than the magnitudes of the LLRs for the other bits, and indicates that the algorithm is less certain about these two bits being $+1$ than it is about the other bits being -1.

Having calculated the LLRs from the second component decoder, the turbo decoder has now completed one iteration. The soft output LLR values from the second component decoder shown in the bottom line of Fig. 11 could now be de-interleaved

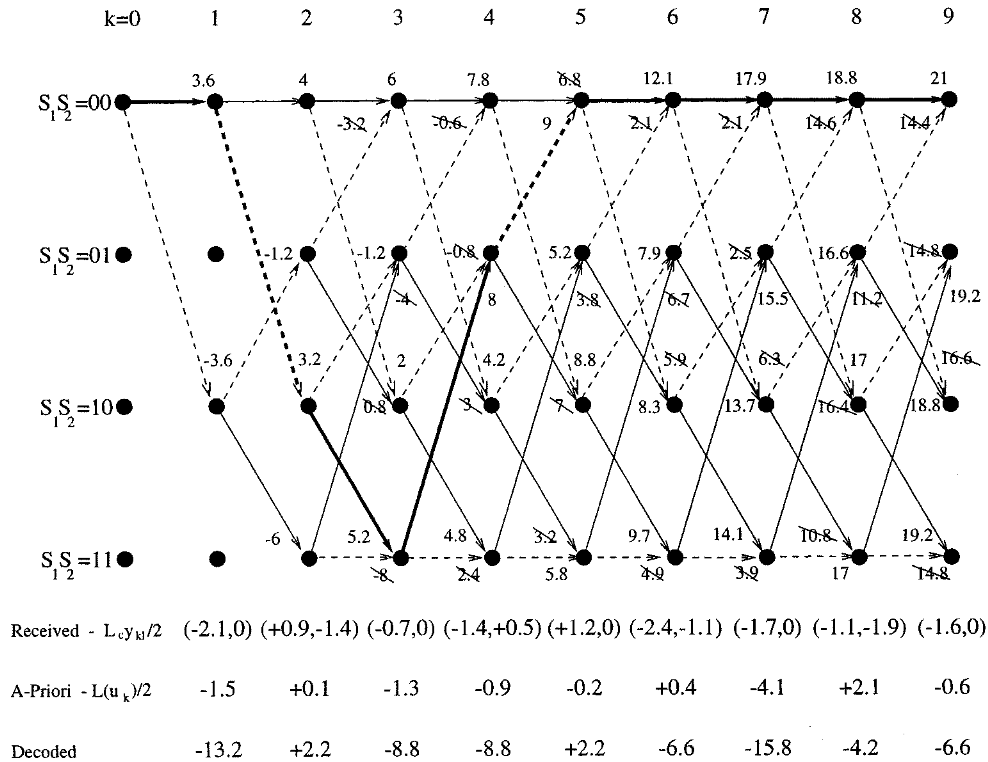

Fig. 11. Trellis diagram for the SOVA decoding in the first iteration of the second decoder.

TABLE II
CALCULATION OF THE EXTRINSIC INFORMATION FROM THE FIRST DECODER IN THE FIRST ITERATION

Trellis Stage k	Output $L(u_k\|\underline{y})$	A-Priori $L(u_k)$	Sys. Info. $L_c y_{ks}$	Extrin. Info.
1	-7.2	0	-4.2	-3
2	-4.6	0	-2.8	-1.8
3	-11.6	0	-3.4	-8.2
4	+2	0	+1.8	+0.2
5	+2	0	+2.4	-0.4
6	+2	0	-2.2	+4.2
7	-4	0	-1.4	-2.6
8	-4	0	-4.8	+0.8
9	-4.4	0	-3.2	-1.2

and used as the output from the turbo decoder. This de-interleaving would result in an output sequence which gave negative LLRs for all the decoded bits except u_4 and u_5, which would be incorrectly decoded as $+1$'s as their LLRs are both $+2.2$. Thus, even after only one iteration, the turbo decoder has decoded the received sequence with one less error than the convolutional decoder did. However, generally better results are achieved with more iterations, and so we now progress to describe the operation of the turbo decoder in the second iteration.

In the second, and all subsequent, iterations the first component decoder is able to use the extrinsic information from the second decoder in the previous iteration as *a-priori* information. Table III shows the calculation of this extrinsic information using (16) from the second decoder in the first iteration. It can be seen that it gives negative LLRs for all the bits except u_2 and u_5, and for these two bits the LLRs are close to zero. This extrinsic information is then de-interleaved and used as the *a-priori* information for the first decoder in the next (second) iteration. The trellis for this decoder is shown in Fig. 12. It can be seen that this decoder uses the same channel information as it did in the first iteration. However now, in contrast to Fig. 9, it also has *a-priori* information, to assist it in finding the correct path through the trellis. The selected ML path is again shown by a bold line, and it can be seen that now the correct all zero path is chosen. The second iteration is then completed by finding the extrinsic information from the first decoder, interleaving it and using it as *a-priori* information for the second decoder. It can be shown that this decoder will also now select the all zero path as the ML path, and hence the output from the turbo decoder after the seond iteration will be the correct all -1 sequence. This concludes our example of the operation of an iterative turbo decoder using the SOVA algorithm, leading on to a comparison of the component decoder algorithms.

VIII. COMPARISON OF THE COMPONENT DECODER ALGORITHMS

In this article, we have described in detail the iterative structure and the component decoders used to decode turbo codes. A numerical example illustrating this decoding was given in the previous section. We now conclude by summarizing the operation of the algorithms which can be used as component decoders, highlighting the similarities and differences between these algorithms, and noting their relative complexities and performances.

The MAP algorithm is the optimal component decoder for turbo codes. It finds the probability of each bit u_k being a $+1$ or -1 by calculating the probability for each transition from state $S_{k-1} = \grave{s}$ to $S_k = s$ that could occur if the input bit was

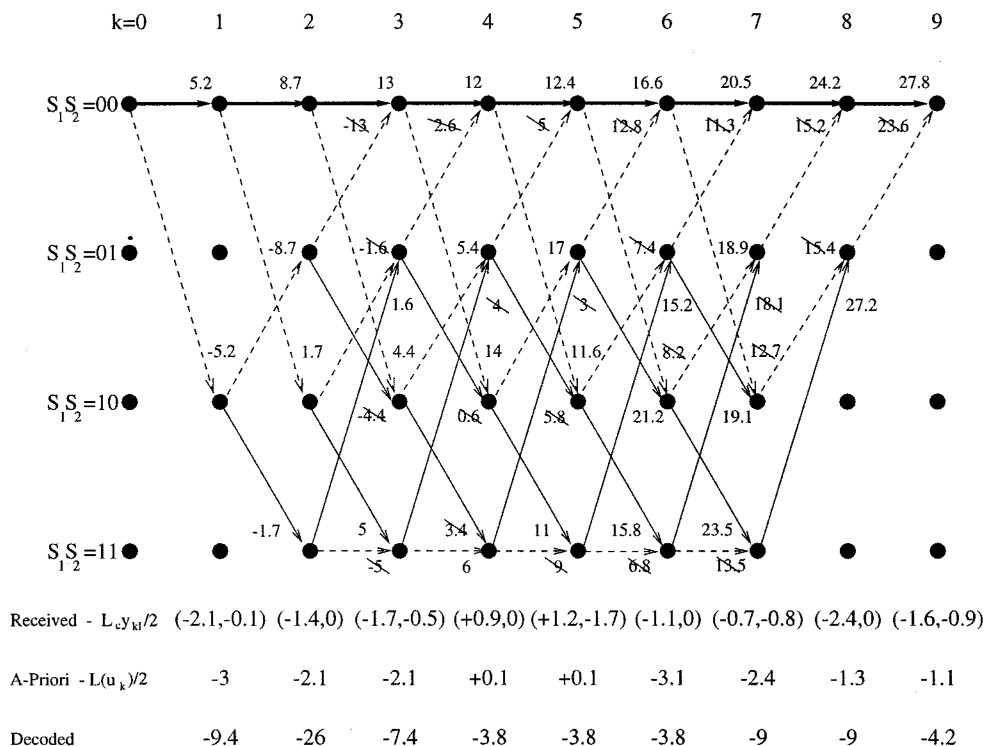

Fig. 12. Trellis diagram for the SOVA decoding in the second iteration of the first decoder.

TABLE III
CALCULATION OF THE EXTRINSIC INFORMATION FROM THE SECOND DECODER IN THE FIRST ITERATION

Trellis Stage k	Output $L(u_k\|\underline{y})$	A-Priori $L(u_k)$	Sys. Info. $L_c y_{ks}$	Extrin. Info.
1	-13.2	-3	-4.2	-6
2	+2.2	+0.2	+1.8	+0.2
3	-8.8	-2.6	-1.4	-4.8
4	-8.8	-1.8	-2.8	-4.2
5	+2.2	-0.4	+2.4	+0.2
6	-6.6	+0.8	-4.8	-2.6
7	-15.8	-8.2	-3.4	-4.2
8	-4.2	+4.2	-2.2	-6.2
9	-6.6	-1.2	-3.2	-2.2

$+1$, and similarly for every transition that could occur if the input bit was -1. As these transitions are mutually exclusive, the probability of any one of them occurring is simply the sum of their individual probabilities, and hence the LLR for a bit u_k is given by the ratio of two sums of probabilities, as in (3).

