2000

ON TURBO CODES AND OTHER CONCATENATED SCHEMES IN COMMUNICATION SYSTEMS

AMBROZE, MARCEL ADRIAN

ON TURBO CODES AND OTHER CONCATENATED SCHEMES IN COMMUNICATION SYSTEMS

M. A. AMBROZE

Ph. D. 2000

ON TURBO CODES AND OTHER CONCATENATED SCHEMES IN COMMUNICATION SYSTEMS

by

MARCEL ADRIAN AMBROZE

A thesis submitted to the University of Plymouth
in partial fulfillment for the degree of

DOCTOR OF PHILOSOPHY

Satellite Research Centre
Department of Communication and Electronic Engineering
Faculty of Technology

August, 2000

On turbo codes and other concatenated schemes in communication systems

by

Marcel Adrian Ambroze

Abstract

The advent of turbo codes in 1993 represented a significant step towards realising the ultimate capacity limit of a communication channel, breaking the link that was binding very good performance with exponential decoder complexity. Turbo codes are parallel concatenated convolutional codes, decoded with a suboptimal iterative algorithm. The complexity of the iterative algorithm increases only linearly with block length, bringing previously unprecedented performance within practical limits.

This work is a further investigation of turbo codes and other concatenated schemes such as the multiple parallel concatenation and the serial concatenation. The analysis of these schemes has two important aspects, their performance under optimal decoding and the convergence of their iterative, suboptimal decoding algorithm.

The connection between iterative decoding performance and the optimal decoding performance is analysed with the help of computer simulation by studying the iterative decoding error events. Methods for good performance interleaver design and code design are presented and analysed in the same way.

The optimal decoding performance is further investigated by using a novel method to determine the weight spectra of turbo codes by using the turbo code tree representation, and the results are compared with the results of the iterative decoder. The method can also be used for the analysis of multiple parallel concatenated codes, but is impractical for the serial concatenated codes. Non-optimal, non-iterative decoding algorithms are presented and compared with the iterative algorithm.

The convergence of the iterative algorithm is investigated by using the Cauchy criterion. Some insight into the performance of the concatenated schemes under iterative decoding is found by separating error events into convergent and non-convergent components. The sensitivity of convergence to the E_b/N_o operating point has been explored.

Contents

List of Abbreviations

BER	Bit Error Rate
DMX	Demultiplexor
d_{free}	free distance
$d_{free-eff}$	effective free distance, minimum possible $(OW_2)_{min}$ over all interleavers I of length N for given component codes
FER	Frame Error Rate (block error rate)
HIWHOW	High Information Weight High Output Weight (error block)
IW	Information Weight
IW_k	Information Weight $IW = k$
LIWHOW	Low Information Weight High Output Weight (error block)
LIWLOW	Low Information Weight Low Output Weight (error block)
ML	Maximum Likelihood
MPCCC	Multiple Parallel Concatenated Convolutional Codes
MX	Multiplexor
NC	Non-recursive Convolutional (code)
$NC(f,g)$	NC with feedforward polynomials f and g
NSC	Non-recursive Systematic Convolutional (code)
$NSC(f)$	NSC with feedforward polynomial f
OW	Output (code) Weight
OW_k	Output (code) Weight for $IW = k$
$(OW_k)_{min}$	minimum OW_k for a given interleaver I of length N
RSC	Recursive Systematic Convolutional (code)
$RSC(f/g)$	RSC with feedforward polynomial f and feedback g
SCCC	Serial Concatenated Convolutional Codes
SISO	Soft Input Soft Output

List of Figures

List of Tables

Acknowledgement

Firstly, I would like to thank my supervisors, Dr. Graham Wade and Prof. Martin Tomlinson for their invaluable guidance and support throughout the course of this research.

I gratefully acknowledge the support of the University of Plymouth and the CVCP committee which through the research scholarships made all this work possible.

I would like to thank my family for their unwavering help and encouragement.

I would also like to thank my friends Levi Tóth, James Slader and Peter van Eetvelt for their company, help and advice.

Author's declaration

At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award.

This study was financed by a University of Plymouth studentship and an Overseas Research Student (ORS) scholarship.

The work has been regularly presented at research seminars, three major journal papers (Ambroze et al., 1998a; Ambroze et al., 1998b; Ambroze et al., 2000c) have been published, one paper was submitted (Ambroze et al., 2000a) and another will be submitted for publication (Ambroze et al., 2000b).

Signed .....

Date 25/10/2000 .....

Message decoded with a rate $R = 1/3$, $N = 1280$, $M = 4$, turbo code on a QPSK, AWGN channel at $E_b/N_o = 1\text{dB}$:

Iteration #1, code #1

```
"The"f}<eamental pro?lem of com-?nucation yrvhat of‘vgppw??#i?g?qL?gmg<
|gm?t eY5hCz e|actlybjRr+ppb??cmat?&{?p?MAssage seleIted at another
 qo)>?+# C&Q.‘V?!j˜on
```

Iteration #1, code #2

```
"The‘fu˜eamenTal problmm of com-dnmcation mr ˜hap o&‘ˆepzgd?Cing a‘ gne
(xomnt ei4hEr u|actly‘jZ0?ppr0?imat%n{ p?mEsscfe qelected at anothe?
 p/i˜v*" C.?. Shannon
```

Iteration #2, code #1

```
"The fundamental problem of com-tnication is that ofh\epr?ducing at Gne
(|gmnt either exactly oR8?‘proximately a message selected at another
 point." C.E. Shannon
```

Iteration #2, code #2

```
"The fundamental problem of communication is that of zeproducing at one
 point either exactly or(!pproximately a message selected at another
 point." C.E. Shannon
```

Iteration #3, code #1

```
"The fundamental problem of communication is that of reproducing at one
 point either exactly or approximately a message selected at another
 point." C.E. Shannon
```

Chapter 1

Introduction

1.1 Background to the investigation

1.1.1 Introduction

Channel capacity

The problem of any communication system, as shown in figure (1.1), is to send data from the transmitter to the receiver through the channel, with as few errors as possible. The errors are due to the channel, which modifies the transmitted values. The probability of bit error (or bit error rate) is defined as

$$\text{BER} = \frac{\text{Number of errors}}{\text{Total number of bits}} \tag{1.1}$$

In order to protect the information bits, they can be separated into blocks of length N, and coded by adding *redundant (parity)* bits to each block. Whilst the information bits are generally independent, the redundant bits should be dependent on all the information bits in the block. Either the information and redundant bits together or

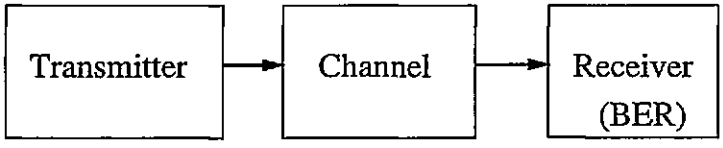


Figure 1.1: Basic communication system

just the redundant bits constitute the *code bits*. The code bits are transmitted over the channel. The code rate R is obtained by dividing the number N of information bits in each block by the number of code bits. Each transmitted block represents a *codeword*. The *block error rate* (*frame error rate*) is defined as:

$$\text{FER} = \frac{\text{Number of blocks decoded with at least one bit error}}{\text{Total number of blocks}} \tag{1.2}$$

Given this transmission system, the question is what is the FER and/or BER that can be achieved, and how can it be reduced.

In 1948, C.E. Shannon introduced to the coding community, confined between the *sphere packing bound* and the *random coding bound*, a fundamental result of channel coding theory: the *Shannon limit* (Shannon and Weaver, 1949). The two bounds are a lower and an upper bound (in this order) on the block error rate, given the channel characteristics and the block length N. Shannon has shown that the block error rate can be reduced to zero as $N \to \infty$ as long as the bit rate (number of information bits transmitted per second) is lower than a value called the *channel capacity*, C. The channel capacity represents the Shannon limit in terms of bit rate. It is dependent on the statistical model of the channel. Equation (1.3) presents the capacity formula for an additive white Gaussian noise channel ($AWGN$),

$$C = W \log_2 \left(1 + \frac{C}{W}\frac{E_b}{N_o}\right) \tag{1.3}$$

where W is the available bandwidth, and E_b/N_o is the information bit to noise energy ratio. By reformulating equation (1.3) as in (1.4),

$$\frac{E_b}{N_o} = \frac{2^{C/W} - 1}{C/W} > \lim_{C/W \to 0} \frac{2^{C/W} - 1}{C/W} = ln(2) = -1.6\text{dB} \tag{1.4}$$

it can be observed that, even for unlimited bandwidth or bit rate reducing to zero, the E_b/N_o cannot be less than $E_b/N_o = -1.6$dB. This value is the Shannon limit for $AWGN$ channels in terms of E_b/N_o. Thus, on an $AWGN$ channel, for any E_b/N_o value higher than -1.6dB and any given bandwidth, information can be transmitted with as few errors as necessary. The conditions are that it is transmitted slower than the value C resulting from equation (1.3), and the block length N is large enough.

Code Rate R	E_b/N_o [dB]
1/2	0
1/3	-0.55
1/4	-0.82
1/6	-1.08
0	-1.6

Table 1.1: Shannon limit for different code rates
Shannon limit for the AWGN channel with QPSK modulation and different code rates.

The value $E_b/N_o = -1.6$dB is the ultimate limit for the AWGN channel. Practical systems employing a given modulation scheme should achieve this limit as the code rate R reduces to zero. For a non-zero code rate, the limit is higher. For QPSK modulation the dependence of this limit on code rate is presented in table (1.1). The values were taken from (Dolinar et al., 1998). The E_b/N_o limit decreases asymptotically with decreasing code rate.

Shannon's result is non-constructive: the random coding bound gives the average performance of randomly chosen codes, based on the idea that there exists a code that performs better or at least as good as the average. Generally, it is deemed that choosing a code at random will give similar performance. The problem is, if a code is chosen at random, it does not have structure to simplify its decoding. The decoding will mean comparing the received sequence with each of the codewords, to find the codeword that resembles the received sequence the most. This means that 2^N codewords should be tried, and thus the complexity of the algorithm depends exponentially on N. In most of the cases, the required value of N makes this option impractical, if not impossible.

Block codes

The impracticality of using randomly chosen codes has led to the construction of codes with algebraic structure, based on simple mathematical rules. They are generally known as *block codes*, since they encode information in independent blocks of length N. The disadvantage of these codes is the way they are decoded, which implies that the received data has to be thresholded before the decoding process can begin (*hard decision decoding*). This results in an information loss that can be significant. Also, the problem of choosing the algebraic structure to maximise performance is non-trivial.

Convolutional codes

As opposed to block codes, convolutional codes do not separate data into blocks, but encode it in (theoretically) infinite streams. The equivalent block length is described by the *constraint length* of the code. Their optimal decoding algorithm is based on a labelled graph (trellis) and it can take unquantised inputs (*soft decision decoding*). The complexity of the trellis depends exponentially on the constraint length. Increasing the constraint length is a necessary but not sufficient condition for improving performance, and further design is required. The block codes can also be decoded based on a trellis, but usually their trellis is much more complex and irregular. The decoding algorithm is *optimal* if it searches the whole code space for the *most likely* codeword, given the received data. Suboptimal algorithms exist for convolutional codes that allow much higher constraint lengths because they do not search the whole code space. The most important group of such algorithms are the *sequential decoding algorithms.* They use a *tree* representation of the code instead of a trellis.

Concatenated codes

Concatenated codes were introduced by Forney (Forney, 1966) in order to obtain higher block lengths with lower decoding complexity. An overall code with higher block length is obtained by encoding the data with a first (outer) code, and then encoding the output of the outer code with a second (inner) code. This type of concatenation is known as *serial concatenation.* The decoding process is performed in two stages: the inner code is decoded first, then the output is decoded by the outer code. The problem of this scheme is that the information that each decoder receives is incomplete relative to the overall code, and thus there is a loss in performance as compared to the decoding of the concatenation as a single overall code.

Turbo codes

Before 1993, the best ways to obtain good performance at low E_b/N_o were (Hagenauer et al., 1996):

- sequential decoding of long constraint length convolutional codes (limited to $E_b/N_o \geq 2$dB, corresponding to the *computational cutoff rate*).

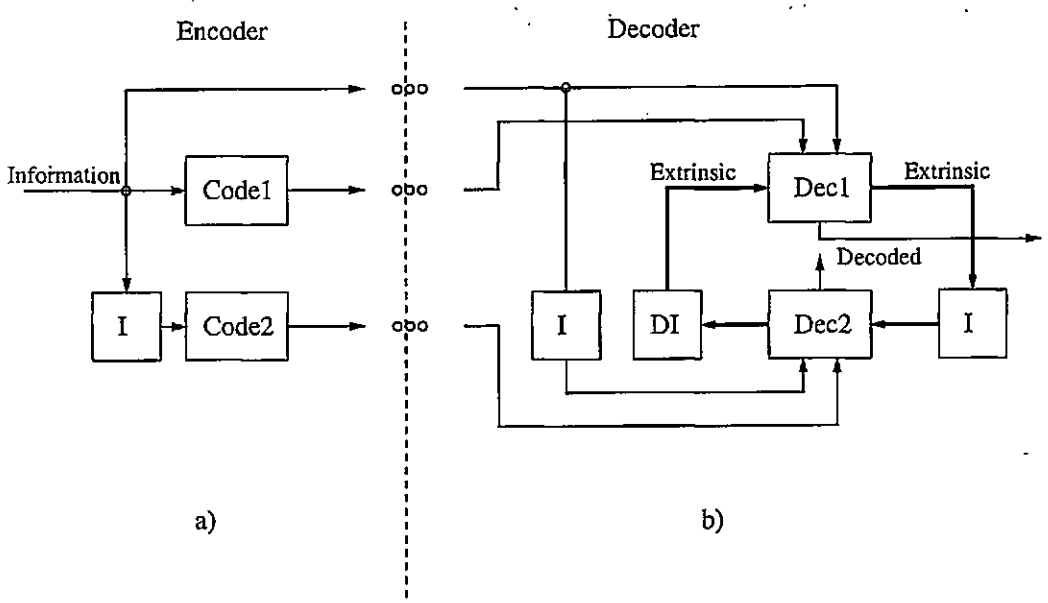


Figure 1.2: Turbo code scheme
Turbo codes are parallel concatenations of convolutional codes: a) encoder and b) decoder. Dec represents a convolutional decoder block, I is the interleaver and DI the inverse interleaver (deinterleaver). Both codes can output decoded information, but only the non-interleaved bits from Dec1 are passed further in the receiving chain.

- concatenated codes of high complexity (NASA: constraint length 14 (16384 trellis nodes for each decoded bit) convolutional code concatenated with long Reed Solomon block code, decoded in 4 iterative stages, has low BER at $E_b/N_o =$ 1.4dB)

The turbo codes were introduced in 1993 by a group of French researchers (Berrou et al., 1993b). They used a block length of $N = 256 * 256 = 65536$ bits, and achieved BER $= 10^{-5}$ at $E_b/N_o = 0.7$dB. The encoder is a *parallel* concatenation of convolutional codes, as shown in figure 1.2(a). The information block is encoded directly by the first code and through an *interleaver* by the second code. The interleaver modifies the order of the information bits in the block. The output (redundant) bits of the two encoders and the information bits are sent over the $AWGN$ channel.

The decoder is presented in figure 1.2(b). The decoding is done in stages. For each received block, the first code is decoded using its corresponding received values, and produces its version of the information bits and also a new type of information called *extrinsic information*. The second code is decoded using its corresponding received values *and* the extrinsic information from the first code (interleaved), producing its

version of the information bits and extrinsic information. Then the first code is decoded again, this time also making use of the extrinsic information from the second code and producing a new version of the information bits and extrinsic information. The process continues in the same fashion for a given number of iterations. The name of turbo codes was inspired by this iterative algorithm with feedback, similar to the process used by a turbo engine. The decoding algorithm is suboptimal, due to the fact that each code can decode only a part of the received values, the part that it has produced in the encoding process. This is characteristic of decoding in stages, and it is the price that was usually payed for lower complexity before turbo codes. Turbo codes instead use the extrinsic information as a link between the two decoders. Each decoder "translates" its part of the received values to the other decoder in terms of probabilities of the information bits, which are common to the two codes. The iteration is needed because what each decoder "understands" from its part of the received values changes with the information about the "invisible" part that it receives from the other decoder. Also, each decoder has to report back only the part of the information that regards its own received values, and not to repeat the information which it has received from the other code, since that will produce a bias in the next decoding. The extrinsic information is calculated to approximate these needs, as opposed to the decoded information, which contains the whole information available after each decoding. By iterating this information exchange, the decoded values should converge to the values that would be produced if the overall code were decoded as a single code. Unfortunately, the extrinsic information defined in (Berrou et al., 1993b) and subsequently used in all iterated schemes is obtained in a rather heuristical fashion, and the performance improvement has been observed by simulation. Also, it is difficult to determine what the overall code would produce, since its complexity depends exponentially on N.

The extrinsic information needs to be "soft" i.e. unquantised (theory) or having a reasonable number of quantisation levels (practice) in order to reduce restrictions in possible values, and allow a smooth convergence. If the extrinsic information was coarsely quantized (and the coarsest quantization is binary), it could happen that the steps the decoder needs to take towards convergence are not in the representable space, and so convergence would be impossible. This is why the decoder for each of the component codes needs to be a Soft Input Soft Output (SISO) decoder.

The relatively low complexity of the turbo decoder is due to the separate decoding of the two codes. The interleaver determines the block length of the overall code, but the decoding complexity for each code depends on its constraint length, which in (Berrou et al., 1993b) was as low as 4 (16 trellis nodes/decoded bit as opposed to 16634 in the NASA code). Thus, the complexity of the algorithm increases linearly with block length (complexity/decoded bit is constant).

Another advantage of turbo codes over previous codes is that very good performance can be obtained without any design effort: the component codes are simple convolutional codes and the interleaver is a randomly chosen permutation. One only has to increase the interleaver length to obtain the desired BER. Of course, a more careful design can produce the same BER with shorter interleavers, and thus shorter receiving delays.

The study of turbo codes has two major parts: the study of their potential performance under the assumption of optimal decoding and the study of the suboptimal iterative decoder.

1.1.2 Optimal decoding

A maximum likelihood decoder maximizes the probability that either a) a codeword or b) each bit in a block separately has been transmitted, given the received values by searching the whole code space (considering every codeword). Case a) describes a *sequence maximum likelihood* decoder and case b) a *bit maximum likelihood* decoder. A maximum likelihood decoder is also known as an *optimal* decoder.

Weight spectra and the union bound

The sequence maximum likelihood error probability can be computed for linear codes by determining their *weight spectra* and using the *union bound* formula to obtain the FER and/or BER (Benedetto and Montorsi, 1996c). A linear code is a code for which the sum of two codewords is also a codeword. Turbo codes are linear codes (Benedetto and Montorsi, 1996c). The information/code weight of a codeword is defined as the number of information/code bits that are one of the given codeword. The weight spectra is a table associating each code weight d with the number of codewords having code weight d, known as the *multiplicity* $a(d)$ of the code weight d. The smallest weight

d in the weight spectra represents the *free distance* of the code, d_{free}. The FER can be reduced in two ways: a) by increasing d_{free} and b) by reducing the multiplicity of low code weights, starting with d_{free}. The BER can be decreased in the same way, and also by reducing the information weight associated to low code weights, starting with d_{free}.

The main design criteria for block and convolutional codes was increasing their d_{free}. One of the goals of code design was to obtain *asymptotically good* codes, codes for which the value of both $\frac{d_{free}}{N}$ and code rate R remain non-zero as the block length $N \to \infty$. This proved to be a very difficult task, although it was shown that such codes do exist (Michelson and Levesque, 1984).

Fortunately, as discussed above, increasing d_{free} is not the only way to obtain good performance. Turbo codes using interleavers chosen at random have the same d_{free} (with high probability) as the interleaver length N is increased (Benedetto and Montorsi, 1996c), and thus they are not asymptotically good. Essential to the performance of turbo codes is that, as the block length is increased, the number of blocks in error and the number of bit errors in an error block remains relatively constant (generally, two bits in error/error block, as shown in (Perez et al., 1996)). In this way, more bits will be transmitted for the same number of errors, and thus

$$\text{BER} = \frac{constant}{N} \to 0 \tag{1.5}$$

as $N \to \infty$. The $\frac{1}{N}$ factor in the BER of turbo codes is called the *interleaver gain*, since the property discussed above is due to the interleaver. The interleaver gain was introduced in (Benedetto and Montorsi, 1995b).

Attempts to determine the optimal decoding performance of turbo codes can be classified by the way they consider the interleaver in a) fixed interleaver methods which study the performance of turbo codes using a given interleaver and b) probabilistic methods which determine the probability of a given performance when the interleaver is chosen at random.

Fixed interleaver methods

In this case, all the parameters of the turbo code are given, and an extensive computer search is performed to obtain the first several components of the weight spectra. Fixed interleaver methods have been presented in (Podemski et al., 1995; Daneshgaran and Mondin, 1997b). In (Seghers, 1995), a similar method is applied to determine the d_{free} of a given turbo code. The appealing aspect of this approach is that it characterizes the error performance of the code exactly for each given interleaver, making possible a direct comparison with the output of the iterative decoder. Unfortunately, their complexity depends strongly on the maximum weight considered, d_{MAX}. The interleaver lengths that could be considered also depend on d_{MAX}. In (Seghers, 1995), a turbo code having $N = 65536$ is considered, but for a very low $d_{MAX} = 6$. Another method is presented in (Ambroze et al., 1998b) and also in this work. Usually, the d_{free} and a few higher weight components of the spectra can be computed for $N < 1000$.

A different fixed interleaver method is presented in (Breiling and Hanzo, 1997a) and in a more complete form in (Breiling and Hanzo, 1997b). It is based on determining a turbo code trellis and performing a computer simulation using an optimal decoder to obtain the BER. The obtained BER is compared with that of the iterative decoder for the same received values, and it was found that the iterative decoder is about 1dB away from the optimal decoder. This method can be applied to short interleavers or longer interleavers that verify a certain constraint. The significance of the result is limited to these types of interleavers, and it is possible that the iterative decoder has better performance for other interleavers, which cannot be approached in this way. The possibility of splitting the hypertrellis into parts that could be decoded separately is suggested as an alternative to the iterative decoder.

The complexity of the hypertrellis is studied in (Benedetto et al., 1997c). The general trellis complexity of block codes is an area that received a lot of interest, before and after turbo codes as in (Manoukian and Honary, 1997; Wolf, 1978; Kiely et al., 1996; Kiely et al., 1995a; Kiely et al., 1995b). Optimal decoding of turbo codes as block codes using the hypertrellis has rekindled the search for the fabled *minimal trellis* of block codes, the least complex possible trellis representation of the code (Benedetto et al., 1997c).

