International Journal of Wireless Information Networks, Vol. 8, No. 4, October 2001 (© 2002)

The UMTS Turbo Code and an Efficient Decoder Implementation Suitable for Software-Defined Radios

M. C. Valenti[1] and J. Sun

This paper provides a description of the turbo code used by the UMTS third-generation cellular standard, as standardized by the Third-Generation Partnership Project (3GPP), and proposes an efficient decoder suitable for insertion into software-defined radio architectures or for use in computer simulations. Because the decoder is implemented in software, rather than hardware, single-precision floating-point arithmetic is assumed and a variable number of decoder iterations is not only possible but desirable. Three twists on the well-known log-MAP decoding algorithm are proposed: (1) a linear approximation of the correction function used by the max* operator, which reduces complexity with only a negligible loss in BER performance; (2) a method for normalizing the backward recursion that yields a 12.5% savings in memory usage; and (3) a simple method for halting the decoder iterations based only on the log-likelihood ratios.

KEY WORDS: Coding; turbo codes; WCDMA (UMTS); 3GPP; software-defined radio (SDR).

1. INTRODUCTION

Due to their near Shannon-capacity performance, turbo codes have received a considerable amount of attention since their introduction [1]. They are particularly attractive for cellular communication systems and have been included in the specifications for both the WCDMA (UMTS) and cdma2000 third-generation cellular standards. At this time, the reasons for the superior performance of turbo codes [2,3] and the associated decoding algorithm [4,5] are, for the most part, understood. In addition, several textbooks [6–8] and tutorial papers [9–11] are now available to provide the interested reader with an understanding of the theoretical underpinnings of turbo codes.

The purpose of this paper is neither to explain the phenomenal performance of turbo codes nor to rigorously derive the decoding algorithm. Rather, the purpose is to clearly explain an efficient decoding algorithm suitable for immediate implementation in a software radio receiver. In order to provide a concrete example, the discussion is limited to the turbo code used by the Universal Mobile Telecommunications System (UMTS) specification, as standardized by the Third-Generation Partnership Project (3GPP) [12]. The decoding algorithm is based on the log-MAP algorithm [13], although many parts of the algorithm have been simplified without any loss in performance. In particular, the branch metrics used in the proposed algorithm are much simpler to compute, and the amount of storage is reduced by 12.5% by an appropriate normalization process. Some critical implementation issues are discussed, in particular the

Note: Portions of this paper were presented at the *IEEE International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, San Diego, California, Oct. 2001. This work was supported by the Office of Naval Research under grant N00014-00-0655.

[1] Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, West Virginia, USA 26506-6109. Tel: (304) 293-0405, ext. 2508. Fax: (304) 293-8602. E-mail: mvalenti@wvu.edu

computation of the max* operator and the dynamic halting of the decoder iterations. Simple, but effective, solutions to both of these problems are proposed and illustrated through simulation.

In the description of the algorithm, we have assumed that the reader has a working knowledge of the Viterbi algorithm [14]. Information on the Viterbi algorithm can be found in a tutorial paper by Forney [15] or in most books on coding theory (e.g., [16]) and communications theory (e.g., [17]).

We recommend that the decoder described in this paper be implemented using single-precision floating-point arithmetic on an architecture with approximately 200 kilobytes of memory available for use by the turbo codec. Because mobile handsets tend to be memory limited and cannot tolerate the power inefficiencies of floating-point arithmetic, this may limit the direct application of the proposed algorithm to only base stations. Readers interested in fixed-point implementation issues are referred to [18], while those interested in minimizing memory usage should consider the *sliding-window* algorithm described in [19] and [20].

The remainder of this paper is organized as follows: Section 2 provides an overview of the UMTS turbo code, and Section 3 discusses the channel model and how to normalize the inputs to the decoder. The next three sections describe the decoder, with Section 4 describing the algorithm at the highest hierarchical level, Section 5 discussing the so-called max* operator, and Section 6 describing the proposed log-domain implementation of the MAP algorithm. Simulation results are given in Section 7 for two representative frame sizes (640 and 5114 bits) in both additive white Gaussian noise (AWGN) and fully interleaved Rayleigh flat-fading. Section 8 describes a simple, but effective, method for halting the decoder iterations early, and Section 9 concludes the paper.

2. THE UMTS TURBO CODE

As shown in Fig. 1, the UMTS turbo encoder is composed of two constraint length 4 recursive systematic convolutional (RSC) encoders concatenated in parallel [12]. The feedforward generator is 15 and the feedback generator is 13, both in octal. The number of data bits at the input of the turbo encoder is K, where $40 \leq K \leq 5114$. Data is encoded by the first (i.e., upper) encoder in its natural order and by the second (i.e., lower) encoder after being interleaved. At first, the two switches are in the up position.

The interleaver is a matrix with 5, 10, or 20 rows and between 8 and 256 columns (inclusive), depending on the size of the input word. Data is read into the interleaver in a rowwise fashion (with the first data bit placed in the upper-left position of the matrix). Intrarow permutations are performed on each row of the matrix in accordance with a rather complicated algorithm, which is fully described in the specification [12]. Next, interrow permutations are performed to change the ordering of rows (without changing the ordering of elements within, . . . , each row). When there are 5 or 10 rows, the interrow permutation is simply a reflection about the center row (e.g., for the 5-row case, the rows {1, 2, 3, 4, 5} become rows {5, 4, 3, 2, 1}, respectively). When there are 20 rows, rows {1, . . . , 20} become rows {20, 10, 15, 5, 1, 3, 6, 8, 13, 19, 17, 14, 18, 16, 4, 2, 7, 12, 9, 11}, respectively, when the number of input bits satisfies either $2281 \leq K \leq 2480$ or $3161 \leq K \leq 3210$. Otherwise, they become rows {20, 10, 15, 5, 1, 3, 6, 8, 13, 19, 11, 9, 14, 18, 4, 2, 17, 7, 16, 12}, respectively. After the intrarow and interrow permutations, data is read from the interleaver in a columnwise fashion (with the first output bit being the one in the upper-left position of the transformed matrix).

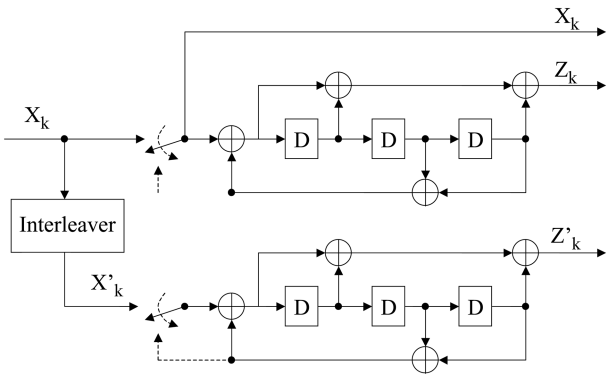


Fig. 1. UMTS turbo encoder.

The data bits are transmitted together with the parity bits generated by the two encoders (the systematic output of the lower encoder is not used and thus not shown in the diagram). Thus, the overall code rate of the encoder is $r = 1/3$, not including the tail bits (discussed below). The first 3 K output bits of the encoder are in the form: $X_1, Z_1, Z_1', X_2, Z_2, Z_2', \ldots, X_K, Z_K, Z_K'$, where X_k is the kth systematic (i.e., data) bit, Z_k is the parity output from the upper (uninterleaved) encoder, and Z_k' is the parity output from the lower (interleaved) encoder.

