Turbo Code System Considerations: Version 1.0

Editor: Sam Dolinar

September 26, 1997

Contents

List of Figures

1 Introduction

This report is a high-level summary of turbo codes from a systems perspective. It first describes some basic concepts underlying turbo codes, and how to construct the encoders and decoders for these codes. Then it discusses some of the inherent tradeoffs involved in the selection of turbo code parameters. It concludes by discussing several system issues that arise with turbo codes.

Because turbo codes are so new, the outstanding issues are in a constant state of flux, so this report can only give our best current understanding of them. The report itself is a work in progress, especially as it is being compiled remotely while the editor is out-of-state with limited computer resources. In particular, the current version contains no bibliography, and all external references are denoted "to be supplied" (TBS). Comments on this draft of the report are solicited from readers for improving later editions. Stay tuned for later developments as they occur.

This report includes material taken verbatim from other sources, especially the draft recommendation on turbo codes prepared by JPL for CCSDS earlier this year. Contributors to the material contained herein are Dariush Divsalar, Sam Dolinar, and Fabrizio Pollara.

2 Brief Description of Turbo Codes

In 1993 a new class of concatenated codes called "turbo codes", was introduced. These codes can achieve near-Shannon-limit error correction performance with reasonable decoding complexity. Turbo codes outperform even the most powerful codes known to date, but more importantly they are much simpler to decode.

A turbo encoder is a combination of two simple recursive convolutional encoders, each using a small number of states. For a block of k information bits, each constituent code generates a set of parity bits. The turbo code consists of the information bits and both sets of parity, as shown in Fig. 1.

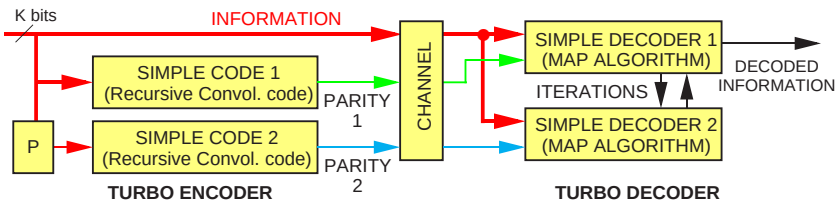


Figure 1: Example of turbo encoder/decoder.

The key innovation is an interleaver P, which permutes the original k information bits before encoding the second code. If the interleaver is well-chosen, information

blocks that correspond to error-prone codewords in one code will correspond to error-resistant codewords in the other code. The resulting code achieves performance similar to that of Shannon's well-known "random" codes, but random codes approach optimum performance only at the price of a prohibitively complex decoder.

Turbo decoding uses two simple decoders individually matched to the simple constituent codes. Each decoder sends likelihood estimates of the decoded bits to the other decoder, and uses the corresponding estimates from the other decoder as *a priori* likelihoods. The constituent decoders use the "MAP" (maximum *a posteriori*) bit-wise decoding algorithm, which requires the same number of states as the well-known Viterbi algorithm. The turbo decoder iterates between the outputs of the two decoders until reaching satisfactory convergence. The final output is a hard-quantized version of the likelihood estimates of either of the decoders.

To achieve their phenomenal performance, turbo codes use large block lengths and correspondingly large interleavers. The size of the interleaver affects buffer requirements and decoding delay, but has little impact on decoding time or decoder complexity. In our initial turbo code studies at JPL, we found good performance for various interleaver sizes ranging from a few thousand bits up to 16384 bits or more. These interleaver sizes are not much larger than those used by current concatenated codes that have a Reed-Solomon outer code with 8-bit code symbols.

More recently, we have reexamined the theoretical performance bounds on codes constrained to have short blocklengths, and have discovered that short-block turbo codes also perform amazingly well with respect to these limits. Thus, turbo codes also offer good performance for applications requiring small block sizes on the order of a few hundreds of bits.

3 Turbo Code Performance

3.1 Simulated Turbo Code Performance Curves

Figure 2 shows the simulated performance of a family of turbo codes of rates 1/2, 1/3, 1/4 and 1/6, constructed for an information block length of 10200 bits. The codes used for these simulations are the same as the ones described in Section 4. For these results, the decoder was forced to make its final decoding decisions after 10 iterations.

To achieve a bit error rate (BER) of 10^{-6}, threshold bit-SNRs of approximately -0.1 dB, $+0.15$ dB, $+0.4$ dB, and $+1.0$ dB, are required by the turbo codes of rates 1/6, 1/4, 1/3, and 1/2, respectively. These same threshold bit-SNRs achieve a codeword error rate (WER) of approximately 10^{-4} for these four codes.

3.2 Comparison to Traditional Concatenated Codes

Turbo codes gain a significant performance improvement over the traditional Reed-Solomon and convolutional concatenated codes currently used by JPL. For example, to

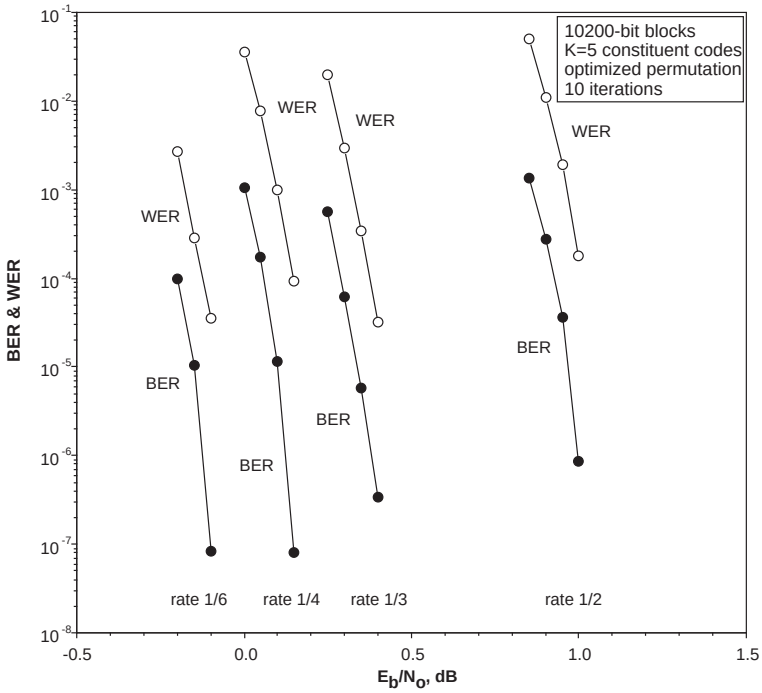


Figure 2: Performance curves for various turbo codes.

achieve an overall BER of 10^{-6} with a block length of 10200 bits (depth-5 interleaving), the required bit-SNRs are approximately 0.8 dB, 1.0 dB, and 2.6 dB for the DSN's standard codes consisting of the (255,223) Reed-Solomon code concatenated with the (15,1/6) convolutional code, the (15,1/4) convolutional code, and the (7,1/2) convolutional code, respectively. The performance gains achieved by the corresponding-rate turbo codes in Figure 2 range from 0.9 dB to 1.6 dB.

3.3 Ultimate Performance Limits

Turbo codes initially generated so much interest because their performance approaches the theoretical Shannon capacity limit more closely than any known codes that are practical to decode.

3.3.1 Code-Rate-Dependent Performance Limits

We initially found that good turbo codes can come within approximately 0.8 dB of the theoretical limit at a bit error rate (BER) of 10^{-6}. In applying this rule of thumb, it is important to keep in mind that the limiting performance depends on the code rate.

Figure 3 shows the Shannon-limit performance curves for a binary-input additive white Gaussian noise (AWGN) channel for rates 1/4, 1/3, 1/2, and 2/3. These curves

show the lowest possible bit-energy-to-noise ratio E_b/N_0 required to achieve a given BER over the binary-input AWGN channel using codes of these rates.