A brute force approach to optimal decoding of a rate $R = 1/4$ turbo code with block

length $N = 16$, by enumerating all codewords is presented in (Divsalar and Pollara, 1995c). It was concluded that in this case, the iterative decoder produced a BER close to that of the optimal decoding, becoming "*slightly suboptimum*" as the E_b/N_o was reduced under $E_b/N_o = 4$dB. Also, determining an incomplete weight spectra by enumerating only codewords with information weight $IW \leq 3$ is mentioned in this paper to be feasible for $N \leq 1024$.

Probabilistic methods

The fixed interleaver methods offer a limited insight on the effect of code parameters on its performance, and thus do not provide design criteria for turbo codes. The most successful methods to characterise the performance of turbo codes based on their parameters are the probabilistic methods. As opposed to fixed interleaver methods, they either determine the *probability of a weight spectra* when the interleaver is chosen at random or the *average weight spectra*, the average of the weight spectra of all turbo codes that have an interleaver of a given length N. To choose an interleaver "at random" is to choose an interleaver with a uniform probability of $1/N!$ where $N!$ is the total number of interleavers of length N.

The probabilistic methods are actually a combinatorial study of *interleaver mappings*. Due to the interleaver, a codeword of the first code is associated (mapped) to a codeword of the second code. The two codewords share the same information weight, as they encode the same information bits in a different order. Since the higher the weights at the start of the weight spectra, the better the performance of the overall code, a codeword with a low code weight from one code should be mapped by the interleaver into a codeword with high weight of the other code. Pushing this idea to the limit, an "ideal" interleaver is introduced in (Svirid, 1995) and also mentioned in (Seghers, 1995). The codewords of each code are separated into groups sharing the same information weight, and ordered according to their code weight. For each group, the ideal interleaver maps the codeword of the first code with highest weight to the codeword of the second code with the lowest weight and so on. The author determines that the interleaver is "ideal" for two reasons: a) it gives the lowest error rate over all turbo codes with the given component codes and any codeword mapping and b) it does not exist. It is also stated that, although the performance of a turbo code using

the ideal interleaver can be used as a lower limit on turbo code performance, it is a very weak bound since it is too far from that of turbo codes using real interleavers.

A more realistic approach is presented in (Divsalar and Pollara, 1995c). It calculates the probability that a given codeword of the first code will be associated with a given codeword of the second code when the interleaver is chosen at random. It has been proved that this probability depends strongly on the information weight of the two codewords and it does not depend on their code weight.

Probably the most powerful and complete method to study turbo codes combines the probabilistic methods with a random coding flavour: the *uniform interleaver* approach, extensively presented in (Benedetto and Montorsi, 1995a; Benedetto and Montorsi, 1995b; Benedetto and Montorsi, 1996c), and subsequently used in most of the papers that study the performance of turbo codes, especially the weight spectra aspect. The uniform interleaver of length N is (Benedetto and Montorsi, 1996c) :

"*A probabilistic device which maps a given input word of weight w into all distinct* $\binom{N}{w}$ *permutations of it with equal probability* $1/\binom{N}{w}$".

It turns out that the weight spectra of a turbo code using such kind of device for an interleaver is in fact the average of the weight spectra of all turbo codes for all interleavers of a given length. The usefulness of this method relies on the argument that the average results must be equaled or outperformed by at least one real turbo code of the given length. Comparisons with results obtained using the iterative algorithm and fixed randomly chosen interleavers show that the performance of turbo codes is close to the average bound.

The exact implementation of the method implies computing the weight spectra of the two (usually identical) block codes which result from truncating the component convolutional codes to the length of the interleaver. This can be made independent of block length, its complexity depending only on d_{MAX} and the complexity of the component code spectra. In this way, large interleaver lengths and high d_{MAX} values can be investigated.

Error bounds

The main method of estimating the performance of turbo codes, as presented above, is by using the weight spectra and the union bound to get an upper bound on the error

probability. Unfortunately, it has been found in (Divsalar et al., 1995) that this bound is not tight at low E_b/N_o, but it diverges, taking values higher than one. Because of this divergence, the union bound cannot be used to characterise the performance of turbo codes at E_b/N_o values close to the limit, although they have good performance at these values. This is why tighter error bounds have been derived, as in (Duman and Masoud, 1998; Viterbi and Viterbi, 1998), based on a bounding technique introduced by Gallager in (Gallager, 1965). An investigation of the new bounds is presented in (Divsalar, 1999). Since these bounds are generally difficult to apply, a simpler (tight) bound is also proposed.

1.1.3 Iterative decoding

Important results in studying the potential performance of turbo codes have been obtained by assuming optimal decoding. Unfortunately, the real decoder is not optimal, but a *suboptimal iterative* algorithm. This raises the problem of convergence and also closeness to the optimal performance.

Convergence

The problem of convergence is the problem whether the output of the iterative decoder stabilises at a fixed value or it keeps changing with iteration. A study of the iterative decoder for very short block lengths, $N \in \{1, 2, 3\}$ is presented in (McEliece et al., 1995). The results show that the iterative decoder, although it always converges to the optimal values for $N \in \{1, 2\}$, does not necessarily converge for $N = 3$, and, if it does converge, it does not always converge to the ML codeword. Unfortunately, the result is limited to inpractical values of N, and it is possible that the situation improves with increasing block length.

In (Moher, 1998a) the iterative algorithm as used in turbo codes is presented as a suboptimal implementation of the principle of iterative cross entropy minimisation.

The impact of *correlation* on convergence is often mentioned (Berrou et al., 1993b; Hagenauer et al., 1996), but not quantified. In (Berrou et al., 1993b), an empirical interleaver design criteria to reduce correlation is mentioned: the correlation is reduced by making sure that bits that are close together in the non-interleaved stream (at the input of the first code) are situated far apart in the interleaved stream (at the input of

the second code).

Turbo codes received a sudden interest from the artificial intelligence community when it was discovered that the turbo decoding algorithm is an instance of *belief propagation in connected graphs* (Frey and MacKay, 1997; Wiberg, 1997; Kschischang and Frey, 1998; McEliece et al., 1998). An optimal algorithm exists to solve these type of graphs, the Pearl belief propagation algorithm. Unfortunately, this algorithm is known to converge only for graphs without loops, whereas turbo code graphs present loops. It was concluded that graphs with loops are actually more interesting and there is a lot of insight to be obtained by studying them.

Closeness to optimal performance

The problem of closeness to optimal performance is the problem of what does the iterative algorithm converge to. The association of the error floor (observed in simulations using the iterative decoder) with the d_{free} of the codes (a property of an optimal decoder) shows that at least for high E_b/N_o the performance of the iterative decoder is close to the optimal decoding performance (Benedetto and Montorsi, 1996c; Perez et al., 1996).

In (Barbulescu, 1998), a qualitative proof is given for the convergence of the iterative decoder to the transmitted data. The proof relies on the property of the MAP algorithm to minimise the bit error probability to show that the MAP functions are *contractions* and thus the output *must* converge to the transmitted data (Sawyer, 1978). One objection to this theory is that the MAP blocks exchange extrinsic information, and not decoded information, and the minimum error probability property applies to the decoded information.

Improving convergence

An iterative decoding suitability (IDS) measurement was recently introduced in (Hokfelt et al., 1998; Hokfelt et al., 1999c; Hokfelt et al., 1999e). It is based on calculating the *linear correlation coefficient* between the extrinsic values at the input and output of the SISO decoders. The IDS characterises the uniformity of input/output correlation values over the code block, based on the idea that a non-uniform distibution of correlation degrades convergence. This measurement has been used to design interleavers

that improve convergence. In (Andersen, 1999) it was observed that the component codes affect the performance of the iterative decoder. Non-optimal codes (as discussed in section 1.1.4) were found by simulation that performed better at low E_b/N_o than the optimal codes, although they performed worse at high E_b/N_o, where the optimal design methods are valid. This was loosely explained by the fact that the iterative decoder converges in small steps between codewords that are close together. Since the optimal codes have better distance properties, the steps of the iterative decoder have to be bigger, as opposed to non-optimal codes. This produces disagreement between the two decoders, and thus nonconvergence. The authors propose the usage of non-optimal codes and concatenating the turbo code with a block code that improves the performance at high E_b/N_o. A compromise is proposed in (Takeshita et al., 1998a) where the use of a non-optimal code concatenated with an optimal code is proposed to obtain a compromise performance in the whole E_b/N_o range.

Another way to improve convergence is by *simulated annealing*, a method usually employed in iterative processes. It was used in the first turbo code (Berrou et al., 1993b), by weighting the extrinsic information with an empirical factor dependent on the statistics of the extrinsic values. Although characterised as a tweak factor in (Robertson, 1994), it was nevertheless used again in (Divsalar and Pollara, 1995a).

A more exotic method was forcing a threshold decision on the extrinsic probabilities of some of the bits after several iterations in (Lin et al., 1997). The authors claim an improvement has been obtained in bit error rate.

Non-iterative suboptimal algorithms

Non-iterative suboptimal decoding algorithms have been used to give a new dimension to iterative decoding. Although suboptimal, they could isolate effects that are characteristic to iterative decoding. Unfortunately, such algorithms are limited to short block lengths ($N \approx 100$). Suboptimal non-iterative algorithms are presented in (Narayanan and Stuber, 1998a; Sadowsky, 1997).

Stopping iteration

Usually, the iterative algorithm finishes after a fixed number of iterations has been performed. In order to save computing time, iteration can be stopped when a block

has been correctly decoded as in (Takeshita et al., 1998b; Shibutani et al., 1999) or when it is determined that continuing the process will not produce significant improvement as in (Hagenauer et al., 1996; Robertson, 1994). Schemes that employ a block code to lower the error floor, as discussed in the next section, are more suitable for the first type of stopping criteria, since the block code can be used to establish when the block has been decoded with no errors.

1.1.4 The error floor

The error floor is a flattening of the FER and BER curves obtained by simulating the encoding/iterative decoding process for turbo codes. It was associated in (Robertson, 1994; Benedetto and Montorsi, 1996c; Perez et al., 1996) with the low d_{free} of turbo codes. This is caused by the fact that the low complexity component codes in the turbo code scheme produce low code weight codewords, and some of the low code weight codewords of the first code are still associated by the interleaver with low code weight codewords of the second code. It was shown in (Benedetto and Montorsi, 1996c; Perez et al., 1996) that this happens with high probability when the interleaver is chosen at random. The error floor has a theoretical advantage and a practical drawback: it shows that the performance of the iterative decoder is close to optimal (at least at high E_b/N_o values), but also it limits the performance of turbo codes with randomly chosen interleavers. There are many approaches to the error floor problem such as interleaver and component code design, serial concatenation with an inner block code and extended concatenated schemes such as the multiple parallel concatenation and the serial concatenation.

Interleaver design

The interleaver is designed to reduce the probability of associating low code weight codewords of the two codes. An iterative method is presented in (Robertson, 1994). It starts with a given interleaver, finds the codeword association with lowest code weight and breaks it by modifying the interleaver. The procedure is repeated until the minimum code weight is increased. Another method based on computer search is presented in (Koora and Betzinger, 1998). Several methods independent of the component codes, of which the most successful is the S interleaver, are presented in (Divsalar

and Pollara, 1995d). Methods based on modifying the row/column interleaver are presented in (Dunscombe and Piper, 1989; Andersen and Zyablov, 1997; Barbulescu and Pietrobon, 1994). Interleaver design methods based on a cost function are presented in (Daneshgaran and Mondin, 1997a; Hokfelt and Maseng, 1997).

Code design

The component codes are designed by trying to maximise the code weight associated with low information weight sequences. It was shown in (Benedetto and Montorsi, 1995a) that the association of codewords of the two codes having information weight $IW = 2$ and minimum code weight possible for the given codes is the most likely to produce the d_{free} of the turbo code. This is not necessarily the minimum code weight possible for the turbo code, but it is the most likely when the interleaver is chosen at random, and this is why it was called the *effective free distance* of the turbo code, $d_{free-eff}$. The component codes that maximise the value of $d_{free-eff}$ are called *optimal* component codes. Tables of optimal component codes are presented in(Benedetto et al., 1998b).

Concatenation with a block code

Another method is concatenating turbo codes with block codes, to correct residual errors. This is based on the observation that the error floor is caused by a small number of bit errors. Block codes are perfectly capable in lowering a small probability of error into a very small one. This was presented in several papers, like (Burkert and Hagenauer, 1997; Lin et al., 1997; Andersen, 1996; Narayanan and Stuber, 1997).

A more exotic method was ignoring several bit positions in the block, thus giving away some code rate (Oberg and Siegel, 1997). This was justified by the fact that the error protection of turbo codes is not uniform, at convergence, only particular bits are in error. The rate loss of this method decreases with interleaver length.

Multiple parallel concatenation

A different direction was to increase the number of codes and interleavers in the parallel concatenated structure. The fundamental idea behind this approach was that, since one interleaver reduces the probability that two parity sequences from two different

encoders would both have low weight, by adding a new interleaver and code, the probability that the new parity sequence would also have low weight is reduced even further. This subject is studied in (Divsalar and Pollara, 1995a). It was proven (using average methods) that the interleaver gain term depends on the number of codes in the concatenated system, and the probability of error is:

$$\text{BER} \sim \frac{1}{N^{m-1}} \tag{1.6}$$

for this extension, where N is the interleaver length and m is the number of component codes.

Serial concatenation

Whereas care is still needed for the MPCCC to avoid interleavers that would produce a low d_{free}, it is not the case with the new type of concatenation proposed in (Benedetto and Montorsi, 1996a). It is, in fact, a revival of the classical serial concatenated scheme, with a different, 'turbo' decoding algorithm. The theoretical analysis, presented in (Benedetto and Montorsi, 1996b), has shown that the interleaver gain is now dependent on the d_{free} of the outer code, and the probability of error is:

$$\text{BER} \sim \frac{1}{N^{\left\lfloor \frac{d^o_{free}+1}{2} \right\rfloor}} \tag{1.7}$$

where d^o_{free} is the free distance of the outer code, and $\lfloor . \rfloor$ denote truncation. It can be seen that, even for a small value of $d^o_{free} = 5$, the probability of error

$$\text{BER} \sim \frac{1}{N^3} \tag{1.8}$$

It was also shown in (Benedetto et al., 1998a) that, similar to the parallel scheme, the number of concatenated codes can be successfully increased, showing a further performance improvement.

1.1.5 Closeness to Capacity

The performance of turbo codes is usually compared to the ultimate capacity limit, obtained as $N \rightarrow \infty$. Since practical systems impose constraints on the maximum block length, the minimum E_b/N_o that can be obtained with a *finite* block length is determined in (Lazic et al., 1997) and also in (Dolinar et al., 1998) by reformulating Shannon's sphere packing bound. In (Dolinar et al., 1998), the notion of *code imperfectness* is introduced as the difference between the E_b/N_o needed by a code to reach a given FER and the limit corresponding to its block length and code rate. It was established, based on simulation results, that turbo codes are "nearly perfect" since they are $\approx$ 0.7dB away from the ultimate limit for block lengths $N \geq 500$ at FER= 10^{-4}. In this light, well known codes of very short block length $N \leq 48$ are shown to be even closer to the limit corresponding to their block length. The advantage of turbo codes is that they are close to the E_b/N_o limit for block lengths that allow this limit to be drastically lowered. Another unprecedented advantage of turbo codes is that they remain nearly perfect for a large range of block lengths. Unfortunately, turbo codes "*lose their luster of near perfectness*" as the FER is decreased (due to the error floor), and also as the code rate is increased. The serial concatenation is mentioned as a possible solution for the error floor problem in (Dolinar et al., 1998).

1.1.6 Soft Input Soft Output algorithms

The optimal SISO algorithm for convolutional codes is the maximum a posteriori algorithm (MAP), presented as early as 1974 in (Bahl et al., 1974). Before the advent of turbo codes, the Viterbi algorithm has been preferred, due to complexity considerations. The MAP algorithm is a bit maximum likelihood decoder, whereas the Viterbi algorithm is a sequence maximum likelihood decoder. The BER improvement for the MAP algorithm was insignificant at the E_b/N_o values at which convolutional codes with optimal decoding were used, and thus the MAP decoder did not justify its complexity. The Viterbi algorithm outputs binary values. A modification of the Viterbi algorithm to output non-binary values corresponding to the decision reliability for any two converging paths in the trellis, is the *soft output Viterbi algorithm* (SOVA) (Hagenauer and Hoeher, 1989; Berrou et al., 1993a). SOVA has soft output, but it is suboptimal and

thus performs worse than the MAP algorithm. The complexity of the MAP algorithm has been reduced by using the logarithmic function to transform its multiplications into additions resulting in the max-log-MAP algorithm. Unfortunately, this algorithm is also suboptimal (in fact, it was shown in (Fossorier et al., 1998) that its decodings are identical to those of the SOVA algorithm). A correction factor that could be implemented as a small one dimensional table has been employed in (Robertson et al., 1997) to transform the max-log-MAP algorithm into the log-MAP algorithm. This correction factor brings the output of the log-MAP algorithm very close to that of the original MAP algorithm.

A different type of simplifications in the MAP algorithm are based on the fact that, at least at high E_b/N_o or in the last iterations, the probability of most of the trellis states are close to zero, and thus they do not need to be investigated (Frey and Kschischang, 1998; Franz and Anderson, 1998).

1.1.7 Trellis termination

The convolutional codes used in the turbo code scheme should to be transformed into block codes by *terminating* their trellis (Benedetto and Montorsi, 1997). This is accomplished by adding a sequence of redundant bits to the information block, sequence known as *data tail.* The length of this sequence is equal to $k-1$ bits, where k is the constraint length of the code. Although the conventional codes have the data tail composed only of bits of zero, the *RSC* codes that have to be used in turbo codes (Benedetto and Montorsi, 1995c) need a non-zero data tail. This is a problem, since terminating the trellis of one of the codes does not guarantee the termination for the second code. This caused a lot of literature, and all possible combinations have been proposed:

- Transmit two separate data tails, one for each code, in (Divsalar and Pollara, 1995c). This method has the advantage that it can be directly used in any concatenated scheme.

- Transmit no data tail and constrain the interleaver to terminate both codes (Berrou and Jezequel, 1996).

- Transmit one data tail for the first code and constrain the interleaver to also

terminate the second code (Koora and Finger, 1997; Barbulescu and Pietrobon, 1995; Blackert et al., 1995; Joerssen and Meyr, 1994; Khandany, 1998).

- Transmit one data tail and do not terminate the second code (Robertson, 1994).
- Transmit no data tail (Reed and Pietrobon, 1996). This has been proposed as the best choice for short blocks, due to the reduction in code rate caused by the data tail, which is more significant for short blocks.

The effect of trellis termination on the (average) optimal decoding performance is studied in (Benedetto and Montorsi, 1997) where it is found that the trellis of at least one code should be terminated, especially for higher constraint length component codes. Also, an alternative to trellis termination is introduced in this paper in the form of continuously decoded turbo codes, which use the *sliding window* SW-MAP algorithm presented in (Benedetto et al., 1996; Benedetto et al., 1997b; Viterbi, 1998) and a convolutional interleaver instead of a block interleaver. Non-block interleavers are also presented in (Hall and Wilson, 1998a).

The impact of interleaver constraints due to trellis termination on optimal decoding performance is studied in (Hokfelt et al., 1999a; Hokfelt et al., 1999b).

1.1.8 Other research directions

Turbo codes are usually studied assuming an AWGN channel, with BPSK/QPSK modulation and coherent reception. Different channels, such as the Rayleigh channel for *multipath* propagation is studied in (Hall and Wilson, 1998b). Turbo code schemes using non-coherent demodulation are studied in (Hall and Wilson, 1997).

Product codes with iterative decoding present an alternative to turbo codes for high code rates (Goalic and Pyndiah, 1997; Pyndiah et al., 1994; Pyndiah et al., 1996; Aitsab and Pyndiah, 1996; Pyndiah, 1997). Higher code rates for turbo codes can also be obtained by *puncturing*. Puncturing was applied in the original turbo codes to increase their rate from $R = 1/3$ to $R = 1/2$, and it is also studied in (Oberg et al., 1997; Acikel and Ryan, 1999).

The turbo decoder needs an estimation of the channel E_b/N_o in the decoding process. A channel estimation scheme is presented in (Summers and Wilson, 1998), where it is also established that an error of up to 6dB in determining the E_b/N_o value is

acceptable. A different method to estimate the E_b/N_o is presented in (Reed and Asenstorfer, 1997).

Construction of bandwidth efficient schemes using turbo codes has received a significant interest (Robertson and Worz, 1995; Benedetto et al., 1995; Ogiwara and Morillo, 1997; Barbulescu et al., 1997; Benedetto et al., 1997a). A bandwidth efficient scheme based on joint interleaver and trellis design is presented in (Wesel and Cioffi, 1997).

The implementation of turbo code algorithms into DSP show their constraint in papers on reducing the memory needed to store the interleaver permutation (Hokfelt et al., 1999d). The low E_b/N_o at which turbo codes can be used impose unprecedented constraints on synchronisation schemes (Yi, 1997).

Turbo codes are low density parity check codes (MacKay and Neal, 1997), a group of iteratively decoded codes introduced by Gallager in (Gallager, 1963). This generalises the turbo code schemes and places an old theory into a new light. It also means that the methods developed by Gallager in his work can be used to analyse turbo codes, adding an unexpected (and significant) contribution to the theory of turbo codes.

1.1.9 Applications

Turbo codes are used for deep space applications with code rates $R = 0.15 - 0.5$ (Divsalar and Pollara, 1995c). Also, they can be used for satellite communications (Divsalar and Pollara, 1995b; Fonseka, 1999; Barbulescu et al., 1997). Different puncturing methods allow turbo codes to provide unequal error protection for GSM speech transmission. The principle of iterative decoding has been successfully applied to CDMA spread spectrum systems (Moher, 1998b). Applications for image transmission are presented in (Fei and Ko, 1997). In (Ambroze et al., 2000a), the application of turbo codes for video watermarking (copyright protection of video material) is proposed and investigated.

A general trend is to include other blocks of the communication system into the iterative loop to provide an overall, more robust transmission scheme. Thus, parts of the system that have been previously included in the channel from the error correction coding point of view are now active blocks of the iterative decoder. Combined iterative demodulation and decoding is presented in (Narayanan and Stuber, 1998b). Combined iterative channel equalisation and decoding is presented in (Raphaeli and Zarai, 1997).

