A Fast and Stable Direct-MAP Implementation of Turbo Decoding

Michael Bandsmer, T. Aaron Gulliver and Vijay K. Bhargava
Dept. of Electrical and Computer Engineering
University of Victoria, P.O. Box 3055, STN CSC, Victoria, BC V8R 3P6
mbandsme@ece.uvic.ca, agullive@ece.uvic.ca, bhargava@ece.uvic.ca

Abstract— **Most implementations of turbo decoding make use of the so-called "log-MAP algorithm", which has been widely viewed in the literature to be faster and more numerically stable than a "direct-MAP" implementation. In this paper, it is demonstrated that the direct-MAP approach can be made numerically stable, and that the iterative process for this direct-MAP implementation is faster than the log-MAP approach on modern microprocessors, even if a lookup table is used to simplify the log-MAP calculations. Furthermore, since the direct-MAP algorithm does not rely on any approximations, the bit error rate performance of the direct-MAP algorithm is slightly better than that produced using the log-MAP algorithm.**

I. Introduction

Turbo codes, first introduced by Berrou, Glavieux, and Thitimajshima [1], are produced by the parallel concatenation of two separate convolutional encoders, C_1 and C_2. A sequence of message bits $\boldsymbol{m}$ is encoded by convolutional encoder C_1 to produce a parity sequence $\boldsymbol{p}$. The same message bits are also permuted by an interleaver, and encoded by convolutional encoder C_2 to produce a second parity sequence $\boldsymbol{q}$. The message bits and the two parity streams are then multiplexed to produce a codeword of the form $(m_1, p_1, q_1, m_2, p_2, q_2, \ldots)$. Higher rate codes can be produced by this encoder through puncturing.

Critical to the good performance of turbo codes is the "iterative decoding" process. Most implementations of turbo decoding make use of the so-called "log-MAP algorithm", which has been widely viewed as being faster and more numerically stable than a "direct-MAP" implementation. In this paper, it is demonstrated that the direct-MAP approach can be made numerically stable, and that the iterative process for this direct-MAP implementation is faster than the log-MAP approach on modern microprocessors, even if a lookup table is used to simplify the log-MAP calculations. Furthermore, since the direct-MAP implementation does not rely on any table-lookup approximations, the bit error rate performance of the direct-MAP implementation is slightly better than that produced using the log-MAP algorithm.

In the rest of this paper, we present an algorithmic description of this direct-MAP implementation, and a complexity comparison with the log-MAP algorithm.

This research is funded in part by an NSERC scholarship and an NSERC Strategic Project Grant.

A. Decoding of turbo codes

Consider a turbo code with $k = 1$ input bits per *trellis section*. Let m_i be the ith transmitted message bit; and let p_i and q_i denote the ith transmitted parity sequences from the first and second decoders, respectively. Let $y_i = (y_m, y_p, y_q)$ denote the sequence of received signal levels when a message/parity sequence (m_i, p_i, q_i) is transmitted. Furthermore, let ν be the number of memory elements in the first convolutional encoder, let s_i be the state of the encoder at time i, for $i = 0, 1, \ldots, T$, and let b_i be the branch of the trellis connecting state s_{i-1} to s_i. Define $L_{in}(b_i)$ as the input bit m_i for branch b_i, and $\sigma^-(b_i)$ and $\sigma^+(b_i)$ as the previous state and next state of the branch, respectively.

For the first encoder, we have the following terms.

$$\alpha_{i-1}(s_{i-1}) = \mathrm{p}(s_{i-1}|\boldsymbol{y}_i^-) \quad (1)$$

$$\gamma_i(b_i) = \mathrm{p}(m_i, y_i|s_{i-1}) \quad (2)$$

$$\beta_i(s_i) = \mathrm{p}(\boldsymbol{y}_i^+|s_i), \quad (3)$$

where $\boldsymbol{y}_i^-$ denotes the part of the received word before the i'th trellis section and $\boldsymbol{y}_i^+$ the part of the received word after the i'th trellis section.