Due to the Markov nature of the trellis and the assumption that the output from the trellis is observed in memoryless noise, the individual probabilities of the transitions in (3) can be expressed as the product of three terms—$\alpha_{k-1}(\grave{s})$, $\beta_k s$ and $\gamma_k(\grave{s}, s)$, as in (5). The $\gamma_k(\grave{s}, s)$ terms can be calculated from the *a-priori* probabilities for the decoded bits, and the received channel information, as in (14) and (15). Then the $\alpha_k(s)$ and $\beta_k(s)$ terms can be calculated recursively as in . (11) and (13). The MAP algorithm is optimal for the decoding of turbo codes, but is extremely complex. Furthermore, because of the multiplications used in the recursive calculation of the $\alpha_{k-1}(\grave{s})$ and $\beta_k(s)$ terms it often suffers from numerical problems in practice. The Log-MAP algorithm is theoretically identical to the MAP algorithm, but transfers its operations to the log domain. Thus multiplications are replaced with additions, and so the numerical problems of the MAP algorithm are avoided and its complexity is dramatically reduced.

The Max-Log-MAP algorithm further reduces the complexity of the Log-MAP algorithm using the maximization approximation given in (20). This has two effects on the operation of the algorithm compared to that of the Log-MAP algorithm. Firstly, as can be seen by examining (27), it means that only two transitions are considered when finding the LLR $L(u_k\,|\,\underline{y})$ for each bit u_k—the best transition from $S_{k-1} = \grave{s}$ to $S_k = s$ that would give $u_k = +1$ and the best that would give $u_k = -1$. Similarly in the recursive calculations of the $A_k(s) = \ln(\alpha_k(s))$ and $B_k(s) = \ln(\beta_k(s))$ terms of (24) and (25) the approximation means that only one transition, the most likely one, is considered when calculating $A_k(s)$ from the $A_{k-1}(\grave{s})$ terms and $B_{k-1}(\grave{s})$ from the $B_k(s)$ terms. This means that although $A_{k-1}(\grave{s})$ should give the logarithm of the probability that the trellis reaches state $S_{k-1} = \grave{s}$ along *any* path from the initial state $S_0 = 0$, in fact it gives the logarithm of the probability of only the *most likely* path to state $S_{k-1} = \grave{s}$. Similarly $B_k(s) = \ln(\beta_k(s))$ should give the logarithm of the probability of the received sequence $\underline{y}_{j>k}$ given only that the trellis is in state $S_k = s$ at stage k. However, the maximization in (25) used in the recursive calculation of the $B_k(s)$ terms means that only the most likely path from state $S_k = s$ to the end of the trellis is considered, and not all paths.

Hence, the Max-Log-MAP algorithm finds the LLR $L(u_k\,|\,\underline{y})$ for a given bit u_k by comparing the probability of the most likely path giving $u_k = +1$ to the probability of the most likely path giving $u_k = -1$. For the next bit, u_{k+1}, again the best path

that would give $u_{k+1} = +1$ and the best path that would give $u_{k+1} = -1$ are compared. One of these "best paths" will always be the ML path, and so will not change from one stage to the next, whereas the other may change. In contrast the MAP and the Log-MAP algorithms consider every path in the calculation of the LLR for each bit. All that changes from one stage to the next is the division of paths into those that give $u_k = +1$ and those that give $u_k = -1$. Thus the Max-Log-MAP algorithm gives a degraded performance compared to the MAP and Log-MAP algorithms.

In the SOVA algorithm the ML path is found by maximizing the metric given in (32). The recursion used to find this metric is identical to that used to find the $A_k(s)$ terms in (24) in the Max-Log-MAP algorithm. Once the ML path has been found, the hard decision for a given bit u_k is determined by which transition the ML path took between trellis stages S_{k-1} and S_k. The LLR $L(u_k \mid \underline{y})$ for this bit is determined by examining the paths which merge with the ML path that would have given a different hard decision for the bit u_k. The LLR is taken to be the minimum metric difference for these merging paths which would have given a different hard decision for the bit u_k. Using the notation associated with the Max-Log-MAP algorithm, once a path merges with the ML path, it will have the same value of $B_k(s)$ as the ML path. Hence, as the metric in the SOVA is identical to the $A_k(s)$ values in the Max-Log-MAP, taking the difference between the metrics of the two merging paths in the SOVA algorithm is equivalent to taking the difference between two values of $(A_{k-1}(\grave{s}) + \Gamma_k(\grave{s}, s) + B_k(s))$ in the Max-Log-MAP algorithm, as in (27). The only difference is that in the Max-Log-MAP algorithm one path will be the ML path, and the other will be the most likely path that gives a different hard decision for u_k. In the SOVA algorithm again one path will be the ML path, but the other may not be the most likely path that gives a different hard decision for u_k. Instead, it will be the most likely path that gives a different hard decision for u_k and survives to merge with the ML path. Other, more likely paths, which give a different hard decision for the bit u_k to the ML path may have been discarded before they merge with the ML path. Thus the SOVA algorithm gives a degraded performance compared to the Max-Log-MAP algorithm. However, as pointed out in [7] by Robertson *et al.* the SOVA and Max-Log-MAP algorithms will always give the same hard decisions, as in both algorithms these hard decisions are determined by the ML path, which is calculated using the same metric in both algorithms.

A comparison of the complexities of the Log-MAP, the Max-Log-MAP, and the SOVA algorithms is given in [7]. The relative complexity of the algorithms depends on the constraint length K of the convolutional codes used, but it is shown that the Max-Log-MAP algorithm is about twice as complex as the SOVA algorithm. The Log-MAP algorithm is slightly more complex than the Max-Log-MAP algorithm due to the look-ups required to find the correction factors $f_c(x)$. The performance of the algorithms when used in the iterative decoding of turbo codes falls in the same order as their complexities, with the best performance given by the Log-MAP algorithm, then the Max-Log-MAP algorithm, and the worst performance given by the SOVA algorithm. In the next section we study the effect of the various parameters on the codec performance.

TABLE IV
STANDARD TURBO ENCODER AND DECODER PARAMETERS USED

Channel	Additive White Gaussian Noise (AWGN)
Modulation	Binary Phase Shift Keying (BPSK)
Component Encoders	2 identical Recursive -Convolutional Codes (RSCs)
RSC Parameters	n=2,k=1,K=3 $G_0 = 7$ $G_1 = 5$
Interleaver	1000 bit random interleaver with odd-even separation [31]
Puncturing Used	Half parity bits from each component encoder transmitted - give half-rate code
Component Decoders	Log-MAP decoder
Iterations	8

TABLE V
INPUT AND TRANSMITTED BITS FOR TURBO DECODING EXAMPLE

Input Bit	Systematic Bit	Parity Bits Coder 1	Parity Bits Coder 2	Received Sequence
-1	-1	-1	–	-2.1,-0.1
-1	-1	–	-1	-1.4,-1.4
-1	-1	-1	–	-1.7,-0.5
-1	-1	–	-1	+0.9,+0.5
-1	-1	-1	–	+1.2,-1.7
-1	-1	–	-1	-1.1,-1.1
-1	-1	-1	–	-0.7,-0.8
-	-1	–	-1	-2.4,-1.9
-	-1	-1	–	-1.6,-0.9

IX. THE EFFECT OF VARIOUS CODEC PARAMETERS

In this section we present simulation results for turbo codes using Binary Phase Shift Keying (BPSK) over Additive White Gaussian Noise (AWGN) channels. We show that there are many parameters, some of which are interlinked, which affect the performance of turbo codes. Some of these parameters are:

- The component decoding algorithm used.
- The number of decoding iterations used.
- The frame-length or latency of the input data.
- The specific design of the interleaver used.
- The generator polynomials and constraint lengths of the component codes.

The standard parameters that we have used in our simulations are shown in Table IV. The turbo encoder uses two component RSCs in parallel. The RSC component codes are $K = 3$ codes with generator polynomials $G_0 = 7$ and $G_1 = 5$ in octal representation. These generator polynomials are optimum in terms of maximizing the minimum free distance of the component codes [30]. The effects of changing these parameters are examined in Section IX-E. The standard interleaver used between the two component RSC codes is a 1000–bit random interleaver with odd-even separation [31]. The effects of changing the length of the interleaver, and its structure, are examined in Sections IX-D and IX-F. Unless otherwise stated, the results in this section are for half-rate codes, where half the parity bits generated by each of the two component RSC codes are punctured. However, for comparison, we also include some results for turbo codes where all the parity bits from both component encoders are transmitted,

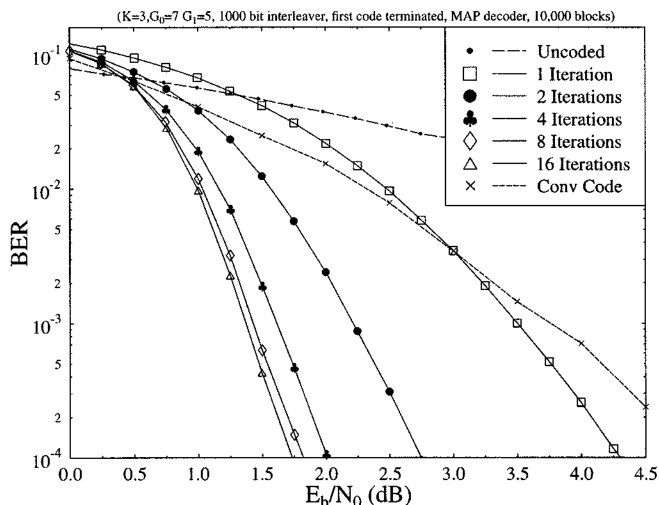

Fig. 13. Turbo coding BER performance using different numbers of iterations of the MAP algorithm. other parameters as in Table IV.