A conventional transmission system comprises two coding parts: source coding, and channel coding. Due to the possibility to use *a priori information* in MAP decoders, combined source/channel coding and decoding is proposed in (Frias and Villasenor, 1997a; Frias and Villasenor, 1997b; Hagenauer, 1995).

1.2 Thesis structure

This work investigates the performance of turbo codes, multiple parallel concatenation (MPCCC) and serial concatenation (SCCC) under optimal and iterative decoding.

Chapter 2 describes the building blocks, the encoding and decoding algorithms for the three concatenated schemes. Their structure is justified using optimal decoding average performance arguments and also computer search results which confirm and extend the average performance theory. Part of the work from this chapter was published in (Ambroze et al., 1998a).

Chapter 3 applies the theory for the average performance of turbo codes and the other concatenated schemes derived for optimal decoding to schemes using practical interleavers chosen at random. The difference between the average performance and the performance of a given interleaver is investigated by analysing the iterative decoding error events obtained by simulation. The results are completed by using fast search algorithms to obtain the distribution of minimum code weight for $IW = 2$ error blocks for MPCCC when the interleaver is chosen at random. It is shown that turbo codes (2PCCC) using an interleaver chosen at random are close to the average performance but the other MPCCC schemes can show large variations. The effect of interleaver and code design criteria is also investigated. A fast algorithm to construct S interleavers is presented. The S interleaver is improved by eliminating some of the "crossed" error event associations that degrade its performance. Formulae to characterise the performance of the S interleaver are derived. Extensive simulation results are presented. The iterative decoding error events are also used to determine what causes the difference between the optimal decoding performance and iterative decoding performance. The schemes producing the best compromise between optimal/iterative decoding performance are compared, and their decoding complexity analysed. Part of the work in this chapter will be sent for publication in (Ambroze et al., 2000a).

Chapter 4 presents a closer look at ML methods to investigate concatenated schemes. A new method based on the tree structure of the codes is presented and used to produce their weight spectra, which is subsequently used together with the union bound to illustrate the dependence of the code performance on turbo code parameters. The code tree is compacted into a trellis structure, and compared with similar investigations in the literature. Non-iterative suboptimal algorithms based on the tree structure are proposed and investigated. Comparisons of the results obtained by tree search and non-iterative decoding with iterative decoding performance curves are performed. Part of the work from this chapter was published in (Ambroze et al., 1998b).

Chapter 5 investigates the convergence of the suboptimal iterative decoder. The iterative decoding performance curves are separated into non-convergent and convergent performance curves by using the Cauchy convergence criterion. The impact of choosing different design parameters on the non-convergent/convergent performance curves is analysed in order to determine the factors that influence convergence. The convergent curves are shown to be close to the performance obtained by determining the weight spectra and applying the union bound. The error blocks are classified from the convergence point of view and analysed. Methods to determine the correlation of the extrinsic information in the iterative decoding process are presented, and used to illustrate the importance of extrinsic information and of the interleaver for the iterative decoder. Part of the work from this chapter was published in (Ambroze et al., 2000c).

Chapter 6 presents the conclusions and ways to improve the current results.

Chapter 2

Turbo codes and other concatenated schemes

2.1 The channel

The encoded bit stream is considered to be transmitted using a BPSK/QPSK modulation with levels $+1$ and -1. The channel is modeled as an AWGN channel, as represented in figure (2.1). The signal to noise ratio, SNR after the matched filter at the receiving side is:

$$\text{SNR} = \frac{\overline{x_i^2}}{\sigma_i^2} = \frac{1}{\sigma^2} = 2\frac{E_b}{N_o}R \tag{2.1}$$

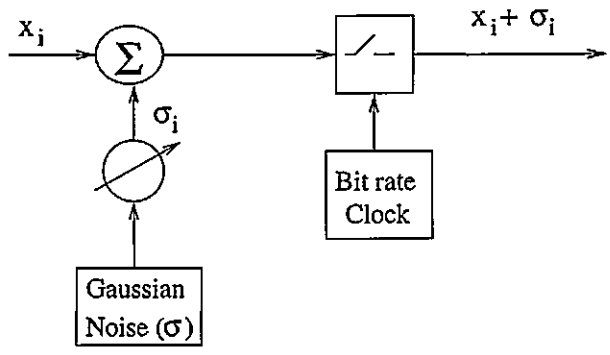


Figure 2.1: AWGN channel model

where R is the code rate. The probability of each level given the received value r_i is:

$$P\{x_i = -1|r_i\} = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(r_i+1)^2}{2\sigma^2}} \tag{2.2}$$

$$P\{x_i = 1|r_i\} = \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(r_i-1)^2}{2\sigma^2}} \tag{2.3}$$

The probabilities can be normalised

$$P\{x_i = -1|r_i\} = \frac{P\{x_i = -1|r_i\}}{P\{x_i = 1|r_i\} + P\{x_i = -1|r_i\}} \tag{2.4}$$

$$P\{x_i = 1|r_i\} = \frac{P\{x_i = 1|r_i\}}{P\{x_i = 1|r_i\} + P\{x_i = -1|r_i\}} \tag{2.5}$$

resulting in

$$P\{x_i = 1|r_i\} = \frac{1}{1 + e^{\frac{-2r_i}{\sigma^2}}} \tag{2.6}$$

$$P\{x_i = -1|r_i\} = 1 - P\{x_i = 1|r_i\} \tag{2.7}$$

where $\sigma = \frac{1}{\sqrt{2\frac{Eb}{No}R}}$ is obtained from equation (2.1).

The pairwise error probability for the $AWGN$ channel for a codeword of code weight ow (assuming the all zeros codeword was transmitted), used in the union bound formula is:

$$P_E\left(\frac{E_b}{N_o}, ow\right) = \frac{1}{2}\text{erfc}\left(\sqrt{R * \frac{E_b}{N_o} * ow}\right) \tag{2.8}$$

where R is the code rate and $\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$ is the complementary error function.

2.2 Turbo codes

2.2.1 The encoder

The encoder for turbo codes is presented in figure (2.2). The main components are the two Recursive Systematic Convolutional (RSC) encoders and the interleaver. These blocks will be discussed in the following sections. The turbo code encodes binary data in a continuous or block fashion, depending on the structure of the interleaver. The

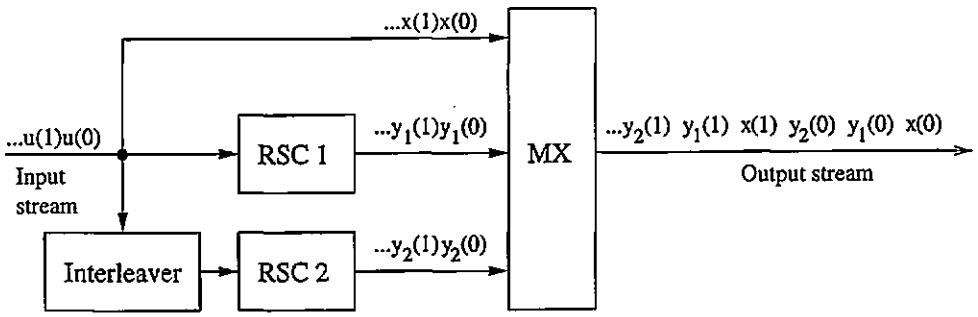


Figure 2.2: The turbo code encoder
A parallel concatenation of Recursive Systematic Convolutional (*RSC*) codes. The output is the multiplex of the information bits (transmitted only once) and the parity bits from the two codes (eventually punctured to reduce code rate). The basic code rate is $R = 1/3$.

information bits are fed directly into the first encoder and through the interleaver into the second encoder. In this way, the second encoder will see a scrambled version of the input bits. The output of the turbo code encoder consists of a multiplex of the information bits and the parity bits of the two *RSC* encoders. The basic rate of the overall code is thus $R = 1/3$ and it can be further increased by puncturing, as in the original paper (Berrou et al., 1993b).

The Recursive Systematic Convolutional Codes

The *RSC* codes are a generalization of systematic convolutional codes. A classical, non-recursive systematic convolutional encoder is shown in figure (2.3). The input of the encoder is a stream of (information) bits, which can be mathematically described as an infinite polynomial by using the delay operator (D):

$$u(D) = \sum_{n=0}^{\infty} u_n D^n \tag{2.9}$$

where the coefficients of the polynomial represent the value ($u_n = 0$ or $u_n = 1$) of the n-th bit in the sequence. The position in the sequence is also given by the exponent n of the delay operator (D). The *weight* of a sequence is defined as the number of bits that are one in the sequence.

As shown in figure (2.3), the encoder for convolutional codes consists of a shift register of $k - 1$ delay elements which store the most recent $k - 1$ information bits.

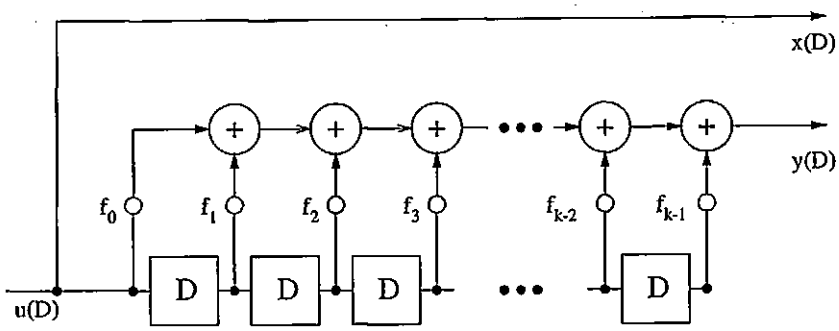


Figure 2.3: NSC(f) encoder

(Non-recursive) systematic convolutional code with constraint length k (memory $M = k - 1$). The input/output and the coefficients of the polynomials are binary, and the additions are performed modulo 2.

The number k is called the *constraint length* of the code, and the value $M = k - 1$ represents the *memory* of the code. Each output bit is obtained at each moment in time as a linear combination of the bits stored in the delay elements and the current information bit. The dependence of the output bits on the information bits can also be expressed in a polynomial form as in the following equations:

$$x(D) = u(D) \tag{2.10}$$

$$y(D) = u(D)f(D) \tag{2.11}$$

where equation (2.10) determines the *systematic* bit and equation (2.11) determines the *parity* bit generated by a systematic convolutional encoder for each information bit. The polynomial $f(D) = \sum_{n=0}^{k-1} f_n D^n$ is the *generator* polynomial of the code. Its binary coefficients determine whether the output of the corresponding delay element will contribute ($f_n = 1$) or not ($f_n = 0$) to the generation of the parity bit.

A recursive systematic convolutional encoder, in its general form is shown in figure (2.4), and for the specific case of $RSC(5/7)$ in figure (2.5). The change from classical systematic convolutional codes consists of the presence of a feedback term denoted $b(D)$ which is computed in a similar way as the parity bit, with the difference that it does not involve the current information bit. The feedback value is added modulo 2 to the information bit. The result becomes the current input bit for the encoding process described above for NSC encoders. The equations describing the new system

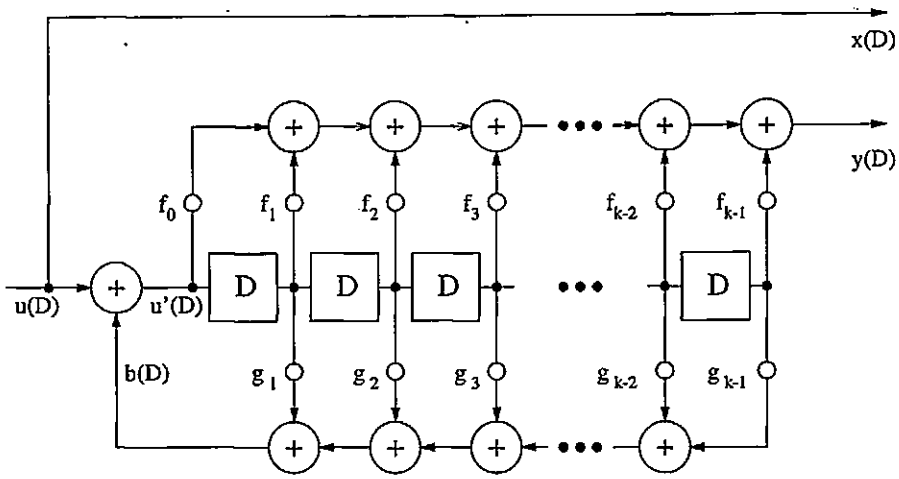


Figure 2.4: $RSC(f/g)$ encoder
Recursive systematic convolutional code with constraint length k (memory $M = k-1$). The input/output and the coefficients of the polynomials are binary, and the additions are performed modulo 2.

are:

$$\begin{aligned} b(D) &= \sum_{n=1}^{k-1} u'(D) g_n D^n \\ &= u'(D)(g(D) - 1) \qquad (2.12) \\ u'(D) &= u(D) + b(D) \qquad (2.13) \\ x(D) &= u(D) \qquad (2.14) \\ y(D) &= u'(D) f(D) \qquad (2.15) \end{aligned}$$

where $g(D) = 1 + \sum_{n=1}^{k-1} g_n D^n$ represents the *feedback polynomial.*

The expression for $u'(D)$ is obtained by combining equations (2.12) and (2.13) :

$$u'(D) = \frac{u(D)}{g(D)} \qquad (2.16)$$

From (2.16), (2.15) the equations describing an RSC encoder become:

$$\begin{aligned} x(D) &= u(D) \qquad (2.17) \\ y(D) &= u(D)\frac{f(D)}{g(D)} \qquad (2.18) \end{aligned}$$

The RSC code is a generalization of the NSC code, because the latter can be obtained

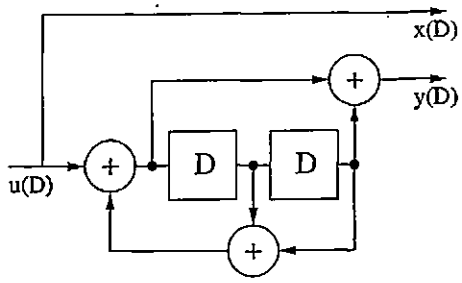


Figure 2.5: $RSC(5/7)$ encoder
Simple constraint length $k = 3$ (memory $M = 2$) RSC encoder, with feed forward polynomial $f = 1+D^2 = 5$ (octal) and primitive feedback polynomial $g = 1+D+D^2 = 7$ (octal).

from the former by setting the feedback polynomial $g(D) = 1$.

One of the most important differences between RSC and NSC codes due to the presence of the feedback term consists of the way they associate the weight of the information sequence with the weight of the parity sequence $y(D)$. More precisely, for an information sequence of weight one, the parity sequence has a *finite* (and low) weight for NSC encoders, as opposed to an *infinite* weight for RSC encoders. This can be mathematically shown by letting $u(D) = D^p$ in equation (2.11), respective (2.15), where p is a positive integer.

For NSC encoders (2.11) becomes:

$$y(D) = D^p f(D) \tag{2.19}$$

The length of $y(D)$ is finite in this case, and it can have at most k bits of one (a weight of k) where k is the constraint length of the code.

For RSC encoders (2.15) becomes:

$$\begin{aligned} y(D) &= \frac{D^p}{g(D)} f(D) \\ &= \frac{D^p}{1 + \sum_{n=1}^{k-1} g_n D^n} f(D) \end{aligned} \tag{2.20}$$

where at least one coefficient $g_n \neq 0$, and $g(D)$ is not a factor of $f(D)$. But $g(D)$ cannot divide D^p either, and thus $y(D)$ has an infinite number of ones. This means that RSC codes will produce a sequence having infinite weight when the input is

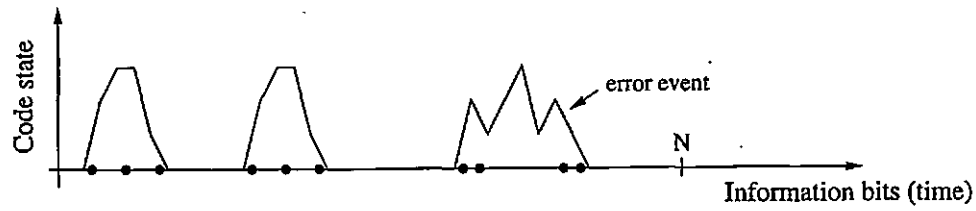


Figure 2.6: Error events
Codeword of a block convolutional code. The dots on the axis represent information bits of 1. All the other information bits are zero.

a sequence having weight one (impulse). Recursive codes can be viewed as Infinite Impulse Response binary filters (IIR), whereas non-recursive codes are Finite Impulse Response binary filters (FIR).

The minimum information weight that determines a finite parity weight for an *RSC* encoder is two, since for any polynomial of the form $g(D) = 1 + \sum_{n=1}^{k-1} g_n D^n$ there exists an integer q so that $g(D)$ divides $D^p + D^{p+q}$. The maximum value of q is obtained when $g(D)$ is *primitive* (Benedetto and Montorsi, 1995c).

Error events

Assuming that the all zero codeword was transmitted, an *error event* of a convolutional code is a sequence of information bits that contains at least one bit of 1. A *terminated* error event causes the encoder to leave the all zeros state and to return to the all zeros state at the end of the sequence. When the convolutional code is transformed into a block code, any non-zero codeword is a concatenation of error events. All error events are terminated with the (possible) exception of the last error event in the block. Terminating the trellis of the convolutional code by appending a data tail means terminating the last error event. A codeword of a block convolutional code is shown in figure (2.6). A given set of error events of the convolutional code can produce several different codewords of the block code, depending on their position in the block. All these codewords share the same information/code weight.

RSC periodicity

The parity sequence $y(D)$ of a *RSC* encoder, corresponding to an information sequence with $IW = 1$ is periodic (Divsalar and Pollara, 1995b). The period T is maximum for

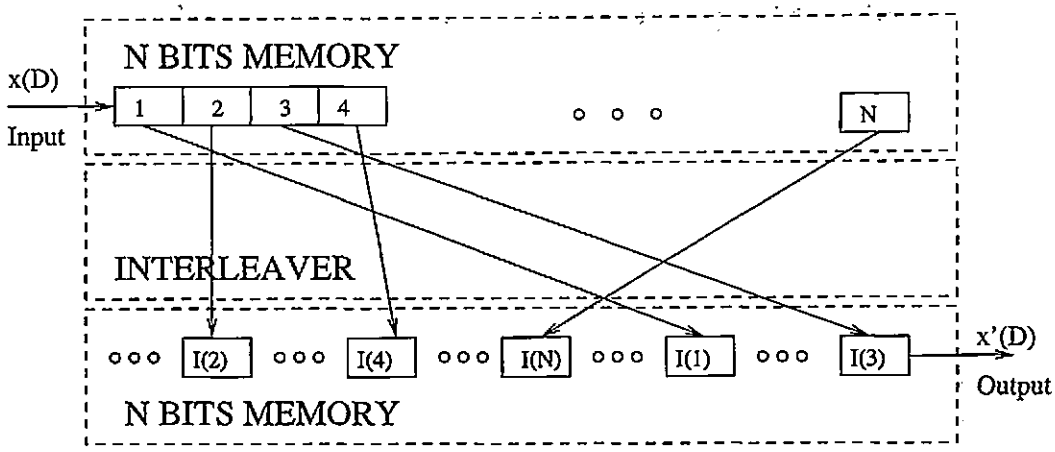


Figure 2.7: Block interleaver

This interleaver is called a *block* interleaver because it is applied separately to each block of N bits and it does not permute the bits outside the given block.

a given constraint length k if the feedback polynomial is primitive. In this case $y(D)$ is a Maximum Length sequence. The parity weight associated with a period is denoted w_T. Due to this periodicity, terminated $IW = 2$ error events can increase their code weight only by multiples of w_T and their length by multiples of T.

The Interleaver

The interleaver structure is represented in figure (2.7). The input bits are first written into an N bit memory. When the memory is filled the bits are read in a different order.

Mathematically, this can be described as follows:

The input sequence,

$$x(D) = \sum_{k=1}^{\infty} x_k D^k = \sum_{k=1}^{\infty} \sum_{i=1}^{N} x_{kN+i} D^{kN+i} \tag{2.21}$$

The output sequence,

$$x'(D) = \sum_{k=1}^{\infty} x'_k D^k = \sum_{k=1}^{\infty} \sum_{i=1}^{N} x_{kN+I(i)} D^{kN+i} \tag{2.22}$$

where I is the interleaver function, $I(i) \neq I(j) \quad if \quad i \neq j \quad , \quad \forall i, j \in 1, ..., N$. For a

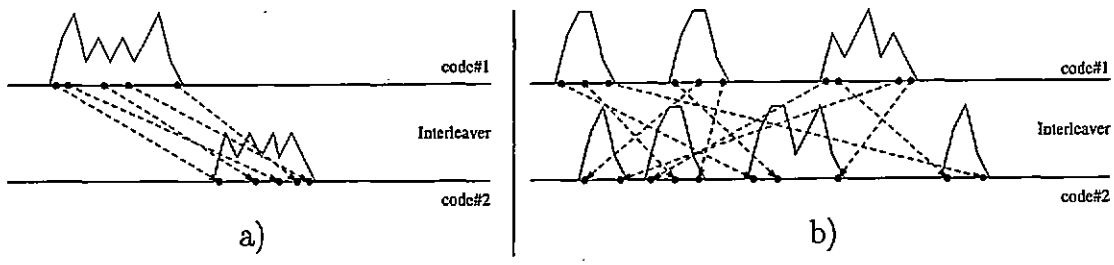


Figure 2.8: Error event mappings
a) given, single error event mapping: a given error event of the first code is associated (mapped) by the interleaver to a given error event of the second code and b) given error events mapping: a set of given error events of the first code is associated by the interleaver to a set of given error events of the second code. They share the same (total) information weight, and the (total) code weight is the same for any position of the error events in the block. Note that different positions in the block give different codewords of the component codes and thus different turbo codeword. The dots on the code axis represent information bits of one, all the other information bits are zero.

block interleaver, the function $I(n)$ can be represented as a permutation,

$$I = \begin{pmatrix} 1 & 2 & 3 & ... & N \\ I(1) & I(2) & I(3) & ... & I(N) \end{pmatrix} \tag{2.23}$$

with $I(k) \in \{1, ..., N\}$, $\forall k \in \{1, ..., N\}$. There are also non-block interleavers, like the convolutional interleavers. They are used for continuous encoding/decoding of turbo codes. They are not treated in this work. Several ways of generating interleaver permutations are presented in the Annex A. The interleaver has a crucial importance in the good performance of turbo codes. It increases the constraint between the code bits so that a code bit depends on many more bits than the short constraint imposed by the component codes. This effect transforms the concatenation into a powerful block code, with block length equal to the interleaver length.