The term $\gamma_i(b_i)$ in (2) is given by

$$\gamma_i(b_i) = \mathrm{p}(m_i) \cdot \mathrm{p}(y_m|m_i) \cdot \mathrm{p}(y_p|m_i, s_{i-1}). \quad (4)$$

The first term of (4), $\mathrm{p}(m_i)$, is the *iterative* information, which is successively refined as the iterations progress. The second term of (4), $\mathrm{p}(y_m|m_i)$, is the *common* information, and is common to both encoders. The third term of (4), $\mathrm{p}(y_p|m_i, s_{i-1})$, is the *private* information for the first encoder, and will be denoted by $\gamma_{i,pr}(b_i)$.

Using these definitions, the complete iterative decoding process is summarized below.

Algorithm 1 (Turbo Decoding)

1. **Setup:** Use the received data and the channel characteristics to calculate the *common* data, and the *private* data for each encoder. Initialize the iterative information $\mathrm{p}(m_i)$ to the *a priori* values (usually 1/2).
2. **Iterate:** Repeatedly update the iterative information $\mathrm{p}(m_i)$ using the "modified BCJR algorithm" as follows:
   (a) Recursively calculate $\alpha_i(s_i)$ for each state, for $i =$

$0, 1, \ldots, T-1$, using the recursion

$$\alpha_0(s_0) = \mathrm{p}(s_0) = \begin{cases} 1 & \text{if } s_0 \text{ is the all-zero state,} \\ 0 & \text{otherwise.} \end{cases} \quad (5)$$

$$\alpha_i(s_i) = \sum_{b_i | \sigma^+(b_i) = s_i} \alpha_{i-1}\left(\sigma^-(b_i)\right) \cdot \gamma_i(b_i). \quad (6)$$

(b) Recursively calculate $\beta_i(s_i)$ for each state, for $i = T, T-1, \ldots, 1$, using the recursion

$$\beta_T(s_T) = \begin{cases} 1 & \text{if } s_T \text{ is the all-zero state,} \\ 0 & \text{otherwise.} \end{cases} \quad (7)$$

$$\beta_i(s_i) = \sum_{b_{i+1} | \sigma^-(b_{i+1}) = s_i} \beta_{i+1}\left(\sigma^+(b_{i+1})\right) \cdot \gamma_{i+1}(b_{i+1}). \quad (8)$$

(c) Calculate the soft probabilities $\mathrm{p}(m_i, \boldsymbol{y})$ for $m_i = 0$ and $m_i = 1$, for $i = 1, 2, \ldots, T$:

$$\hat{\mathrm{p}}(m_i, \boldsymbol{y}) = \sum_{b_i | L_{in}(b_i) = m_i} \alpha_{i-1}(s_{i-1}) \cdot \gamma_{i,pr}(b_i) \cdot \beta_i(s_i). \quad (9)$$

This is called the *extrinsic information* (or *extrinsic output*) from the first decoder.

(d) Repeat steps (a)-(c) for the second decoder, using the interleaved extrinsic output $\hat{\mathrm{p}}(m_i, \boldsymbol{y})$ from the first decoder as the probabilities $\mathrm{p}(\widetilde{m}_i)$ for the second decoder.

(e) Deinterleave the second decoder's extrinsic output, $\hat{\mathrm{p}}(\widetilde{m}_i, \boldsymbol{y})$, to get the input $\mathrm{p}(m_i)$ for the first decoder.

3. **Terminate:** After the iterative process has refined $\mathrm{p}(m_i)$ "sufficiently", the final bit probabilities $\mathrm{p}(m_i, \boldsymbol{y})$ can be calculated as

$$\mathrm{p}(m_i, \boldsymbol{y}) = \frac{\hat{\mathrm{p}}(m_i, \boldsymbol{y}) \cdot \hat{\mathrm{p}}(\widetilde{m}_i, \boldsymbol{y}) \cdot \mathrm{p}(m_i | y_m)}{\hat{\mathrm{p}}(m_i, \boldsymbol{y}) \cdot \hat{\mathrm{p}}(\widetilde{m}_i, \boldsymbol{y}) \cdot \mathrm{p}(m_i | y_m) + (1 - \hat{\mathrm{p}}(m_i, \boldsymbol{y})) \cdot (1 - \hat{\mathrm{p}}(\widetilde{m}_i, \boldsymbol{y})) \cdot (1 - \mathrm{p}(m_i | y_m))}.$$