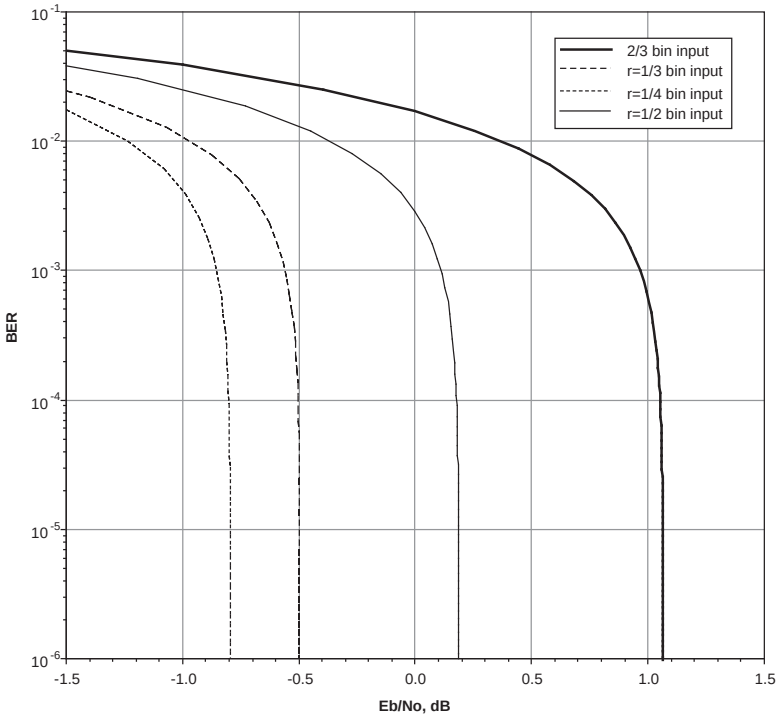


Figure 3: Capacity limits on the BER performance for codes with rates 1/4, 1/3, 1/2 and 2/3 operating over a binary input AWGN channel).

For low BER, each of these capacity-limited performance curves approaches a vertical asymptote dependent on the code rate. The asymptotes are at 1.1 dB for rate 2/3, 0.2 dB for rate 1/2, -0.5 dB for rate 1/3, and -0.8 dB for rate 1/4. The vertical asymptote for the ultimate Shannon limit on performance (i.e., rate $\rightarrow$ 0) is -1.6 dB. A comparison of these limits shows the improvement that is theoretically possible as a result of lowering the code rate. For example, for a binary-input AWGN channel, rate-1/2 codes suffer an inherent 0.7 dB disadvantage relative to rate-1/3 codes, a 1.0 dB disadvantage relative to rate-1/4 codes, and a 1.8 dB disadvantage relative to the ultimate limit (rate $\rightarrow$ 0).

3.3.2 Block-Size-Dependent Performance Limits

Just as a constraint on code rate raises the minimum threshold for reliable communication above the ultimate unconstrained capacity limit, so does a constraint on codeblock length. The theoretical limits shown in Figure 3 assume no constraint on block size. Approaching these limits requires that block sizes grow arbitrarily large.

We evaluated some classic Shannon sphere packing lower bounds on the performance of arbitrary codes of a given block size and code rate on the additive white Gaussian noise channel with unconstrained input (i.e., not necessarily binary-input as in Figure 3). The

results are shown in Figure 4. The curves labeled “bound” are the block-size-dependent bounds for each code rate. The horizontal asymptotes labeled “capacity” are the rate-dependent capacity limits. These asymptotes are slightly different from the vertical asymptotes in Figure 3 because they represent capacity limits for an unconstrained-input channel instead of a binary-input channel.

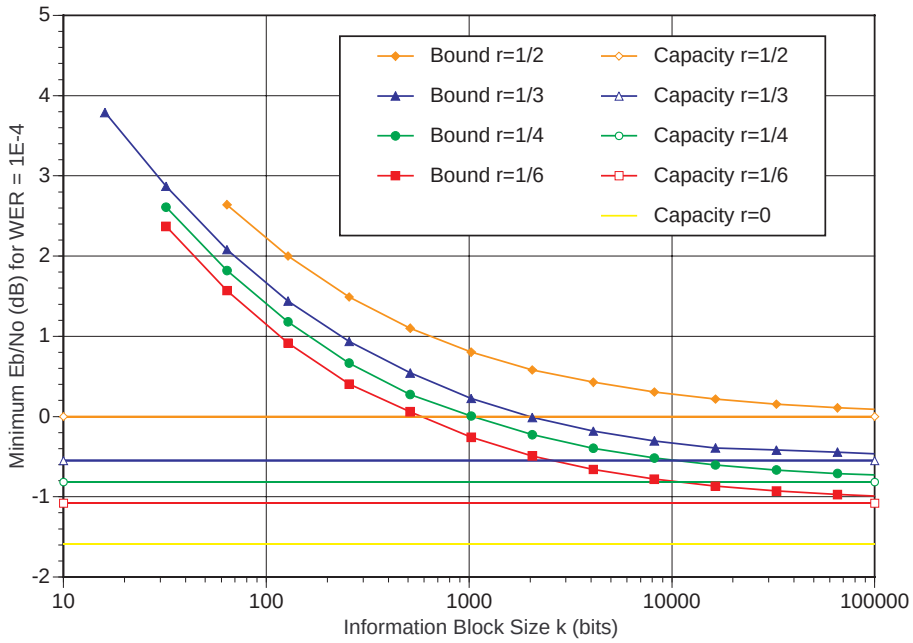


Figure 4: Shannon sphere-packing lower bounds on the BER performance for codes with varying information block length k and rates 1/6, 1/4, 1/3, 1/2, operating over an unconstrained-input AWGN channel.

This figure shows that, for any given code rate, the minimum threshold for reliable communication is significantly higher than the corresponding ultimate limit for that code rate, if the codeblock length is constrained to a given finite size. For example, 1000-bit blocks have an inherent advantage of about 1.3 dB compared to 100-bit blocks for each of the four code rates plotted. An additional gain of just over 0.5 dB is potentially obtained by going from 1000-bit blocks to 10000-bit blocks, and another 0.2 dB by going to 100000-bit blocks. After that, there is less than another 0.1 dB of improvement available before the ultimate capacity limit for unlimited block sizes is reached.

3.4 The Turbo Decoder Error Floor

Although turbo codes can be found to approach the Shannon-limiting performance at very small required bit error rates, the turbo code’s performance curve does not stay steep forever as does that of a convolutional/Reed-Solomon concatenated code. When it reaches the so-called “error floor,” the curve flattens out considerably and looks from

that point onward like the performance curve for a weak convolutional code. In the error floor region, the weakness of the constituent codes takes charge, and the performance curve flattens out from that point onward. The error floor is not an absolute lower limit on achievable error rate, but it is a region where the slope of the turbo code's error rate curve becomes dramatically lower.

We have developed transfer function bounds on turbo code performance [TBS] that accurately predict the actual turbo decoder's performance in the error floor region above the so-called "computational cutoff rate" threshold, below which the bounds diverge and are useless. However, we have found empirically that the error floor appears to be extrapolatable backwards through the computational cutoff rate barrier to some (as yet undetermined) lower value of signal-to-noise ratio where it finally stops being an accurate predictor of turbo code performance. Furthermore, the extrapolated error floor in this region appears to be computable as a "false convergence" plateau that we noted in our initial attempts to evaluate the bound in the region where it diverges. Thus, we have serendipitously obtained an accurate prediction of the error floor over its entire range by considering both the full transfer function bound and the false convergence plateau computed from some of the terms of this bound.

Fig. 5 provides an illustration of the transition of a turbo code performance curve from a steep "waterfall" region into a much flatter "error floor" region for two of the turbo codes we have analyzed at JPL. This figure shows the actual simulated turbo code performance compared with the full transfer function bound (labeled "analytical upper bound" or "all input weights") and also the extrapolation of the theoretical bound along the "false convergence plateau" (labeled "low input weights, error floor"). [Note that the information block length in this figure is denoted by N rather than k, the notation used elsewhere in this report.]

The original turbo codes developed by Berrou *et al* had error floors starting at a BER of about 10^{-5}. By using our theoretical predictors as guides, we have been able to design good turbo codes that lower the error floor to possibly insignificant levels (e.g., as low as 10^{-9} bit error rate).

4 Turbo Encoder Implementation on a Spacecraft

A turbo encoder is a combination of two simple encoders. The input is a frame of k information bits. The two component encoders generate parity symbols from two simple recursive convolutional codes, each with a small number of states. The information bits are also sent uncoded. A key feature of turbo codes is an interleaver, which permutes bit-wise the original k information bits before input to the second encoder.

4.1 Currently Recommended Turbo Encoder

JPL presented a preliminary turbo code recommendation to the CCSDS standards committee. A draft of our recommendation for the turbo encoder is included here.