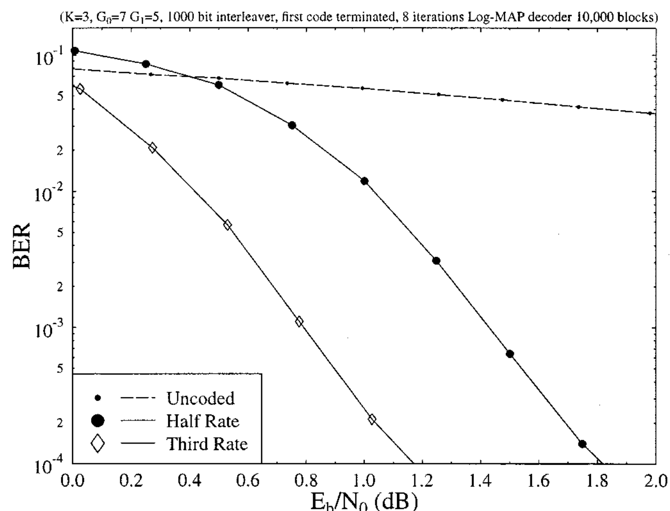

Fig. 14. BER performance comparison between one-third and half-rate turbo codes using parameters of Table IV.

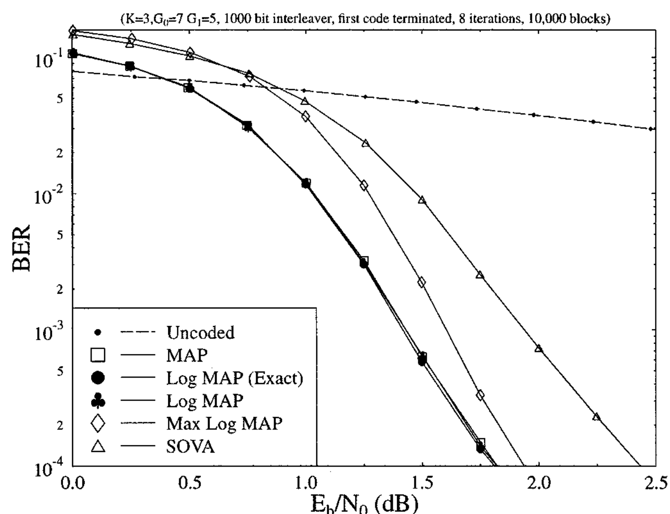

Fig. 15. BER performance comparison between different component decoders for a random interleaver with $L = 1000$. Other parameters as in Table IV.

leading to a one-third rate code. At the decoder two component, soft-in soft-out, decoders are used in parallel in the structure shown in Fig. 1. In most of our simulations we use the Log-MAP decoder, but the effect of using other component decoders is investigated in Section IX-C. Usually 8 iterations of the component decoders are used, but in the next section we consider the effect of the number of iterations.

A. *The Effect of the Number of Iterations Used*

Fig. 13 shows the performance of a turbo decoder using the MAP algorithm versus the number of decoding iterations which were used. For comparison, the uncoded BER and the BER obtained using convolutional coding with a standard $(2, 1, 3)$ nonrecursive convolutional code, are also shown. Like the component codes in the turbo encoder, the convolutional encoder uses the optimum octal generator polynomials of 7 and 5. It can be seen that the performance of the turbo code after one iteration is roughly similar to that of the convolutional code at low SNRs, but improves more rapidly than that of the convolutional coding as the SNR is increased. As the number of iterations used by the turbo decoder increases, the turbo decoder performs significantly better. However, after eight iterations there is little improvement achieved by using further iterations. For example, it can be seen from Fig. 13 that using 16 iterations rather than eight gives an improvement of only about 0.1 dB. Similar results are obtained when using the SOVA algorithm—again there is little improvement in the BER performance of the decoder from using more than eight iterations. Hence, for complexity reasons usually only about eight iterations are used, and so, unless otherwise stated, in our future simulations we have used eight iterations. In the next section, we consider the effect of puncturing.

B. *The Effect of Puncturing*

Again, in a turbo encoder two or more component encoders are used to generate parity information from an input data sequence. We have used two RSC component encoders, and this is the arrangement most commonly used for turbo codes with rates below two-thirds. Typically, in order to give a half-rate code, half the parity bits from each component encoder are punctured. This was the arrangement used in their original paper by Berrou *et al.* on turbo codes [1]. However, it is of course possible to omit the puncturing and transmit all the parity information from both component encoders, which gives a one-third rate code. The performance of such a code, compared to the corresponding half-rate code, is shown in Fig. 14. In this figure, the encoders use the same parameters as were described above for Fig. 13. It can be seen that transmitting all the parity information gives a gain of about 0.6 dB, in terms of E_b/N_0, at a BER of 10^{-4}. This corresponds to a gain of about 2.4 dB in terms of channel SNR. Very similar gains are seen for turbo codes with different frame-lengths. Let us now consider the performance of the various soft-in soft-out component decoding algorithms.

C. *The Effect of the Component Decoding Algorithm Used*

It can also be seen from Fig. 15 that the Max Log MAP and the SOVA algorithms both give a degradation in performance compared to the MAP and Log MAP algorithms. At a BER of

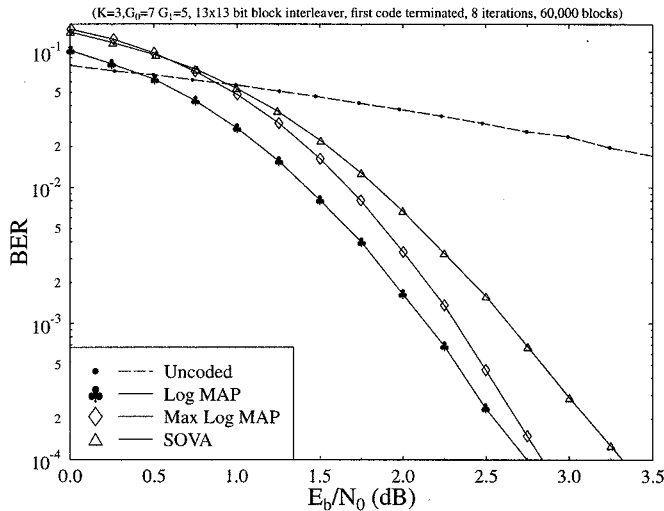

Fig. 16. BER performance comparison between different component decoders for a $L = 169$, 13×13, block interleaver. Other parameters as in Table IV.

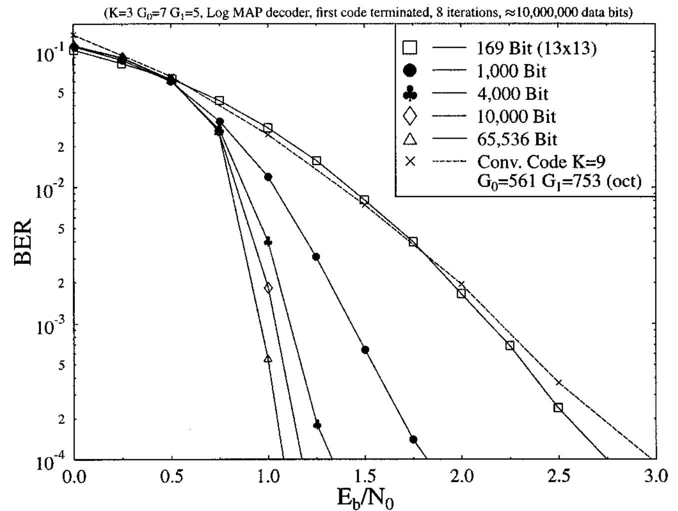

Fig. 18. Effect of frame-length on the BER performance of turbo coding. All Interleavers except $L = 169$ block interleaver use random separated interleavers [31]. Other parameters as in Table IV.

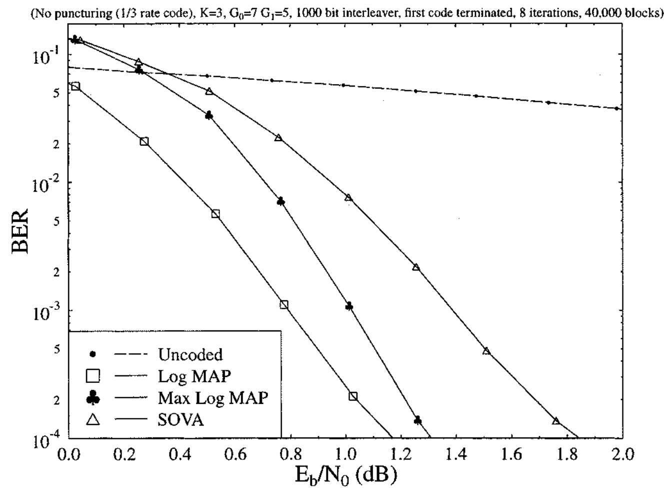

Fig. 17. BER performance comparison between different component decoders for a random interleaver with $L = 1000$ Using a $1/3$ rate code. Other parameters as in Table IV.

10^{-4} this degradation is about 0.1 dB for the Max Log MAP algorithm, and about 0.6 dB for the SOVA algorithm.