The codewords of the turbo code are associations of codewords of the component codes, determined by the interleaver. The way the interleaver associates codewords of the first code with codewords of the second code is illustrated in figure (2.8).

2.2.2 Optimal decoding performance

The parallel concatenation that forms the turbo code can be considered as an overall, very powerful, single code. The performance of an optimal decoder for this code is

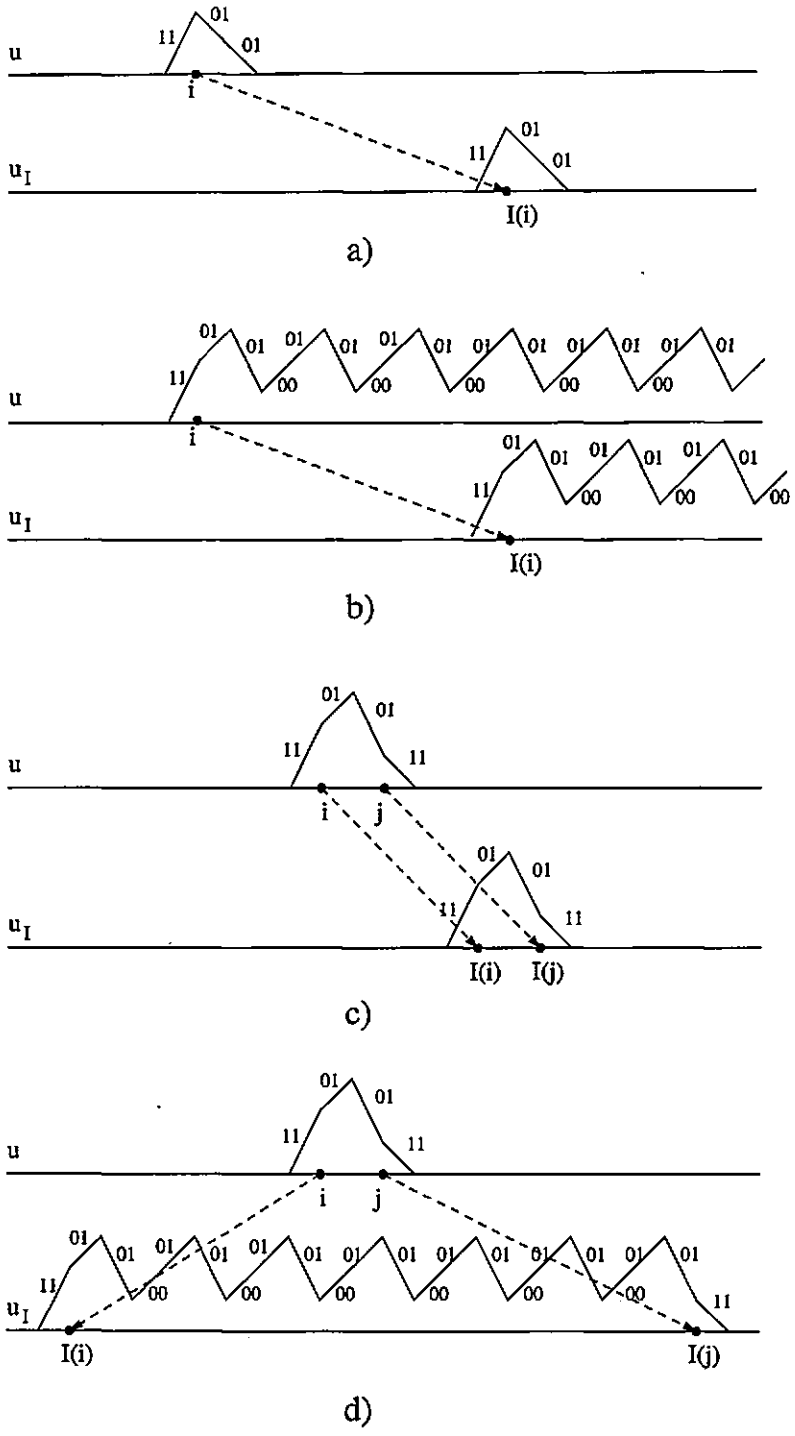


Figure 2.9: The interleaver effect on error events
The interleaver effect on a) $IW = 1$, non-recursive, systematic $NSC(7)$ turbo code b) $IW = 1$, $RSC(5/7)$ turbo code c) $d_{free,eff} = 2 + 4 + 4 = 10$ for a $RSC(5/7)$ turbo code and d) higher overall code weight $IW = 2$ mapping for a $RSC(5/7)$ turbo code. The first bit on each transition is the systematic bit, the second bit is the parity bit.

estimated by computing its weight spectra and using the union bound formula:

$$\text{FER} \leq \sum_{iw,ow} a(iw,ow) P_E\left(\frac{E_b}{N_o}, ow\right) \tag{2.24}$$

$$\text{BER} \leq \sum_{iw,ow} \frac{a(iw,ow) * iw}{N} P_E\left(\frac{E_b}{N_o}, ow\right) \tag{2.25}$$

where $a(iw, ow)$ is the number (multiplicity) of codewords having information weight iw and code weight ow and $P_E\left(\frac{E_b}{N_o}, ow\right)$ is the *pairwise* error probability for a code weight ow. $P_E\left(\frac{E_b}{N_o}, ow\right)$ depends on the channel and it decreases with ow. Its expression for an $AWGN$ channel with BPSK/QPSK modulation is given in equation (2.8).

Consider the following cases:

The component codes are $NSC(7)$

The codeword with the smallest weight for the component code contains a single error event of $(IW = 1, OW = 4)$. As presented in figure 2.9(a), this is always associated by the interleaver with a codeword of the second code containing the same error event in a different position. Thus the weight spectra of the overall code will always contain the $(IW = 1, OW = 1 + 3 + 3)$=$(IW = 1, OW = 7)$ codeword and its d_{free} cannot be higher than 7, independent of the interleaver choice. Also, the multiplicity of the $(IW = 1, OW = 7)$ codewords is at least $a(1, 7) = N$. This is because there are N distinct codewords of the first code caused by all the possible positions of the $(IW = 1, OW = 4)$ error event in the block, and each of them is associated with a different $(IW = 1, OW = 4)$ codeword of the second code, producing an overall codeword with $(IW = 1, OW = 7)$.

The component codes are $RSC(5/7)$

In this case, the code weight caused by sequences with $IW = 1$ is only limited by their position in the block, as shown in figure 2.9(b). Even if truncation happens, the contribution to each overall code weight has a small multiplicity, independent of N.

A more interesting situation is presented in figure 2.9(c). A codeword consisting of an error event with $(IW = 2, OW = 6)$ is associated by the interleaver with a codeword containing the same single error event in a different position in the block. This error

event associates the smallest code weight to an $IW = 2$ and thus generates the overall codeword with the smallest code weight for an $IW = 2$. It can be shown (Divsalar and Pollara, 1995d) that the probability of such an association when the interleaver is chosen at random depends weakly on N, approximately as:

$$P(2, N) = 1 - (1 - 2/N)^N \tag{2.26}$$

As N is increased, $P(2, N)$ converges asymptotically to

$$\lim_{N \to \infty} P(2, N) = 1 - e^{-2} \approx 0.86 \tag{2.27}$$

and thus about 9 out of 10 interleavers will map at least one such pair of codewords. In this way, the d_{free} of this concatenation is with high probability not higher than $OW = 2+4+4 = 10$, a limit which is independent of N. The interleaver does not map all the $(IW = 2, OW = 6)$ error events into themselves. Figure 2.9(d) shows another possibility which generates a higher overall code weight. It can be calculated that the average number of $(IW = 2, OW = 6)$ to $(IW = 2, OW = 6)$ error event mappings over all interleavers of length N is independent of N and it is approximately (Perez et al., 1996)

$$a(2, 10) \approx \frac{\binom{N}{1}\binom{N}{1}}{\binom{N}{2}} \approx 2 \tag{2.28}$$

The d_{free} of the $RSC(5/7)$ component code is caused by an $(IW = 3, OW = 5)$ error event. An $(IW = 3, OW = 5)$ to $(IW = 3, OW = 5)$ codeword mapping would cause a lower d_{free}, determined by an overall $(IW = 3, OW = 3+2+2 = 7)$ codeword. This value is lower than the minimum value corresponding to $IW = 2$, but the probability that at least one such mapping occurs is (Divsalar and Pollara, 1995d):

$$P(3, N) = 1 - (1 - 6/N^2)^N \approx 6/N \tag{2.29}$$

For $N = 600$ only 1 in 100 interleavers performs this mapping at all. This probability decreases with N. The fact that not many interleavers map this pair is the reason for

its average multiplicity being less than one, as computed in (Perez et al., 1996):

$$a(3,7) = \frac{\binom{N}{1}\binom{N}{1}}{\binom{N}{3}} \approx 6/N \tag{2.30}$$

From the examples above it can be observed that at the basis of turbo code performance is the way the interleaver maps a given error event of the first code into a given error event of the second code based on the common information weight of the two error events:

- $IW = 1$: The interleaver always does N mappings.
- $IW = 2$: The interleaver does a small number of mappings with high probability $P \approx 0.86$.
- $IW \geq 3$: The number of mappings decreases with N as $1/N^{IW-2}$.

The interleaver effect presented above is oblivious of code weight. It is the role of the component codes to adapt the code weight associated to each information weight such that improvement can be obtained in performance by exploiting the interleaver effect.

Non-recursive codes (such as $NSC(7)$) associate low weight to $IW = 1$ error events. In this case $P_E\left(\frac{E_b}{N_o}, ow\right)$ in equations (2.24) and (2.25) has a high value. The multiplicity increases linearly with N and thus their contribution to the overall FER increases with N. The contribution to BER remains constant with N and thus the overall BER cannot be reduced to zero as $N \to \infty$. Recursive codes (such as $RSC(5/7)$) associate high code weight to $IW = 1$ error events, which makes $P_E\left(\frac{E_b}{N_o}, ow\right) \approx 0$.

Recursive codes associate low weight with $IW = 2$ error events. In this case, the FER remains relatively constant with N, whereas BER $\sim 1/N$. Since the $1/N$ term is due to the interleaver effect, it was called *interleaver gain.* Higher IW error events have secondary effects, due to the fact that their multiplicity reduces with N.

Due to the interleaver effect, the error events of the component code that have $IW = 2$ and minimum code weight are the most likely to cause the d_{free} of the overall code. This is why the turbo code codeword obtained by associating one such error event of each component code has been defined as the *effective free distance,* $d_{free-eff}$ of the overall code in (Benedetto and Montorsi, 1996c).

Note that this value is determined by the choice of the component codes, and is the same for all interleavers, although not all of them produce it. In this work $(OW_2)_{min}$ will denote the minimum code weight associated to an $IW = 2$ for *a given interleaver.* It differs from $d_{free-eff}$ because not all the interleavers produce $d_{free-eff}$, and thus $(OW_2)_{min} \geq d_{free-eff}$. A $d_{free-eff}$ codeword for a turbo code using the $RSC(5/7)$ component code is presented in figure 2.9(c). The main rule for component code design for turbo codes is to maximize $d_{free-eff}$. This does not always mean choosing codes with the highest d_{free}, as discussed in (Benedetto and Montorsi, 1996c; Divsalar and Pollara, 1995c).

The probability of $(OW_2)_{min}$ for a $RSC(5/7)$ turbo code with different interleaver lengths was simulated by generating a large number of randomly chosen interleavers (obtained as described in Annex A) and counting their $IW = 2$ mappings with lowest code weight. The results are presented in figure 2.10(a). It can be seen that a proportion of 0.86 of the total number of interleavers has $(OW_2)_{min} = d_{free-eff} = 10$, as discussed above. Also, about 0.15 of the total number of interleavers has $(OW_2)_{min} = 12$ and very few of them have $(OW_2)_{min} = 14$ and $(OW_2)_{min} = 16$. Figure 2.10(b) shows the computed multiplicity of $d_{free-eff}$ mappings. It can be observed that they are concentrated, as expected, around $a(2, 10) = 2$, which actually results as an average

$$\begin{aligned} a(2,10) &= 0.28 * 1 + 0.27 * 2 + 0.18 * 3 + 0.09 * 4 + 0.04 * 5 + 0.01 * 6 \\ &= 0.28 + 0.54 + 0.54 + 0.36 + 0.2 + 0.06 + ... \approx 2 \end{aligned}$$

The distribution is practically independent of the value of N. The low d_{free} causes an error floor in the FER and BER performance of turbo codes, which can be observed in the iterative decoder performance as shown in figure 2.10(c) and 2.10(d) for a turbo code using $RSC(5/7)$. Due to the interleaver gain in BER, this error floor is lowered with increasing N, whereas the error floor for FER remains the same. This effect has been described in (Perez et al., 1996) as *spectral thinning.*

The association of the turbo code error floor with its low d_{free} has shown for the first time that the iterative decoder performance is close to the optimal performance at least at high E_b/N_o values. Also, the unusual bend of the performance curves, different from the usual optimal curves, suggests that the iterative decoder misbehaves at low

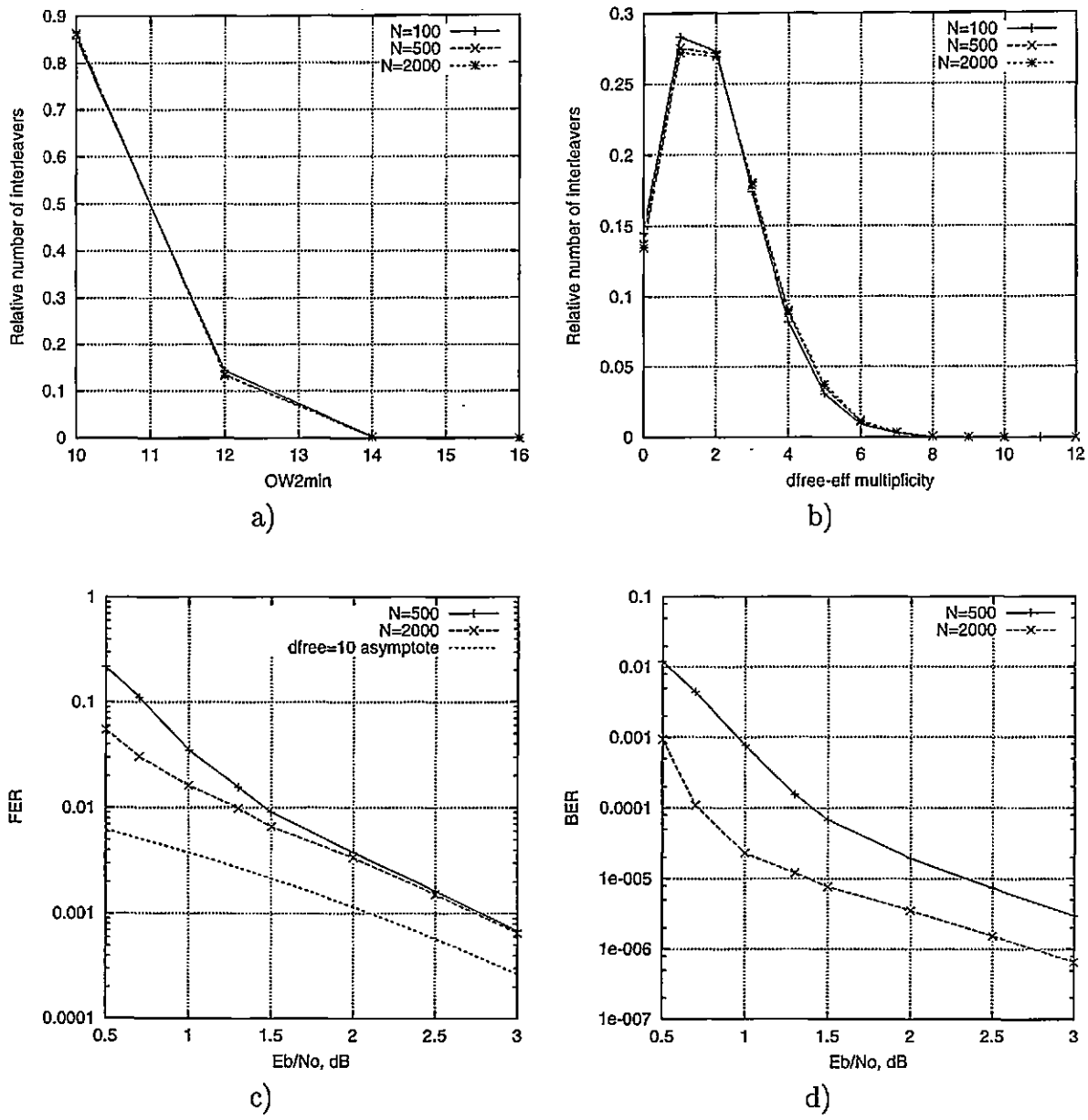


Figure 2.10: $IW = 2$ error events mapping probability
The values are determined for an $RSC(5/7)$ turbo code: a) $(OW_2)_{min}$ distribution, b) multiplicity of error events causing the $d_{free-eff}$, c) FER performance for different block lengths and d) BER performance for different block lengths

E_b/N_o, producing non-optimal decodings.

There is another explanation of the shape of the curves, which does not blame the iterative decoder, presented in (Perez et al., 1996). The weight spectra of turbo codes has a non-uniform distribution due to the interleaver effect: the interleaver tends to create a high concentration of codewords in a region of the spectra, which causes the sudden performance change when the E_b/N_o is low enough for their contribution to the error rate to be significant. This region shifts to higher code weights with increasing interleaver lengths, and thus the bend in the curve moves to the left in E_b/N_o (see figure 2.10(d)). This is also part of the spectral thinning. Although they are likely to have the same d_{free}, the FER performance of turbo codes improves with N at low E_b/N_o due to this effect, but the improvement is limited by the error floor.

2.2.3 The turbo decoder

The main advantage of turbo codes is the way they can be decoded. As described above, the parallel concatenation results in a powerful equivalent block code of length N. Instead of attempting to decode the received block as a single equivalent code, the two component codes are decoded separately. In this case, a method is necessary to obtain the output of the equivalent decoder from the output of the two component decoders. The decoding of turbo codes is performed by repeatedly decoding the received values for each component encoder, using a Soft Input Soft Output (SISO) algorithm. At each decoding the SISO algorithm produces a special kind of soft information called *extrinsic information* which is used by the other decoder to improve its own output extrinsic information. This transforms an exponential dependence of the decoding complexity on the block length N into a linear dependency, allowing for much longer block lengths. The decoding complexity still increases exponentially with component code memory, but this is not so important since good performance can be obtained with low memory component codes.

The Soft Input Soft Output algorithm

The SISO block for turbo codes is shown in figure (2.11). In order to perform the turbo decoding it is necessary to compute the probabilities of the information bits given the

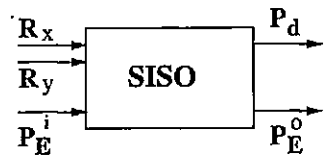


Figure 2.11: The SISO decoder
The input and output connections for the SISO decoder. The MAP algorithm is used as a SISO decoder.

received values and the code constraints.

$$P_d\{u_i = 0\} = P\{u_i = 0|\mathbf{R}_1^N\} \tag{2.31}$$

This can be done by using the MAP algorithm (Bahl et al., 1974), presented in Annex B. This algorithm, also called the *forward-backward algorithm*, relies on two recursive inspections of the code trellis, in order to determine the dependence of the current bit on previous bits (the α recursion) and on the future bits in the block (the β recursion). This is done by breaking relation (2.31) into three terms:

$$\begin{aligned} \alpha_i(m) &= P\{S_i = m, \mathbf{R}_1^{i-1}\} \\ \beta_i(m) &= P\{\mathbf{R}_{i+1}^N | S_i = m\} \\ \gamma_i(m, m_+) &= P\{S_i = m_+, R_i | S_{i-1} = m\} \end{aligned} \tag{2.32}$$

with these notations, equation (2.31) becomes

$$P_d\{u_i = 0\} = K_i \sum_{m,m_+|u(m,m_+)=0} \alpha_{i-1}(m)\gamma_i(m, m_+)\beta_i(m_+) \tag{2.33}$$

where K_i does not depend on u_i. The α (forward) recursion is given by the formula:

$$\alpha_{i+1}(m_+) = \alpha_i(m')\gamma_{i+1}(m', m_+) + \alpha_i(m'')\gamma_{i+1}(m'', m_+) \tag{2.34}$$

where m' and m'' are the two code states from which the encoder can reach state m_+. The β (backward) recursion is given by the formula:

$$\beta_i(m) = \beta_{i+1}(m'_+)\gamma_{i+1}(m, m'_+) + \beta_{i+1}(m''_+)\gamma_{i+1}(m, m''_+) \tag{2.35}$$

where m'_+ and m''_+ are the two code states that the encoder can reach from state m, and γ is the transition probability given by:

$$\gamma_i(m, m_+) = P\{R_{y_i}|y(m, m_+)\}P\{R_{x_i}|x(m, m_+)\}P\{S_i = m_+|S_{i-1} = m\} \tag{2.36}$$

where $x()$ and $y()$ are functions that associate a value of zero or one with each possible transition, and represent the encoded bits.

$$\begin{aligned} P_d\{u_i = 0\} &= K_i \sum_{m,m_+|u(m,m_+)=0} \alpha_{i-1}(m)\gamma_i(m, m_+)\beta_i(m_+) \\ &= K_i \sum_{m,m_+|u(m,m_+)=0} \alpha_{i-1}(m)P\{R_{y_i}|y(m, m_+)\} * \\ &\quad P\{R_{x_i}|x(m, m_+)\}P\{S_i = m_+|S_{i-1} = m\}\beta_i(m_+) \end{aligned} \tag{2.37}$$

The probability $P\{S_i = m_+|S_{i-1} = m\}$ can be seen as the probability of the information bit that caused the transition, $P\{u_i = 0\}$ and thus:

$$\begin{aligned} P_d\{u_i = 0\} &= K_i P\{R_{x_i}|0\}P\{u_i = 0\} * \\ &\quad \left[\sum_{m,m_+|u(m,m_+)=0} \alpha_{i-1}(m)P\{R_{y_i}|y(m, m_+)\}\beta_i(m_+) \right] \end{aligned} \tag{2.38}$$

where $u()$ associates each transition with the information bit that caused it. Since the encoder is systematic, $u() = x()$ and thus $P\{R_{x_i}|x(m, m_+)\}$ can be factored out of the summation. This is necessary since both codes use it as channel input. This would not be necessary if the codes were nonsystematic, since then there would be no shared channel values between the two codes.