After the K data bits have been encoded, the trellises of both encoders are forced back to the all-zeros state by the proper selection of tail bits. Unlike conventional convolutional codes, which can always be terminated with a tail of zeros, the tail bits of an RSC will de-

pend on the state of the encoder. Because the states of the two RSC encoders will usually be different after the data has been encoded, the tails for each encoder must be separately calculated and transmitted. The tail bits are generated for each encoder by throwing the two switches into the down position, thus causing the inputs to the two encoders to be indicated by the dotted lines. The tail bits are then transmitted at the end of the encoded frame according to X_{K+1}, Z_{K+1}, X_{K+2}, Z_{K+2}, X_{K+3}, Z_{K+3}, X'_{K+1}, Z'_{K+1}, X'_{K+2}, Z'_{K+2}, X'_{K+3}, Z'_{K+3}, where X represents the tail bits of the upper encoder, Z represents the parity bits corresponding to the upper encoder's tail, X' represents the tail bits of the lower encoder, and Z' represents the parity bits corresponding to the lower encoder's tail. Thus, when tail bits are taken into account, the number of coded bits is $3K + 12$, and the code rate is $K/(3K + 12)$.

3. CHANNEL MODEL

BPSK modulation is assumed, along with either an AWGN or flat-fading channel. The output of the receiver's matched filter is $Y_k = a_k S_k + n_k$, where $S_k = 2X_k - 1$ for the systematic bits, $S_k = 2Z_k - 1$ for the upper encoder's parity bits, $S_k = 2\,Z'_k - 1$ for the lower encoder's parity bits, a_k is the channel gain ($a_k = 1$ for AWGN and is a Rayleigh random variable for Rayleigh flat-fading), n_k is Gaussian noise with variance $\sigma^2 = 1/(2E_s/N_o) = (3K + 12)/(2K(E_b/N_o))$, E_s is the energy per code bit, E_b is the energy per data bit, and N_o is the one-sided noise spectral density.

The input to the decoder is assumed to be in log-likelihood ratio (LLR) form, which assures that the channel gain and noise variance have been properly taken into account. Thus, the input to the decoder is in the form

$$R_k = \ln\left(\frac{P[S_k = +1|Y_k]}{P[S_k = -1|Y_k]}\right) \tag{1}$$

By applying Bayes rule and assuming that $P[S = +1] = P[S = -1]$

$$R_k = \ln\left(\frac{f_Y(Y_k|S_k = +1)}{f_Y(Y_k|S_k = -1)}\right) \tag{2}$$

where $f_Y(Y_k|S_k)$ is the conditional probability density function (pdf) of Y_k given S_k, which is Gaussian with mean $a_k S_k$ and variance σ^2. Substituting the expression for the Gaussian pdf and simplifying yields

$$R_k = \frac{2a_k}{\sigma^2} Y_k \tag{3}$$

Thus, the matched filter coefficients must be scaled by a factor $2a_k/\sigma^2$ before being sent to the decoder. For the remainder of the discussion, the notation $R(X_k)$ denotes the received LLR corresponding to systematic bit X_k, $R(Z_k)$ denotes the received LLR for the upper parity bit Z_k, and $R(Z'_k)$ denotes the received LLR corresponding to the lower parity bit Z'_k.

4. DECODER ARCHITECTURE

The architecture of the decoder is as shown in Fig. 2. As indicated by the presence of a feedback path, the decoder operates in an iterative manner. Each full iteration consists of two half-iterations, one for each constituent RSC code. The timing of the decoder is such that RSC decoder #1 operates during the first half-iteration, and RSC decoder #2 operates during the second half-iteration. The operation of the RSC decoders is described in Section 6.

The value $w(X_k)$, $1 \le k \le K$, is the *extrinsic information* produced by decoder #2 and introduced to the input of decoder #1. Prior to the first iteration, $w(X_k)$ is initialized to all zeros (since decoder #2 has not yet acted on the data). After each complete iteration, the values of $w(X_k)$ will be updated to reflect *beliefs* regarding the data propagated from decoder #2 back to decoder #1. Note that because the two encoders have independent tails, only information regarding the actual data bits is passed between decoders. Thus, $w(X_k)$ is not defined for $K + 1 \le k \le K + 3$ (if it were defined it would simply be equal to zero after every iteration).

The extrinsic information must be taken into account by decoder #1. However, because of the way that the branch metrics are derived, it is sufficient to simply add $w(X_k)$ to the received systematic LLR, $R(X_k)$, which forms a new variable, denoted $V_1(X_k)$. For $1 \le k \le K$, the input to RSC decoder #1 is both the combined systematic data and extrinsic information, $V_1(X_k)$, and the received parity bits in LLR form, $R(Z_k)$. For $K + 1 \le k \le K + 3$ no extrinsic information is available, and thus the

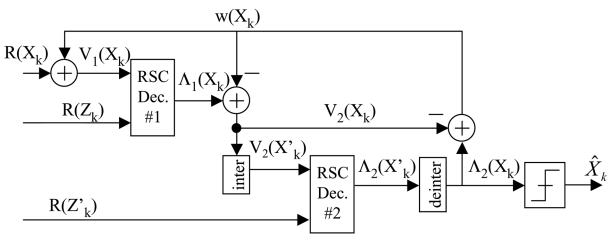


Fig. 2. Proposed turbo decoder architecture.

input to RSC decoder #1 is the received and scaled upper encoder's tail bits, $V_1(X_k) = R(X_k)$, and the corresponding received and scaled parity bits, $R(Z_k)$. The output of RSC decoder #1 is the LLR $\Lambda_1(X_k)$, where $1 \leq k \leq K$ since the LLR of the tail bits is not shared with the other decoder.

By subtracting $w(X_k)$ from $\Lambda_1(X_k)$, a new variable, denoted $V_2(X_k)$, is formed. Similar to $V_1(X_k)$, $V_2(X_k)$ contains the sum of the systematic channel LLR and the extrinsic information produced by decoder #1 (note, however, that the extrinsic information for RSC decoder #1 never has to be explicitly computed). For $1 \leq k \leq K$, the input to decoder #2 is $V_2(X_k')$, which is the interleaved version of $V_2(X_k)$, and $R(Z_k')$, which is the channel LLR corresponding to the second encoder's parity bits. For $K + 1 \leq k \leq K + 3$, the input to RSC decoder #2 is the received and scaled lower encoder's tail bits, $V_2(X_k') = R(X_k')$, and the corresponding received and scaled parity bits, $R(Z_k')$. The output of RSC decoder #2 is the LLR $\Lambda_2(X_k')$, $1 \leq k \leq K$, which is deinterleaved to form $\Lambda_2(X_k)$. The extrinsic information $w(X_k)$ is then formed by subtracting $V_2(X_k)$ from $\Lambda_2(X_k)$ and is fed back to use during the next iteration by decoder #1.

Once the iterations have been completed, a hard bit decision is taken using $\Lambda_2(X_k)$, $1 \leq k \leq K$, where $\hat{X}_k = 1$ when $\Lambda_2(X_k) > 0$ and $\hat{X}_k = 0$ when $\Lambda_2(X_k) \leq 0$.

5. THE MAX* OPERATOR

The RSC decoders in Fig. 2 are each executed using a version of the classic MAP algorithm [21] implemented in the log-domain [13]. As will be discussed in Section 6, the algorithm is based on the Viterbi algorithm [14] with two key modifications: First, the trellis must be swept through not only in the forward direction but also in the reverse direction, and second, the add-compare-select (ACS) operation of the Viterbi algorithm is replaced with the Jacobi logarithm, also known as the max* operator [19]. Because the max* operator must be executed twice for each node in the trellis during each half-iteration (once for the forward sweep, and a second time for the reverse sweep), it constitutes a significant, and sometimes dominant, portion of the overall decoder complexity. The manner that max* is implemented is critical to the performance and complexity of the decoder, and several methods have been proposed for its computation. Below, we consider four versions of the algorithm: log-MAP, max-log-MAP, constant-log-MAP, and linear-log-MAP. The only difference among these algorithms is the manner in which the max* operation is performed.