Note that if only the hard output is desired, this equation can be greatly simplified by observing that $\mathrm{p}(M_i = 0, \boldsymbol{y}) < 1/2$ (i.e. the decoded bit is a '0') if and only if

$$\hat{\mathrm{p}}(\widetilde{m}_i, \boldsymbol{y}) \hat{\mathrm{p}}(m_i, \boldsymbol{y})\, \mathrm{p}(m_i | y_m) \Big|_{m_i = 0} < (1 - \hat{\mathrm{p}}(\widetilde{m}_i, \boldsymbol{y}))\, (1 - \hat{\mathrm{p}}(m_i, \boldsymbol{y}))\, (1 - \mathrm{p}(m_i | y_m)) \Big|_{m_i = 0}. \quad (10)$$

Note, that if $\gamma_i(b_i)$ is used in step 2(c) instead of $\gamma_{i,pr}(b_i)$, then steps 2(a)-(c) describe the well-known BCJR algorithm for the MAP decoding of a convolutional code, which calculates the final bit probabilites $\mathrm{p}(m_i, \boldsymbol{y})$ directly. This provides a slightly more efficient way of calculating the final bit probabilities $\mathrm{p}(m_i, \boldsymbol{y})$ if the extrinsic information is not needed (e.g. for the final half-iteration).

II. Description of Direct-MAP Implementation

The usual approach for implementing turbo decoding is the log-MAP algorithm, which makes use of logarithms of probabilities (called *metrics*) in its calculations. This simplifies the process of probability multiplication and division, which in the log-MAP domain are converted to the addition and subtraction of metrics, respectively. However, the addition of probabilities, as required by (6), (8), and (9), becomes much more complicated, requiring time-consuming logarithm and exponentiation operations.

The direct-MAP implementation presented in the following sections avoids these time-consuming operations by using probabilities directly.

A. Decoding Step 1: Setting Up

Step 1 of the turbo decoding algorithm is the calculation of the *common* data $\mathrm{p}(y_m | m_i)$ and the *private* data for each encoder. (From here on, we deal only with the first encoder's private data, i.e. $\mathrm{p}(y_p | m_i, s_{i-1})$. Calculations for the second encoder are analagous.)

First, each received signal level y is converted into the probability $\mathrm{p}(\mathtt{bit} = 0 | y)$, and scaled by a factor of $\sqrt{2}$, for reasons discussed in Section II-B.1. For coherent detection of BPSK signals on an AWGN channel, it can be shown that this scaled probability is given by the expression

$$\text{"scaled"}\ \ \mathrm{p}(\mathtt{bit} = 0 | y) = \frac{\sqrt{2}}{1 + \exp\left(-4 \frac{y}{\sqrt{N_0}} \sqrt{\frac{E_b}{N_0}'}\right)}, \quad (11)$$

where $\frac{E_b}{N_0}'$ is the estimated SNR, per *coded* bit, at the receiver.

Scaled versions of the common data and private data are then calculated as follows:

$$\text{"scaled"}\ \ \mathrm{p}(y_m | m_i) = \sqrt{2}\, \mathrm{p}(m_i | y_m) \quad (12)$$

$$\text{"scaled"}\ \ \mathrm{p}(y_p | m_i, s_{i-1}) = \sqrt{2}\, \mathrm{p}(p_i | y_p). \quad (13)$$

B. Decoding Step 2: Iterating

The direct-MAP implementation calculates the α and β values using (6) and (8) directly. (For numerical stability issues regarding these calculations, see Section II-B.1.)

The extrinsic output is calculated using (9). The sum is performed by stepping through each of the $2^k \cdot 2^\nu$ branches, adding $\alpha_{i-1}(s_{i-1}) \cdot \gamma_{i,pr}(b_i) \cdot \beta_i(s_i)$ to the appropriate measure of the message bit, either $\mu(M_i = 0, \boldsymbol{y})$ or $\mu(M_i = 1, \boldsymbol{y})$. Once the sums have been completed, these measures are converted to the probability $\mathrm{p}(M_i = 0, \boldsymbol{y}) = \frac{\mu(M_i = 0, \boldsymbol{y})}{\mu(M_i = 0, \boldsymbol{y}) + \mu(M_i = 1, \boldsymbol{y})}$, which is the extrinsic output required for the next half-iteration.