The extrinsic information

From equation (2.38), by denoting

$$P^i_E\{u_i = 0\} = P\{u_i = 0\} \tag{2.39}$$

$$P^o_E\{u_i = 0\} = K_i \sum_{m,m_+|u(m,m_+)=0} \alpha_{i-1}(m)P\{R_{y_i}|y(m, m_+)\}\beta_i(m_+) \tag{2.40}$$

$RSC(5/7)$

$u(m, m_+)$ / $x(m, m_+)$		m			
		0	1	2	3
m_+	0	0	1		
	1			1	0
	2	1	0		
	3			0	1

$y(m, m_+)$		m			
		0	1	2	3
m_+	0	0	1		
	1			0	1
	2	1	0		
	3			1	0

Table 2.1: Code tables for the $RSC(5/7)$ code
The blank entries in the table represent impossible transitions. They do not contribute to the sums in the MAP equations.

one obtains

$$P_d\{u_i = 0\} = P\{R_{x_i}|0\}P_E^i\{u_i = 0\}P_E^o\{u_i = 0\} \tag{2.41}$$

Equation (2.40) defines the *extrinsic information* produced by the decoder. It depends on all channel inputs and a priori probabilities, excepting the systematic value and a priori probability for the current bit. Also, equation (2.39) symbolizes the fact that the a priori information could be the extrinsic output of another decoder. The two equations form the basis of including such an algorithm in an iterative loop: it could take information from a previous decoder and produce *new* (extrinsic) information to be used by the next decoder. Note that the term new must be interpreted bitwise. The extrinsic information of a bit still depends on the input extrinsic information from all the other bits. The fact that it does not depend on itself is essential for the ability to break the transition probability into products (equation (2.36)). The difference between the iterative decoding exchanging extrinsic (P_E)/complete information (P_d) is presented in figure (2.13) in terms of BER. It can be observed that the algorithm using complete information exchange also improves with iteration, but saturates at a level much higher than the one using extrinsic information exchange.

As an example, the formula above can be written for the $RSC(5/7)$ code based on table (2.1) as presented below:

The α recursion

Using the values in table (2.1), and equation (2.36), equation (2.34) becomes:

$$\begin{cases} \alpha_i(0) = & \alpha_{i-1}(0)P_E\{u_i=0\}P\{R_{x_i}|0\}P\{R_{y_i}|0\} \\ & + \alpha_{i-1}(1)P_E\{u_i=1\}P\{R_{x_i}|1\}P\{R_{y_i}|1\} \\ \alpha_i(1) = & \alpha_{i-1}(2)P_E\{u_i=1\}P\{R_{x_i}|1\}P\{R_{y_i}|0\} \\ & + \alpha_{i-1}(3)P_E\{u_i=0\}P\{R_{x_i}|0\}P\{R_{y_i}|1\} \\ \alpha_i(2) = & \alpha_{i-1}(0)P_E\{u_i=1\}P\{R_{x_i}|1\}P\{R_{y_i}|1\} \\ & + \alpha_{i-1}(1)P_E\{u_i=0\}P\{R_{x_i}|0\}P\{R_{y_i}|0\} \\ \alpha_i(3) = & \alpha_{i-1}(2)P_E\{u_i=0\}P\{R_{x_i}|0\}P\{R_{y_i}|1\} \\ & + \alpha_{i-1}(3)P_E\{u_i=1\}P\{R_{x_i}|1\}P\{R_{y_i}|0\} \end{cases} \tag{2.42}$$

The β recursion

Using the values in table (2.1) and equation (2.36), equation (2.35) becomes:

$$\begin{cases} \beta_i(0) = & \beta_{i+1}(0)P_E\{u_i=0\}P\{R_{x_i}|0\}P\{R_{y_i}|0\} \\ & + \beta_{i+1}(2)P_E\{u_i=1\}P\{R_{x_i}|1\}P\{R_{y_i}|1\} \\ \beta_i(1) = & \beta_{i+1}(0)P_E\{u_i=1\}P\{R_{x_i}|1\}P\{R_{y_i}|1\} \\ & + \beta_{i+1}(2)P_E\{u_i=0\}P\{R_{x_i}|0\}P\{R_{y_i}|0\} \\ \beta_i(2) = & \beta_{i+1}(1)P_E\{u_i=1\}P\{R_{x_i}|1\}P\{R_{y_i}|0\} \\ & + \beta_{i+1}(3)P_E\{u_i=0\}P\{R_{x_i}|0\}P\{R_{y_i}|1\} \\ \beta_i(3) = & \beta_{i+1}(1)P_E\{u_i=0\}P\{R_{x_i}|0\}P\{R_{y_i}|1\} \\ & + \beta_{i+1}(3)P_E\{u_i=1\}P\{R_{x_i}|1\}P\{R_{y_i}|0\} \end{cases} \tag{2.43}$$

The extrinsic information

Using the values in table (2.1), equation (2.40) becomes:

$$\begin{aligned} P_E^o\{u_i=0\} = & \quad \alpha_{i-1}(0)P\{R_{y_i}|0\}\beta_i(0) \\ & + \alpha_{i-1}(1)P\{R_{y_i}|0\}\beta_i(2) \\ & + \alpha_{i-1}(2)P\{R_{y_i}|1\}\beta_i(3) \\ & + \alpha_{i-1}(3)P\{R_{y_i}|1\}\beta_i(1) \end{aligned} \tag{2.44}$$

The iterative algorithm

The iterative decoding algorithm is schematically presented in figure (2.12). The received stream of channel samples are demultiplexed and grouped into blocks of length

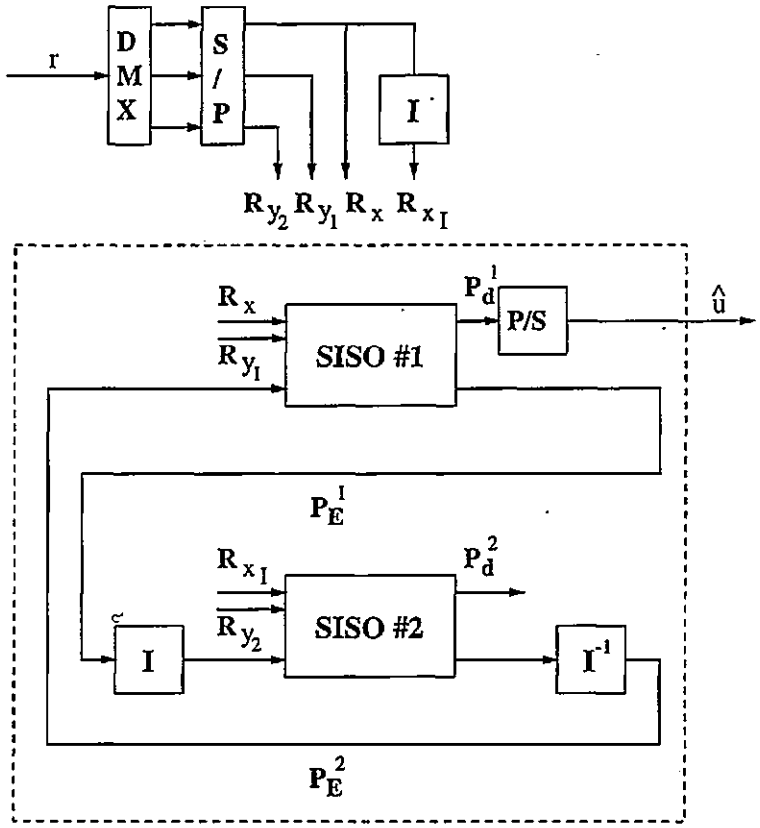


Figure 2.12: The turbo decoder

N. For each block, the iterative decoder executes several iterations before the decoded output is thresholded and passed further on in the receiving chain. The iterative algorithm consists of the following steps:

1. The channel values are transformed into probabilities
2. The received values for the systematic bit are passed directly to the first decoder and interleaved to the second decoder. Each decoder acts on the received values for the corresponding parity bit.
3. The a priori probabilities are initialized to 0.5 (uniform probabilities)
4. The first decoder produces its extrinsic information and decoded values based on channel values and a priori information
5. The extrinsic information (interleaved) is passed to the second decoder as a priori information
6. The second decoder produces its extrinsic information and decoded values based on channel values and a priori information
7. The extrinsic information (deinterleaved) is passed to the first decoder as a priori information
8. Loop from step (4) a given number of times (iterations)
9. The decoded values from the first decoder (or the interleaved decoded values from the second decoder) are passed further in the receiving chain (eventually thresholded)

In the algorithm presented above, steps (5) and (7) describe the extrinsic information exchange between the two decoders. These are critical points for the iterative algorithm. There are several possibilities:

- The probabilities $P_E^o\{u_i = 1\}$ and $P_E^o\{u_i = 0\}$ are fed directly as inputs to the next decoder,

$$P_{E_{next}}^i\{u_i = k\} = P_E^o\{u_i = k\} \quad , \quad k = 0, 1 \tag{2.45}$$

The term K_i in equation (2.40) is common for $k = 0, 1$ so that it does not affect the result on an infinite precision machine. Still, the cumulated product of these factors leaves open a normalisation problem.

- In order to solve the above problem, the output probabilities could be normalized, so that

$$P^i_{E_{next}}\{u_i = k\} = \frac{P^o_E\{u_i = k\}}{P^o_E\{u_i = 0\} + P^o_E\{u_i = 1\}} \quad k = 0, 1 \tag{2.46}$$

in which which the K_i term cancels out, and the obtained values verify

$$P^i_{E_{next}}\{u_i = 0\} + P^i_{E_{next}}\{u_i = 1\} = 1 \tag{2.47}$$

- Compute the *log likelihood ratio* as in (Robertson, 1994),

$$\lambda\{u_i\} = \log\left(\frac{P^o_E\{u_i = 0\}}{P^o_E\{u_i = 1\}}\right) \tag{2.48}$$

From this, the next decoder receives:

$$\begin{aligned} P^i_{E_{next}}\{u_i = 0\} &= \frac{e^{\lambda\{u_i\}}}{1 + e^{\lambda\{u_i\}}} \\ P^i_{E_{next}}\{u_i = 1\} &= \frac{1}{1 + e^{\lambda\{u_i\}}} \end{aligned} \tag{2.49}$$

This is mathematically equivalent to the previous alternative. The log likelihood ratio is mathematically attractive because it supplies a 'pseudochannel' value, under the assumption of a Gaussian distribution at the decoder output. This was useful in the first turbo decoder (Berrou et al., 1993b) since it used a SOVA decoder with input channel values and not probabilities (in this case, the input to the next decoder is given directly by (2.48)). It is also useful for simplified versions of the MAP algorithm working in the log domain (log-MAP).

2.2.4 The convergence issue

The usual way of describing the properties of a code is by assuming an optimal decoder at the receiving side. In this case, a study of several characteristics of the codewords

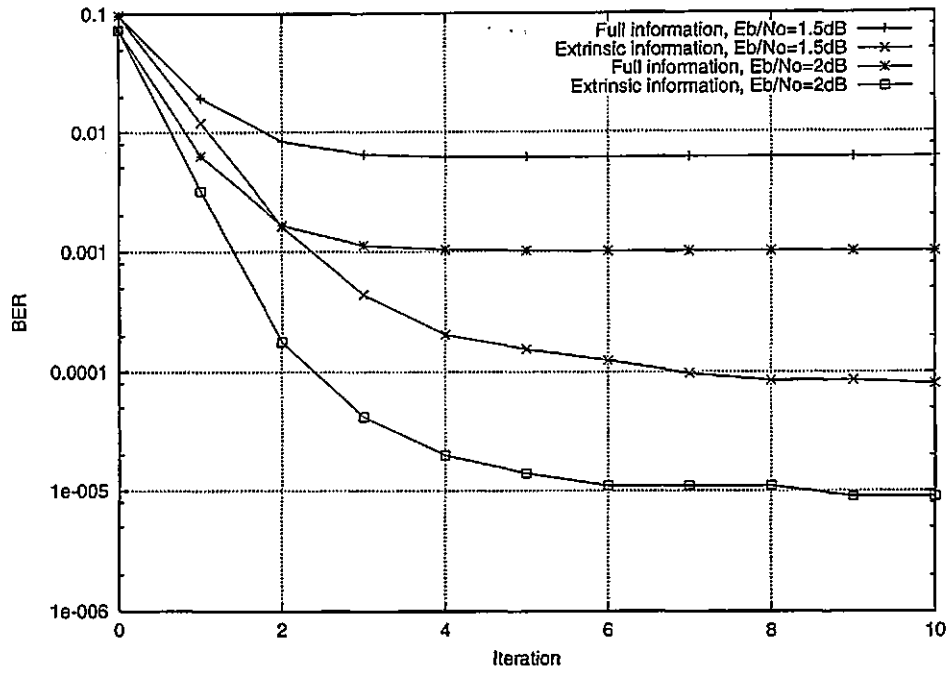


Figure 2.13: Extrinsic vs complete information exchange
BER vs iteration for the iterative process using extrinsic (P_E) or complete (P_d) MAP information exchange for an $N = 500$, $RSC(5/7)$ turbo code at different E_b/N_o values

(code weight, minimum distance) can give an idea about the expected performance, and indicate ways to improve it. The real decoder in the case of turbo codes is the iterative decoder, which is simpler but nonoptimal. This raises several questions:

- How close is the output of the iterative decoder to the output of an optimal decoder for turbo codes? What are the design constraints to make it closer?

- Since the turbo decoder is iterative, it is important to know if it converges or not. This is useful for determining when to stop iterating and choosing a decoded output to work with.

- What is the link between convergence and the closeness to the optimal decoding? Figure (2.13) presents the difference between the iterative algorithm using complete (P_d) information exchange, and the iterative algorithm using extrinsic information. It can be observed that the first case *is* convergent. It converges quicker than the second case, but it converges more times to the wrong sequence. It is not only a case of convergence, but also a case of what the algorithm con-

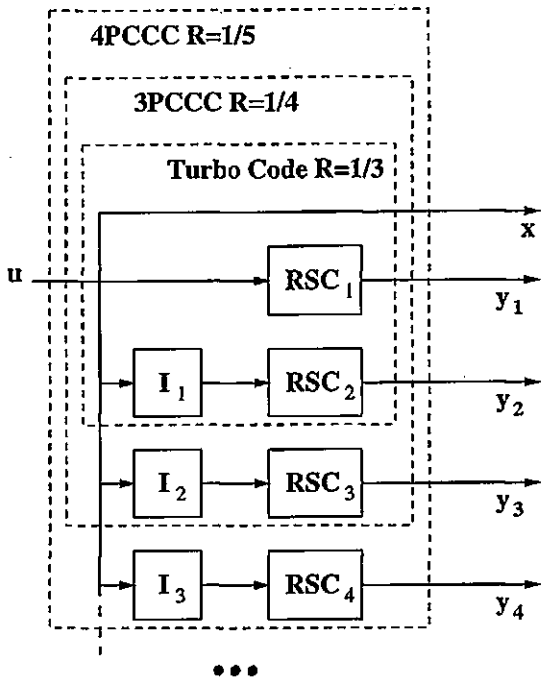


Figure 2.14: MPCCC

verges to.

2.3 The multiple parallel concatenation

2.3.1 The encoder

The MPCCC are a straightforward extension of turbo codes, by adding one or more interleaver/code pair in the concatenated structure, as shown in figure (2.14). By adding one interleaver/code pair a 3PCCC scheme is obtained, by adding two interleaver/code pairs a 4PCCC scheme, and so on. Note the difference in indexing the interleaver and *RSC* blocks. Some publications prefer to add a (constructively unnecessary) interleaver for the first code, for symmetry reasons (Divsalar et al., 1995; Divsalar and Pollara, 1995a).

One of the main problems that limits the number of codes that could be added is the decrease in code rate. An unpunctured 3PCCC scheme has a code rate $R = 1/4$ and for 4PCCC, $R = 1/5$. In these cases, the use of systematic convolutional codes is more critical than in the case of turbo codes, since otherwise the code rates would be $R = 1/6$ for 3PCCC and $R = 1/8$ for 4PCCC.

2.3.2 Optimal decoding performance

The reason for increasing the number of codes/interleavers in the concatenated structure was given in (Divsalar and Pollara, 1995a) in terms of the probabilities that the interleavers will associate given error events of the component codes, depending on their information weight. If m is the number of codes in the structure, the mapping probability (interleaver effect) is:

- $IW = 1$: The interleavers always do N such mappings.
- $IW = 2$: The interleavers do an average number of mappings $a(2, ow) \sim 1/N^{m-2}$ with a probability $P(2, ow) \sim 1/N^{m-2}$.
- $IW \geq 3$: The interleavers do an average number of mappings $a(iw, ow) \sim 1/N^{m-2+iw}$ with a probability $P(iw, ow) \sim 1/N^{m-2+iw}$.

The likelihood of associating given error events of the component codes reduces with the number of interleavers. It was observed in (Divsalar and Pollara, 1995a) that *"Increasing either the weight of the data sequence or the number of codes has roughly the same effect on lowering this probability"*.

It can be seen that, similar to the turbo code case, the $IW = 1$ error events should be associated with high code weights by using recursive codes. The $IW = 2$ still dominate the performance, but this time, as long as $m > 2$, the FER can be reduced to zero with increasing block lengths as fast as

$$\text{FER} \sim \frac{1}{N^{m-2}} \tag{2.50}$$

and BER even quicker,

$$\text{BER} \sim \frac{1}{N^{m-3}} \tag{2.51}$$

due to the $1/N$ factor in the union bound formula for BER (equation (2.25)). Note that the $m = 2$, 2PCCC case describes turbo codes.

The $d_{free-eff}$ for MPCCC schemes can be defined in a similar way as for turbo codes, and the design criteria for component codes are identical. As an example, the $d_{free-eff}$ for a 3PCCC using the $RSC(5/7)$ component code corresponds to the association of

the $(IW = 2, OW = 6)$ for each code, resulting in $d_{free} = 2+4+4+4 = 14$, as opposed to $d_{free} = 2 + 4 + 4 = 10$ for the turbo code using $RSC(5/7)$ as component code.

These conclusions are based on the optimal decoding assumption, and do not consider the performance of the iterative decoder for the MPCCC schemes.

2.3.3 The decoder

Turbo decoding of the MPCCC schemes with more than two codes presents the problem of how the extrinsic information should be exchanged between decoders. Several possible situations for 3PCCC are shown in figure (2.15). The first case is a direct extension of the turbo decoder: each code is decoded separately and the extrinsic information is fed into the next code. The extrinsic information of the 3^{rd} code is fed back to the first code and the process is repeated. In the second case, the first code supplies extrinsic information to the two other codes. These codes are decoded in parallel and their extrinsic information is fed back to the first code, and the process is repeated. The third case, all codes are decoded separately, but use extrinsic information from both previous codes. Finally, in the last case all codes are decoded in parallel and supply extrinsic information to all the other codes.

Simulation results for a 3PCCC scheme with $N = 500$ at $E_b/N_o = 1$dB are presented in figure (2.16). The x axis represents each decoding for each iterations, in the order: code1, code2, code3, code1,...

It can be observed that the third case gives the best performance. This can be explained by the fact that it uses all the available extrinsic information at any moment in time. The fourth case has the closest performance to the third case. The 'step' shape of the curve shows that for each iteration, the BER for any of the component code is similar, due to parallel decoding. The worst case is the first case, where the improvement due to iteration is almost nonexistent after the first iteration. The difference in performance for the same encoding scheme (and thus same optimal decoding performance), shows the importance of carefully designing the extrinsic information exchange schedule, especially when the number of codes in the structure is increased. Due to the fact that the third case has the best performance, it has been chosen as the preferred decoding scheme for 3PCCC. It also can be easily extended to general MPCCC schemes.