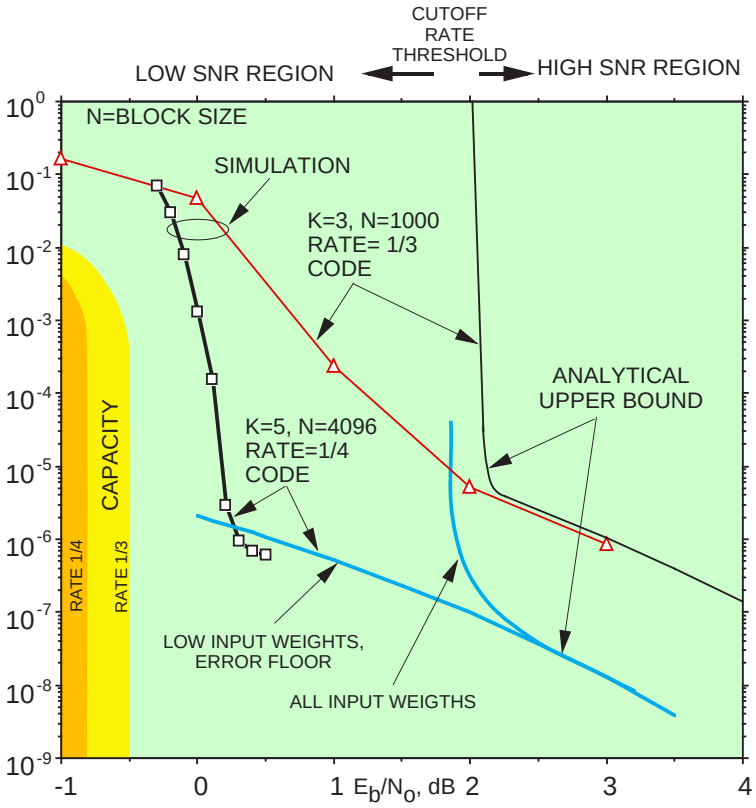


Figure 5: Illustration of turbo code error floor

Our recommendation to CCSDS was in the form of establishing a standard "compatible" family of turbo codes for which the code rate is an adjustable parameter that can be user-selected without affecting the structure, the complexity, or the performance (relative to respective theoretical limits) of the encoder or the decoder. The block size in this recommendation was fixed at 10200 bits, at the initial insistence of CCSDS. However, the compatible family of codes is easily generalized to include different block lengths.

The recommended turbo code is a systematic code with the following specifications:

1. Code type: Systematic parallel concatenated turbo code
2. Number of component codes: 2 (plus an uncoded component to make the code systematic)
3. Type of component codes: Recursive convolutional codes
4. Number of states of each convolutional component code: 16
5. Nominal[1] Code Rates: r = 1/2, 1/3, 1/4, or 1/6 bit per symbol (selectable)
6. Information Block length: $k = 10200$ bits
7. Codeblock length: $(k+4)/r$ = 20408, 30612, 40816, or 61224 bits for rates 1/2, 1/3, 1/4, or 1/6, respectively
8. Turbo Code Permutation

   The permuter or interleaver is a fundamental component of the turbo encoding and decoding process. The interleaver for turbo codes is a fixed bit-by-bit permutation of the the entire block of data. Unlike the symbol-by-symbol rectangular interleaver used with Reed-Solomon codes, the turbo code permutation scrambles individual bits and resembles a randomly selected permutation in its lack of apparent orderliness.

   The recommended permutation for the block length of 10200 is specified by a particular reordering of the integers $1, 2, \ldots, 10200$. The full permutation sequence is available on the World Wide Web at URL:
   `http://www331.jpl.nasa.gov/public/CCSDSinterleaver.html`
9. Backward and Forward Connection Vectors

   (a) Backward connection vector for both component codes and all code rates: G0 = 10011

[1]Because of "trellis termination" symbols (see next subsection), the true code rates (defined as the ratios of the information block lengths in item 6 to the codeblock lengths in item 7) are slightly smaller than the nominal code rates. In this recommendation, the symbol r and the terminology "code rate" always refers to the nominal code rates, r = 1/2, 1/3, 1/4, or 1/6.

(b) Forward connection vector for both component codes and rates 1/2 and 1/3: G1 = 11011. Puncturing of every other symbol from each component code is necessary for rate 1/2. No puncturing is done for rate 1/3.

(c) Forward connection vectors for rate 1/4: G2 = 10101, G3 = 11111 (1st component code); G1 = 11011 (2nd component code). No puncturing is done for rate 1/4.

(d) Forward connection vectors for rate 1/6: G3 = 11111, G4 = 11101, G5 = 10111 (1st component code); G1 = 11011, G3 = 11111 (2nd component code). No puncturing is done for rate 1/6.

4.2 Encoder Block Diagram

The recommended encoder block diagram is shown in Fig. 6. Each input frame of $k = 10200$ information bits is held in a frame buffer, and the bits in the buffer are read out in two different orders for the two component encoders. The first component encoder (a) operates on the bits in unpermuted order ("in a"), while the second component encoder (b) receives the same bits permuted by the interleaver ("in b"). The read-out addressing for "in a" is a simple counter, while the addressing for "in b" is specified by the turbo code permutation.

The component encoders are recursive convolutional encoders realized by feedback shift registers as shown in Fig. 6. The circuits shown in this figure implement the backward connection vector, G0, and the forward connection vectors, G1, G2, G3, G4, G5, specified in item 9 above. A key difference between these convolutional component encoders and a conventional standalone convolutional encoder is their recursiveness. In the figure this is indicated by the signal (corresponding to the backward connection vector G0) fed back into the leftmost adder of each component encoder.

Both component encoders in Figure 6 are initialized with 0s in all registers, and both are run for a total of $k + 4$ bit times, producing an output codeblock of $(k + 4)/r$ encoded symbols, where r is the nominal code rate. For the first k bit times, the input switches are in the lower position (as indicated in the figure) to receive input data. For the final 4 bit times, these switches move to the upper position to receive feedback from the shift registers. This feedback cancels the same feedback sent (unswitched) to the leftmost adder and causes all four registers to become filled with zeros after the final 4 bit times. Filling the registers with zeros is called terminating the trellis. During trellis termination the encoder continues to output nonzero encoded symbols. In particular, the "systematic uncoded" output (line "out 0a" in the figure) includes an extra 4 bits from the feedback line in addition to the k information bits.

In Fig. 6, the encoded symbols are multiplexed from top-to-bottom along the output line for the selected code rate to form the turbo codeblock. For the rate 1/3 code, the output sequence is (out 0a, out 1a, out 1b); for rate 1/4, the sequence is (out 0a, out 2a, out 3a, out 1b); for rate 1/6, the sequence is (out 0a, out 3a, out 4a, out 5a, out 1b, out 3b). These sequences are repeated for $(k + 4)$ bit times. For the rate 1/2 code, the

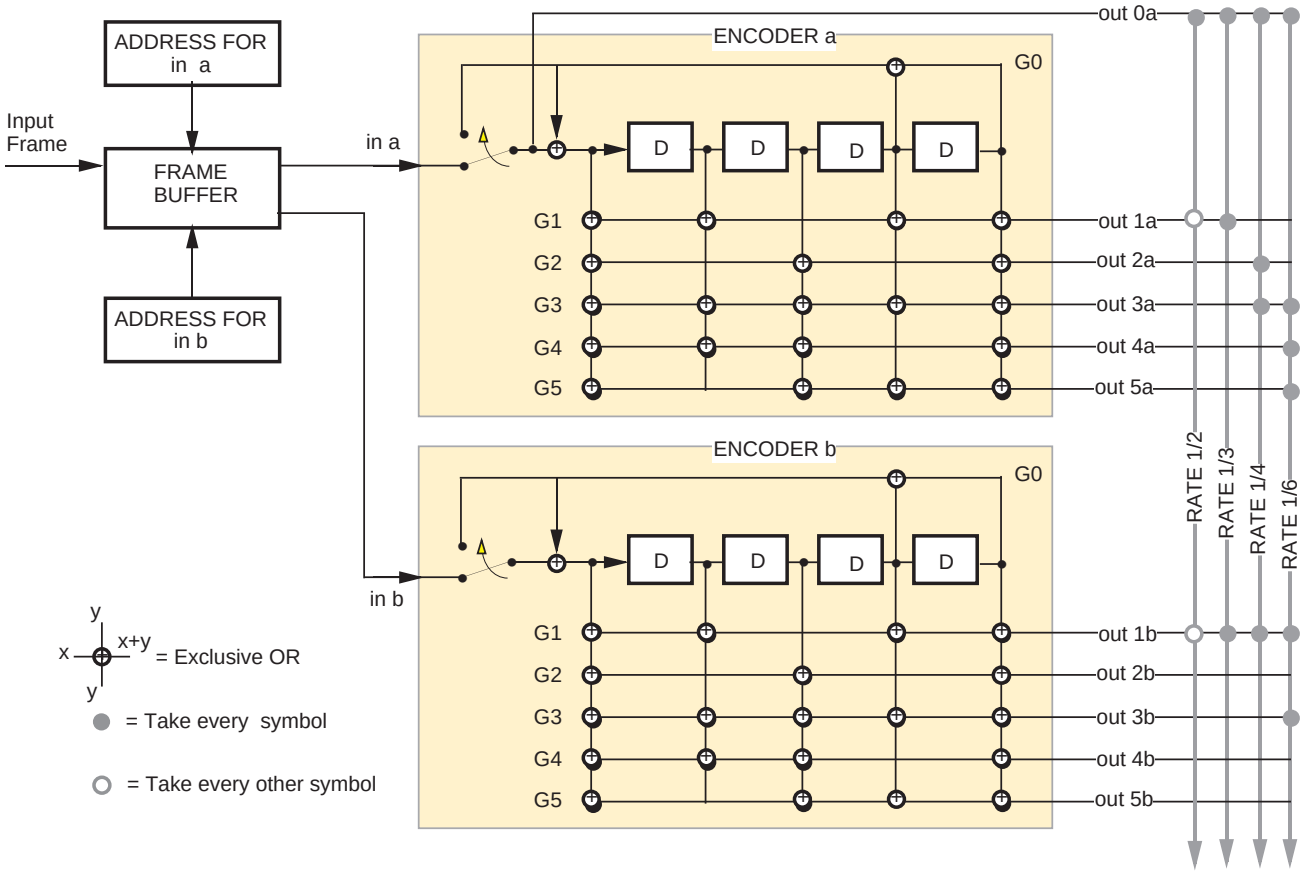


Figure 6: Turbo Encoder Block Diagram

output sequence is (out 0a, out 1a, out 0a, out 1b), repeated $(k+4)/2$ times. The turbo codeblocks constructed from these output sequences are depicted in Figure 7 for the four nominal code rates.