5.1. Log-MAP Algorithm

With the log-MAP algorithm, the Jacobi logarithm is computed exactly using

$$\begin{aligned} \max^*(x,y) &= \ln(e^x + e^y) \\ &= \max(x,y) + \ln(1 + e^{-|y-x|}) \\ &= \max(x,y) + f_c(|y-x|) \end{aligned} \tag{4}$$

which is the maximum of the function's two arguments plus a nonlinear correction function that is only a function of the absolute difference between the two arguments. The correction function $f_c(|y - x|)$ can be implemented using the log and exp functions in C (or the equivalent in other languages) or by using a large look-up table. The log-MAP algorithm is the most complex of the four algorithms when implemented in software, but as will be shown later, generally offers the best bit error rate (BER) performance. The correction function used by the log-MAP algorithm is illustrated in Fig. 3, along with the correction functions used by the constant-log-MAP and linear-log-MAP algorithms.

5.2. Max-log-MAP Algorithm

With the max-log-MAP algorithm, the Jacobi logarithm is loosely approximated using

$$\max^*(x,y) \approx \max(x,y) \tag{5}$$

i.e., the correction function in (4) is not used at all. The max-log-MAP algorithm is the least complex of the four algorithms (it has twice the complexity of the Viterbi algorithm for each half-iteration) but offers the worst BER performance. The max-log-MAP algorithm has the addi-

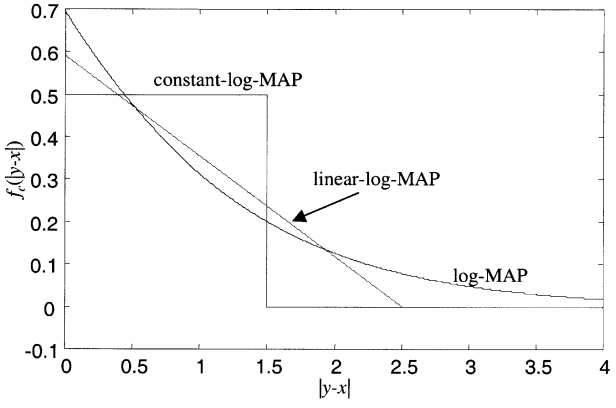


Fig. 3. Correction functions used by log-MAP, linear-log-MAP, and constant-log-MAP algorithms.

tional benefit of being tolerant of imperfect noise variance estimates when operating on an AWGN channel.

5.3. Constant-log-MAP Algorithm

The constant-log-MAP algorithm, first introduced in [22], approximates the Jacobi logarithm using

$$\max^*(x,y) \approx \max(x,y) + \begin{cases} 0 & \text{if } |y-x| > T \\ C & \text{if } |y-x| \le T \end{cases} \quad (6)$$

where it is shown in [23] that the best values for the UMTS turbo code are $C = 0.5$ and $T = 1.5$. This algorithm is equivalent to the log-MAP algorithm with the correction function implemented by a 2-element look-up table. The performance and complexity is between that of the log-MAP and max-log-MAP algorithms.

5.4. Linear-log-MAP Algorithm

The linear-log-MAP algorithm, first introduced in [24], uses the following linear approximation to the Jacobi logarithm:

$$\max^*(x,y) \approx \max(x,y) + \begin{cases} 0 & \text{if } |y-x| > T \\ a(|y-x| - T) & \text{if } |y-x| \le T \end{cases} \quad (7)$$

In [24], the values of the parameters a and T were picked for convenient fixed-point implementation. We are assuming a floating-point processor is available, so a better solution would be to find these parameters by minimizing the total squared error between the exact correction function and its linear approximation. Performing this minimization, which is detailed in the Appendix, yields $a = -0.24904$ and $T = 2.5068$. The linear-log-MAP algorithm offers performance and complexity between that of the log-MAP and constant-log-MAP algorithms. As will be shown in the simulation results, a key advantage of the linear-log-MAP algorithm is that it converges faster than constant-log-MAP.

6. MAP ALGORITHM IN THE LOG DOMAIN

Each of the two RSC decoders in Fig. 2 operates by sweeping through the code trellis twice, once in each of the forward and reverse directions. Each sweep uses a modified version of the Viterbi algorithm to compute partial path metrics, where the modifications is that the ACS operations are replaced with the max* operator. During the second sweep, an LLR value is computed for each stage of the trellis and its corresponding data bit.

Two key observations should be pointed out before going into the details of the algorithm: (1) It does not matter whether the forward sweep or the reverse sweep is performed first; and (2) while the partial path metrics for the entire first sweep (forward or backward) must be stored in memory, they do not need to be stored for the entire second sweep. This is because the LLR values can be computed during the second sweep, and thus partial path metrics for only two stages of the trellis (the current and previous stages) must be maintained during the second sweep.

Because of these observations, we recommend sweeping through the trellis in the reverse direction first. While performing this sweep, the partial path metric at each node in the trellis must be saved in memory (with an exception noted below). After completing the reverse sweep, the forward sweep can proceed. As the forward sweep is performed, LLR estimates of the data can be produced. Because the LLR estimates are produced during the forward sweep, they are output in the correct ordering (if the forward sweep was completed first, then the LLRs would be produced during the reverse sweep and would therefore be in reversed order).

6.1. Trellis Structure and Branch Metrics

The trellis of the RSC encoder used by the UMTS turbo code is shown in Fig. 4. Solid lines indicate data $X_k = 1$ and dotted lines indicate data $X_k = 0$. The branch metric associated with the branch connecting states S_i (on the left) and S_j (on the right) is $\gamma_{ij} = V(X_k)X(i,j) + R(Z_k)Z(i,j)$, where $X(i,j)$ is the data bit associated with the branch and $Z(i,j)$ is the parity bit associated with the branch. Because the RSC encoder is rate 1/2, there are only four distinct branch metrics:

$$\begin{aligned} \gamma_0 &= 0 \\ \gamma_1 &= V(X_k) \\ \gamma_2 &= R(Z_k) \\ \gamma_3 &= V(X_k) + R(Z_k) \end{aligned} \quad (8)$$

where for decoder #1 $V(X_k) = V_1(X_k)$ and for decoder #2 $V(X_k) = V_2(X'_k)$ and $R(Z_k) = R(Z'_k)$.

6.2. Backward Recursion

The proposed decoder begins with the backward recursion, saving normalized partial path metrics at all the nodes in the trellis (with an exception noted below),

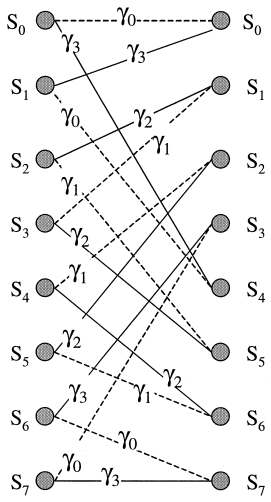


Fig. 4. Trellis section for the RSC code used by the UMTS turbo code. Solid lines indicate data 1 and dotted lines indicate data 0. Branch metrics are indicated.

which will later be used to calculate the LLRs during the forward recursion. The backward partial path metric for state S_i at trellis stage k is denoted $\beta_k(S_i)$, with $2 \leq k \leq K + 3$ and $0 \leq i \leq 7$. The backward recursion is initialized with $\beta_{K+3}(S_0) = 0$ and $\beta_{K+3}(S_i) = -\infty \; \forall i > 0$.