Note that a more natural implementation of turbo decoding would use the "unmodified" BCJR algorithm to calculate the "complete output" $\mathrm{p}(m_i, \boldsymbol{y})$, and then use this complete output to calculate the extrinsic output $\hat{\mathrm{p}}(m_i, \boldsymbol{y})$. This works well and is quite efficient for the log-MAP approach. However, for the direct-MAP approach, the extrinsic output is related to the complete output by the cumbersome relation

$$\hat{\mathrm{p}}(m_i, \boldsymbol{y}) = \frac{1}{1 + \frac{\mathrm{p}(m_i)}{1 - \mathrm{p}(m_i)} \cdot \frac{\mathrm{p}(m_i | y_m)}{1 - \mathrm{p}(m_i | y_m)} \cdot \frac{1 - \mathrm{p}(m_i, \boldsymbol{y})}{\mathrm{p}(m_i, \boldsymbol{y})}}. \quad (14)$$

Not only is (14) unwieldy, but also, any real implementation causes "0/0" numerical instability at high SNR's; for example, we could have both $\mathrm{p}(m_i, \boldsymbol{y}) = 0$ and $\mathrm{p}(m_i | y_m) =$

0. Altering the form of (14) does not seem to help its stability.

Instead of applying the BCJR algorithm directly, we use the modified form of the BCJR algorithm as outlined in Algorithm 1, to produce the extrinsic output directly. This suffers from no numerical instability.

B.1 Renormalization

Another potential source of numerical instability arises from the fact that the α and β values calculated in (6) and (8) can grow or decay exponentially, as a function of the number of bits processed. The standard log-MAP approach is well suited to handle this, since this exponential growth is converted to a linear growth in the log-MAP domain. However, with the direct-MAP implementation, it becomes necessary to renormalize the α and β values periodically (i.e. scale them by a multiplicative constant), to prevent overflow/underflow of the data type used to store them. Since renormalization is a time-consuming process, the question becomes, "What is the largest number of trellis sections we can process before renormalizing the α's/β's without incurring an overflow or underflow?"

Although the following is not a rigorous proof, it provides a reasonable explanation for the worst-case (shortest) number of trellis sections we can process before renormalization becomes necessary. The following explanation is developed for the α calculations; the β calculations are analogous.

The approach used is to consider the "worst-case" received words which cause the α values to decay or grow the fastest. Let λ_i be the α *growth rate* for one trellis section i, defined as

$$\lambda_i = \frac{\sum_{s_i} \alpha_i(s_i)}{\sum_{s_{i-1}} \alpha_{i-1}(s_{i-1})}. \tag{15}$$

It will be shown that the decoding of pure noise generally causes the α's to grow the slowest (low λ_i), and the decoding of a pure, noiseless, signal generally causes the α's to grow the fastest (high λ_i). Although there exist "pathological" examples which could cause a slower or faster growth rate than the pure-noise and pure-signal cases (e.g. a trellis section for which all $\mathrm{p}(b_i, \boldsymbol{y}) = 0$), the probability of receiving such a pathological codeword in a real communications environment is insignificant. Thus, we limit ourselves to the most extreme real-world situations, the pure-noise and the pure-signal cases. The expected growth rate λ_i for each of these cases is derived below.

Observe that from (4), (12), and (13), the measure of the complete γ value is calculated as

$$\text{"scaled" } \gamma_i(b_i) = \mathrm{p}(m_i) \cdot [l \cdot \mathrm{p}(m_i|y_m)] \cdot [l \cdot \mathrm{p}(p_i|y_p)]\,,$$

for the scale factor $l = \sqrt{2}$. For the more general case with m_i consisting of k input bits, and p_i consisting of r parity bits, with the scale factor l applied to each bit's measure, this equation becomes

$$\text{"scaled" } \gamma_i(b_i) = l^{n_i} \cdot \mathrm{p}(m_i) \cdot \mathrm{p}(m_i|y_m) \cdot \mathrm{p}(p_i|y_p), \tag{16}$$

where $n_i = k + r$ is the total number of output bits (message + parity) for that trellis section. We now derive the optimum scale factor l for this general case.