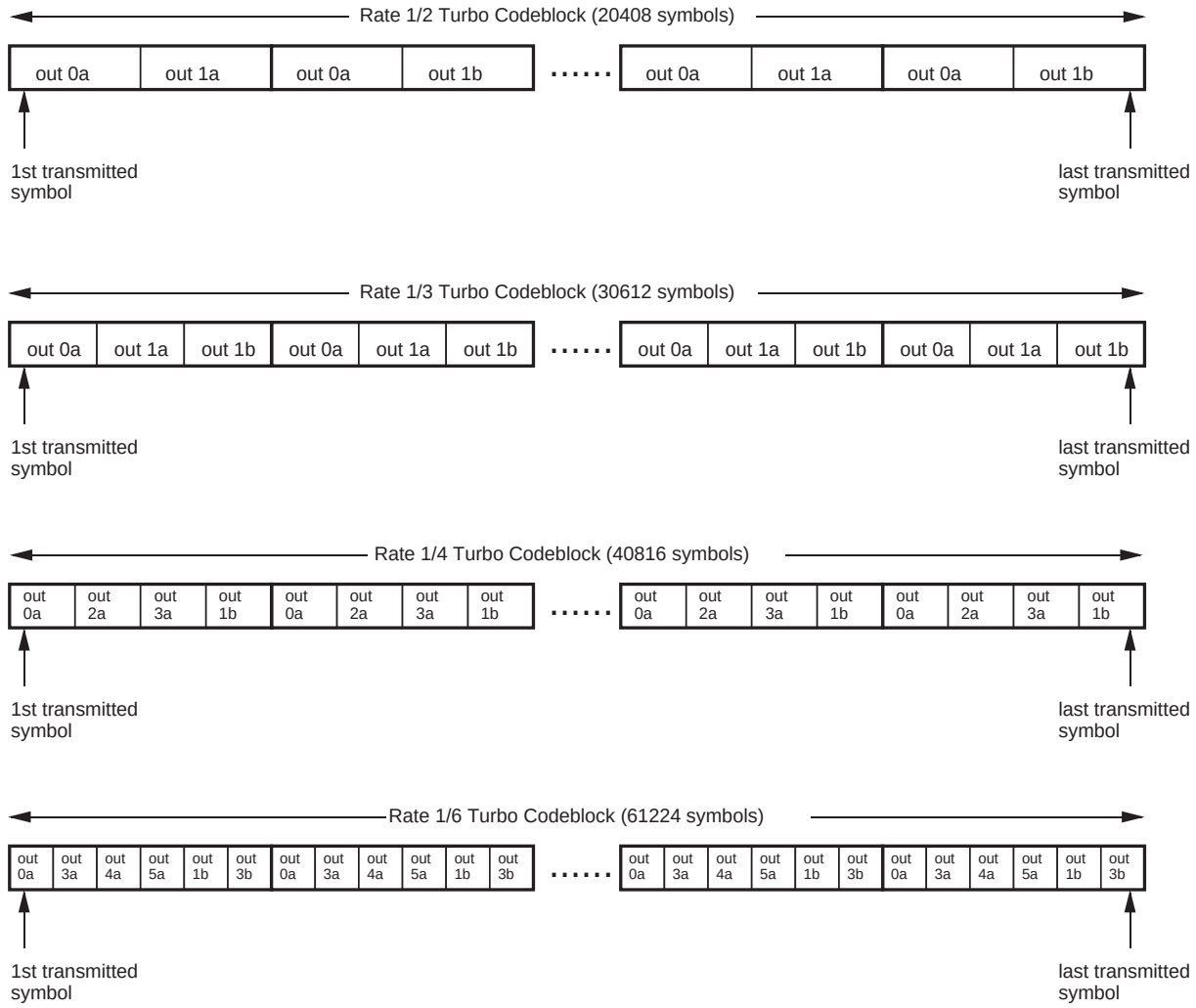


Figure 7: Turbo Codeblocks for Different Code Rates.

4.3 Encoder Hardware and Software Requirements

The turbo code introduces a couple of unique encoder complexity issues. The information block needs to be buffered and read out in a permuted order as part of the encoding process. This buffering has no analog in the plain convolutional encoder, but the size of this buffer is comparable to that required for an interleaved Reed-Solomon code block of the same size. The difference is that the traditional JPL concatenated coding architecture completely separates the Reed-Solomon encoder (with its associated buffer) from

the convolutional encoder. Thus, the turbo encoder cannot be regarded as a plug-in replacement for the convolutional encoder hardware. The turbo encoder actually replaces the Reed-Solomon/convolutional encoder combination.

Another spacecraft complexity consideration is how to implement the permutation. The best permutations for turbo codes look very random, and this requires specifying a random-looking readout order via a ROM. Alternatives exist, and we have found some good permutations that can be generated by simple rules rather than from a lookup table.

Some specific hardware and software considerations are:

1. Encoder Memory Requirements

   The two main encoder memory requirements are (a) storing an information data block while it is being encoded, and (b) storing the permutation (unless it is computed on-the-fly). Both of these requirements are driven primarily by the information block size k (in bits). The RAM required to hold the pre-encoded data must be at least k bits, and the size of the ROM that holds a pre-computed permutation would be at least $k \log_2 k$ bits.

2. Encoder Processing Requirements

   Encoder processing requirements are very modest, consisting of a small number of modulo-2 additions to implement the constituent recursive convolutional encodings. However, if the permutation is computed on-the-fly to save ROM, there will be additional processing requirements to compute the permutation.

3. Encoder Latency

   An entire information block of k bits must be read in before the encoding can proceed, because some of the bits in the tail end of block will be permuted to the front and need to be encoded first. Thus, there is a fundamental encoding latency of at least k bits in the encoding process.

5 Turbo Decoder Implementation on the Ground

The turbo decoder uses an iterative decoding algorithm based on simple decoders individually matched to the two simple constituent codes. Each constituent decoder makes likelihood estimates derived initially without using any received parity symbols not encoded by its corresponding constituent encoder. The (noisy) received uncoded information symbols are available to both decoders for making these estimates. Each decoder sends its likelihood estimates to the other decoder, and uses the corresponding estimates from the other decoder to determine new likelihoods by extracting the "extrinsic information" contained in the other decoder's estimates based on the parity symbols available only to it. Both decoders use the "maximum *a posteriori*" (MAP) bitwise decoding algorithm, which requires the same number of states as the well-known Viterbi algorithm. The turbo decoder iterates between the outputs of the two constituent

decoders until reaching satisfactory convergence. The final output is a hard-quantized version of the likelihood estimates of either of the decoders.

5.1 Recommended Turbo Decoding Algorithm

The CCSDS standards committee did not discuss a recommendation for the turbo decoder. However, the code performance curves in Section 3 are based on a certain decoding algorithm, so we recommend its usage. Variations from this algorithm will result in performance tradeoffs.

The recommended turbo decoder (for the turbo code family currently recommended to CCSDS) has the following characteristics:

1. Decoder type: Iterative "turbo" decoding using two 16-state component decoders (see Reference [TBS])
2. Type of component decoders: Soft-input, soft-output MAP decoders (see Reference [TBS])
3. Quantization of channel symbols: At least 6 bits/symbol
4. Quantization of decoder metrics: At least 8 bits
5. Number of decoder iterations: Variable depending on signal-to-noise ratio.

5.2 Decoder Block Diagram

The overall turbo decoding procedure is as depicted in Figure 1 and described earlier. The "simple decoders 1 and 2" each compute likelihood estimates using a version of the MAP or log-MAP algorithm as described in [TBS]. A diagram showing the structure of the turbo decoder in more detail is shown in Fig. 8.
Fig. 9 shows the basic circuits needed to implement the log-MAP algorithm.