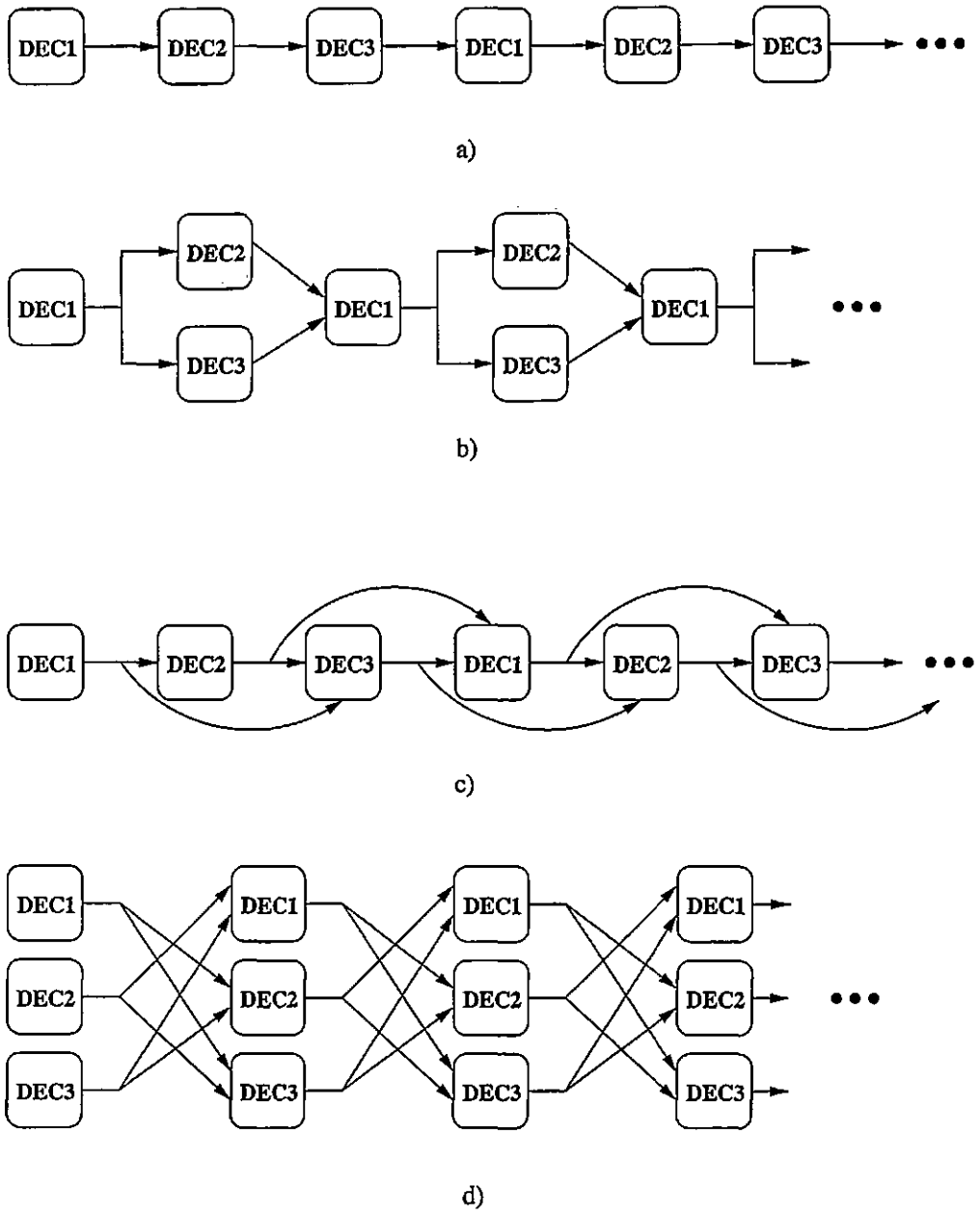


Figure 2.15: 3PCCC decoding schemes
Different possibilities to exchange extrinsic information between the decoders: a) serial, b) serial-parallel, c) full serial and d) parallel extrinsic information exchange

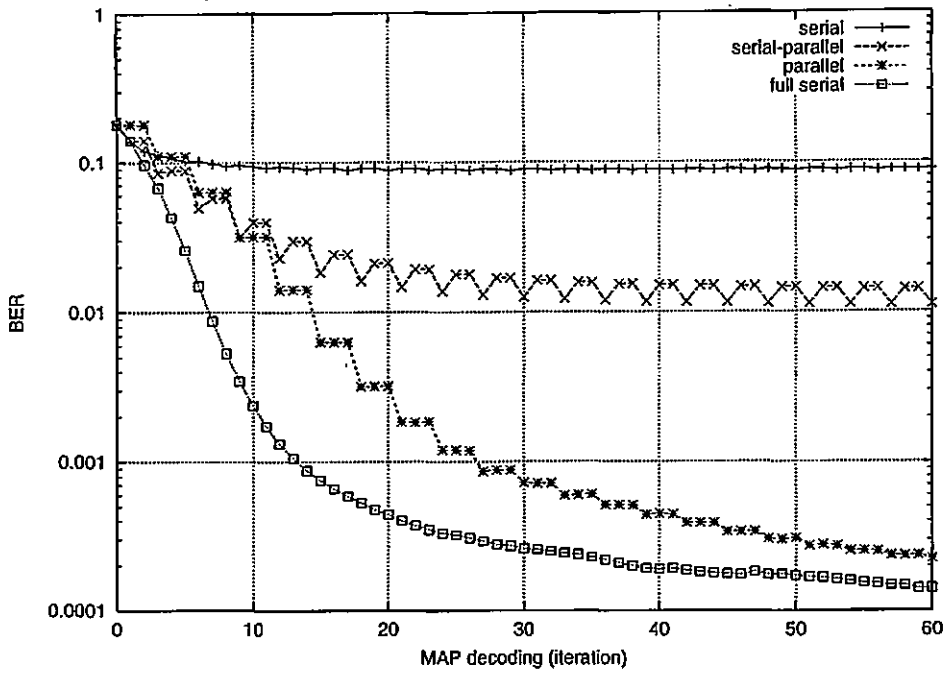


Figure 2.16: 3PCCC decoding schemes performance comparison
BER improvement with iteration for an $N = 500$, $RSC(5/7)$ 3PCCC scheme using different extrinsic information exchange schemes

The decoder for the 3PCCC scheme is presented in figure (2.17). The iterative algorithm is detailed below:

1. The channel values are transformed into probabilities
2. The received values for the systematic bit are passed directly to the first decoder and interleaved to the second and third decoder. Each decoder acts on the received values for the corresponding parity bit.
3. The a priori probabilities are initialized to **0.5** (uniform probabilities)
4. The first decoder produces its extrinsic information and decoded values based on channel values and a priori information. Its input extrinsic vector is reset to **0.5**
5. The extrinsic information (suitably interleaved) is combined with the input extrinsic vector of the second and third decoder
6. The second decoder produces its extrinsic information and decoded values based on channel values and its input (a priori) extrinsic information vector. Its a priori vector is reset to **0.5**

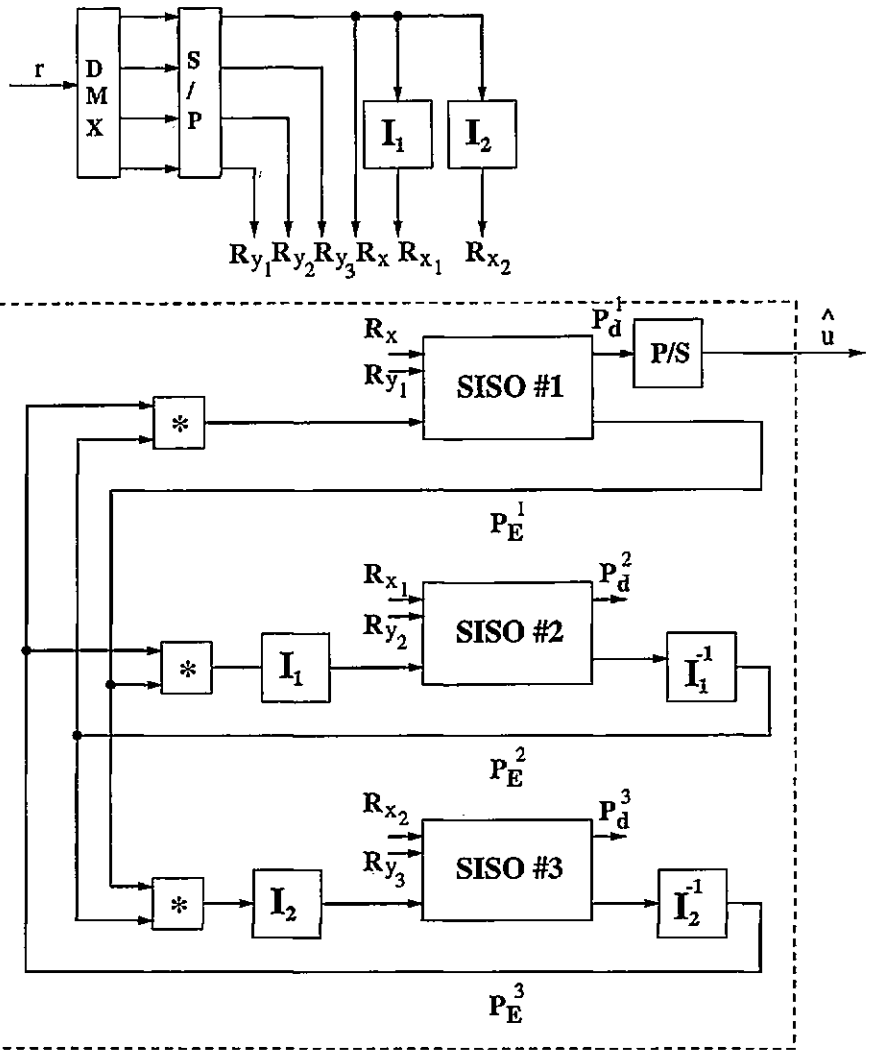


Figure 2.17: 3PCCC decoder

3PCCC decoder using full serial extrinsic information exchange. The '*' block represents the multiplication (and normalisation) of the extrinsic information from the other decoders, as presented in equation (2.52) or (2.53).

7. The extrinsic information (suitably interleaved) is combined with the a priori input vector for the first and third decoder

8. The third decoder produces its extrinsic information based on the combined extrinsic information from the first and second decoder. Its input extrinsic information vector is set to **0.5**

9. The extrinsic information from the third decoder (suitably interleaved) is combined with the a priori vector of the first and second decoder

10. Loop from step (4) a given number of times (iterations)

11. The decoded values from the first decoder (or the interleaved decoded values from the second decoder) are passed further in the receiving chain (eventually thresholded)

Note that the first two decodings use incomplete extrinsic information: the first decoding has no a priori information and the second decoding has a priori information only from the first decoder. Subsequently, each decoder takes two input extrinsic informations from the previous codes, generally denoted as P^o_{E1} and P^o_{E2}. The two probabilities could be combined by simply multiplying them:

$$P^i_E = P^o_{E1} P^o_{E2} \tag{2.52}$$

or by using a normalised product:

$$P^i_E = \frac{P^o_{E1} P^o_{E2}}{P^o_{E1} P^o_{E2} + (1 - P^o_{E1})(1 - P^o_{E2})} \tag{2.53}$$

For an increased number of codes, the equivalent input extrinsic information can be obtained by multiplying all probabilities together or successively applying formula (2.53). For example, for a 4PCCC scheme, P^i_E is obtained from the set P^o_{E1}, P^o_{E2}, P^o_{E3}. By applying (2.53) for P^o_{E1}, P^o_{E2} and intermediary value P^i_{E12} is obtained, and the final value P^i_E results by combining P^i_{E12} and P^o_{E3} using (2.53). The second formula has the advantage of normalised probabilities, but it might have more numerical problems, since it uses division. If log likelihood ratios were used, the product would become a sum.

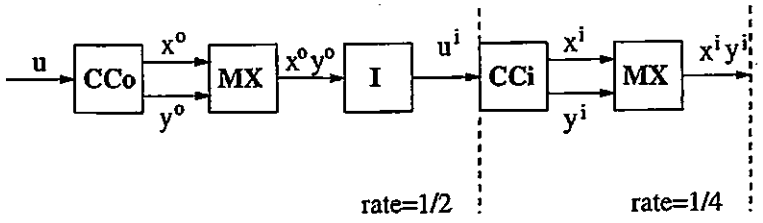


Figure 2.18: SCCC encoder

2.4 The serial concatenation

2.4.1 The encoder

An alternative to MPCCC schemes are the serial concatenated convolutional codes, SCCC. The concatenation is shown in figure (2.18). In this case, the output of the first (outer) encoder is multiplexed, interleaved and used as input for the second (inner) encoder. The code rate depends on the rate of the component codes. In the case of rate $R = 1/2$ component codes, the code rate for the unpunctured system is $R = 1/4$, the same rate as an unpunctured 3PCCC scheme. In this case, there is a difference between the interleaver length and the block length, the interleaver length being twice as long as the (information) block length, $N_I = 2N$.

2.4.2 Optimal decoding performance

From the previous sections it can be observed that the associations of single, given error events that interleavers produce most often have low information weight. The higher the information weight, the lower the mapping probability. It is the number of bits of one that the interleaver sees in its input/output block that determine this probability. But what if this number is never less than 5 and can be easily increased by choosing the right code? This is the case of the serial concatenated codes, where the *code* bits (parity+systematic) rather than just the information bits of the outer code end up in the input block of the interleaver. This means that their number cannot be lower than the d_{free} of the outer code. For a simple $RSC(5/7)$ component code, $d_{free} = 5$, and it can be easily increased by choosing codes with higher memory. Reasoning as before,

the interleaver gain for the SCCC scheme would be

$$\text{BER} \sim \frac{1}{N_I^{d^o_{free}-1}} \tag{2.54}$$

where d^o_{free} is the free distance of the outer code. Unfortunately, the output block of the interleaver is still the *input* of the inner code, and nothing forces this code to consider the d^o_{free} bits of one as a single error event. Instead, the inner code splits the single outer code error event into several error events which can be positioned independently in the block whilst producing the same overall inner code weight. This increases their mapping probability and hence the interleaver gain is only (Benedetto and Montorsi, 1996b):

$$\text{BER} \sim \frac{1}{N_I^{\left\lfloor \frac{d^o_{free}+1}{2} \right\rfloor}} \tag{2.55}$$

If d^o_{free} is odd, the inner code can separate it into several $IW = 2$ error events and one $IW = k$ error event, where $k \in \{1, 3\}$. The $k = 1$ case is excluded if the inner code is recursive, since it produces infinite code weight. Another case is separating $d^o_{free}+1$ into several $IW = 2$ error events. This case has a higher mapping probability, dominating the interleaver gain, and this is why $d^o_{free}+1$ appears in equation (2.55).

Since the input of the outer code is not involved in the interleaver gain, this code can be non-recursive. The inner code still needs to be recursive. It is no longer useful to use systematic codes, since the code rate cannot be reduced by using systematic codes. In (Benedetto and Montorsi, 1996b) it is argued that the outer code should be non-recursive (tends to associate lower information weights to low code weights), non-systematic (can have higher d^o_{free}). The inner code is designed in a similar way as for turbo codes.

2.4.3 The decoder

The decoder is based on the observation that the MAP algorithm as presented in (Bahl et al., 1974) does not necessary refer to the information bits as decoded bits, but can also be particularised to parity bits or any signal that can be associated to a Markov model transition. The decoder for SCCC schemes is presented in figure (2.20). As it

Figure 2.19: SISO decoder for the outer code
The input and output connections for the SISO decoder. The MAP algorithm is used as a SISO decoder.

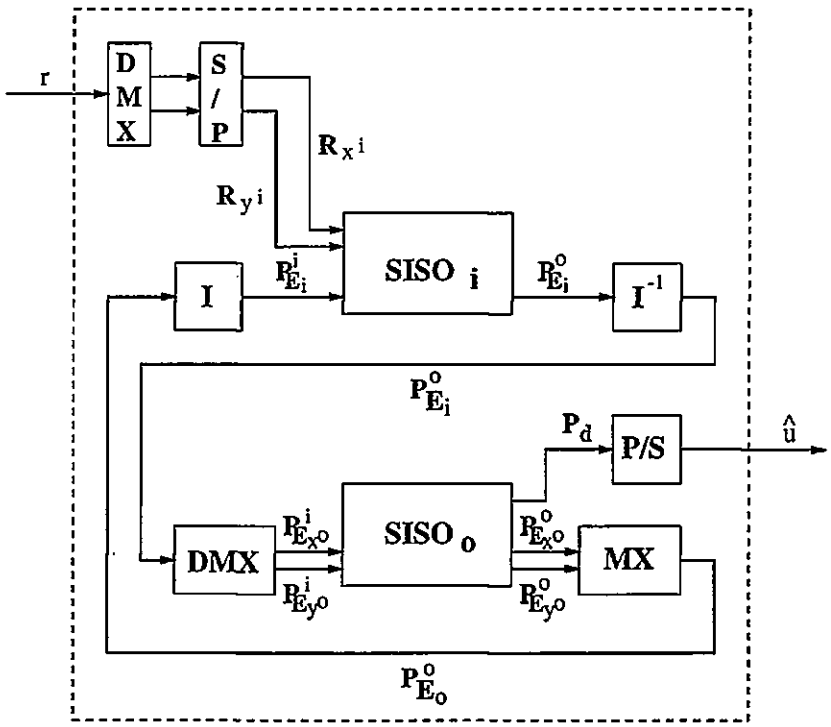


Figure 2.20: SCCC decoder

can be seen, a generalized MAP decoder is used to generate extrinsic information for the parity bits of the outer encoder, whereas the classical MAP decoder is still used for the inner code.

The outer code

The SISO block for the outer code is shown in figure (2.19). The equations for the parity bits can be derived from the general MAP equations presented in Annex B, and are shown below. The algorithm uses the same α and β recursions as presented for turbo codes, the difference being in the way the computed values are used to produce the extrinsic information for the parity bits. Also, the transition probability consists now only of the a priori information for the systematic/parity bits:

$$\gamma_i(m, m_+) = P_E^i\{y_i = y(m, m_+)\}P_E^i\{x_i = x(m, m_+)\} \tag{2.56}$$

The probability that the parity bit is zero if $\mathbf{R}_1^N$ was received is:

$$P_d\{y_i = 0\} = P\{y_i = 0|\mathbf{R}_1^N\} = K_i \sum_{m,m_+|y(m,m_+)=0} \alpha_{i-1}(m)\gamma_i(m, m_+)\beta_i(m_+) \tag{2.57}$$

Replacing the transition probability γ from equation (2.56) gives:

$$\begin{aligned} P_d\{y_i = 0\} &= K_i \sum_{m,m_+|y(m,m_+)=0} \alpha_{i-1}(m)P_E^i\{y_i = y(m, m_+)\} * \\ &\quad P_E^i\{x_i = x(m, m_+)\}\beta_i(m_+) \\ &= K_i P_E^i\{y_i = 0\} * \\ &\quad \left[\sum_{m,m_+|y(m,m_+)=0} \alpha_{i-1}(m)P_E^i\{x_i = x(m, m_+)\}\beta_i(m_+)\right] \end{aligned} \tag{2.58}$$

The product can be split into extrinsic information and intrinsic information (information dependent on the current bit):

$$P_d\{y_i = 0\} = K_i P_E^i\{y_i = 0\}P_E^o\{y_i = 0\} \tag{2.59}$$

where the output extrinsic information is:

$$P_E^o\{y_i = 0\} = \sum_{m,m_+|y(m,m_+)=0} \alpha_{i-1}(m) P_E^i\{x_i = x(m,m_+)\}\beta_i(m_+) \tag{2.60}$$

A similar expression can be derived for the systematic/other parity bit:

$$P_E^o\{x_i = 0\} = \sum_{m,m_+|x(m,m_+)=0} \alpha_{i-1}(m) P_E^i\{y_i = y(m,m_+)\}\beta_i(m_+) \tag{2.61}$$

The decoded bit is obtained as:

$$P_d\{u_i = 0\} = K_i \sum_{m,m_+|u(m,m_+)=0} \alpha_{i-1}(m) P_E^i\{x_i = x(m,m_+)\} P_E^i\{y_i = y(m,m_+)\}\beta_i(m_+) \tag{2.62}$$

The output P_d of the SISO block in figure (2.19) corresponds to equation (2.62).

The inner code

The inner code can be systematic or nonsystematic. In the systematic case, the turbo code formulae apply, but it is not straight forward whether the systematic probability should be excluded from the extrinsic information or not. Thus in this case the output extrinsic formula are either:

$$P_E^o\{u_i = 0\} = K_i \sum_{m,m_+|u(m,m_+)=0} \alpha_{i-1}(m) P\{R_{y_i}|y(m,m_+)\}\beta_i(m_+) \tag{2.63}$$

as for turbo codes, or

$$P_E^o\{u_i = 0\} = K_i P\{R_{x_i}|0\} \sum_{m,m_+|u(m,m_+)=0} \alpha_{i-1}(m) P\{R_{y_i}|y(m,m_+)\}\beta_i(m_+) \tag{2.64}$$

In the nonsystematic case, the formula becomes:

$$P_E^o\{u_i = 0\} = K_i \sum_{m,m_+|u(m,m_+)=0} \alpha_{i-1}(m) P\{R_{x_i}|x(m,m_+)\} P\{R_{y_i}|y(m,m_+)\}\beta_i(m_+) \tag{2.65}$$

whilst the α and β recursions are similar to turbo codes. Note that in equation (2.65) $x(m, m_+) = 0$ is not always valid for $m, m_+ | u(m, m_+) = 0$, since x now denotes a parity bit and $x() \neq u()$.

The systematic case opens some interesting research directions. The first regards the question whether using the channel information in the extrinsic values would help the iterative decoding or not. Simulations show that using the channel values improves the start of the iterations, but not the final result. The second question is that of which code is decoded first. It has been argued that the inner code should have certain properties since it is the first to be decoded. If the outer code receives the channel information, it could be decoded first. Simulations show that it does not make much difference which of the codes is decoded first, even for codes with different characteristics.

The iterative algorithm for SCCC schemes is detailed below:

1. The received values are transformed into probabilities, using the channel estimation E_b/N_o and supplied to the inner decoder. The input extrinsic values for the inner decoder are set to **0.5**.

2. The inner code is decoded, producing the extrinsic information corresponding to its information bits.

3. The extrinsic information from the inner decoder is deinterleaved and demultiplexed, and supplied as a priori information for the parity bits of the outer code.

4. The outer code is decoded, producing the extrinsic information for the parity bits and the decoded information for its information bits.

5. The extrinsic information from the outer code is multiplexed and interleaved and supplied to the outer decoder as a priori information for its information bits.

6. The process is repeated from step 2 a fixed number of iterations.

7. The decoded information from the outer code is passed further in the receiving chain, where it will eventually be thresholded.

Applied to an SCCC scheme with $NC(5, 7)$ as outer code and $RSC(5/7)$ as inner code, the above formulas become:

$NC(5,7)$

$u(m, m_+)$		m			
		0	1	2	3
m_+	0	0	0		
	1			0	0
	2	1	1		
	3			1	1

$x(m, m_+)$		m			
		0	1	2	3
m_+	0	0	1		
	1			0	1
	2	1	0		
	3			1	0

$y(m, m_+)$		m			
		0	1	2	3
m_+	0	0	1		
	1			1	0
	2	1	0		
	3			0	1

Table 2.2: Code tables for the $NC(5,7)$ convolutional code
The blank entries in the table represent impossible transitions. They do not contribute to the sums in the MAP equations.

The outer code $NC(5,7)$

The code constraints are presented in table (2.2).