Consider first the decoding of pure noise. In this case,

$$\mathrm{p}(m_i) \approx \mathrm{p}(m_i|y_m) \approx \frac{1}{2^k} \tag{17}$$

$$\mathrm{p}(p_i|y_p) \approx \frac{1}{2^r}, \tag{18}$$

so that the expected scaled value is

$$\text{"scaled" } \gamma_{i,noise}(b_i) = \frac{1}{2^k}\left(\frac{l}{2}\right)^{n_i}. \tag{19}$$

Substituting (19) into (15) and simplifying leads to an expected α growth rate of

$$\lambda_{i,noise} = \left(\frac{l}{2}\right)^{n_i}. \tag{20}$$

Next, consider the decoding of a noiseless signal. In this case, the codeword defines a single, unique, path through the trellis, with only one possible branch for each trellis section. In this case, only one state at time $i-1$ (say state s) satisfies $\alpha_{i-1}(s) \neq 0$, and only one branch b leading from that state satisfies $\gamma_i(b) \neq 0$. Moreover, for a noiseless signal, all probabilities in (16) are identically 1 for the branch b (for all iterations except the first, for which $\mathrm{p}(m_i) = 1/2$). This leads to an expected α growth rate for the "pure" signal case of

$$\lambda_{i,pure} = l^{n_i}. \tag{21}$$

To minimize the effects of the decay/growth for as long as possible for both the pure noise and the noiseless cases, we choose each bit's scale factor l so that the geometric average of these worst case growth rates is 1. Thus, the maximum growth rate will "match" the maximum decay rate, so $\sum_{s_i} \alpha_i(s_i)$ will remain within the range of the data type used for as long as possible. Performing this geometric average gives an optimum scale factor of

$$l = \sqrt{2}, \tag{22}$$

and an expected bound on the α growth rate of

$$\left(\frac{1}{\sqrt{2}}\right)^{n_i} \leq \lambda_i \leq \left(\sqrt{2}\right)^{n_i}. \tag{23}$$

In our implementation, C's `double` type was used. Since the positive range of C's `double` type is from 2^{-1023} to 2^{+1023}, (23) implies that the α's will decay to 0 or grow to `+Inf` after processing about 2064 bits of the output, in the worst case. Note that the typical case can usually go much further; nevertheless, our implementation renormalized after every 2000 output bits, allowing plenty of room for intermediate calculations, and for a wide variation in the range of the individual $\alpha_i(s_i)$'s.

Note also that the extrinsic output requires the computation of the product $\alpha_{i-1}(s_{i-1}) \cdot \gamma_{i,pr}(b_i) \cdot \beta_i(s_i)$ in (9). Thus, the entire $\alpha \cdot \gamma \cdot \beta$ product must also not exceed the range of a `double`. This is ensured in our implementation by keeping track of the indices i at which the $\alpha_i(s_i)$'s are renormalized, and then renormalizing the $\beta_i(s_i)$'s at the same indices. This ensures that the "extreme" α's (which occur at indices $i-1$) are always multipled by the normalized β's (at indices i), and vice versa.

III. Complexity and Comparison With Log-MAP

In the past, analysis of implementation complexity has focused mainly on the complexity of the BCJR algorithm (i.e. the number of operations required to calculate the α's, β's, γ's, and final output probabilities), as in [4]. However, for turbo decoding, this approach lumps the setup time together with the time required for one half-iteration of the decoding algorithm, without taking into account the fact that the setup takes place only once, while the iterative process is usually done many times.

Thus, in this complexity analysis, we separate the time required for setup and the time required for one half-iteration. We do this for both the log-MAP implementation and the direct-MAP implementation.

Our complexity analysis counts the number of mathematical operations required. Addition and subtraction operations are lumped collectively into the "Additions" category, and other operations are counted separately. Table I summarizes the number of operations required for the direct-MAP implementation and the log-MAP implementation. Note that the overall complexity is dominated by the $2^k \cdot 2^\nu$ terms, which is the number of branches in the trellis section. The termination step is not included in the table, as the complexity of the terminating step is about the same as that for any intermediate half-iteration.