Fig. 16 compares the Log MAP, Max Log MAP and SOVA algorithms for a turbo decoder with a frame-length of only 169 b, rather than 1000 b as was used for Fig. 15. It can be seen that although all three decoders give a worse BER performance than those shown in Fig. 15, the differences in the performances between the decoders are very similar to those shown in Fig. 15. Similarly, Fig. 17 compares these three decoding algorithms for a one-third rate code, and again the degradations relative to a decoder using the Log-MAP algorithm are about 0.1 dB for the Max-Log-MAP algorithm, and about 0.6 dB for the SOVA algorithm.

D. *The Effect of the Frame-Length of the Code*

In the original paper on turbo coding by Berrou *et al.* [1], and many of the subsequent papers, impressive results have been presented for coding with very large frame lengths. Dolinar *et al.* analyzed the associated theoretical performance limits as a function of the coded frame length in [32]. However, for many applications, such as speech transmission systems, the large delays inherent in using high frame-lengths are unacceptable. Therefore, an important area of turbo coding research is achieving as impressive results with short frame-lengths as have been demonstrated for long frame-length systems. Fig. 18 shows how dramatically the performance of turbo codes depends on the frame-length L used in the encoder. The 169-bit code would be suitable for use in a speech transmission systems at approximately 8 kb/s with a 20-ms frame-length [33], while the 1000–bit code would be suitable for video transmission. The larger frame-length systems would be useful in data or nonreal time transmission systems. It can be seen from Fig. 18 that the performance of turbo codes is very impressive for systems with long frame lengths. However, even for a short frame-length system, using 169 b per frame, it can be seen that turbo codes give good results, comparable to or better than a constraint length $K = 9$ convolutional code. The use of the $K = 9$ convolutional code as a bench-marker is justified below.

Fig. 15 shows a comparison between turbo decoders using the parameters described above. In this figure the "Log MAP (exact)" curve refers to a decoder which calculates the correction term $f_c(x)$ in (28) of Section 5 exactly, i.e., using

$$f_c(x) = \ln(1 + e^{-x}) \tag{38}$$

rather than using a look-up table as described in [7]. The Log MAP curve refers to a decoder which does use a look-up table with eight values of $f_c(x)$ stored, and hence introduces an approximation to the calculation of the LLRs. It can be seen that, as expected, the MAP and the Log-MAP (exact) algorithms give identical performances. Furthermore, as Robertson found [7], the look-up procedure for the values of the $f_c(x)$ correction terms introduces no degradation to the performance of the decoder.

It can be shown that a single decoding operation with the Log-MAP decoder, including using the *a-priori* information, is

about four times as complex as decoding the same code using a standard Viterbi decoder. The curves shown in Fig. 18, and in most of our results, were generated using two component decoders with eight iterations. Therefore, the overall complexity of a turbo decoder is approximately $2 \times 8 \times 4 = 64$ times that of a Viterbi decoder for one of the component convolutional codes. This means that the complexity of our turbo decoder using eight iterations of two $K = 3$ component codes is approximately the same as the complexity of a Viterbi decoder for an ordinary $K = 9$ convolutional code. However, this ignores the fact that, at any given iteration after the first, the component decoders re-use the same channel information they used in the first iteration. Only the *a-priori* information inputs to the component decoders change from one iteration to the next. Therefore, the complexity of iterations after the first one can be reduced by storing information used in the first iteration, rather than recalculating it. This allows us to reduce the complexity of the turbo decoder by about a third. In order to give a comparison between the performance of turbo codes and convolutional codes for similar complexity decoders, we will compare our $K = 3$ turbo codes with an eight iteration decoder to a $K = 9$ convolutional code. However, it should be noted that it is possible to reduce the complexity of the turbo decoder below that of the $K = 9$ convolutional decoder.

Fig. 18 shows the performance of such a convolutional code. A nonrecursive $(2, 1, 9)$ convolutional code using the generator polynomials $G_0 = 561$ and $G_1 = 753$ in octal notation, which maximize the free distance of the code [30], was used. These generator polynomials provide the best performance in the AWGN channels we use in this section. A frame-length of 169 b is used, and the code is terminated. It can be seen that even for the short frame-length of 169 b, turbo codes out-perform similar complexity convolutional codes. As the frame-length is increased, the performance gain from using turbo codes, rather than high constraint length convolutional codes, increases dramatically.

Fig. 19 shows how the performance of a one-third rate turbo code varies with the frame-length of the code. Again, the performance of the turbo code is better the longer the frame-length of the code, but impressive results are still obtained with a frame length of only 169 b. Again the results for a $K = 9$ convolutional code are shown, this time using a third rate $n = 3, k = 1$ code with the optimal generator polynomials of $G_0 = 557$, $G_1 = 663$ and $G_2 = 711$ [30] in octal notation. Again it can be seen that the high constraint length convolutional code is out-performed by turbo codes with frame-lengths of 169 and higher.

Let us now consider the effect of using different RSC component codes.

E. The Component Codes

Both the constraint length and the generator polynomials used in the component codes of turbo codes are important parameters. Often in turbo codes the generator polynomials which lead to the largest minimum free distance for ordinary convolutional codes are used, although when the effect of interleaving is considered these generator polynomials do not necessarily lead to the best minimum free distance for turbo codes. Fig. 20 shows

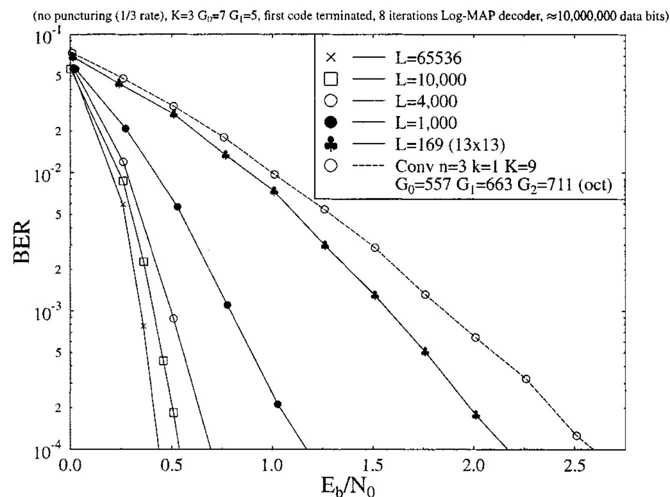

Fig. 19. Effect of frame-length on BER performance of $1/3$ rate turbo coding. All interleavers except $L = 169$ block interleaver use random interleavers. Other parameters as in Table IV.

the huge difference in performance that can result from different generator polynomials being used in the component codes. The other parameters used in these simulations were the same as detailed above in Table IV.

Most of the results given in this report were obtained using constraint length three component codes. For these codes we have used the optimum generator polynomials in terms of maximizing the minimum free distance of the component convolutional codes, i.e., 7 and 5 in octal representation. These generator polynomials were also used for constraint length three turbo coding by Hagenauer *et al.* in [16] and Jung in [34]. It can be seen from Fig. 20 that the order of these generator polynomials is important—the octal value 7 should be used for the feedback generator polynomial of the encoder (denoted here by G_0). If G_0 and G_1 are swapped round, the performance of a convolutional code (both regular and recursive systematic codes) would be unaffected, but for turbo codes this gives a significant degradation in performance.

The effect of increasing the constraint length of the component codes used in turbo codes is shown in Fig. 21. For the constraint length four turbo code we again used the optimum minimum free distance generator polynomials for the component codes (15 and 17 in octal, 13 and 15 in decimal representations). The resulting turbo code gives an improvement of about 0.25 dB at a BER of 10^{-4} over the $K = 3$ curve.

For the constraint length 5 turbo code we used the octal generator polynomials 37 and 21 (31 and 17 in decimal), which were the polynomials used by Berrou *et al.* [1] in the original paper on turbo coding. We also tried using the octal generator polynomials 23 and 35 (19 and 29), which are again the optimum minimum free distance generator polynomials for the component codes, as suggested by Hagenauer *et al.*in [16]. We found that these generator polynomials gave almost identical results to those used by Berrou *et al.*It can be seen from Fig. 21 that increasing the constraint length of the turbo code does improve its performance, with the $K = 4$ code performing about 0.25 dB better than the $K = 3$ code at a BER of 10^{-4}, and the $K = 5$ code giving a further improvement of about 0.1 dB. However,

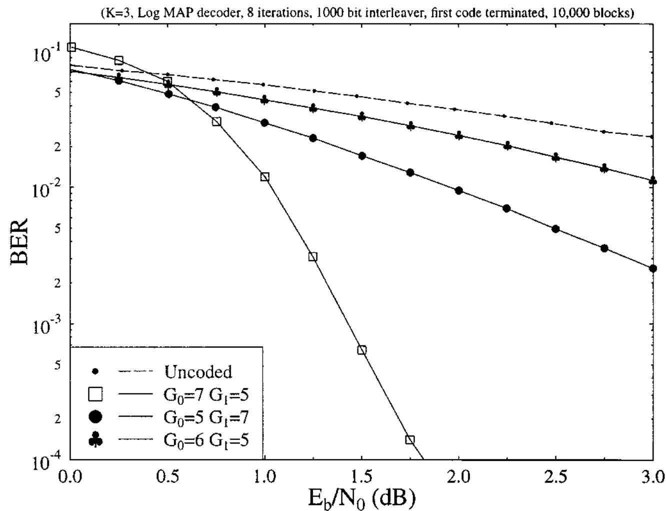

Fig. 20. Effect of generator polynomials on BER performance of turbo coding. Other parameters as in Table IV.