Beginning with stage $k = K + 2$ and proceeding through the trellis in the backward direction until stage[2] $k = 2$, the partial path metrics are found according to

$$\tilde{\beta}_k(S_i) = \max{}^*\left\{\left(\beta_{k+1}(S_{j_1}) + \gamma_{ij_1}\right), \left(\beta_{k+1}(S_{j_2}) + \gamma_{ij_2}\right)\right\} \quad (9)$$

where the tilde above $\beta_k(S_i)$ indicates that the metric has not yet been normalized and S_{j_1} and S_{j_2} are the two states at stage $k + 1$ in the trellis that are connected to state S_i at stage k. After the calculation of $\tilde{\beta}_k(S_0)$, the partial path metrics are normalized according to

$$\beta_k(S_i) = \tilde{\beta}_k(S_i) - \tilde{\beta}_k(S_0) \quad (10)$$

Because after normalization $\beta_k(S_0) = 0 \; \forall k$, only the other seven normalized partial path metrics $\beta_k(S_i)$, $1 \leq i \leq 7$, need to be stored. This constitutes a 12.5% savings in memory relative to either no normalization or other common normalization techniques (such as subtracting by the largest metric).

6.3. Forward Recursion and LLR Calculation

During the forward recursion, the trellis is swept through in the forward direction in a manner similar to the Viterbi algorithm. Unlike the backward recursion, only the partial path metrics for two stages of the trellis must be maintained: the current stage k and the previous stage $k - 1$. The forward partial path metric for state S_i at trellis stage k is denoted $\alpha_k(S_i)$, with $0 \leq k \leq K - 1$ and $0 \leq i \leq 7$. The forward recursion is initialized by setting $\alpha_0(S_0) = 0$ and $\alpha_0(S_i) = -\infty \; \forall i > 0$.

Beginning with stage $k = 1$ and proceeding through the trellis in the forward direction until stage[3] $k = K$, the unnormalized partial path metrics are found according to

$$\tilde{\alpha}_k(S_j) = \max{}^*\{(\alpha_{k-1}(S_{i1}) + \gamma_{i1_j}), (\alpha_{k-1}(S_{i2}) + \gamma_{i2_j})\} \quad (11)$$

where S_{i1} and S_{i2} are the two states at stage $k - 1$ that are connected to state S_j at stage k. After the calculation of $\tilde{\alpha}_k(S_0)$, the partial path metrics are normalized using

$$\alpha_k(S_i) = \tilde{\alpha}_k(S_i) - \tilde{\alpha}_k(S_0) \quad (12)$$

As the αs are computed for stage k, the algorithm can simultaneously obtain an LLR estimate for data bit X_k. This LLR is found by first noting that the likelihood of the branch connecting state S_i at time $k - 1$ to state S_j at time k is

$$\lambda_k(i, j) = \alpha_{k-1}(S_i) + \gamma_{ij} + \beta_k(S_j) \quad (13)$$

The likelihood of data 1 (or 0) is then the Jacobi logarithm of the likelihood of all branches corresponding to data 1 (or 0), and thus:

$$\Lambda(X_k) = \max_{(S_i \to S_j):X_i=1}{}^*\{\lambda_k(i, j)\} - \max_{(S_i \to S_j):X_i=0}{}^*\{\lambda_k(i, j)\} \quad (14)$$

where the max* operator is computed recursively over the likelihoods of all data 1 branches $\{(S_i \to S_j):X_i = 1\}$ or data 0 branches $\{(S_i \to S_j):X_i = 0\}$. Once $\Lambda(X_k)$ is calculated, $\alpha_{k-1}(S_i)$ is no longer needed and may be discarded.

7. SIMULATION RESULTS

Simulations were run to illustrate the performance of all four variants of the decoding algorithm. Two representative frame/interleaver sizes were used, K = 640 and K = 5114 bits. For the smaller interleaver, up to 10 decoder iterations were performed, while for the larger interleaver, up to 14 decoder iterations were performed. To speed up the simulations, the decoder was halted once all of the errors were corrected (the next section discusses practical ways to halt the decoder). Results

[2] The backward metrics $\beta_k(S_i)$ at stage $k = 1$ are never used and therefore do not need to be computed.

[3] Note that α_k does not need to be computed for $k = K$ (it is never used), although the LLR $\Lambda(X_k)$ must still be found.

for both AWGN and fully interleaved Rayleigh flat-fading channels were produced. All four algorithms were implemented in C, with the log-MAP algorithm implementing (3) using log and exp function calls. In order to present a fair comparison, all four algorithms decoded the same received code words, and thus the data, noise, and fading were the same for each family of four curves. Enough trials were run to generate 100 frame errors for the best algorithm (usually log-MAP) at each value of E_b/N_o (more errors were logged for the other algorithms because the same received frames were processed by all four algorithms). This translates to a 95% confidence interval of $(1.25\hat{p}, 0.8\hat{p})$ for the worst-case estimate of the frame error rate (FER) (the confidence interval will be slightly tighter for the BER) [25]. Because the same received code word was decoded by all four algorithms, and because such a large number of independent error events were logged, any difference in performance among algorithms is due primarily to the different correction functions that were used, rather than to the vagaries of the Monte Carlo simulation.

7.1. BER and FER Performance

The bit error rate (BER) is shown for the K = 640 bit UMTS turbo code in Fig. 5 and for the K = 5114 bit code in Fig. 6. Likewise, the frame error rate (FER) is shown in Figs. 7 and 8 for the 640 and 5114 bit codes, respectively. The E_b/N_o required to achieve a BER of 10^{-5} is tabulated in Table I. In each case, the performance of max-log-MAP is significantly worse than the other algorithms, requiring between 0.3 to 0.54 dB higher E_b/N_o than the log-MAP algorithm. The gap between max-log-MAP and log-MAP is about 0.13 dB wider for fading than it is for AWGN, and about 0.1 dB wider for the K = 5114 bit code than it is for the K = 640 bit code. The performance of both constant-log-MAP and linear-log-MAP are close to that of the exact computation of the log-MAP algorithm. The constant-log-MAP algorithm is between 0.02 and 0.03 dB worse than log-MAP, regardless of channel or frame size. The linear-log-MAP shows performance that is almost indistinguishable from log-MAP, with performance ranging from 0.01 dB worse to 0.01 dB better than log-MAP.

The fact that linear-log-MAP can sometimes be slightly better than log-MAP is an interesting and unexpected result. At first, one might infer that because this discrepancy is within the confidence intervals, then it is simply due to the random fluctuations of the Monte Carlo simulation. However, the simulation was carefully constructed such that the same received frames were decoded by all four algorithms. Thus, there must be a different reason for this phenomenon. We believe the reason for this discrepancy is as follows: although each of the two MAP decoders shown in Fig. 2 is optimal in terms of minimizing the "local" BER, the overall turbo decoder is not guaranteed to minimize the "global" BER. Thus, a slight random perturbation in the computed partial path metrics and corresponding LLR values could result in a perturbation in the BER. The error caused by the linear-log-MAP approximation to the Jacobi algorithm induces such a random perturbation both within the algorithm and into the BER curve. Note that this perturbation is very minor and the performance of

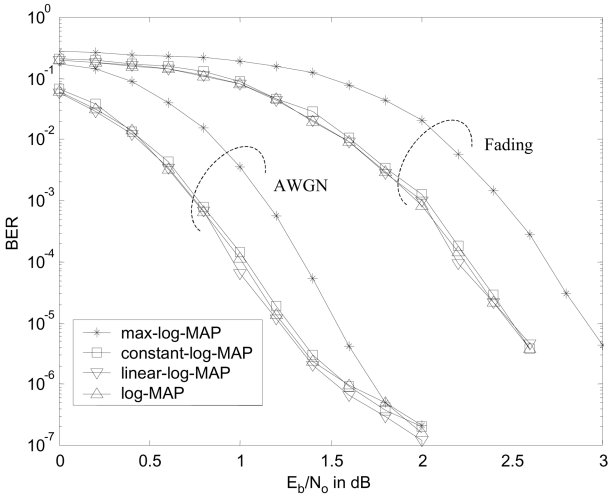


Fig. 5. BER of K = 640 UMTS turbo code after 10 decoder iterations.