The α recursion

By using the values in table (2.2) and equations (2.34) and (2.56), the α recursion becomes:

$$\begin{cases} \alpha_i(0) &= \alpha_{i-1}(0)P_E\{x_i=0\}P_E\{y_i=0\} + \alpha_{i-1}(1)P_E\{x_i=1\}P_E\{y_i=1\} \\ \alpha_i(1) &= \alpha_{i-1}(2)P_E\{x_i=0\}P_E\{y_i=1\} + \alpha_{i-1}(3)P_E\{x_i=1\}P_E\{y_i=0\} \\ \alpha_i(2) &= \alpha_{i-1}(0)P_E\{x_i=1\}P_E\{y_i=1\} + \alpha_{i-1}(1)P_E\{x_i=0\}P_E\{y_i=0\} \\ \alpha_i(3) &= \alpha_{i-1}(2)P_E\{x_i=1\}P_E\{y_i=0\} + \alpha_{i-1}(3)P_E\{x_i=0\}P_E\{y_i=1\} \end{cases} \quad (2.66)$$

The β recursion

By using the values in table (2.2) and equations (2.35) and (2.56), the β recursion becomes:

$$\begin{cases} \beta_i(0) &= \beta_{i+1}(0)P_E\{x_i=0\}P_E\{y_i=0\} + \beta_{i+1}(2)P_E\{x_i=1\}P_E\{y_i=1\} \\ \beta_i(1) &= \beta_{i+1}(0)P_E\{x_i=1\}P_E\{y_i=1\} + \beta_{i+1}(2)P_E\{x_i=0\}P_E\{y_i=0\} \\ \beta_i(2) &= \beta_{i+1}(1)P_E\{x_i=0\}P_E\{y_i=1\} + \beta_{i+1}(3)P_E\{x_i=1\}P_E\{y_i=0\} \\ \beta_i(3) &= \beta_{i+1}(1)P_E\{x_i=1\}P_E\{y_i=0\} + \beta_{i+1}(3)P_E\{x_i=0\}P_E\{y_i=1\} \end{cases} \quad (2.67)$$

The extrinsic informations

By using the values in table (2.2) and equations (2.60) and (2.61) the extrinsic informations are:

$$\begin{cases} P_E^o\{x_i=0\} = & \alpha_{i-1}(0)P_E\{y_i=0\}\beta_i(0) + \alpha_{i-1}(1)P_E\{y_i=0\}\beta_i(2) \\ \quad + & \alpha_{i-1}(2)P_E\{y_i=1\}\beta_i(1) + \alpha_{i-1}(3)P_E\{y_i=1\}\beta_i(3) \\ P_E^o\{y_i=0\} = & \alpha_{i-1}(0)P_E\{x_i=0\}\beta_i(0) + \alpha_{i-1}(1)P_E\{x_i=0\}\beta_i(2) \\ \quad + & \alpha_{i-1}(2)P_E\{x_i=1\}\beta_i(3) + \alpha_{i-1}(3)P_E\{x_i=1\}\beta_i(1) \end{cases} \tag{2.68}$$

The decoded value

From table (2.2) and equation (2.62) the probability of the decoded bit is:

$$\begin{aligned} P_d\{u_i=0\} = & \quad \alpha_{i-1}(0)P_E\{x_i=0\}P_E\{y_i=0\}\beta_i(0) \\ & + \alpha_{i-1}(1)P_E\{x_i=1\}P_E\{y_i=1\}\beta_i(0) \\ & + \alpha_{i-1}(2)P_E\{x_i=0\}P_E\{y_i=1\}\beta_i(1) \\ & + \alpha_{i-1}(3)P_E\{x_i=1\}P_E\{y_i=0\}\beta_i(1) \end{aligned} \tag{2.69}$$

The inner code RSC(5/7)

The decoding formulae for this code are identical to those for turbo codes.

2.5 Summary

- The block components (*RSC* and interleaver) of a concatenated scheme have been described and their equations derived.
- The SISO block for the iterative decoder has been described. The block MAP decoder is used for the SISO algorithm. The original MAP algorithm equations in (Bahl et al., 1974), also presented in Annex (B), were used to derive the formula for turbo codes, MPCCC and SCCC. Example equations for particular codes were also presented.
- The encoder/decoder for turbo codes, MPCCC and SCCC have been presented. The exchange of extrinsic information in the iterative decoder is described for each scheme. Several extrinsic information exchange schemes are shown for the 3PCCC and the choice of a particular scheme is justified by simulation.

- The structure of each concatenated scheme is justified from an optimal decoder approach. The equations used are derived from the equations in (Divsalar and Pollara, 1995d) and (Perez et al., 1996). The interleaver gain is presented for each scheme, and exemplified for turbo codes by presenting the results of a computer search for $IW = 2$ error events. This has the novelty of illustrating the shape of the probability distributions for error events, rather than just the average. The computer search results are integrated with the average performance theory.

Chapter 3

Simulated concatenated schemes

3.1 Introduction

The probabilistic approach for determining the performance of turbo codes and their derivations is based on the likelihood that interleaver(s) chosen at random will associate codewords of the component codes having a given information/code weight. This has two factors:

- **The interleaver factor** This is given by the likelihood that interleaver(s) chosen at random would associate (map) given error events of the first code into error events of the other code(s) and the number of these mappings. This likelihood decreases with the information weight of the error events and is (almost) independent of code weight.

- **The code factor** This factor consists of the code weight and number of distinct error events associated to each information weight.

The performance of turbo codes is dictated by the combination of the two factors, and each factor influences the design of the other: because of the first factor, short, low information weight error events should be associated with a code weight as high as possible. Because higher code weights mean longer error events (second factor), the interleaver should be designed to reduce the probability of mapping short error events of the first code to short error events of the second code.

In this chapter the performance of the PCCC/SCCC schemes is determined by simulation and analysed in the light of theoretical statistics. The likelihood of obtaining

good performance when randomly choosing the interleaver and the improvement that can be obtained by designing the interleaver/codes for each scheme is investigated. The analysis is based on the observation of the *iterative decoding error events*, presented in the following. The effect of increasing block length is determined for each scheme, with the emphasis on two block length values, a "short" block ($N = 500$) and a "long" block ($N = 2000$).

Some of the design methods in the random interleaver approach do not improve the worst case but the chance that, by choosing an interleaver at random, a better performance will be obtained.

3.2 Iterative decoding error events

In order to compare the output of the iterative decoder with the ML performance it is important to define the error events for the iterative decoder. Due to the linearity of the code, the all zero sequence can be considered to be transmitted during simulations. The output of the decoder is thresholded, thus obtaining the error sequence, which has an information weight defined as the number of ones in the sequence. This can be associated with a code weight by re-encoding the sequence. It is expected for maximum likelihood errors to have a low information weight and also a low code weight associated to it, although it is difficult to specify the upper limit without knowing the weight spectra of the equivalent code.

For each observed error event, its structure can be analysed and insight into the cause of the error event can be obtained. The observed error events have been loosely classified into three categories:

- (LIWLOW) Low information weight low code weight error events. An error event of this type is presented in figure (3.1). It has information weight $IW = 2$ and code weight $OW = 10$.
- (LIWHOW) Low information weight high code weight error events. An error event of this type is presented in figure (3.2). It has information weight $IW = 3$ and code weight $OW = 107$.
- (HIWHOW) High information weight high code weight error events. An error

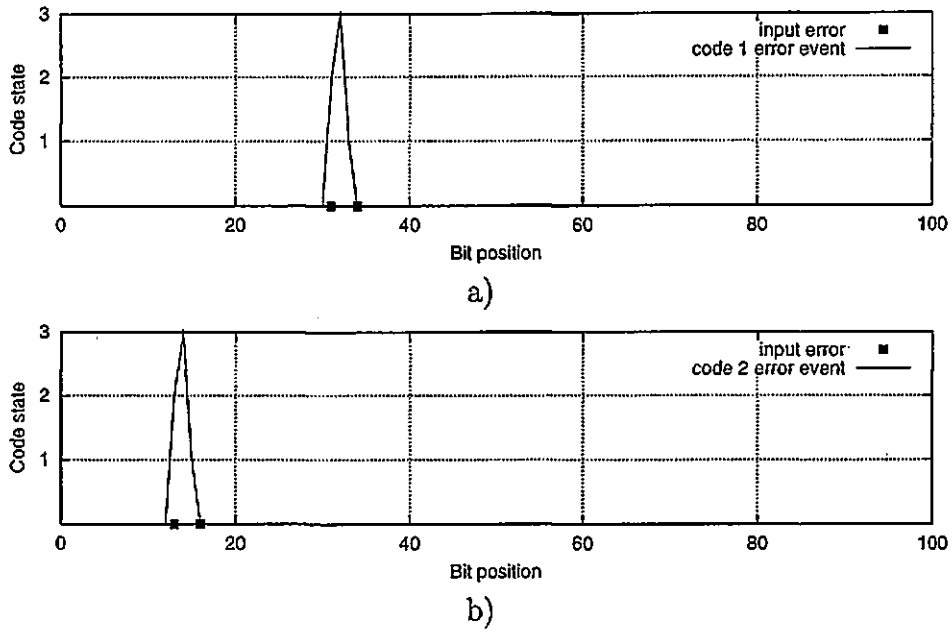


Figure 3.1: LIWLOW error event
Low information weight low code weight error event for an $N = 100$, RSC(5/7) turbo code, with IW=2, OW=10 (d_{free}). a) error events of code 1 and b) error events of code 2.

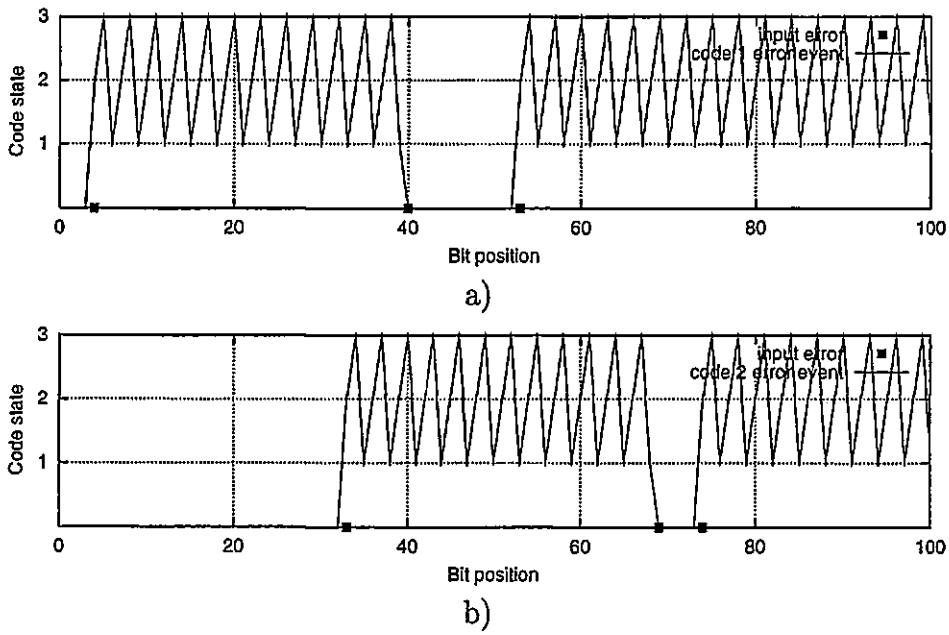


Figure 3.2: LIWHOW error event
Low information weight high code weight error event for an $N = 100$, RSC(5/7) turbo code, with IW=3, OW=107. a) error events of code 1 and b) error events of code 2.

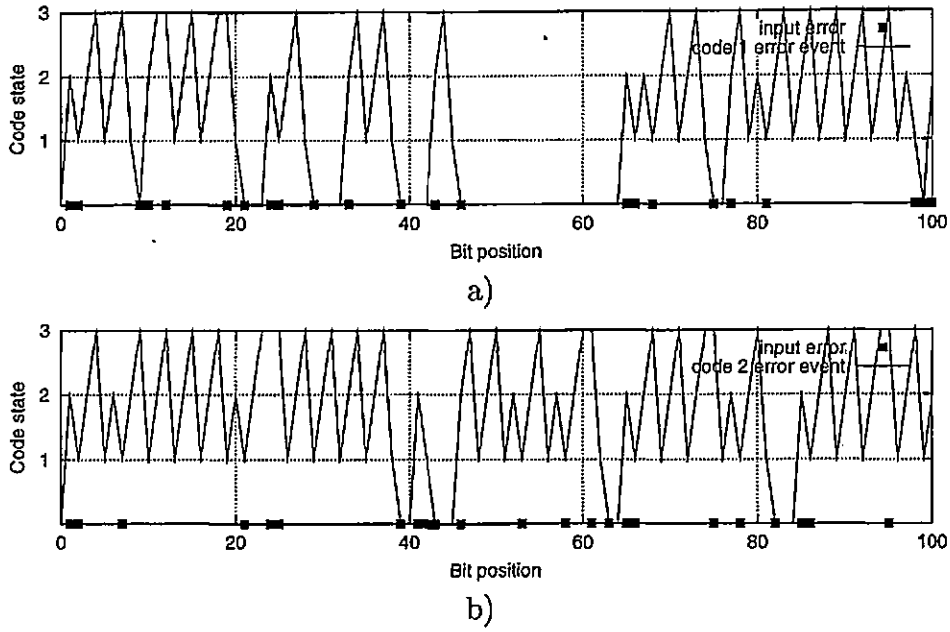


Figure 3.3: HIWHOW error event
High information weight high code weight error event for an $N = 100$, RSC(5/7) turbo code, with IW=23, OW=125. a) error events of code 1 and b) error events of code 2.

event of this type is presented in figure (3.3). It has information weight $IW = 23$ and code weight $OW = 125$.

The distinction between the three types of error events is not definite. Generally, an error event has high information weight if $IW > 20$, and high code weight if $OW > 100$. In simulations, it is also important to determine if the error events terminate the trellis of the component codes. This is why, in the following simulations, the termination status for the trellis of each component code has been determined for each error event. Also, it is interesting to record the length of the error events, this information can be used to determine the effect of interleaver design. The error events can be used to upper limit the value of d_{free} for the code: once an error event having a given code weight W has been observed, $d_{free} \leq W$.

3.3 Turbo codes

3.3.1 Interleaver factor

The interleaver factor is only dependent on the information weight, and thus an ($IW = 2, OW = 6$) error event will be mapped by the interleaver into itself just as often as it will be mapped to an ($IW = 2, OW = 20$) error event (for a $RSC(5/7)$ turbo code). The difference between the two cases for the interleaver is just the length of the error events: the ($IW = 2, OW = 20$) is slightly longer than the ($IW = 2, OW = 6$), but usually they are both much smaller than the block length, and thus their length will have a weak influence on the mapping probability. But from the code spectra point of view, the second mapping is more desirable than the first. The association between the interleaver point of view and code spectra point of view is: *the longer the error event, the higher its code weight.* It is relatively easy to design interleavers that increase the likelihood of mapping short error events of one of the codes into long error events of the other code, thus obtaining a higher overall code weight.

The S interleaver

The S (Semirandom) interleaver was introduced in (Divsalar and Pollara, 1995d) with the purpose of obtaining an interleaver that was still (partially) "randomly chosen", but had a bias towards associating (mapping) short error events of one of the codes to long error events of the second code. In fact, this interleaver will not map *at all* short to short error events. The "short" and "long" terms are defined by using the S parameter, a positive integer value. An error event is "long" if it has at least S information bits, otherwise it is "short". The S condition is realized by ensuring that any two bit positions in the direct input stream that are closer than S bits have their interleaved positions further away than S bits. Mathematically, this is expressed as:

$$\begin{aligned} &\forall \quad i, j \in \{0, .., N-1\}, \quad i \neq j \\ &if \qquad |i - j| < S \\ &then \quad |I(i) - I(j)| > S \end{aligned} \tag{3.1}$$

A more localised, algorithmic condition is presented in table (3.1).

$S(I,k,n)$
$\forall i \in \{k-S, k+S\}, i \neq k, \lvert I(i)-n\rvert > S$

Table 3.1: The S condition

The S condition for interleaver I, position k and corresponding interleaved value n. For clarity, the interleaver edge tests have been omitted. Also, in the case of designing the interleaver, the condition is considered satisfied for the values of i for which $I(i)$ does not yet exist.

	$getI(N,S)$
1.	$k \leftarrow 1, I \leftarrow \mathbf{0}, P \leftarrow \{1,...,N\}$
2.	**if** $P = \emptyset$ **then** *(deadlock)* **go to** 1.
3.	$n = rand(P), P \leftarrow P - \{n\}$
4.	**if** $!S(I,k,n)$ **then go to** 2.
5.	$I(k) \leftarrow n$
6.	**if** $k < N$ **then** $P \leftarrow \{1,...,N\} - \{I(1),...,I(k)\}, k \leftarrow k+1$, **go to** 2.
7.	return I

Table 3.2: S interleaver generator

It is clear that the aim of the design is to increase the value of S. A condition of $S = 0$ simply specifies a randomly chosen interleaver. Two algorithms to construct interleavers having a given value of parameter S are presented in tables (3.2) and (3.3). In order to find the practical limit, one can start with an estimated value of S, construct an interleaver and then increase the value of S by one. This process is repeated until it takes too long to construct the interleaver.

The implementation of the S interleaver algorithm presented in table (3.2) tries to follow the brief description of the S interleaver presented in (Divsalar and Pollara, 1995d). The aim of the algorithm is to design a permutation in which each position verifies the $S()$ condition described in table (3.1). The algorithm starts in position $k = 1$ with a completely undesigned permutation (all permutation values are set to zero to indicate an undesigned value, $I \leftarrow \mathbf{0}$). The set of all values available for the current position, P, is initialised to all available permutation values, which at the start are all the numbers from 1 to N, where N is the interleaver length. For each value of k, a random value n is taken from the set P, and excluded from it, to mark that it has been tried (step 3). If n verifies the local S condition $S(I,k,n)$ (step 4), then it is assigned to $I(k)$, $I(k) \leftarrow n$ (step 5). If not, a new value n is randomly chosen and excluded from P,

	getI(N, S)
1.	$k \leftarrow 1, I \leftarrow \mathbf{0}, P \leftarrow \{1, ..., N\}$
2.	**if** $P = \emptyset$ **then** *(swap)*
	if $\exists j \in \{1, ..., k-1\}$ and $\exists n \in \{1, ..., N\} - \{I(1), ..., I(k-1)\}$ so that $S(I, I(j), k)$ and $S(I, n, j)$ **then** $I(k) \leftarrow I(j)$ and $I(j) \leftarrow n$, **go to** 6. **else** *(deadlock)* **go to** 1.
3.	$n = rand(P), P \leftarrow P - \{n\}$
4.	**if** $!S(I, k, n)$ **then go to** 2.
5.	$I(k) \leftarrow n$
6.	**if** $k < N$ **then** $P \leftarrow \{1, ..., N\} - \{I(1), ..., I(k)\}, k \leftarrow k + 1$, **go to** 2.
7.	return I

Table 3.3: Fast S interleaver generator

until a value verifying the S condition is found, or P becomes the empty set, indicating that all available values were tried, but none of them was good (step 2). In the latter case, the algorithm has reached a deadlock, and it is restarted from the beginning. In the former case, a value of n has been found eventually, and it is assigned to $I(k)$. The fact that several values have been tried and excluded from P leads to the necessity of restoring P to the set of all values available for the next position, which is the set of all possible values $\{1, ..., N\}$ less the set of values already assigned, $\{I(1), ..., I(k)\}$ (step 6). If the end of the interleaver was reached, the algorithm finishes, returning an S interleaver (step 7). Otherwise, the next position is designed in the same way.

The amount of time needed to construct an interleaver having a given value of S can only be determined in statistical terms. In (Divsalar and Pollara, 1995d) it was specified that the maximum value of S that can be obtained in *reasonable* time depends on the interleaver length N as $S = \sqrt{N/2}$. In this work, by using the algorithm presented in table (3.2) it was found that the time needed to obtain such values of S was long (days), increasing with interleaver size. A closer examination of the algorithm has shown that, for the above values of S, the algorithm reaches a deadlock at a number of positions from the end of the interleaver which is around S, and thus rather small comparable to the length of the interleaver N. It was assumed that the algorithm fails so close to the end because of edge effects: the S condition is less restrictive at the edges of the interleaver, and thus these values are preferred, leading to non-uniform choices. This is why the swapping code was added in table (3.3). The cause of the deadlock is that

the values of n that could satisfy the S condition in the deadlock position were already assigned to previous positions.

The idea is that one of the available values could satisfy the S condition in a previously designed position, thus freeing a value that could satisfy the S condition in the current position. The new code searches for this pair, and swaps the values, thus pushing the algorithm forward. If such a pair cannot be found, the algorithm reaches a deadlock, and is restarted. The swapping code has provided the small number of positions necessary to reach the end of the interleaver, almost without any deadlock for values close to (and sometimes over) the $\sqrt{N/2}$ limit. Only a small number of trials are needed, leading to a fast algorithm.

This problem was also mentioned in (Lee et al., 1999) where it is solved in a different and interesting way, by starting with the square interleaver (see Annex A.2) and introducing randomness by performing random swaps that verify an S condition. The authors suggest that the algorithm can produce interleavers having any possible S value. The flaw in the argument is that, as the value of S is increased, the number of actual swaps decreases as compared to the number of trials performed, and the algorithm becomes very time consuming. An interleaver with the given value of S is indeed obtained, but it is not very random. This raises the interesting question of *how random can the S interleaver be, given the value of S*, question for which the number of swaps that are performed in a given number of trials can be an approximate answer.

A theoretical upper bound for the value of S for a given interleaver length N can be obtained by considering the fact that any S consecutive bits in the direct stream have to be interleaved at least S bits apart, thus the space occupied is $(S+1)^2 < N$ resulting in the upper bound

$$S < \sqrt{N} - 1 \tag{3.2}$$

Figure (3.4) presents the maximum values of the parameter S for different values of the block length N obtained by using the algorithm in table (3.3), together with the upper bound.

The square interleaver reaches the maximum value of S, and this is why it is used in (Lee et al., 1999) as a starting permutation. Still, it cannot be called an 'S' interleaver, since it is not at all random. One of the main reasons the maximum S value

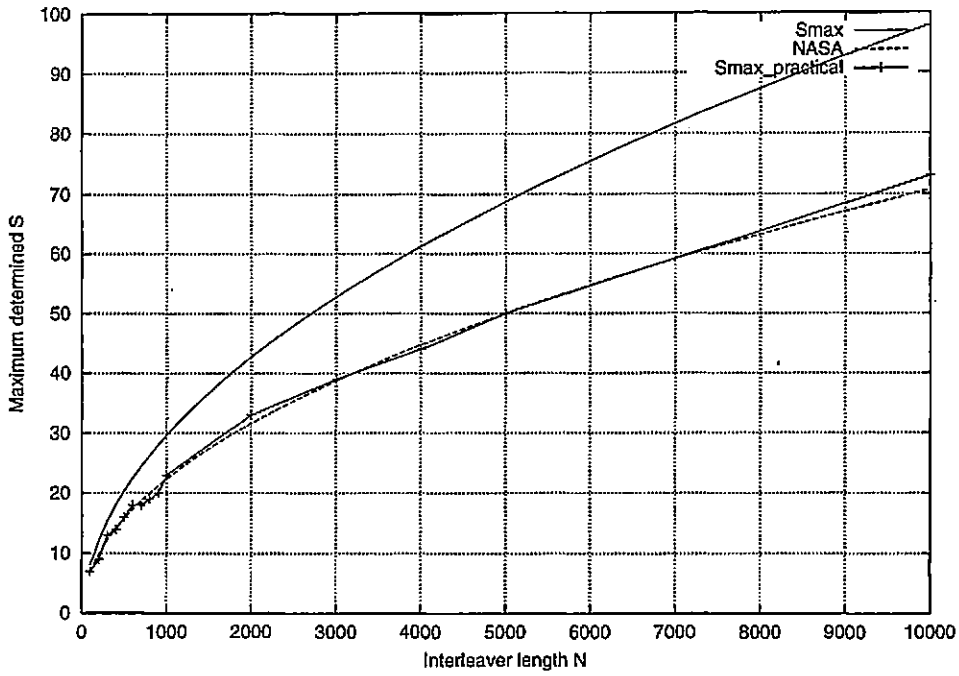


Figure 3.4: Practical S values
Maximum determined value for parameter S for different interleaver lengths and comparisons with the limit from literature and maximum possible value which is obtained for the square interleaver

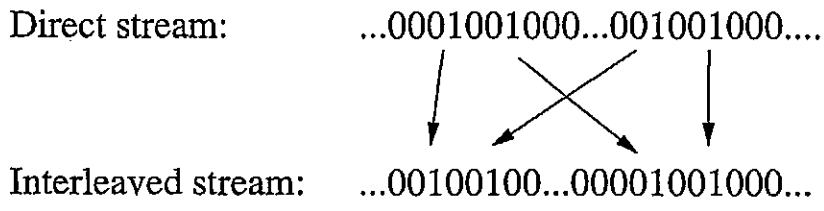


Figure 3.5: $IW = 2 + 2$ "crossed" error event
$IW = 2 + 2$ "crossed" error event observed for turbo codes using S interleavers, the $RSC(5/7)$ component code and any interleaver length

D	N=400			N=2500			N=10000		
	S=0	S=15	S=18	S=0	S=33	S=48	S=0	S=70	S=98
$\{2\}$	5	6	393	6	7	2493	9	3	9993
$\{2,5\}$	66	66	1548	61	80	9948	84	64	39948

Table 3.4: $IW = 2 + 2$ "crossed" error events multiplicity
The multiplicity of $IW = 2 + 2$ crossed error events for $RSC(5/7)$, based on $IW = 2$ error events having 2 bits of zero (first line) and 2 or 5 bits of zero (second line). Note that the multiplicity is cumulative (line two includes line one).

obtained by the square interleaver is sacrificed for "randomness" are the $IW = 2 + 2$ "crossed" error events. The information sequences that cause these error events with minimum code weight are presented in figure (3.5) for a turbo code using the $RSC(5/7)$ code. Each code produces two $(IW = 2, OW = 6)$ error events situated more than S bits away from each other. Since the two bits of 1 in an error event of the interleaved code belong to different error events of the first code, the S condition is fulfilled. The minimum code weight which is possible with such an arrangement is $OW_{2+2} \geq 2 * d_{free-eff}$, value which is independent of S. In the case of the $RSC(5/7)$ turbo code, this value is $2 * d_{free-eff} = 20$.