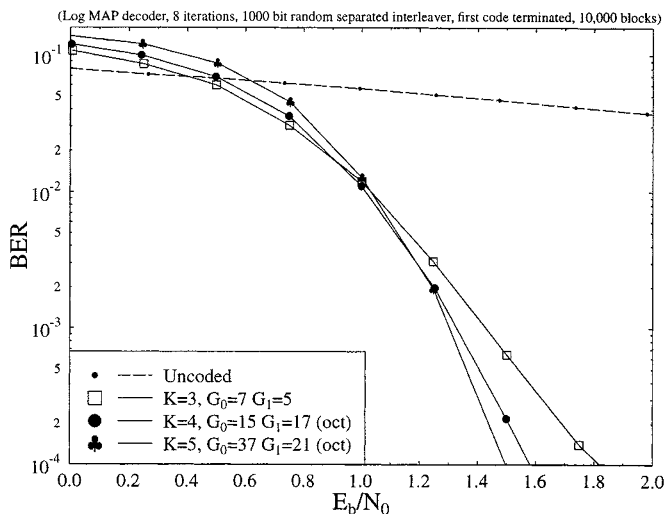

Fig. 21. Effect of constraint length on the BER performance of turbo coding. Other parameters as in Table IV.

these improvements are provided at the cost of approximately doubling or quadrupling the decoding complexity. Therefore, unless otherwise stated, we have used component codes with a constraint length of 3 in our work. Let us now focus on the effects of the interleaver used within the turbo encoder and decoder.

F. The Effect of the Interleaver

It is well known that the interleaver used in turbo codes has a vital influence on the performance of the code. The interleaver design together with the generator polynomials used in the component codes, and the puncturing used at the encoder, have a dramatic affect on the free distance of the resultant turbo code. Several algorithms have been proposed, for example, in [35] and [36], that attempt to choose good interleavers based on maximizing the minimum free distance of the code. However, this process is complex, and the resultant interleavers are not necessarily optimum. For example, in [37] random interleavers designed using the technique given in [36] are compared to a

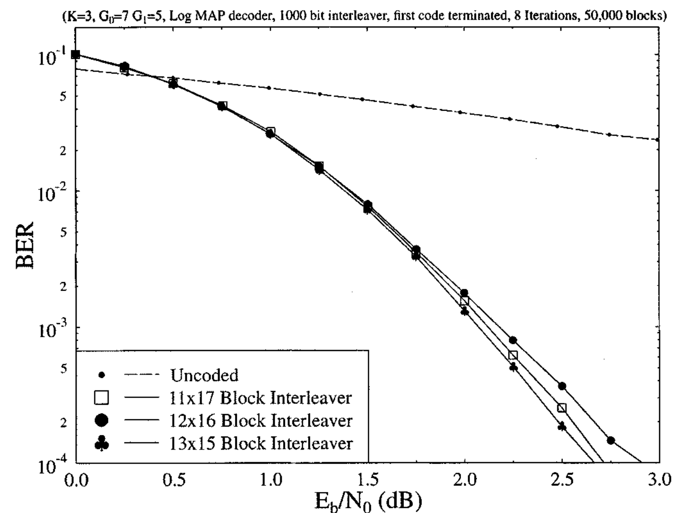

Fig. 22. Effect of block interleaver choice for $L \approx 190$ frame-length turbo codes. Other parameters as in Table IV.

12×16 block interleaver, and the "optimized" interleavers are found to perform worse than the block interleaver.

In [31], a simple technique for designing good interleavers, which is referred to as "odd-even separation" is proposed. With alternate puncturing of the parity bits from each of the component codes, which is the puncturing most often used, if an interleaver is designed so that the odd and even input bits are kept separate, then it can be shown that one (and only one) parity bit associated with each information bit will be left unpunctured. This is preferable to the more general situation, where some information bits will have their parity bits from both component codes transmitted, whereas others will have neither of their parity bits transmitted.

A convenient way of achieving odd-even separation in the interleaver is to use a block interleaver with an odd number of rows and columns [31]. The benefits of using an odd number of rows and columns with a block interleaver can be seen in Fig. 22. This shows a comparison between turbo coders using several block interleavers with frame-lengths of approximately 190 b. The 12×16 block interleaver, proposed for short frame transmission systems in [37] and used by the same authors in other papers such as [34], [38], [39], clearly has a somewhat lower performance than the other block interleavers, which use an odd number of rows and columns. It is also interesting to note that of the two block interleavers with an odd number of rows and columns, the interleaver which is closer to being square (i.e., the 13×15 interleaver) performs better than the more rectangular 11×17 interleaver.

We also attempted using random interleavers of various frame-lengths. The effect of the interleaver choice for a turbo coding system with a frame-length of approximately 960 b is shown in Fig. 23. It can be seen from this figure that, as was the case with the codes with frame-lengths around 192 b shown in Fig. 22, the block interleaver with an odd number of rows and columns (the 31×31 interleaver) performs significantly better than the interleaver with an even number of rows and columns (the 30×32 interleaver). However, both of these interleavers are outperformed by the two random interleavers.

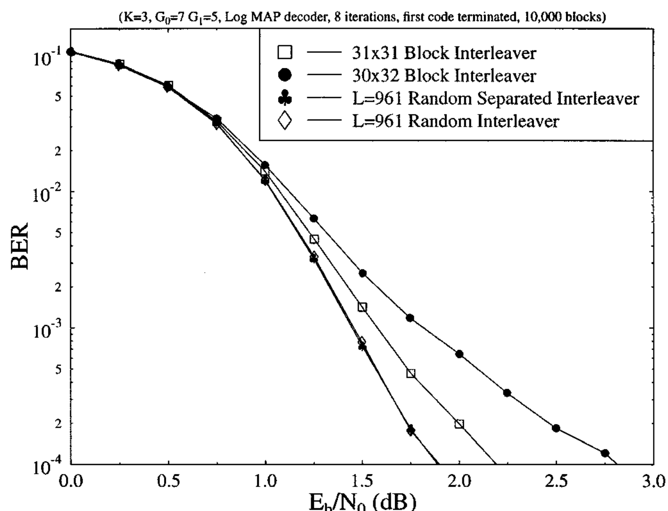

Fig. 23. Effect of interleaver choice for $L \approx 961$ frame-length turbo codes. Other parameters as in Table IV.

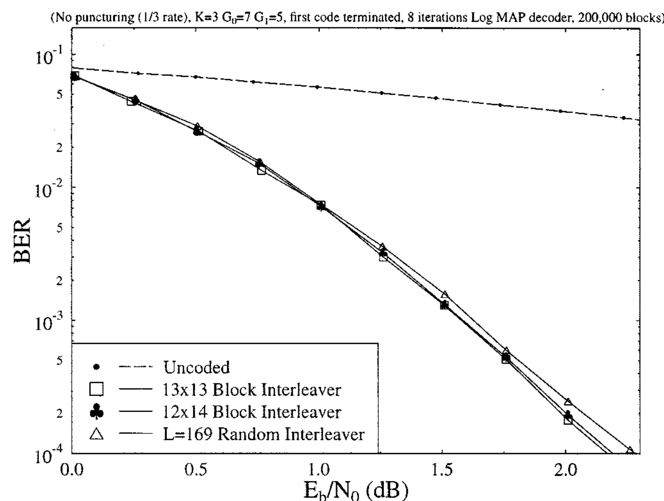

Fig. 25. Effect of interleaver choice for third-rate $L \approx 169$ frame-length turbo codes. Other parameters as in Table IV.

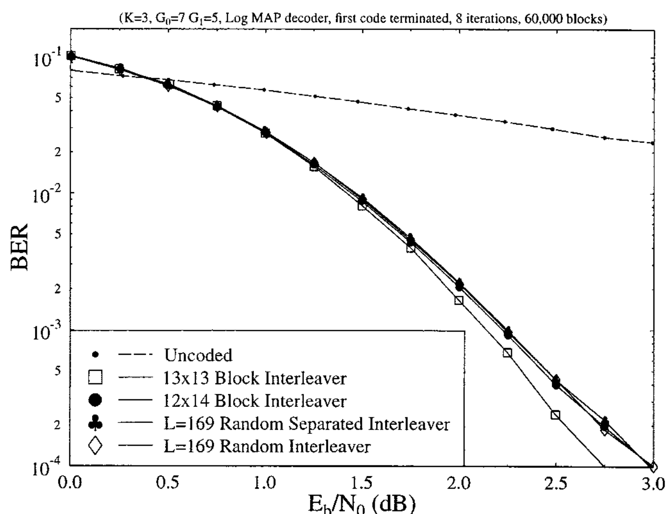

Fig. 24. Effect of interleaver choice for $L \approx 169$ frame-length turbo codes. Other parameters as in Table IV.

In the "random separated" interleaver odd-even separation, as proposed by Barbulescu and Pietrobon [31], is used. This interleaver performs very slightly better than the other random interleaver, which does not use odd-even separation. However, the effect of odd-even separation is much less significant for the random interleavers than it is for the block interleavers.

Similar curves are shown in Fig. 24 for turbo coding schemes with approximately 169 b per frame. It can be seen again that the scheme using block interleaving with odd-even separation (i.e., the 13×13 interleaver) performs better than the the scheme using block interleaving without odd-even separation (i.e., the 12×14 interleaver). However, for this short frame-length system the two random interleavers perform worse than the best block interleaver. From our results it appears that although random interleavers give the best performance for turbo codes with long frame-lengths, for short frame-length systems the best performance is given using a block interleaver with an odd number of rows and columns.