5.3 Decoder Hardware and Software Requirements

The turbo code's selling point is that it provides near-Shannon-limit performance at low coding complexity. There is a dramatic reduction in complexity compared to the long-constraint-length ($K = 15$) convolutional codes that have been designed for recent missions in our prior attempts to squeeze performance toward the Shannon limit.

To first order, the complexity of a turbo decoder relative to a convolutional decoder using the *same* number of trellis states and branches can be estimated by multiplying several factors: (a) a factor of 2 because the turbo code uses two component decoders; (b) another factor of 2 because the individual decoders use forward and backward recursions compared to the Viterbi decoder's forward-only recursion; (c) another small factor because the turbo decoder's recursions require somewhat more complex calculations

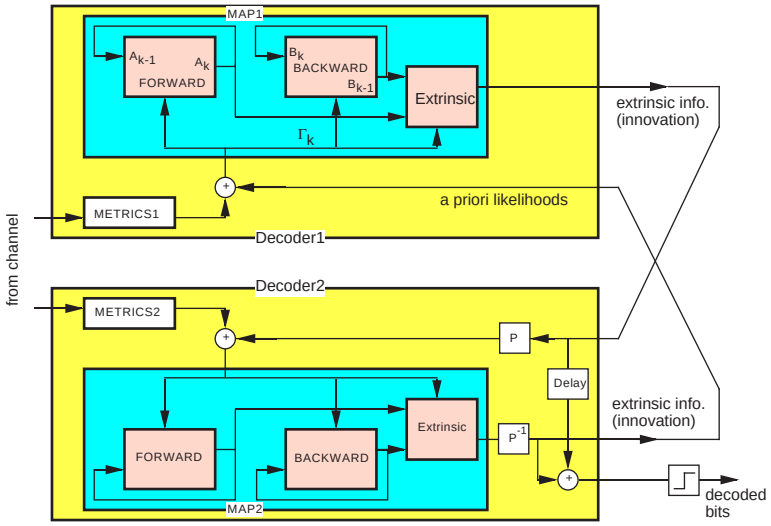
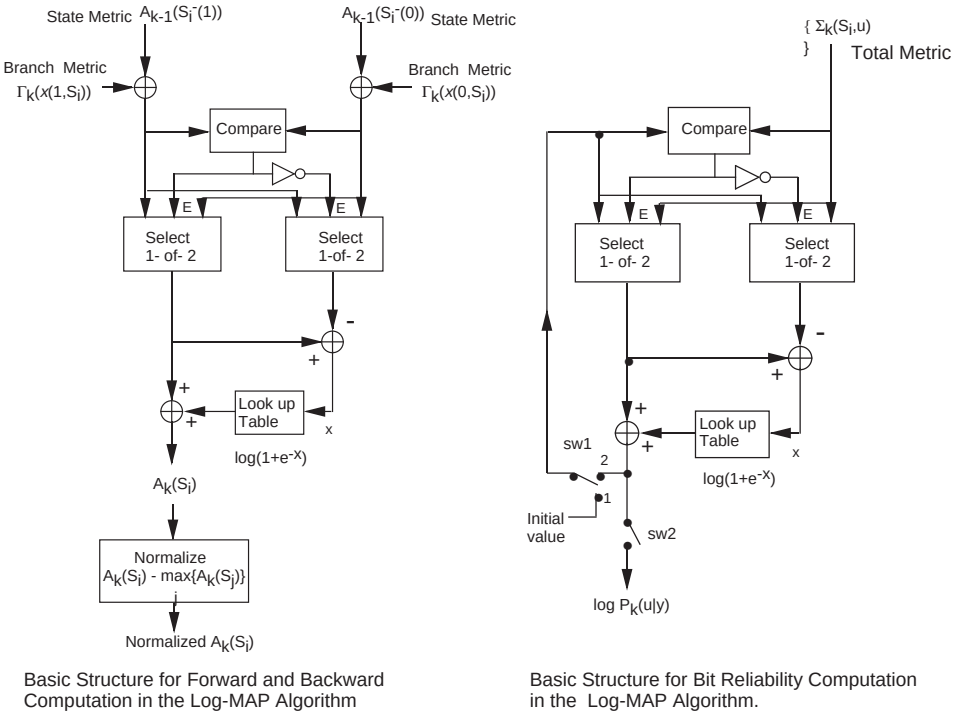


Figure 8: Structure of the turbo decoder.



Basic Structure for Forward and Backward Computation in the Log-MAP Algorithm

Basic Structure for Bit Reliability Computation in the Log-MAP Algorithm.

Figure 9: Basic Circuits to Implement the Log-MAP Algorithm

than the Viterbi decoder's; and (d) a factor to account for the turbo decoder's multiple iterations compared to the Viterbi decoder's single iteration. The relative decoding complexity for two *different* turbo codes or two *different* convolutional codes can be estimated by multiplying two additional factors: (e) the number of trellis states; and (f) the number of trellis branches per input bit into each state. We are trying to evaluate how big factor (c) should be, and whether factors (b) and (d) might be reduced on the average by using a smart turbo decoding algorithm. Such an algorithm might allow the decoder to stop its iterations early if a given codeword can already be decoded reliably, or to skip over portions of the forward and backward recursions for some iterations.

Factors (a) through (d) are 1 for Viterbi decoders of convolutional codes. For the CCSDS standard constraint-length-7 convolutional decoder, factor (e) is $2^6 = 64$, and factor (f) is $2/1 = 2$. For the Cassini constraint-length 15, rate 1/6 convolutional decoder, factor (e) is $2^{14} = 16384$ and factor (f) is $6/1 = 6$. For the turbo codes specified in Section 4, factor (e) is $2^4 = 16$ and and factor (f) ranges from $2/1 = 2$ to $6/1 = 6$.

Figure 10 shows estimated complexity versus performance tradeoffs for various rate-2/3 turbo codes, relative to that of a baseline $K = 7$ convolutional code. Although the specific data in this figure was derived for a high-code-rate, high-data-rate application (e.g., very low number of iterations allowed), similar complexity versus performance tradeoff curves can be drawn for the codes specified in Section 4.

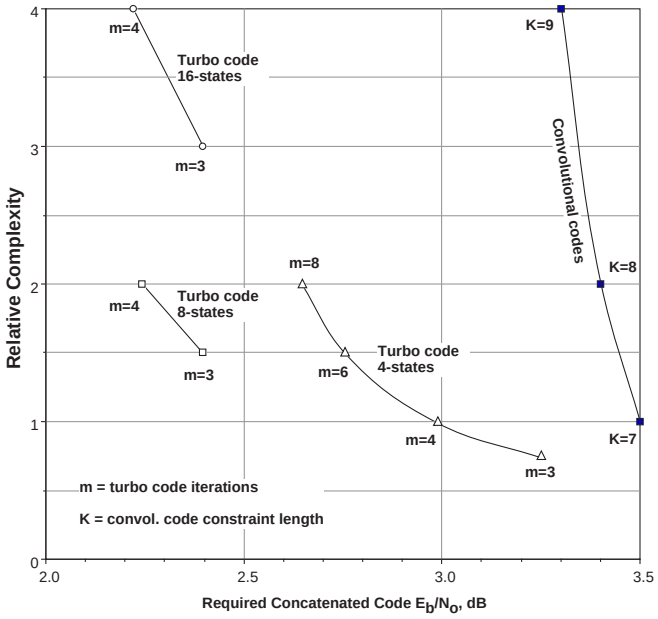


Figure 10: Downlink Turbo Code Complexity (relative to the baseline convolutional code) for Various Turbo Codes versus Required E_b/N_0

The estimates in Fig. 10 take into account complexity factors (a), (b), (d), and (e) as appropriate for each individual turbo code considered. Factor (c) is assumed to be

1, because we at present do not have a good estimate of how closely the complexity of a MAP-decoder recursion can be pushed toward that of a Viterbi decoder. This assumption causes the results in Figure 10 to be slightly optimistic. On the other hand, there is a significant potential for improvement in factors (d) and (b) if we can eventually design a smart turbo decoder that can correctly decide to terminate its iterations early for the vast majority of codewords that require fewer iterations than the maximum, or to skip portions of the forward or backward recursions in sections of the trellis that have already converged. In this case, the turbo code complexities depicted in Figure 10 might actually turn out to be slightly pessimistic.

Some specific hardware and software considerations are:

1. Decoding Speed

   The decoder processing requirements are proportional to complexity factors (a) through (f). The overall complexity factor translates directly to decoder processing time or to additional hardware to perform parallel decodings. Because the turbo decoder operates on discrete blocks of k bits, arbitrarily high decoder throughput may be gained by processing whole blocks in parallel devices. The price for this is additional hardware and additional latency. For more information on the detailed processing requirements for each step of the log-MAP decoding algorithm, see Reference [TBS].