The following assumptions about the implementation are made:

- Both the log-MAP and the direct-MAP implementation precalculate measures for the *partial* γ products $\gamma_{i,partial}(b_i) = \mathrm{p}(y_m|m_i) \cdot \mathrm{p}(y_p|m_i, s_{i-1})$, which are required in every iteration of the algorithm (see (4)). The cost of computing these products is incorporated into the "Setup" category of Table I.
- The log-MAP implementation uses a table-lookup for addition of probabilities, i.e., the complex operation $\delta_{sum} = \ln(e^{\delta_1} + e^{\delta_2})$ is replaced with the simpler form

$$\delta_{sum} = \max(\delta_1, \delta_2) + f_c(|\delta_2 - \delta_1|) \tag{24}$$

where $f_c(x)$ is implemented as a lookup table. (We assume that the determination of $\max(\delta_1, \delta_2)$ obtains the value $|\delta_2 - \delta_1|$ with negligible extra work.) Note that a table-lookup operation is much less efficient than an "ordinary" mathematical operation, since a table-lookup has an implied floating-point to integer conversion, which is "very slow on all [Pentium-type] processors" [2], not to mention the scaling and range-checking that must be done to ensure that the lookup value is within the range contained in the table.
- The log-MAP implementation is optimized by calculating the extrinsic bit probability from the complete bit probability. This calculation is done using (14), which in a log-MAP implementation becomes numerically stable due to the fact that probabilities extremely close to 0 which would have caused a "0/0" error in (14) can still be expressed accurately as a log-likelihood ratio, e.g. $\Lambda(M_i = 1) = \log \frac{\mathrm{p}(M_i=1)}{\mathrm{p}(M_i=0)}$.
- The number of operations required for renormalizing the α's and β's as required by the direct-MAP implementation are considered negligible and are not included in this table, since the renormalizations occur only once every ≈ 2000 output bits, as described in Section II-B.1.

TABLE I
Operations per trellis section required for turbo decoding

AWGN BPSK SETUP	Direct-MAP Operations	Log-MAP Operations
Additions	$2 \cdot n_i - 1$	$2(2^{n_i} - 1)$
Multiplications	$n_i + 2(2^{n_i} - 1)$	n_i
Divisions	n_i	-
"exp(a)" ops	n_i	-
ITERATING	**Direct-MAP Operations**	**Log-MAP Operations**
Additions	$2 \cdot (2^k - 1)2^\nu + 2^k 2^\nu + 1$	$2 \cdot (2^k - 1)2^\nu + 6 \cdot 2^k 2^\nu + 3$
Multiplications	$6 \cdot 2^k 2^\nu$	-
Divisions	1	-
"max(a,b)" ops	-	$2 \cdot (2^k - 1)2^\nu + 2^k 2^\nu$
Table Lookups	-	$2 \cdot (2^k - 1)2^\nu + 2^k 2^\nu$

Table I suggests that the log-MAP approach would be more efficient for small numbers of iterations, due to its more efficient setup procedure, whereas the direct-MAP approach is more efficient for larger numbers of iterations, due to its more efficient iterative process. (We have assumed that multiplication is reasonably fast, which is the case for modern processors.) Moreover, the direct-MAP approach performs slightly better in terms of bit error rate than the log-MAP method, since the log-MAP method uses a lookup table, which is only an approximation.

IV. Conclusion

In this paper, we have presented a direct-MAP implementation of turbo decoding which outperforms the classical log-MAP/table-lookup approach on modern processors, in terms of speed and bit error rate performance.

References

[1] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo Codes", Proc. ICC, pp. 1064–1070, 1993.

[2] A. Fog, "How to optimize for the Pentium family of microprocessors", http://www.agner.org/assem/pentopt.htm, July 3, 2000.

[3] C. Heegard and S. Wicker, *Turbo Coding*, Boston: Kluwer Academic Publishers, 1999.

[4] B. Vucetic and J. Yuan, *Turbo Codes Principles and Applications*, Boston: Kluwer Academic Publishers, 2000.