When puncturing is not used, and we have a third rate code, the benefit of using odd-even separation with block interleavers, i.e., using block interleavers with an odd number of rows and columns, disappears. This can be seen from Fig. 25, which compares the performance of a turbo code with no puncturing using three different interleavers, all with a length of approximately 169 b. As in the case of the half-rate turbo codes using puncturing in Fig. 24, for a small frame length, such as 169 b, the best performance is given by using a block rather than a random interleaver. However, it can be seen from Fig. 25 that, unlike for half-rate codes, for turbo codes without puncturing there is little difference between the block interleavers with and without odd-even separation, i.e., between the 13×13 and 12×14 interleavers.

In [40], Herzberg suggests that a "reverse block" interleaver, i.e., a block interleaver in which the output bits are read from the block in the reverse order relative to an ordinary block interleaver, gives an improved performance over ordinary block interleavers. He also suggests that for high SNRs, and low BERs, reverse block interleavers with a small frame-length give a better performance than random interleavers with a much longer frame length. However, as can be seen from Fig. 26, which shows the performance of ordinary and reverse block interleavers for various frame-lengths, we found very little difference between the performances of block and reverse block interleavers. One difference between our results and those in [40] is that we have used punctured half-rate turbo codes, whereas Herzberg used turbo codes without puncturing. However, we found that even with third-rate turbo codes using no puncturing, and using 14×14 interleavers as Herzberg did, the performance of block and reverse block interleavers were almost identical. It appears in [40] that for turbo codes with long random interleavers, and with an ordinary block interleaver, Herzberg used the generator polynomials $G_0 = 5$ and $G_1 = 7$, whereas for the reverse block interleaver he used the generator polynomials $G_0 = 7$ and $G_1 = 5$. The generator polynomials $G_0 = 5$ and $G_1 = 7$ were used so that the performance of turbo codes with long random interleavers could be approximated using the Union bound and the error coefficients calculated by Benedetto and Montorosi in

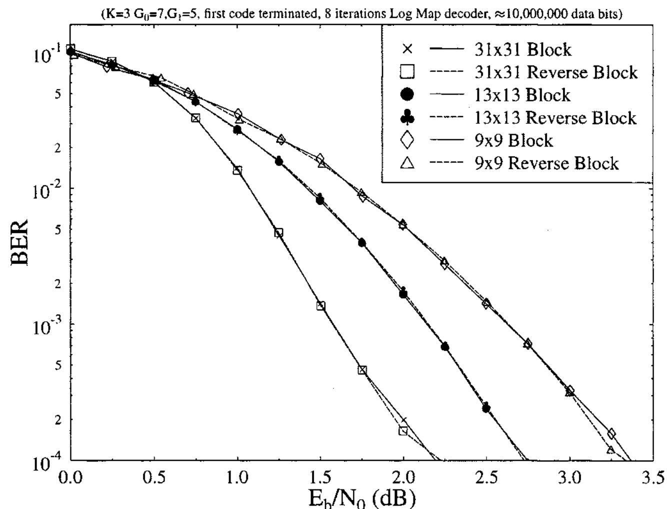

Fig. 26. BER performance of block and reverse block interleavers. Other parameters as in Table IV.

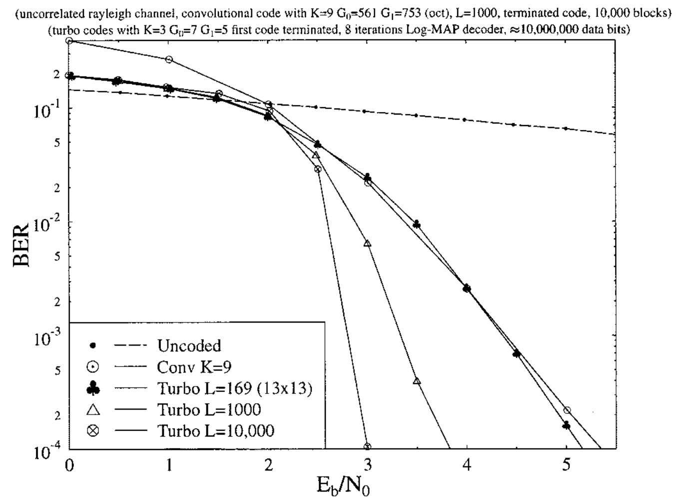

Fig. 27. BER performance of turbo codes with different frame-lengths L over perfectly interleaved Rayleigh fading channels. Other turbo codec parameters as in Table IV.

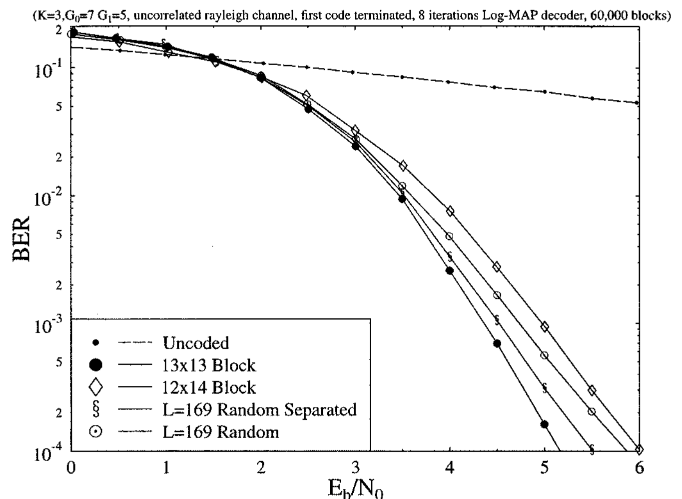

Fig. 28. BER performance of turbo codes with different interleavers over perfectly interleaved Rayleigh fading channels. Other turbo codec parameters as in Table IV.

[15] for these generator polynomials. However, as was seen in Fig. 20, these generator polynomials give a significantly worse performance that the generator polynomials $G_0 = 7$ and $G_1 = 5$ we have used for most of our simulations, and Herzberg used with his reverse block interleaver. Thus it appears that the reason Herzberg found such promising results for the reverse block interleaver was not because of this interleaver's superiority, but because of the inferiority of the generator polynomials he used with random and block interleavers.

Having investigated the performance of turbo codes when used with BPSK modulation over AWGN channels, in the next section we will briefly characterize the expected performance over Rayleigh-fading channels.

X. Turbo Coding over Rayleigh Channels

In the previous sections, we have discussed the performance of turbo coded BPSK over AWGN channels. In this section we have trasmitted over Rayleigh fading channels, assuming that the receiver has exact estimates of the fading amplitude and phase inflicted by the channel. This assumption is justified as several techniques, e.g., Pilot Symbol Assisted Modulation (PSAM) [4], are available which provide practical CSI recovery with performance very close to that assuming perfect recovery. In Section X-A we consider the performance of various turbo codes over Rayleigh fading channels, which are perfectly interleaved. Then in Section X-B we characterize the effects of various Doppler frequencies.

A. Turbo Coded Perfomance over Perfectly Interleaved Rayleigh Channels

Fig. 27 shows the performance of three turbo codes with different frame-lengths L over a perfectly interleaved Rayleigh fading channel using BPSK modulation. All the turbo codecs use two $K = 3$ RSC component codes with generator polynomials $G_0 = 7$ and $G_1 = 5$. At the decoder eight iterations of the Log-MAP decoder are used. Also shown in Fig. 27 is the performance of a constraint length $K = 9$ convolutional code which, as explained earlier, has a decoder complexity which is similar or slightly higher than that of the turbo decoder. It can be seen that the turbo codes with frame-lengths of $L = 1000$ or $L = 10\ 000$ give a significant increase in performance over the convolutional code. Even the turbo code with a short frame length of 169 b outperforms the convolutional code for BERs below 10^{-3}.

Comparing the performance of the $L = 169$, $L = 1000$, and $L = 10\ 000$ turbo codes in Fig. 27 to those in Fig. 18 for the same codes over an AWGN channel, we see that the perfectly interleaved fading of the received channel values degrades the BER performance of the code by around 2 dB at a BER of 10^{-4}, with a larger degradation for the shorter frame-length codes.

The short frame-length $L = 169$ turbo codec in Fig. 27 uses a 13×13 block interleaver. We found in Section IX–F that with a Gaussian channel for short frame-lengths a block interleaver with an odd number of rows and columns should be used. Fig. 28 shows the effect of using different interleavers, all with a frame-length of approximately 169 b, on the BER performance of a turbo codec over a perfectly interleavered Rayleigh channel. It

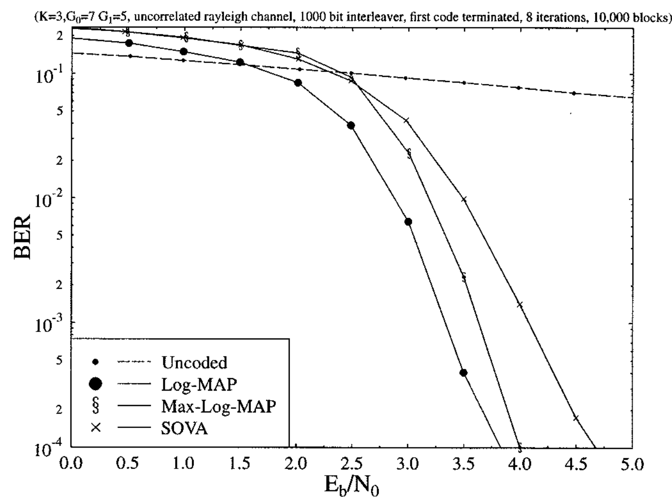

Fig. 29. BER performance of turbo codes with $L = 1000$ Using different component decoders over perfectly interleaved Rayleigh fading channels. Other turbo codec parameters as in Table IV.