2. Decoder Memory

   The decoder memory requirements are proportional to complexity factors (a) and (e). For more information on the detailed memory requirements of the log-MAP decoding algorithm, see Reference [TBS].

3. Decoding Delay

   Because the decoder processes whole blocks of k bits at a time, there is a minimum decoding delay of k bits. This latency is further increased by the time required for the decoder to process each block. If parallel decoders are used to increase decoding throughput, the latency increases in proportion to the number of parallel decoders.

6 Selection of Turbo Code Parameters

Because turbo codes can achieve great performance over a wide range of parameter values, the selection of reasonable code parameters is a major systems issue. The system design must assess all the parameter-space tradeoffs as they affect both the performance of the code and systems-related considerations. Turbo codes give the system designer vast flexibility to choose any desirable combination of parameters without sacrificing performance more than intrinsically necessary.

6.1 Code Rate

The code rate of the currently recommended turbo encoder is selectable from 1/2, 1/3, 1/4, or 1/6. Lower code rates are also possible to achieve even better performance if the receivers can work at the correspondingly lower channel-symbol SNR (E_s/N_0). The rule of thumb is that the potential coding gain for using lower code rates pretty much follows the corresponding gain for the ultimate code-rate-dependent theoretical limits given in Section 3.3.

6.2 Block Size

Larger block sizes than the current CCSDS recommendation (10200 bits) can be desirable for obtaining the utmost performance. On the other hand, some applications (e.g., emergency links, short-lived probes) may require much shorter blocks.

We have already seen in Figure 4 how some fundamental theoretical lower bounds on the performance of arbitrary codes on the additive white Gaussian noise channel vary with code block length. Amazingly, this variation is mirrored by the empirically determined dependence on block length of the performance of a large family of good turbo codes.

Figure 11 shows simulation results compared to the lower bound for a family of rate-1/3 turbo codes with different block lengths (using the generator polynomials specified in Section 4). This comparison is approximate, because the simulations were performed for a bit error rate (BER) of 10^{-6}, while the bounds are computed for a codeword error rate (WER) of 10^{-4}. [These two requirements give equivalent threshold bit-SNRs (within a few hundredths of a dB) for the 10200-bit turbo codes whose performance is plotted in Figure 2. For smaller blocks, the BER requirement of 10^{-6} is overly stringent compared to a WER requirement of 10^{-4} and thus it produces slightly pessimistic simulation results compared to the bound.]

Although there is a 2 dB performance differential between the simulation results for 256-bit blocks and 49152-bit blocks, we see that the difference between the simulations and the lower bounds remains approximately the same. The simulation results are about 0.5 dB to 1.0 dB from the theoretical limits for all code rates ranging from 1/6 to 1/2 and at all codeblock sizes ranging from 256 to 49152 information bits. We have similar but less extensive results for turbo codes in the same family with rates 1/2, 1/4, and 1/6.

The significance of this half-theoretical, half-empirical result is that we might expect to construct families of turbo codes that approach the ultimate theoretical limits uniformly regardless of code rate or block size. Many people have been hearing the wrong message about turbo codes. It is true that the worldwide excitement about turbo codes began with the discovery that they approach the ultimate Shannon limit at very large blocklengths. However, it is false to jump to the conclusion that turbo codes are therefore only good for large blocks. In fact, what we now know is just the opposite: turbo codes appear to be uniformly good for short blocks as well as long blocks.

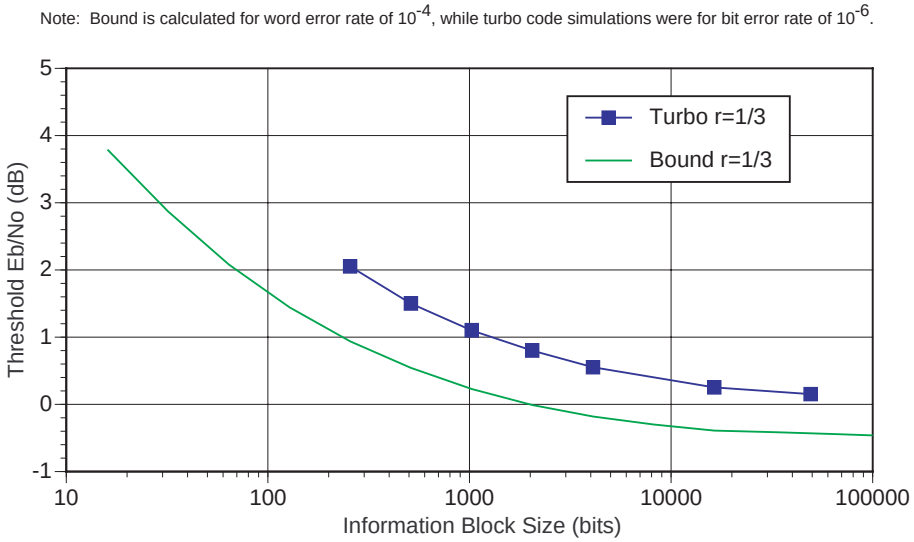


Figure 11: Comparison of turbo code performance with blocklength-constrained lower bound.

The meaning of "uniformly good" is that both short-block and long-block turbo codes can approach the ultimate theoretical performance limits for arbitrary codes *constrained to the same block size*. In contrast, fixed non-turbo codes such as the $(15, r)$ convolutional codes can only approach the ultimate theoretical limits at their particular "natural" block size. For a non-block code like the $(15, r)$ convolutional code, the natural block size is not precisely defined, but is on the order of 100 to 200 bits, i.e., the length of the decoding delay required for the Viterbi decoder to perform well.

For example, for blocks on the order of 1000 bits, a 1000-bit turbo code can exploit the intrinsic theoretical performance advantage of 1000-bit blocks over 150-bit blocks (just over 1 dB of advantage according to Figure 4). On the other hand, a $(15, r)$ convolutional code is stuck with essentially the same performance for a 1000-bit decoding delay that it obtains for a 150-bit delay. At 1000 bits, therefore, a $(15, r)$ convolutional code cannot compete in performance with a 1000-bit rate-r turbo code.

The following table shows some rough estimates of the comparative performance of $K = 5$ turbo codes versus $K = 15$ convolutional codes for 1000-bit blocks:

	Convol (15,*r*) BER=10^{-6}	Turbo (15,*r*) BER=10^{-6}	Convol (15,*r*) BER=10^{-3}
r=1/3	2.7 dB	1.1 dB	1.15 dB
r=1/4	2.45 dB	0.85 dB	0.9 dB
r=1/6	2.2 dB	0.6 dB	0.65 dB

Figure 12: E_b/N_0 required for convolutional and turbo codes defined on 1000-bit blocks

As seen from columns 1 and 2 of the table, turbo codes beat the corresponding-rate

convolutional codes by about 1.6 dB in E_b/N_0 at BER=10^{-6}. Alternatively, turbo codes beat the corresponding convolutional codes by three orders of magnitude in BER when both codes are constrained to operate at roughly the same E_b/N_0 (columns 2 and 3 of the table). Either way, it is a huge advantage in favor of turbo codes for 1000-bit blocks.

Interestingly enough, the same weakness also shows up in the other direction for a *fixed* convolutional code. If one were to insist on using the $(15, r)$ convolutional code with a 50-bit block (and correspondingly reduce the decoding delay of the Viterbi decoder), the decoder's performance would fall apart dramatically. [Of course, in this situation one would likely switch to a $(7, r)$ convolutional code, for which the "natural" block length is on the order of 32 to 64 bits.] On the other hand, the performance of a 50-bit turbo code would degrade gracefully by the amount mandated theoretically for any 50-bit code.

Extrapolating from Figure 11, it is likely that one single turbo code (i.e., using the same two component encoders but different permutations for different block lengths) would be uniformly good for 50-bit blocks, 150-bit blocks, 1000-bit blocks, 10000-bit block, or 100000-bit blocks. This is much simpler and also provides a performance advantage compared to the present coding alternatives: 1) choosing $(7, r)$ convolutional codes for 50-bit blocks; 2) choosing $(15, r)$ convolutional codes for 150-bit blocks; 3) choosing (255,223) Reed-Solomon + (K, r) convolutional codes for 10000-bit blocks; and 4) forcing one of these three codes to be ill-matched to the required block size, when the required block size is not 50-bits, 150-bits, or 10000-bits.

6.3 Constituent Codes

Effective turbo codes can be constructed from a wide variety of constituents. For our preliminary recommendation to CCSDS, we settled on a particular choice for the constituent codes. Here are some of the factors underlying this choice.