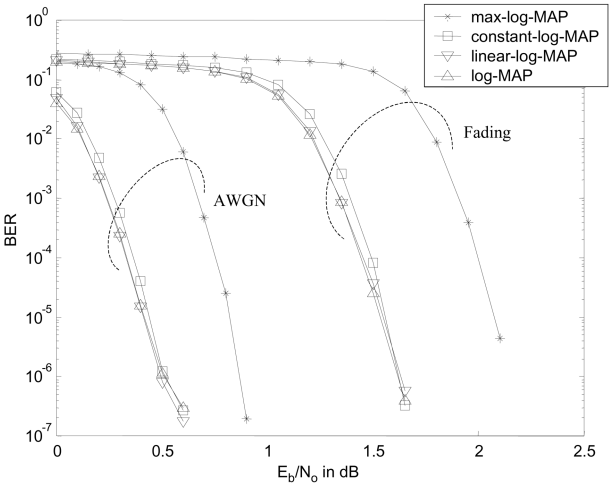


Fig. 6. BER of K = 5114 UMTS turbo code after 14 decoder iterations.

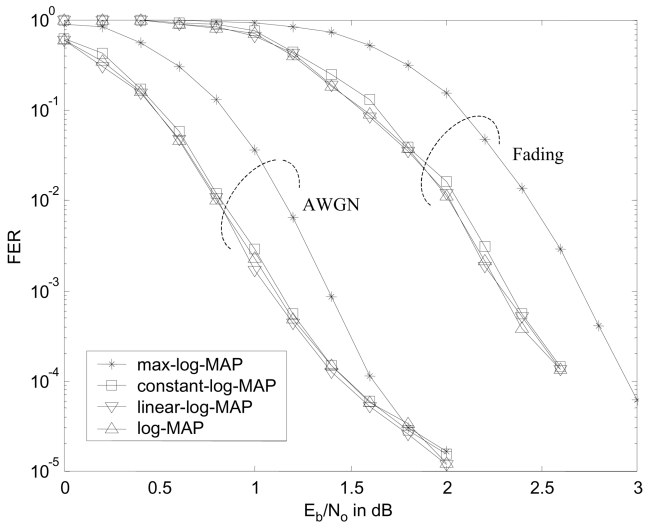


Fig. 7. FER of K = 640 UMTS turbo code after 10 decoder iterations.

linear-log-MAP is always within 0.1 dB of the log-MAP algorithm.

If the simulations were run to a much lower BER, an error floor would begin to appear [3]. The beginning of a floor can be seen in the simulation of the K = 640 bit code in AWGN. In the floor region, all four algorithms will perform roughly the same. It can be seen in Figs. 5 and 7 that the algorithms are beginning to converge as the BER and FER curves begin to flare into a floor. Thus, while the choice of algorithm has a critical influence on performance at low signal-to-noise ratio (SNR), the choice becomes irrelevant at high SNR.

Table I. E_b/N_o Required for the UMTS Turbo Code to Achieve a BER of 10^{-5}

	AWGN		Fading	
Algorithm	K = 640	K = 5114	K = 640	K = 5114
max-log-MAP	1.532 dB	0.819 dB	2.916 dB	2.073 dB
constant-log-MAP	1.269 dB	0.440 dB	2.505 dB	1.557 dB
linear-log-MAP	1.220 dB	0.414 dB	2.500 dB	1.547 dB
log-MAP	1.235 dB	0.417 dB	2.488 dB	1.533 dB

This suggests that in a software implementation, perhaps the algorithm choice should be made adaptive (e.g., choose linear-log-MAP at low SNR and max-log-MAP at high SNR).

7.2. Average Number of Iterations

The average number of iterations required for each algorithm to converge (i.e., correct all the errors in a frame) is shown in Fig. 9 for the 640-bit code and Fig. 10 for the 5114-bit code. A value of 11 iterations for the smaller code and 15 iterations for the larger code indicates that the algorithm does not converge. In all cases, the max-log-MAP algorithm requires more decoder iterations than the other algorithms at any particular value of E_b/N_o. The other three algorithms require roughly the same number of iterations, with the constant-log-MAP algorithm requiring slightly more iterations than the linear-log-MAP or log-MAP algorithms. As with the BER and FER curves, the distinction among algorithms becomes less pronounced at higher SNR as the error

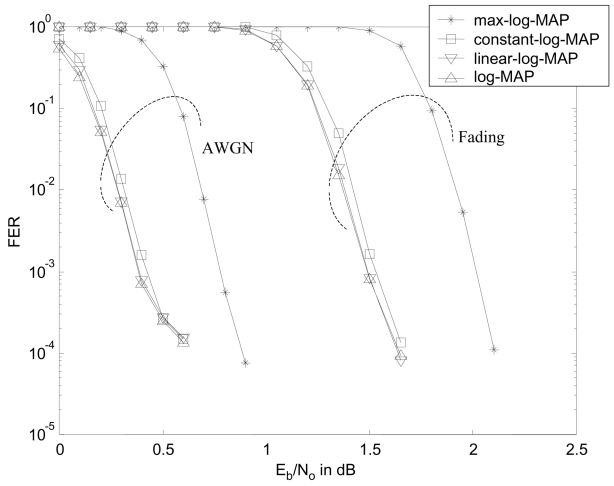


Fig. 8. FER of K = 5114 UMTS turbo code after 14 decoder iterations.

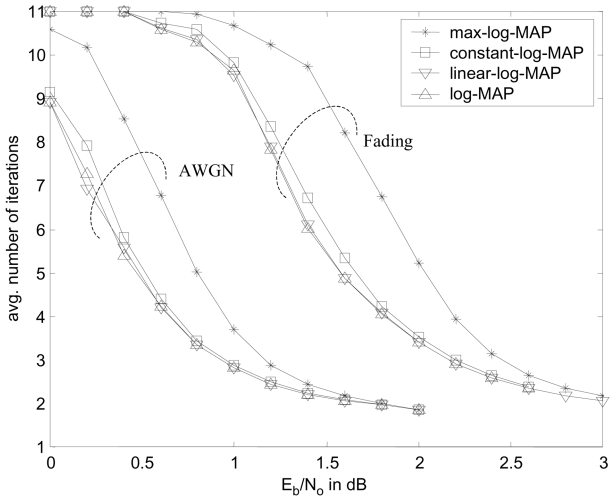


Fig. 9. Average number of decoder iterations required for the K = 640 UMTS turbo code to converge.