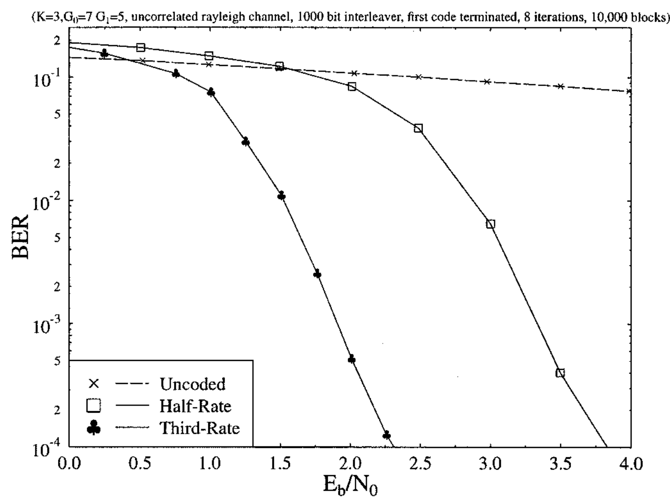

Fig. 30. The BER performance comparison between one-third and one-half rate of turbo codes over perfectly interleaved Rayleigh fading channels. Other turbo codec parameters as in Table IV.

can be seen from this figure that again for the short frame-length of 169 b the best performance is given by a block interleaver with an odd number of rows and columns. This block interleaver acheives odd-even separation [31] so that each data bit has one, and only one, of the two parity bits associated with it transmitted. The 12×14 block interleaver shown in Fig. 28 does not give odd-even separation, and so even through its frame-length is almost identical to that of the 13×13 block interleaver (168 rather than 169 b) it performs almost 1 dB worse than the 13×13 block interleaver at a BER of 10^{-4}. The two random interleavers shown in Fig. 28 also perform worse than the 13×13 block interleaver, although the random interleaver with odd-even separation does perform better than the nonseparated interleaver.

Fig. 29 shows how the choice of the component decoders used at the turbo decoder affects the performance of the codec over a perfectly interleavered Rayleigh channel. It can be seen that again the Log-MAP decoder gives the best performance, followed by the Max-Log-MAP decoder with the SOVA decoder, the simplest of the three, giving the worst performance. It can also be seen that the differences in performances between the different decoders are slightly larger than they were over an AWGN channel—the Max-Log-MAP decoder performs about 0.2 dB worse than the Log-MAP decoder, and the SOVA decoder is about 0.8 dB worse than the Log-MAP decoder.

Fig. 30 shows the effect of puncturing on a turbo code with frame-length $L = 1000$ over the perfectly interleaved Rayleigh channel. In Fig. 14 we saw that over the AWGN channel the third-rate code outperformed the half-rate code by about 0.6 dB in terms of E_b/N_0. We see from Fig. 30 that again for the perfectly interleaved Rayleigh channel the difference in performance is bigger—about 1.5 dB in terms of E_b/N_0 or about 3.25 dB in terms of channel SNR.

B. Turbo Coded Performance over Correlated Rayleigh Channels

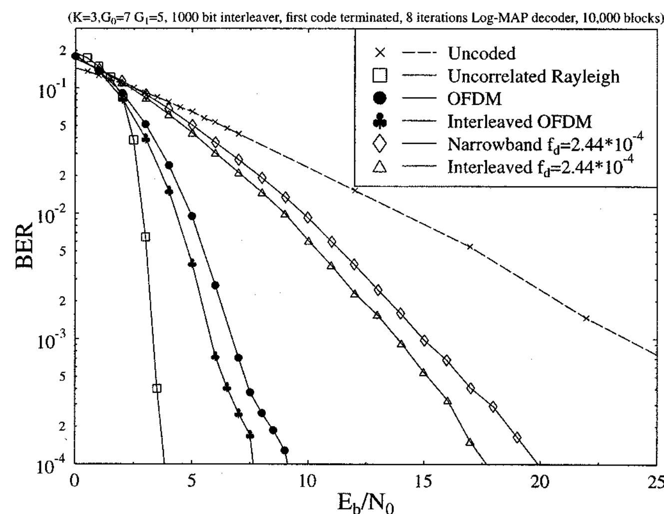

Fig. 31. Performance of turbo coding over Rayleigh fading channels. Turbo codec parameters as in Table IV.

Fig. 31 shows the performance of a half-rate turbo coding system with $L = 1000$ over various Rayleigh fading channels. It can be seen that by far the best performance is given by the perfectly interleaved Rayleigh channel where there are no correlations between sucessive fading values. The narrowband Rayleigh channel shown has a normalized Doppler frequency of $f_d = 2.44 * 10^{-4}$, given by assuming a carrier frequency of 1.9 GHz, a symbol rate of 360 Kbaud and a vehicular speed of 50 km/h. These values correspond to a RACE ATDMA long macro-cell type system. It can be seen that the turbo codes give a significant coding gain over the uncoded BER results even for this channel. We found that for Rayleigh fading channels with faster fading, i.e., a higher normalized Doppler frequency, the coding gain increased. Also it can be seen that interleaving the output bits from the turbo encoder before transmission over the Rayleigh fading channel improves the performance for the narrowband system by about 2.5 dB at a BER of 10^{-4}. This gain was acheived by merely interleaving over the length of the output block from the turbo encoder (2000 b). Higher interleaving gains can be acheived, at the cost of extra delay, by interleaving over longer periods. Perfect interleaving over a much

longer period would give the performance shown by the uncorrelated Rayleigh curve in Fig. 31.

Also shown in Fig. 31 is the performance of our turbo codec over an Orthogonal Frequency Division Multiplexing (OFDM) system with Rayleigh fading. The effects of OFDM with turbo coding will be explored in the next section, but it can be seen that the OFDM gives a coded performance much closer to that for the perfectly interleaved Rayleigh channel. Again interleaving over the 2000 output bits from the turbo encoder improves the coded performance.

Having portrayed the performance of turbo codes using BPSK modulation over Rayleigh channels, in the next section we offer our conclusions.

XI. Conclusion

In this article, we have described the techniques used for the decoding of turbo codes. Although it is possible to optimally decode turbo codes in a single noniterative step [21], [22], for complexity reasons a nonoptimum iterative decoder is almost always preferred. Such an iterative decoder employs two component soft-in soft-out decoders, and we have described the MAP, Log-MAP, Max-Log-MAP and SOVA algorithms, which can all be used as component decoders. The MAP algorithm is optimal for this task, but it is extremely complex. The Log-MAP algorithm is a simplification of the MAP algorithm, and offers the same optimal performance with a reasonable complexity. The other two algorithms, the Max-Log-MAP and the SOVA, are both less complex again, but give a slightly degraded performance. In order to gauge the expected coding performance we also provided a range of performance results using a variety of codec parameters.

Research continues in a range of areas, attempting to improve the complexity versus performance tradeoffs. For example, while innovating in the field of block-based turbo codes, near-Shannonian performance was achieved by Hagenauer *et al.* using a near-unity coding rate, although at a high decoding complexity. Other efforts are in the field of incorporating turbo codes in practical speech and video systems. A range of application examples can be found in the context of interactive and broadcast video systems as well as local area networks in [4] and [5], and in [42]–[52]. In short, an exciting era at the 50th anniversary of Shannonian information theory, witnessing the first practical systems performing close to information theoretical limits, stimulating the research community to aspire to similar performance over dispersive, fading wireless channels.

Acknowledgment

Sincere thanks are due to the EPSRC, U.K. and to the Virtual Centre of Excellence in Mobile Communications.

References

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

[2] C. Berrou and A. Glavieux, "Near optimum error correcting coding and decoding. Turbo-codes," *IEEE Trans. Commun.*, vol. 44, no. 10, pp. 1261–1271, 1996.

[3] R. Steele and L. Hanzo, Eds., *Mobile Radio Communications: Second and Third Generation Cellular and WATM Systems*, 2nd ed. New York: Wiley, 1999, ISBN 07273-1406-8.

[4] L. Hanzo, W. T. Webb, and T. Keller, *Single- and Multi-Carrier Quadrature Amplitude Modulation: Principles and Applications for Personal Communications, WATM and Broadcasting*. New York: Wiley, 2000.

[5] L. Hanzo, P. Cherriman, and J. Streit. Video compression and communications over wireless channels: From second to third generation systems, WLAN's and beyond. *TITLE?* [Online]. Available: http://www-mobile.ecs.soton.ac.uk

[6] L. R. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate," *IEEE Trans. Inform. Theory*, vol. VOL. NO?, pp. 284–287, Mar. 1974.

[7] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain," in *Proc. Int. Conf. Communications*, June 1995, pp. 1009–1013.

[8] G. Battail, "Ponderation des symboles decodes par l'algorithme de Viterbi," *Ann. Telecommun.*, vol. 42, no. 1–2, pp. 31–38, Jan. 1987. (in French).

[9] C. Berrou, P. Adde, E. Angui, and S. Faudeil, "A low complexity soft-output Viterbi decoder architecture," in *Proc. Int. Conf. Communications*, May 1993, pp. 737–740.

[10] S. Le Goff, A. Glavieux, and C. Berrou, "Turbo-codes and high spectral efficiency modulation," *Proc. IEEE Int. Conf. Communications*, pp. 645–649, 1994.