6.3.1 Number and Type of Constituent Codes

Turbo codes with more than two constituent codes are feasible in principle, but to this point they have not been well studied — mainly because two-component turbo codes already perform so well. The best performing and best understood constituent codes discovered thus far are the class of recursive convolutional codes, as recommended in Section 4 and in the original turbo code paper by Berrou *et al*.

6.3.2 Constraint Length

Our currently recommended turbo code is formed from two recursive convolutional codes with constraint length $K = 5$. Higher constraint lengths are more complex to decode, and thus far they seem to offer negligible performance improvement. In the other direction, constituent codes with constraint lengths less than 5 may be desirable to achieve higher decoding speeds.

We have some recent results indicating that turbo codes using a particular set of $K = 4$ constituents may perform almost as well as the ones we are currently recommending based on $K = 5$ constituents. This merits further study, because the corresponding decoder is only half as complex or twice as fast. For very high data rate applications, we are also studying super-low-complexity codes constructed from two-state ($K = 2$) constituents.

6.3.3 Code Generator Polynomials

Considerable theory has been developed to guide the choice of constituent code generator polynomials. This theory is based on the transfer function bounds that are used to predict the turbo decoder error floor. The error floor can be lowered the most if the divisor polynomial (G0 in Figure 6) is a primitive polynomial. Additional theoretical considerations guide the choice of the remaining polynomials.

6.3.4 Code Transparency

Turbo codes are inherently non-transparent, meaning that a totally inverted codeblock cannot be an exact codeword. However, a turbo code can be made "approximately transparent" except near the edges of the codeblock. It is a system design issue to decide whether an approximately transparent turbo code would be preferred, at some sacrifice of performance, to one designed without any transparency constraints. The codes recommended in Section 6 are *not* constructed to be approximately transparent.

6.4 Permutation

The permutation included in our preliminary recommendation to CCSDS looks very random. However, it was manually optimized to yield the best performance for the fixed block size $k = 10200$, and it outperforms a purely random permutation by a small amount. Such an optimized permutation needs to be stored onboard in ROM because it is infeasible to recompute it on the fly for every codeword.

The best understood, and only slightly poorer performing, permutations for turbo codes are completely random. However, we have discovered that pseudo-random sequences generated by some simple randomizing algorithms (such as simple feedback shift register sequences) are not sufficiently random for generating good turbo code permutations. But there are some slightly more complex algorithms for generating pseudo-random permutations that perform better. We are currently examining an on-the-fly permutation generation algorithm of moderate complexity that performs within about 0.1 dB of the optimized permutation.

6.5 Decoder Stopping Rules

Presently our simulated turbo decoders stop iterating after a predetermined number of iterations. For some codewords (or sections of codewords), the predetermined number of iterations may be too many or too few. We need to study methods to stop the decoder's iterations when convergence is satisfactory, i.e., without wasting iterations when the decoder has already converged, and without halting iterations prematurely when the decoder needs a little more time.

An efficient decoder stopping rule will reduce the average number of iterations and increase the average decoding throughput. This will come at the expense of a slightly more complicated decoding algorithm and increased decoder buffering requirements to accommodate variable decoding times.

6.6 Parallel versus Serial Concatenation

Unlike conventional concatenated codes, turbo codes are "self-concatenated" in that they envelop their concatenation. For conventional concatenated codes (such as JPL's traditional Reed-Solomon and convolutional code concatenation) the concatenation is external to the separate constituent codes, and the two codes are decoded separately.[2] For turbo codes, the concatenation is internal to the definition of the code, and the decoder is constructed to decode the whole code at the same time, not in two distinct pieces.

The original turbo codes were based on "parallel" concatenations of constituent codes, as specified here in Section 4. Later, similar codes were developed at JPL based on serial concatenation. Serially concatenated codes offer the potential of somewhat better performance than parallel concatenated codes, including lower error floors. To date, there are fewer simulation results for serially concatenated codes, and this is the main reason they were not presented as part of our preliminary recommendation to CCSDS. Study is ongoing to determine the relative benefits of serial concatenation versus parallel concatenation.

7 Overall System Issues

In addition to the basic tradeoff considerations in selecting the fundamental turbo code parameters, there are a number of broader system issues that arise when turbo codes are used. This section discusses synchronization issues and various types of non-ideal performance issues.

[2] An exception to this general rule is the special Feedback Concatenated Decoder (FCD) developed for the Galileo S-Band mission, in which some information is passed back and forth between the convolutional decoder and the Reed-Solomon decoder in a 4-stage iterative process. In this sense, the 4-stage code developed for Galileo can be regarded as a less powerful precursor of turbo codes. However, turbo decoders pass soft information more efficiently between their constituent decoders, and the turbo code uses simpler constituent codes while achieving better performance.

7.1 Synchronization for Turbo Codes

Codeblock synchronization is necessary for proper decoding of turbo codeblocks. Synchronization of the turbo codeblock is achieved by using a stream of fixed-length codeblocks with an attached sync marker between them. The code symbols comprising the sync marker for the turbo code are attached directly to the encoder output without being encoded.

Synchronization is acquired on the receiving end by recognizing the specific bit pattern of the sync marker in the raw (undecoded) telemetry channel data stream. Synchronization is then confirmed by making further checks. Frame synchronizers should be set to expect a marker at a recurrence interval equal to the length of the sync marker plus that of the turbo codeblock.

A diagram of a turbo codeblock with attached sync marker is shown in Fig. 13. Note that the lengths of the turbo codeblock and the sync marker are both inversely proportional to the nominal code rate r. This yields roughly equivalent synchronization performance independent of code rate.

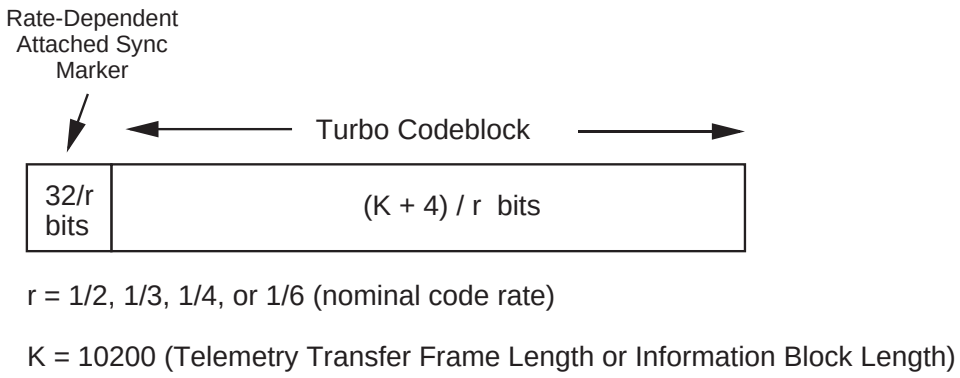


Figure 13: Turbo Codeblock with Attached Sync Marker

At JPL frame sync has traditionally been acquired using a sync marker defined in the information bit domain rather than the encoded symbol domain, and detected after Viterbi decoding. This method relies on the fact that frame sync is not required for successful operation of the Viterbi decoder but is necessary for decoding the Reed-Solomon code. The Viterbi decoder is capable of finding its own "node sync" without the aid of known sync markers in the data stream. In contrast, the Reed-Solomon decoder has no effective method (other than trial and error) for determining frame sync on its own. Thus, frame sync has traditionally been performed by looking for the presence of a known sync marker in the Viterbi-decoded bit stream, without any assistance from the Reed-Solomon decoder

A similar approach will not work efficiently for turbo codes, because each constituent convolutional code is too weak by itself to detect a reasonable size marker reliably, and because the powerful combined turbo decoding operation needs to know the code

block boundaries before it can iterate between permuted and unpermuted data domains. Therefore, turbo code applications will need to use channel-symbol-domain frame sync methods similar to those used by the Galileo S-band mission. In addition, the frame sync algorithms for turbo codes will have to work at much higher data rates.

Turbo decoders see the same type of soft channel information as Viterbi decoders, and should in principle be capable of processing it to determine sync, just as Viterbi decoders can find their own node sync without looking for external sync markers. However, this will not be easy. A turbo decoder needs to determine the correct "phase" of the incoming symbol stream modulo N, where N is the total number of channel symbols per turbo codeblock, while the Viterbi decoder only needs to determine the correct "phase" modulo n, where $1/n$ is the rate of the code. These numbers n and N typically differ by three or four orders of magnitude. We are investigating whether there are circumstances for which markerless synchronization methods might be feasible for turbo codes.

7.2 Non-ideal Receiver Issues

An ideal receiver hands to the turbo decoder perfectly synchronized, perfectly soft symbols corrupted only by pure additive white Gaussian noise. The low-SNR operating conditions achievable by turbo codes require the systems designer to avoid over-optimizing the code performance at the expense of declining receiver performance.