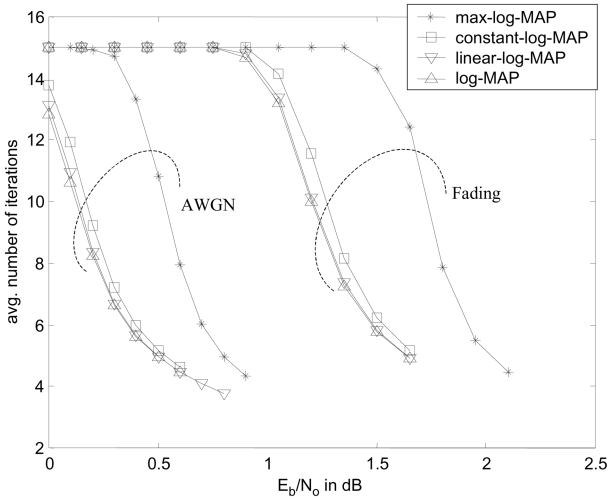


Fig. 10. Average number of decoder iterations required for the K = 5114 UMTS turbo code to converge.

curves begin to reach the error-floor region. However, for sufficiently low SNR, we found that in AWGN the max-log-MAP takes about two more iterations to converge for the smaller code and about six more iterations for the larger code (with even more iterations required in Rayleigh fading). The constant-log-MAP algorithm typically requires about more iterations than log-MAP, while linear-log-MAP requires the same number of iterations as log-MAP.

7.3. Processing Rate

The simulations were run on a PC with a 933-MHz Pentium III and the Windows 2000 operating system. The average throughput, measured in bits per second (bps) per iteration is listed in Table II. Clearly, the log-MAP algorithm is the least-efficient algorithm, requiring more than seven times the processing power of the max-log-MAP algorithm, which is the fastest algorithm per iteration. As the only difference between log-MAP and max-log-MAP is the calculation of the correction function, it stands to reason that calculating the correction function using the log and exp function calls accounts for over 7/8 = 85% of the complexity of the log-MAP decoder. The other three algorithms required roughly the same complexity, with max-log-MAP offering the highest throughput per iteration.

Note that the algorithm with the highest throughput per iteration will not necessarily be the algorithm with the highest overall throughput. For instance, at E_b/N_o = 0.5 dB, the K = 5114 code received over an AWGN channel requires an average of 10.8 iterations of the max-log-MAP algorithm, 5.2 iterations of the constant-log-MAP algorithm, and 4.85 iterations of each of the log-MAP and linear-log-MAP algorithm. This implies that the overall throughput of the max-log-MAP algorithm will only be about 34 kbps, while the constant-log-MAP and linear-log-MAP algorithms will offer an overall throughput of 57 and 54 kbps, respectively. Thus, it appears that the constant-log-MAP and linear-log-MAP algorithms offer the best tradeoff in terms of complexity and performance, with the linear-log-MAP algorithm offering slightly better error rate performance at the cost of slightly lower overall throughput.

Table II. Processing Rate of the Algorithms Running on a 933-MHz P3

Algorithm	Throughput
max-log-MAP	366 kbps/iteration
constant-log-MAP	296 kbps/iteration
linear-log-MAP	262 kbps/iteration
log-MAP	51 kbps/iteration

7.4. Sensitivity to Noise Variance Estimation Errors

With the exception of the max-log-MAP algorithm, the log-MAP algorithm and its approximations require knowledge of the noise variance σ^2. In [23], it is shown that one of the disadvantages of the constant-log-MAP algorithm is that it is rather sensitive to errors in the noise variance estimate. We tested the sensitivity of the proposed algorithm to noise variance estimate errors by giving the decoder an estimate of $\hat{\sigma}^2 = \epsilon\sigma^2$ of the true variance σ^2. We varied ϵ from 0.1 to 2.0 (with 1.0 indicating a perfect estimate of the noise variance), and plotted the results for the K = 5114 bit turbo code operating in AWGN in Fig. 11 for two values of E_b/N_o and all four algorithms. This figure indicates that, at least for the 5114-bit UMTS turbo code, all three algorithms behave similarly in the presence of noise variance estimate errors, with the constant-log-MAP consistently worse than the log-MAP and linear-log-MAP algorithms (which had similar performance to one another).

8. DYNAMIC HALTING CONDITION

The simulation results from the previous section assumed that the decoder halted as soon as it converged, i.e., when all the errors in the frame were corrected. This requires knowledge of the data, which is available when

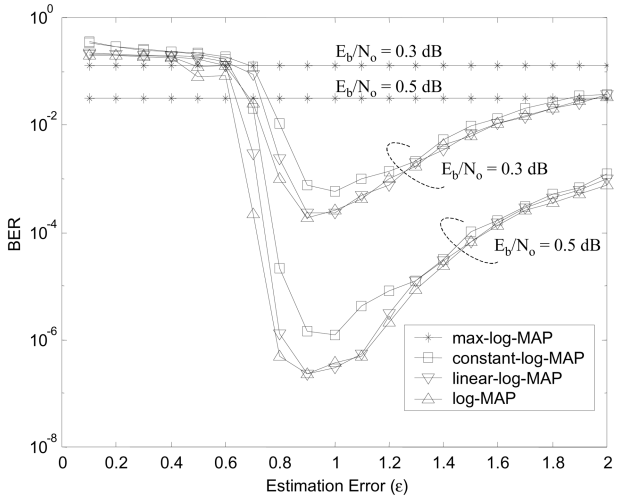


Fig. 11. Sensitivity of the K = 5114 UMTS turbo code to noise variance estimation errors in AWGN.

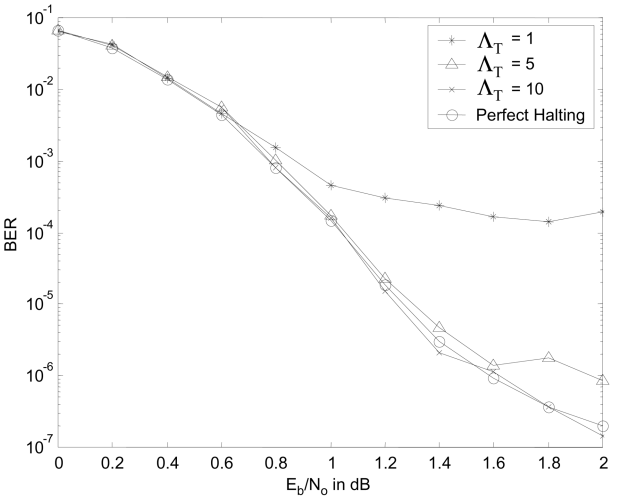


Fig. 12. BER of K = 640 UMTS turbo code with constant-log-MAP decoding in AWGN with various halting thresholds.

running a computer simulation. However, in practice, the decoder will not have knowledge of the data, and thus a blind method for halting the iterations must be employed. Because the decoder rarely requires the maximum number of iterations to converge, using an early stopping criterion will allow a much greater throughput in a software radio implementation.

Several early stopping criteria have been proposed based on the cross entropy between iterations or on the sign-difference ratio [26]. The decoder considered here uses a simpler, but still effective, stopping criteria based only on the log-likelihood ratio. The decoder stops once the absolute value of all of the LLRs are above a threshold, Λ_T; i.e., the decoder halts once

$$\min_{1 \le k \le K} \{|\Lambda_2(X_k)|\} > \Lambda_T \tag{15}$$

The performance of the stopping condition is highly dependent on the choice of Λ_T. If it is too small, then the decoder will tend to not perform enough iterations and BER performance will suffer. If, however, it is too large, then the decoder will tend to overiterate, and the throughput will suffer.

The K = 640-bit UMTS turbo code was simulated in AWGN using both ideal halting (i.e., halt once the decoder converges) and halting using various values for Λ_T. The decoder used a maximum of 10 iterations of the constant-log-MAP algorithm, and each curve was generated using the same received code words. BER results are shown in Fig. 12, FER results are shown in Fig. 13, and the average number of decoder iterations is shown in Fig. 14. As can be seen, $\Lambda_T = 1$ and $\Lambda_T = 5$ are too small and raise the BER floors, while $\Lambda_T = 10$ raises the FER floor only slightly and has only a negligible effect on the BER floor. Using the threshold $\Lambda_T = 10$ requires, on average, less than one extra iteration compared to ideal halting.