[11] U. Wachsmann and J. Huber, "Power and bandwidth efficient digital communications using turbo codes in multilevel codes," *Eur. Trans. Telecommun.*, vol. 6, pp. 557–567, Sept.–Oct. 1995.

[12] P. Robertson and T. Worz, "Bandwidth-efficient turbo trellis-coded modulation using punctured component codes," *IEEE J. Select. Areas Commun.*, vol. 16, pp. 206–218, Feb. 1998.

[13] S. Benedetto and G. Montorsi, "Design of parallel concatenated convolutional codes," *IEEE Trans. Commun.*, vol. 44, pp. 591–600, May 1996.

[14] L. C. Perez, J. Seghers, and D. J. Costello, "A distance spectrum interpretation of turbo codes," *IEEE Trans. Inform. Theory*, vol. 42, pp. 1698–1709, Nov. 1996.

[15] S. Benedetto and G. Montorsi, "Unveiling turbo codes: Some results on parallel concatenated coding schemes," *IEEE Trans. Inform. Theory*, vol. 42, pp. 409–428, Mar. 1996.

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

[17] R. Pyndiah, "Iterative decoding of product codes: Block turbo codes," in *Proc. Int. Symp. Turbo Codes and Related Topics*, Brest, France, Sept. 1997, pp. 71–79.

[18] B. Sklar, "A primer on turbo code concepts," *IEEE Commun. Mag.*, pp. 94–102, Dec. 1997.

[19] J. Hagenauer and P. Hoeher, "A Viterbi algorithm with soft-decision outputs and its applications," *IEEE Globecom*, pp. 1680–1686, 1989.

[20] G. D. Forney, "The Viterbi algorithm," *Proc. IEEE*, vol. 61, pp. 268–278, Mar. 1973.

[21] M. Breiling and L. Hanzo, The super-trellis structure of turbo codes, in IEEE Trans. Inform. Theory, Sept. 2000. to appear.

[22] ——, "Optimum noniterative turbo-decoding," in *Proc. PIMRC'97*, Helsinki, Finland, Sept. 1997, pp. 714–718.

[23] C. Berrou, "Some clinical aspects of turbo codes," in *Proc. Int. Symp. Turbo Codes and Related Topics*, Brest, France, Sept. 1997, pp. 26–31.

[24] H. Nickl, J. Hagenauer, and F. Burkett, "Approaching Shannon's capacity limit by 0.27 dB using simple Hamming codes," *IEEE Commun. Lett.*, vol. 1, pp. 130–132, Sept. 1997.

[25] W. Koch and A. Baier, "Optimum and sub-optimum detection of coded data disturbed by time-varying inter-symbol interference," *IEEE Globecom*, pp. 1679–1684, Dec. 1990.

[26] J. A. Erfanian, S. Pasupathy, and G. Gulak, "Reduced complexity symbol dectectors with parallel structures for ISI channels," *IEEE Trans. Commun.*, vol. 42, pp. 1661–1671, 1994.

[27] A. Viterbi, "Approaching the Shannon limit: Theorist's dream and practitioner's challenge," in *Proc. Int. Conf. Millimeter Wave and Far Infrared Science and Technology*, 1996, pp. 1–11.

[28] A. J. Viterbi, "An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes," *IEEE J. Select. Areas Commun.*, pp. 260–264, Feb. 1997.

[29] J. Hagenauer, "Source-controlled channel decoding," *IEEE Trans. Commun.*, vol. 43, pp. 2449–2457, Sept. 1995.

[30] J. G. Proakis, *Digital Communications*, 3rd ed. New York: McGraw-Hill, 1995.

[31] A. S. Barbulescu and S. S. Pietrobon, "Interleaver design for turbo codes," *IEE Electron. Lett.*, pp. 2107–2108, Dec. 1994.
[32] S. Dolinar, D. Divsalar, and F. Pollara. Code Performance as a Function of Block Size. SPL, NASA. [Online]. Available: http://tmo.jpl.nasa.gov/tmo/progress_report/42-133/title.htm
[33] J. P Woodard, T. Keller, and L. Hanzo, "Turbo-coded orthogonal frequency division multiplex transmission of 8 kbps encoded speech," in *Proc. ACTS'97*, Aalborg, Denmark, Oct. 1997, pp. 894–899.
[34] P. Jung, "Comparison of turbo-code decoders applied to short frame transmission systems," *IEEE J. Select. Areas Commun.*, pp. 530–537, 1996.
[35] P. Robertson, "Illuminating the structure of code and decoder of parallel concatenated recursive systematic (turbo) codes," *IEEE Globecom*, pp. 1298–1303, 1994.
[36] P. Jung and M. Naßhan, "Performance evaluation of turbo codes for short frame transmission systems," *IEE Electron. Lett.*, pp. 111–112, Jan. 1994.
[37] ——, "Dependence of the error performance of turbo-codes on the interleaver structure in short frame transmission systems," *IEE Electron. Lett.*, pp. 287–288, Feb. 1994.
[38] ——, "Results on turbo-codes for speech transmission in a joint detection CDMA mobile radio system with coherent receiver antenna diversity," *IEEE Trans. Veh. Technol.*, vol. 46, pp. 862–870, Nov. 1997.
[39] P. Jung, M. Naßhan, and J. Blanz, "Application of turbo-codes to a CDMA mobile radio system using joint detection and antenna diversity," *Proc. IEEE Conf. Veh. Technol.*, pp. 770–774, 1994.
[40] H. Herzberg, "Multilevel turbo coding with short interleavers," *IEEE J. Select. Areas Commun.*, vol. 16, pp. 303–309, Feb. 1998.
[41] T. A. Summmers and S. G. Wilson, "SNR mismatch and online estimation in turbo decoding," *IEEE Trans. Commun.*, vol. 46, pp. 421–423, Apr. 1998.
[42] L. Hanzo, P. J. Cherriman, and E. L. Kuan, "Interactive cellular and cordless video telephony: State-of-the-art, system design principles and expected performance," *Proc. IEEE*, vol. 88, pp. 1388–1415, Sept. 2000.
[43] C. H. Wong, T. H. Liew, and L. Hanzo, "Blind-detection assisted, block turbo coded, decision-feedback equalised burst-by-burst adaptive modulation," *IEEE J. Select. Areas Commun.*, submitted for publication.
[44] C. S. Lee, T. Keller, and L. Hanzo, "OFDM-based turbo-coded hierarchical and nonhierarchical terrestrial mobile digital video broadcasting," *Proc. IEEE Trans. Broadcasting*, vol. 46, pp. 1–22, Mar. 2000.
[45] P. Cherriman, E. L. Kuan, and L. Hanzo, "Turbo coded burst-by-burst adaptive joint-detection CDMA based video telephony," *IEEE Trans. Circuits Syst. Video Technol.*, submitted for publication.
[46] T. Keller and L. Hanzo, "Turbo-coded adaptive modulation techniques for duplex OFDM transmission," *IEEE Trans. Veh. Technol.*, to be published.
[47] B.-L. Yeap, T.-H. Liew, J. Hamorsky, and L. Hanzo, "Comparative study of convolutional coded as well as convolutional-based and block-based turbo coded turbo equalisers," *IEEE J. Select. Areas Commun.*, submitted for publication.
[48] T. Keller and L. Hanzo, "Adaptive multicarrier modulation: A convenient framework for time-frequency processing in wireless communications," *Proc. IEEE*, vol. 88, pp. 611–642, May 2000.
[49] T. Keller, M. Münster, and L. Hanzo, "A turbo-coded burst-by-burst adaptive wideband speech transceiver," *IEEE J. Select. Areas Commun.*, pp. 2363–2372, Nov. 2000.
[50] P. J. Cherriman, T. Keller, and L. Hanzo, "Subband-adaptive turbo-coded OFDM-based interactive video telephony," *IEEE Trans. Circuits Syst. Video Technol.*, submitted for publication.
[51] C. S. Lee, S. Vlahoyiannatos, and L. Hanzo, "Satellite based turbo-coded, blind-equalised 4-QAM and 16-QAM digital video broadcasting," *Proc. IEEE Trans. Broadcasting*, pp. 23–34, Mar. 2000.
[52] L. Hanzo, C. H. Wong, and P. Cherriman, "Burst-by-burst adaptive wideband wireless video telephony," *IEEE Signal Processing Mag.*, vol. 17, pp. 2212–2228, July 2000.

Jason P. Woodard was born in Northern Ireland in 1969. He received the B.A. degree in physics in 1991 from Oxford University, Oxford, U.K., and the M.Sc. degree in electronics in 1992 from Southampton University, Southampton, U.K., where he is currently pursuing the Ph.D. degree in speech coding.

Lajos Hanzo received the Dipl. Ing. degree in electronics in 1976 and the Ph.D. degree in 1983, both from the Technical University of Budapes, Budapest, Hungary.

During his 24-year career in telecommunications, he has held various research and academic posts in Hungary, Germany, and the U.K. Since 1986, he has been with the Department of Electronics and Computer Science, University of Southampton and has been a Consultant to Multiple Access Communications Ltd., U.K. Currently, he holds the chair in Telecommunications. He has coauthored five books on mobile radio communications, published in excess of 300 research papers, organized and chaired conference sessions, presented overview lectures, and was awarded a number of distinctions. Currently he is managing a research team, working on a range of research projects in the field of wireless multimedia communications under the auspices of the Engineering and Physical Sciences Research Council (EPSRC) U.K, the European IST Programme, and the Mobile Virtual Centre of Excellence (VCE). He also provides a range of industrial training courses.