Some specific issues arising from the interplay of the code selection and receiver performance are the following.

7.2.1 Lower symbol SNR

To take advantage of the improved performance of turbo codes, the receiving system must operate at a significantly lower symbol signal-to-noise ratio (SSNR) than that of a less powerful code with the same code rate. This imposes more stringent demands on the receiver's ability to perform symbol synchronization. The performance advantages of turbo coding may be negated if the receiver cannot lock onto the lower-SSNR symbols.

Since the threshold SSNR drops in direct proportion with the code rate, whereas the threshold bit signal-to-noise ratio (BSNR) converges to a fixed limit as code rate $\rightarrow 0$ (see Section 3.3.1), lowering the code rate too far toward 0 produces diminishing returns in overall code performance while continuing to tax the receiver heavily. It is a systems issue to decide on the code rate that provides the best tradeoffs. For turbo codes, the variation of code performance with code rate more or less mirrors that of the ultimate limits on performance, as given in Section 3.3.1.

7.2.2 Performance with Non-Ideal Tracking Loops

We have not yet assessed how much the turbo decoder's performance degrades when there are small errors in tracking and detecting the received symbols. However, with turbo codes, there is also a possibility to improve the receiver's tracking performance by

feeding back soft information from the decoding process to assist the receiver's tracking loops. Preliminary assessments [TBS] of potential improvements are encouraging.

7.3 Decoder Performance Issues

Turbo codes and their associated decoders are designed to operate as standalone coding systems providing good performance without any further assistance. However, several systems issues related to code performance do arise.

7.3.1 Residual Error Detection and/or Correction

In applications requiring extremely low error rates, the error rate of a turbo code in the error floor region may be unacceptable despite our best efforts to lower it. The solution may be to add an outer code to work in conjunction with the turbo code as the inner code. The outer code would ideally be a binary code such as a BCH code rather than a nonbinary Reed-Solomon code. Because of the sparseness of errors on the error floor (typically a handful of bit errors per block), the outer code could have a very high code-rate and would shift the required SNR just a tiny amount. We are just starting to look at issues pertaining to the selection of a good outer code and its resulting performance. To correctly analyze the performance of the concatenated system, we will first need to amass and study a large collection of simulated turbo decoder error statistics that are much more detailed than the average error rates we have determined previously.

7.3.2 Lowering the Turbo Code's Error Floor

Even without using an outer BCH code, we have been able to design good turbo codes that lower the error floor to possibly insignificant levels (e.g., 10^{-9} bit error rate). Such performance may be sufficiently good for JPL applications to obviate the need for an outer error-correcting code. In that case, a simpler outer code (such as a CRC code) may still be desirable for error detection only.

Our insights into lowering the error floor came from analyzing the theoretical predictors of turbo code performance mentioned in Section 3.4. These theoretical predictors are important because they give us insight not only into the overall error rate but also into the detailed error statistics. Our theoretical model for the error floor predicts sparse, nearly independent bit errors in the turbo decoded blocks that are decoded erroneously. Under this error model, the performance of an outer code should be accurately computable by summing a binomial probability distribution. The main question remaining is the degree of validation necessary to confirm that the model indeed matches the actual error floor behavior. Because of the rarity of errors in the error floor region, it is very difficult to obtain sufficient simulation data to confirm all aspects of the predicted error statistics.

Conversely, in the "waterfall" region of the turbo code performance curve where simulated errors are plentiful enough to determine the detailed error statistics, the im-

plications are bad with respect to the efficacy of an outer code. In the waterfall region, the turbo decoder occasionally encounters a code block for which its iterative decoding algorithm fails to converge. The result is an avalanche of errors in a few bad blocks that cannot be corrected by any reasonable outer code. The message is that an outer code will provide very little benefit at signal-to-noise ratios below the error floor region.

7.3.3 Frame Error Rate (FER) or Word Error Rate (WER)

For many applications, especially when the data has been compressed, the frame error rate (FER) is a more relevant measure of code performance than the bit error rate (BER). When there is a one-to-one correspondence of data frames and code blocks, the frame error rate is synonymous with the turbo code's word error rate (WER).

Most of our turbo code simulations have measured the performance of the code by its BER. Only recently have we begun assembling a comparable set of simulation data on WER for turbo codes. Our preliminary results indicate that WERs on the order of 10^{-4} are feasible with careful selection of the generator polynomials of the constituent codes. Further simulations of obtainable WERs are needed.

7.3.4 Incomplete Frames

If a long received turbo codeblock is incomplete, is there any way to decode part of the data? We are looking into ways to modify the turbo algorithm to recover part of a codeblock in case some critical data needs to be salvaged.

7.3.5 Unequal Error Protection

Headers of a packetized system (such as SCPS) might need stronger protection than the body of the data. This might also allow an easing of the BER requirement on the non-header data. Turbo codes lend themselves naturally (in principle) to providing unequal error protection. In a standard turbo code such as the ones presented in Section 4, all information is encoded twice, once permuted and once unpermuted. Improved error rates on small portions of a codeblock might be obtained by encoding selected information tidbits three or more times within the same codeblock. We have not yet analyzed the performance of such a code.

7.3.6 Decoder Sensitivity to Encoder Errors

We briefly looked at a problem heretofore not considered in our turbo code design or performance analysis. Evidently there is a small but non-vanishing probability that the RAM holding the block of information bits might be struck by a cosmic particle causing one of the information bits to flip. If this bit flip should occur after that bit has been encoded by one of the turbo code's component encoders but before it is encoded by the

second component, might this have a catastrophic effect on the turbo decoder, causing corruption of the entire information block?

The answer to this question is apparently no — provided that the decoder allows for the possibility that this type of error might occur. Our first attempts at decoding a block corrupted in this manner did indeed manifest error propagation throughout the block. However, we showed that we can eliminate this error propagation by introducing minor modifications to the decoding algorithm that account for the possibility of a bit flip before encoding. For one such modification, we assumed that the location of the corrupted bit is known to the decoder (e.g., by using error detection onboard the spacecraft). In this case, each component decoder should "erase" the corrupted bit by always reporting to the other component decoder a log-likelihood ratio of zero for that bit, regardless of the values of the received symbols or any extrinsic information obtained on previous iterations. For the second modification, we assumed no knowledge about the location of the corrupted bit but we allowed the decoder to consider that all the bits in the block might be suspect with a certain probability. As a first-cut model, we simply saturated the decoder's log-likelihood ratio for every bit at a reliability value that would not exceed the *a priori* confidence that each component encoder actually encoded the same bit value. For example, with one bit flip per 10000 bits, it is appropriate to saturate the likelihood ratio at approximately 10000:1, because, no matter how confidently one component decoder determines the value of a given bit, there is a 1 in 10000 chance that the bit will have the opposite value in the eyes of the second decoder.

With either of these algorithmic modifications in place, we did not see any error propagation beyond the corrupted bit itself. Thus, the turbo decoder seems to be robust enough to adapt to different types of noise models, as long as its log-likelihood-ratio computations account for all the types of noise actually present.

We need to further investigate these effects in several ways. First, we need a more quantitative appraisal of the degree to which the decoder is able to overcome the bit-flip problem. Is our crude solution of saturating the log-likelihood ratios good enough, or should the decoder calculate the log-likelihood ratios more exactly, taking into account the bit-flip probability and the Gaussian noise probabilities simultaneously? More generally, we want to learn how susceptible turbo decoders are to unanticipated and therefore unmodeled noise sources, and whether there might be "universal" safeguards that could be built into the decoding algorithm to protect against a wide range of unknown corruptions.

7.3.7 Imperfect Computation of Receiver Metrics

Due to hardware constraints and requirements on the speed of the decoder, a real-life turbo decoder will not compute the MAP algorithm to the maximum possible precision, and decoder performance will suffer as a result. It is another systems issue to decide how to get the maximum possible performance at the least hardware cost for the required decoder speed.

8 Summary

Turbo codes represent a major paradigm shift in JPL's approach to coding systems. They offer performance improvements of 1 dB or more compared to the codes currently used by the DSN, and they are more than an order-of-magnitude easier to decode than the most complex of the current DSN codes. Furthermore, they sustain their near-optimum performance over a wide range of fundamental code parameters (such as code rate and block size) that are important to system designers.

Future JPL coding systems will likely be based on families of turbo codes[3] with adjustable parameters appropriate for different applications. Such integrated coding families will be an improvement over the present-day hodge-podge of different codes. They will provide better performance, reduce the complexity of decoding, and simplify system integration. To reach this goal, we must continue to study the basic principles governing turbo codes, and to assess the implications of turbo codes on system design. The current version of this report is just a small step toward the latter goal.

[3] or similar codes such as serial concatenated convolutional codes with soft-input, soft-output iterative decoding