It is interesting to note that the BER is sometimes lower with $\Lambda_T = 10$ than with ideal halting. The reason for this is as follows: The number of errors at the output of a turbo decoder will sometimes oscillate from one iteration to the next [6]. If the received code word is too corrupted to successfully decode, "ideal halting" will al-

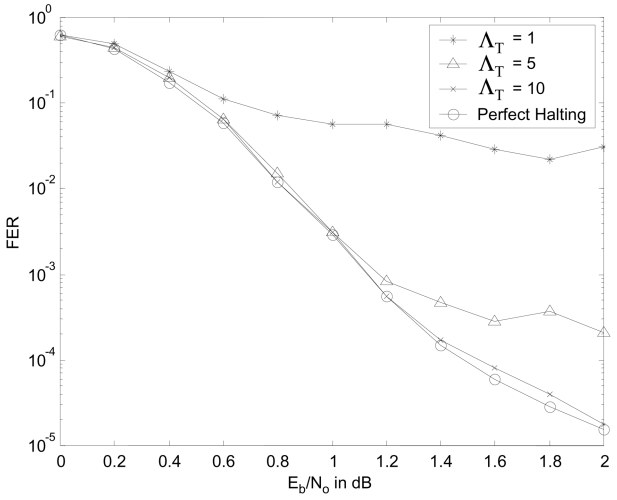


Fig. 13. FER of K = 640 UMTS turbo code with constant-log-MAP decoding in AWGN with various halting thresholds.

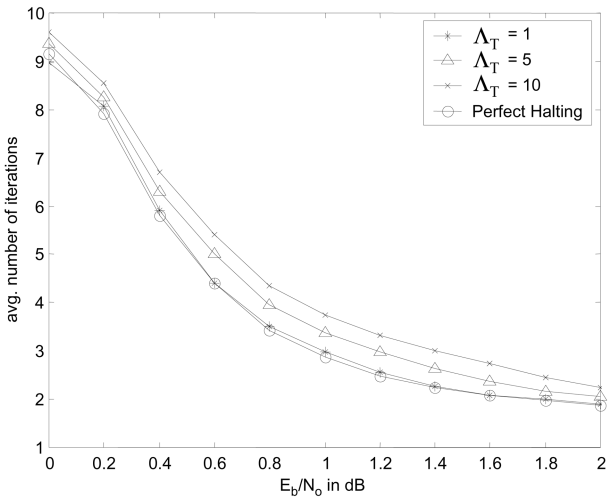


Fig. 14. Average number of decoder iterations required for the K = 640 UMTS turbo code to converge in AWGN using constant-log-MAP decoding and various halting thresholds.

ways run the full number of iterations; thus, the number of bit errors will be dictated by the performance at the last iteration, which due to the oscillatory nature of the decoder, could be quite high. On the other hand, the early-halting decoder will stop the iterations when the LLRs are high, even if the BER is not identically zero. Thus, although early halting cannot lower the FER, it can lower the BER by having fewer bit errors when there is a frame error.

9. CONCLUSIONS

This paper has discussed a complete turbo decoding algorithm that can be implemented directly in software. To provide a concrete example, the discussion focused on the turbo code in the UMTS specification. However, the same algorithm can be used for any BPSK- or QPSK-modulated turbo code, including that used by cdma2000 and by the CCSDS deep-space telemetry standard. Note, however, that if the modulation does not have a constant phase (e.g., QAM modulation), the input LLR normalization process and branch metrics discussed in this paper must be modified.

In addition to providing an overview of the UMTS turbo code and a generic decoder implementation, three aspects regarding turbo codec implementation have been studied in this paper. First, a simple, but effective, linear approximation to the Jacobi logarithm was proposed. Simulation results that this approximation offers better performance and faster convergence than the constant-log-MAP algorithm at the expense of only a modest increase in complexity. Second, a method for normalizing the partial path metrics was proposed that eliminates the need to store the metrics for state S_0. Finally, a method for halting the decoder iterations based only on the current value of the LLRs was proposed and shown through simulation to require only one more decoding iteration compared to ideal halting.

APPENDIX

The parameters used by the linear approximation to the Jacobi logarithm are chosen to minimize the total squared error between the true function (3) and its approximation (7):

$$\Phi(a,T)=\int_0^T\left[a(x-T)-\ln(1+e^{-x})\right]^2dx + \int_T^{\infty}\left[\ln(1+e^{-x})\right]^2dx \quad (16)$$

This function is minimized by setting the partial derivatives with respect to a and T equal to zero. First, take the partial with respect to *a:*

$$\begin{aligned}\frac{\partial\Phi(a,\ T)}{\partial a} &= \int_0^T 2[a(x-T)-\ln(1+e^{-x})](x-T)\,dx\\ &= 2\int_0^T a(x-T)^2\ dx - 2\int_0^T(x-T)\ln(1+e^{-x})dx\\ &= \frac{2}{3}a(x-T)^3\Big|_0^T - 2\int_0^T(x-T)\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n}e^{-nx}\ dx\\ &= \frac{2}{3}aT^3 - 2\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n}\int_0^T(x-T)e^{-nx}\ dx\\ &= \frac{2}{3}aT^3 + 2\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^2}\left[T+\left(\frac{1}{n}\right)(e^{-nT}-1)\right]\\ &= \frac{2}{3}aT^3 + 2K_1T - 2K_2 + 2\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^3}e^{-nT} \quad (17)\end{aligned}$$

where K_1 and K_2 are constants:

$$K_1=\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^2}=\frac{\pi^2}{12}$$
$$K_2=\sum_{n=1}^{\infty}\frac{(-1)^{n+1}}{n^3}$$

Next, take the partial derivative of (16) with respect to *T:*

$$\begin{aligned}\frac{\partial\Phi(a,T)}{\partial T} &= \int_0^T 2[a(x-T)-\ln(1+e^{-x})](-a)\,dx\\ &\quad + [a(x-T)-\ln(1+e^{-x})]^2\big|_{x=T}\\ &\quad - [\ln(1+e^{-x})]^2\big|_{x=T}\\ &= \int_0^T 2[a(x-T)-\ln(1+e^{-x})]\ (-a)\,dx \quad (18)\end{aligned}$$

Because the above expression is being set to zero, it can be divided by $-2a$:

$$\begin{aligned}\int_0^T [a(x-T) - \ln(1+e^{-x})]dx &= \int_0^T a(x-T)dx - \int_0^T \ln(1+e^{-x})\,dx \\ &= \frac{a}{2}(x-T)^2\Big|_0^T - \int_0^T \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} e^{-nx}\,dx \\ &= -\frac{a}{2}T^2 - \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n} \int_0^T e^{-nx}\,dx \\ &= -\frac{a}{2}T^2 - \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n^2} [e^{-nT} - 1] \\ &= -\frac{a}{2}T^2 - K_1 + \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n^2} e^{-nT} \end{aligned} \quad (19)$$

Multiply (17) by 3/2 and add to $2T$ times (19) to obtain

$$g(T) = K_1 T - 3K_2 + 3\sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n^3} e^{-nT} + 2T \sum_{n=1}^{\infty} \frac{(-1)^{n+1}}{n^2} e^{-nT} \quad (20)$$

By setting the above monotonically increasing function of T to zero, we arrive at the optimal value for T. However, because T is embedded in a sum of exponentials, an iterative approach to solving for T must be taken. The iterative solution is found by first choosing two values T_1 and T_2 that satisfy $g(T_1) < 0$ and $g(T_2) > 0$. Let T_0 be the midpoint between T_1 and T_2. If $g(T_0) < 0$, then set $T_1 = T_0$; otherwise set $T_2 = T_0$ and repeat the iteration until T_1 and T_2 are very close. Although the upper limit of the summations is infinity, an error of less than 10^{-10} results if the upper limit is truncated to 30. By iteratively solving (20) with the upper limit of the summations set to 30, we find

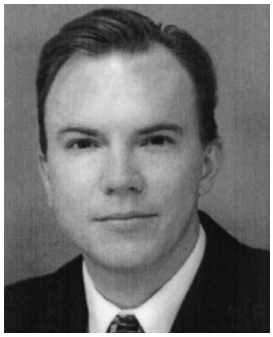
$$T = 2.50681640022001 \quad (21)$$

Once the optimal value of T has been found, a can easily be found by setting either (17) or (19) equal to zero and solving for a, which results in

$$a = -0.24904181891710 \quad (22)$$

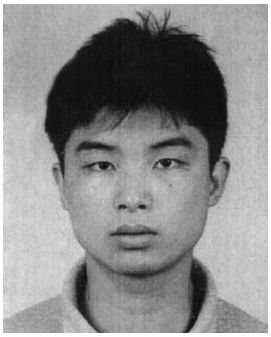
REFERENCES

1. C. Berrou, A. Glavieux, and P. Thitimasjshima, Near Shannon limit error-correcting coding and decoding: Turbo-codes(1), *Proc. IEEE Int. Conf. on Commun.* (Geneva, Switzerland), pp. 1064–1070, May 1993.
2. 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.
3. L. C. Perez, J. Seghers, and D. J. Costello, A distance spectrum interpretation of turbo codes, *IEEE Trans. Inform. Theory*, Vol. 42, pp. 1698–1708, Nov. 1996.
4. D. Divsalar, S. Dolinar, and F. Pollara, Iterative turbo decoder analysis based on density evolution, *IEEE J. Select. Areas Commun.*, Vol. 19, pp. 891–907, May 2001.
5. S. ten Brink, Convergence behavior of iteratively decoded parallel concatenated codes, *IEEE Trans. Commun.*, Vol. 49, pp. 1727–1737, Oct. 2001.
6. C. Heegard and S. B. Wicker, *Turbo Coding*, Kluwer Academic Publishers, Dordrecht, the Netherlands, 1999.
7. B. Vucetic and J. Yuan, *Turbo Codes: Principles and Applications*, Kluwer Academic Publishers, Dordrecht, the Netherlands, 2000.
8. K. Chugg, A. Anastasopoulos, and X. Chen, *Iterative Detection: Adaptivity, Complexity Reduction, and Applications*, Kluwer Academic Publishers, Dordrecht, the Netherlands, 2001.
9. J. Hagenauer, The turbo principle: Tutorial introduction and state of the art, *Proc., Int. Symp. on Turbo Codes and Related Topics* (Brest, France), pp. 1–11, Sept. 1997.
10. B. Sklar, A primer on turbo code concepts, *IEEE Commun. Magazine*, Vol. 35, pp. 94–102, Dec. 1997.
11. M. C. Valenti, Turbo codes and iterative processing, *IEEE New Zealand Wireless Communications Symposium* (Auckland, New Zealand), Nov. 1998.
12. European Telecommunications Standards Institute, Universal mobile telecommunications system (UMTS): Multiplexing and channel coding (FDD), *3GPP TS 125.212 version 3.4.0*, pp. 14–20, Sept. 23, 2000.
13. P. Robertson, P. Hoeher, and E. Villebrun, Optimal and sub-optimal maximum a posteriori algorithms suitable for turbo decoding, *European Trans. on Telecommun.*, Vol. 8, pp. 119–125, Mar./Apr. 1997.
14. A. J. Viterbi, Error bounds for convolutional codes and an asymptotically optimum decoding algorithm, *IEEE Trans. Inform. Theory*, Vol. 13, pp. 260–269, Apr. 1967.
15. G. D. Forney, The Viterbi algorithm, *Proc. IEEE*, Vol. 61, pp. 268–278, Mar. 1973.
16. S. Wicker, *Error Control Systems for Digital Communications and Storage*, Prentice Hall, Englewood Cliffs, NJ, 1995.
17. J. Proakis, *Digital Communications*, 4th ed., McGraw-Hill, New York, 2001.
18. T. Blankenship and B. Classon, Fixed-point performance of low-complexity turbo decoding algorithms, *Proc. IEEE Veh. Tech. Conf. (VTC)* (Rhodes, Greece), May 2001.
19. A. J. Viterbi, An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes, *IEEE J. Select. Areas Commun.*, Vol. 16, pp. 260–264, Feb. 1998.
20. M. Marandian, M. Salehi, J. Fridman, and Z. Zvonar, Performance analysis of turbo decoder for 3GPP standard using the sliding window algorithm, *Proc. IEEE Personal Indoor and Mobile Radio Commun. Conf.* (San Diego, CA), Oct. 2001.
21. 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. 20, pp. 284–287, Mar. 1974.
22. W. J. Gross and P. G. Gulak, Simplified MAP algorithm suitable for implementation of turbo decoders, *Electronics Letters*, Vol. 34, pp. 1577–1578, Aug. 6, 1998.
23. B. Classon, K. Blankenship, and V. Desai, Turbo decoding with the constant-log-MAP algorithm, *in Proc., Second Int. Symp. Turbo Codes and Related Appl.* (Brest, France), pp. 467–470, Sept. 2000.
24. J.-F. Cheng and T. Ottosson, Linearly approximated log-MAP algorithms for turbo coding, *Proc. IEEE Veh. Tech. Conf. (VTC)* (Houston, TX), May 2000.
25. M. C. Jeruchim, P. Balaban, and K. S. Shanmugan, *Simulation of Communication Systems: Modeling, Methodology, and Techniques*, 2nd ed., Kluwer Academic Publishers, Dordrecht, the Netherlands, 2001.
26. Y. Wu, B. D. Woerner, and W. J. Ebel, A simple stopping criterion for turbo decoding, *IEEE Commun. Letters*, Vol. 4, pp. 258–260, Aug. 2000.



Matthew C. Valenti received a B.S.E.E. in 1992 from Virginia Tech (Blacksburg, VA), a M.S.E.E. in 1995 from the Johns Hopkins University (Baltimore, MD), and a Ph.D. in electrical engineering in 1999 from Virginia Tech, where he was a Bradley Fellow.

He is currently an assistant professor in the Lane Department of Computer Science and Electrical Engineering at West Virginia University (Morgantown, WV). His research interests are in the areas of communication theory, error correction coding, applied information theory, and wireless multiple-access networks. He also acts as a consultant to several companies engaged in various aspects of turbo codec design, including software radio, FPGA, and ASIC implementations for military, satellite, and third-generation cellular applications. Prior to attending graduate school at Virginia Tech, he was an electronics engineer at the United States Naval Research Laboratory, Washington, DC, where he was engaged in the design and development of a spaceborne adaptive antenna array and a system for the collection and correlation of maritime ELINT signals.



Jian Sun received his B.S.E.E. in 1997 and M.S.E.E. in 2000, both from Shanghai Jiaotong University (Shanghai, China). He is currently pursuing a Ph.D. in the Lane Department of Computer Science and Electrical Engineering at West Virginia University (Morgantown, WV). His research interests are in wireless communications, wireless networks, and DSP applications.