Note that any combination of two $IW = 2$ error events of the component code can cause a "crossed" error event. For the $RSC(5/7)$ code, the basic $IW = 2$ error events contain $\{2, 5, 8, ...\}$ zeros, and the "crossed" error events can result as any combination of these error events. Two error events having 2 zeros and 5 zeros in the direct stream, interleaved into two error events having 2 zeros each will produce the next higher OW_{2+2} code weight, equal to $8 + 6 + 4 + 4 = 22$ and so on.

It can be easily shown that the square interleaver produces a number of such mappings that increases with N and this is why the BER of a turbo code that uses it does not have an interleaver gain (Perez et al., 1996). A randomly chosen interleaver will do these mappings with a high probability, independent of N, but with a much smaller multiplicity. The S interleaver is a compromise between the need for a high value of S and a number of $IW = 2 + 2$ "crossed" error events as small as possible. The fact that an S interleaver produced by using the algorithm in table (3.3) is "random enough" is verified by the results presented in table (3.4). This table presents the number $IW = 2 + 2$ mappings for a randomly chosen interleaver, an S interleaver

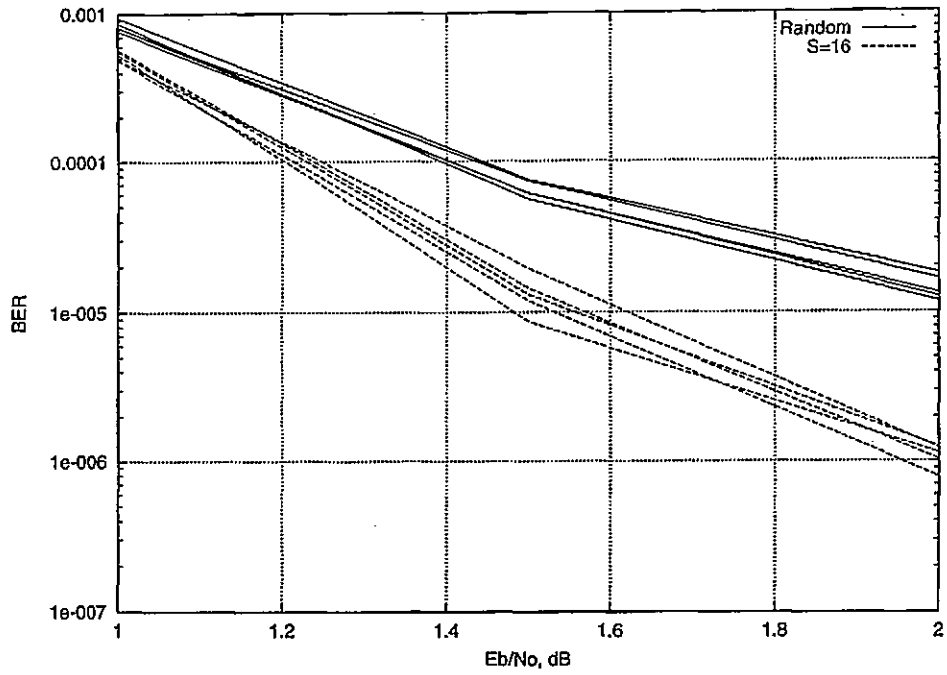


Figure 3.6: Random/S interleaver performance
Random versus S class interleavers performance for $RSC(5/7)$, $N = 500$ turbo codes. Five interleavers were chosen at random from each class. The separation in performance is visible.

and the square interleaver for different interleaver lengths. It can be observed that the values for the S interleaver are close to those for the randomly chosen interleaver. The value of S was roughly $S = \sqrt{N/2}$.

The advantage of using an S-class interleaver as opposed to a randomly chosen interleaver is shown in figure (3.6) where the performance of 5 interleavers from each class is presented in terms of BER, for an $RSC(5/7)$ turbo code with $N = 500$. A difference of an order of magnitude is obtained at $E_b/N_o = 2$dB. This difference

Block Size	500	2000	10000
S	16	26	39
IW(d_{free})	2	4	4
d_{free}	20	20	20
$(OW_2)_{min}$	20	30	-

Table 3.5: Turbo code S/random interleaver d_{free}
Variation of observed d_{free} of turbo codes using S-class interleavers for increasing values of N and S. IW(d_{free}) is the corresponding input weight for the d_{free} error event.

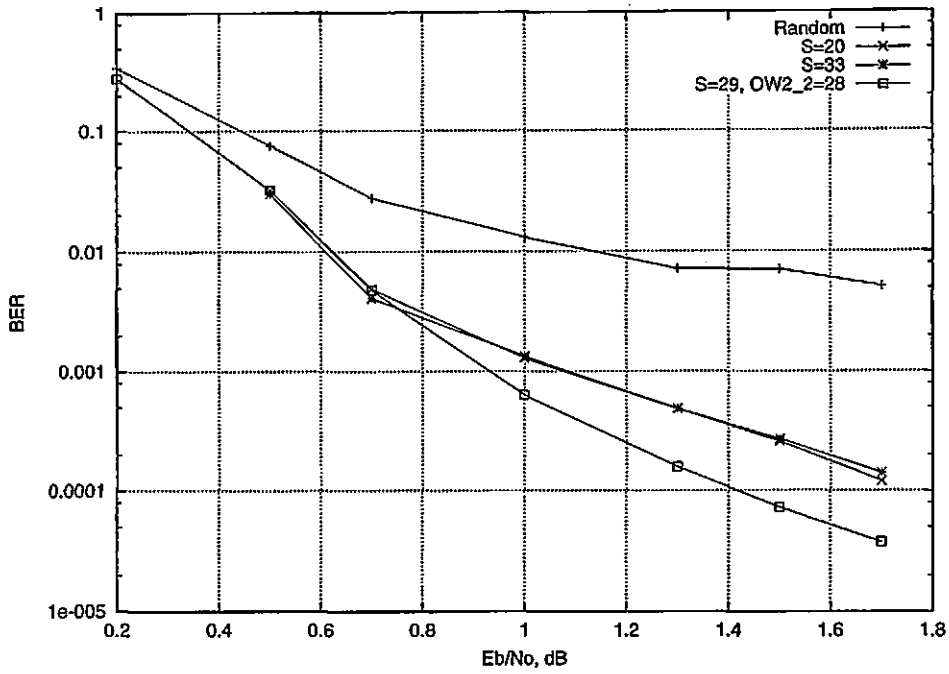


Figure 3.7: Improved S interleaver performance
Performance comparisons for an $N = 2000$, $M = 2$, $RSC(5/7)$ turbo code using an S interleaver, for different values of parameter S. Increasing S does not always improve performance, and more complicated design is necessary.

decreases with decreasing E_b/N_o. The effect of increasing the value of S on turbo code performance is presented in figure (3.7) for a block length $N = 2000$ and a component code RSC(5/7). It can be observed that there is a significant improvement in performance in going from $S = 0$ (randomly chosen interleaver) to $S = 20$ and no visible improvement as S is increased to $S = 33$. A similar effect is shown in Table (3.5), which shows the increase in d_{free} for turbo codes with increasing block length and thus higher practical limit of parameter S. The results indicate that there is a limit value of S over which the improvement in performance due to eliminating low code weight, IW=2 sequences is masked by the contribution of $IW = 2 + 2$ "crossed" error events. Above this limit, increasing S without improving the other error events has limited effect. A visualisation of the effect of increasing the S value until it reaches the $IW = 2 + 2$ limit is shown in figure (3.8). The figure shows the probability of producing a minimum weight by $IW = 2$ and $IW = 2 + 2$ error events for different values of S when the S interleaver is chosen at random. It can be seen that the probability curve for $IW = 2$ shifts right until it reaches the $IW = 2 + 2$ curve which

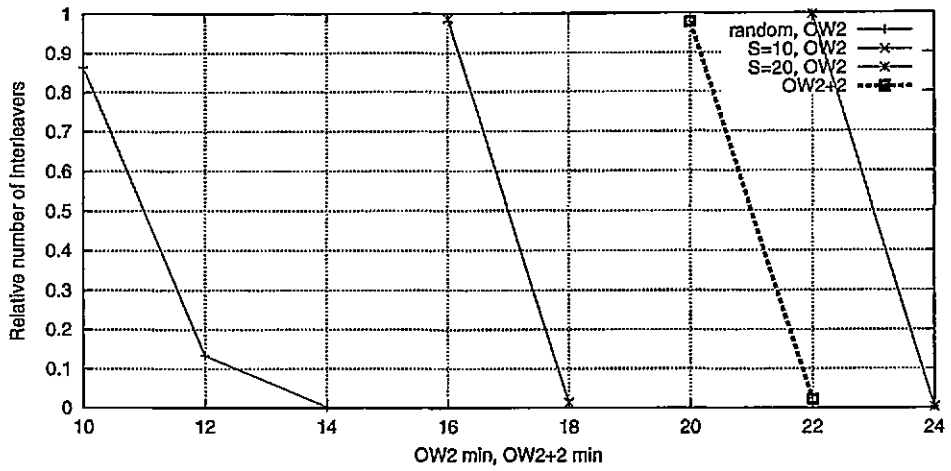


Figure 3.8: Turbo code $(OW_2)_{min}$ probability distributions
Minimum OW_2, OW_{2+2} probability distributions for turbo codes using $RSC(5/7)$ and random/S interleavers with different values of S. The block length is $N = 2000$.

is independent of S.

This limit can be determined due to the periodic way in which $IW = 2$ error events of RSC codes accumulate code weight, illustrated in figure (3.9) for the $RSC(5/7)$ and $RSC(7/5)$ codes. The number n of periods T of a parity sequence generated by an $IW = 2$ input sequence having at least S input zeros between the two ones is

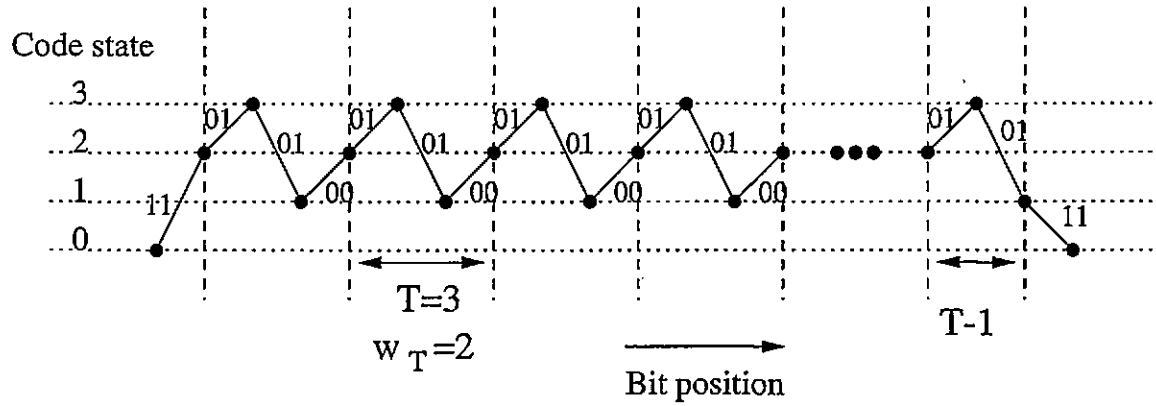
$$n \geq \left\lfloor \frac{S}{T} \right\rfloor + 1 \tag{3.3}$$

The worst case $IW = 2$ error event for the turbo code using the S interleaver associates an $IW = 2$ error event of the first code containing only one period with an error event of the second code containing n periods, resulting in

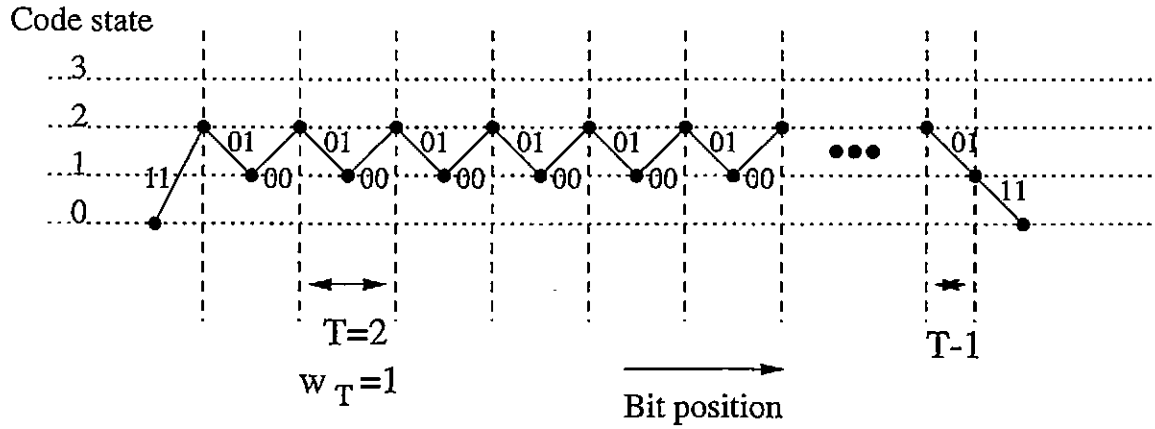
$$OW_2 \geq (n+1)w_T + IW + 2w_e \tag{3.4}$$

where $IW = 2$ is the information weight, and w_e is the 'edge' parity weight which for $RSC(f/g)$ codes with feedforward polynomials f having $f_0 = f_{k-1} = 1$ is always $w_e = 2$ (Divsalar and Pollara, 1995b). By replacing w_e and using equation (3.3) equation (3.4) becomes

$$OW_2 \geq (n+1)w_T + 6 \geq \left(\left\lfloor \frac{S}{T} \right\rfloor + 2 \right) w_T + 6 \tag{3.5}$$



a)



b)

Figure 3.9: $IW = 2$ periodic weight cumulation
Periodic cumulation of weight for $IW = 2$ sequences for a) RSC(5/7) and b) RSC(7/5).

The $d_{free-eff}$ of the turbo code is produced by the association of an $IW = 2$ error event containing one period for each code and thus,

$$d_{free-eff} = 2w_T + 6 \tag{3.6}$$

Equation (3.5) becomes:

$$OW_2 \geq \left\lfloor \frac{S}{T} \right\rfloor w_T + d_{free-eff} \tag{3.7}$$

The value of S can be seen as an "extension" factor for $d_{free-eff}$. Note that codes that have a period higher than the value of S will not change their minimum OW_2. Without the S condition (and also for any $S < T$), $OW_2 \geq d_{free-eff}$. From (3.7) it can be concluded that if an $IW = 2$ turbo code error event is to cumulate a weight higher than a given value W, the S interleaver condition has to be:

$$S \geq T \left(\left\lceil \frac{W-6}{w_T} \right\rceil - 2 \right) \tag{3.8}$$

Now suppose we want to determine the value of S for which OW_2 is higher than the smallest "crossed" error event code weight. Because this error event is composed of two $d_{free-eff}$ error events, it has $OW_{2+2} = 2 * d_{free-eff}$. Then equation (3.8) becomes

$$S_{2+2} \geq T \left(\left\lceil \frac{2d_{free-eff} - 6}{w_T} \right\rceil - 2 \right) = T \left(\left\lceil \frac{6}{w_T} \right\rceil + 2 \right) \tag{3.9}$$

where the second equality makes use of equation (3.6). The RSC(5/7) code, for which the periodic cumulation of weight is shown in figure 3.9(a), has $T = 3$, $w_T = 2$ and thus $S_{2+2} \geq 15$ whereas the RSC(7/5) code, shown in figure 3.9(b), has $T = 2$, $w_T = 1$ resulting in $S_{2+2} \geq 16$.

From figure (3.4) it can be observed that these two codes can reach their S_{2+2} for any $N \geq 500$. The small multiplicity of "crossed" error events in table (3.4) suggests the possibility of increasing the $(OW_{2+2})_{min}$ by rejecting some "crossed" error events of low code weight. This could be accomplished in several ways:

- By serial concatenation of the turbo code with a block error correcting code, capable of correcting 4 errors wherever they are positioned in the block. In the

$iw2x2(I,k,D)$
$\exists d_1,d_2,d_3,d_4 \in D$, so that $I^{-1}(I(I^{-1}(I(k)\pm d_1)\pm d_2)\pm d_3)\pm d_4 = k$

Table 3.6: $IW = 2+2$ "crossed" error event condition
In designing the interleaver, if any of the values of I or I^{-1} involved in the condition does not yet exist, the condition is considered *not* satisfied. The set D is characteristic to the component code. The interleaver edge conditions should be tested.

case of a BCH code, (Andersen, 1996) mentions a required number of $16*4 = 64$ parity bits. The resulting decrease in code rate becomes less significant with increasing block length.

- By *forcing* the value of one of the bits in the error event to zero. This will not reduce the value of S. The encoder always transmits zero in that position and the iterative decoder forces the value of the extrinsic probability to zero. Simulation shows that forcing only one bit out of 4 is enough to clear the error. This is an improvement to the method reported in (Oberg and Siegel, 1997), since a smaller number of bits need to be used to improve error correction. Nevertheless, it still results in a reduction of code rate which increases with the number of "crossed" error events but decreases with block size.

- Modifying the algorithm that constructs the S interleaver (table (3.3)) to include a supplementary condition: an interleaver position is accepted only if it verifies the S condition and *it does not close an* $IW = 2+2$, *two error event loop*, condition formalised in table (3.6). This does not reduce the code rate. The number of zeros in each of the basic $IW = 2$ error events belongs to a limited set of values, D. The larger the set, the higher the $(OW_{2+2})_{min}$, but the more difficult to obtain a high S value for the interleaver. This has the effect of balancing the S value between two conditions, leading to a compromise between $IW = 2$ and $IW = 2+2$ error events.

The first method is ideal if the code rate can be reduced (for long blocks). The second method is just interesting for research and it is better than the first method only if just the first line in table (3.4) needs to be cleared. The third method has the advantage of not reducing the code rate, but only a limited number of "crossed" error events can be eliminated.

The third method has been applied to improve a turbo code using an $RSC(5/7)$ component code with block length $N = 2000$. Comparative results are presented in figure (3.7). They show the bit error rate curves for $S = 0$ (random), $S = 20$, $S = 33$ and for an $S = 29$ interleaver. The $S = 29$ interleaver has been designed to exclude the "crossed" error events caused by all the combinations of $IW = 2$ error events with $D = \{2, 5, 8, 11\}$ zeros. This results in $(OW_{2+2})_{min} = 28$ and also causes the reduction of S from $S = 33$ to $S = 29$, allowing for $(OW_2)_{min} = 28$. Simulations show an expected $d_{free} = 28$ for this interleaver, and a corresponding improvement. The experiment has been done for $N = 2000$ in order to allow for the decrease in S. Trying to increase the free distance to $d_{free} = 30$ at the same N is not possible in this way because S would decrease and OW_2 is already smaller than 30. For longer interleavers, the attempt fails to obtain a $d_{free} > 30$ because of the large number of $IW = 2 + 2$ loops which reduces S to very low values. Also, $IW = 2 + 2 + 2$, triple $IW = 2$ "crossed" error events have their minimum code weight equal to $3 * d_{free-eff} = 30$, adding to the number of loops that should be rejected. Thus, $d_{free} = 30$ is the limit for the $RSC(5/7)$ turbo codes designed in this way.

Characteristic to the S interleaver is that the component code error events are usually groups of low weight error events (mostly $IW = 2$), and not higher IW single error events. This is because these error events for the component code, at least for relatively small code weights, are short, and as a consequence their information bits are interleaved far apart, with a total distance increasing with information weight, resulting in a high total error event length, and thus a high code weight. This is why excluding the $IW = 2$ groups of error events results in an increase of the free distance.

The algorithms presented for designing the interleavers belong to the category of algorithms based on *rigid* conditions, leading to the design of an interleaver with uniform properties. Different methods based on more flexible conditions, such as a *cost function* which should be maximised over the whole interleaver are presented in (Daneshgaran and Mondin, 1997a; Hokfelt and Maseng, 1997). Algorithms with rigid conditions are usually approximations of a cost function too complicated to implement or even to determine.

Memory	Optimal code					Non-optimal code				
	RSC	$d_{free,eff}$	T	w_T	S_{2+2}	RSC	$d_{free,eff}$	T	w_T	S_{2+2}
2	5/7	10	3	2	15	7/5	8	2	1	16
3	17/13	14	7	4	28	11/17	8	2	1	16
4	37/23	22	15	8	45	21/37	10	5	2	25
5	45/67	38	31	16	93	-	-	-	-	-

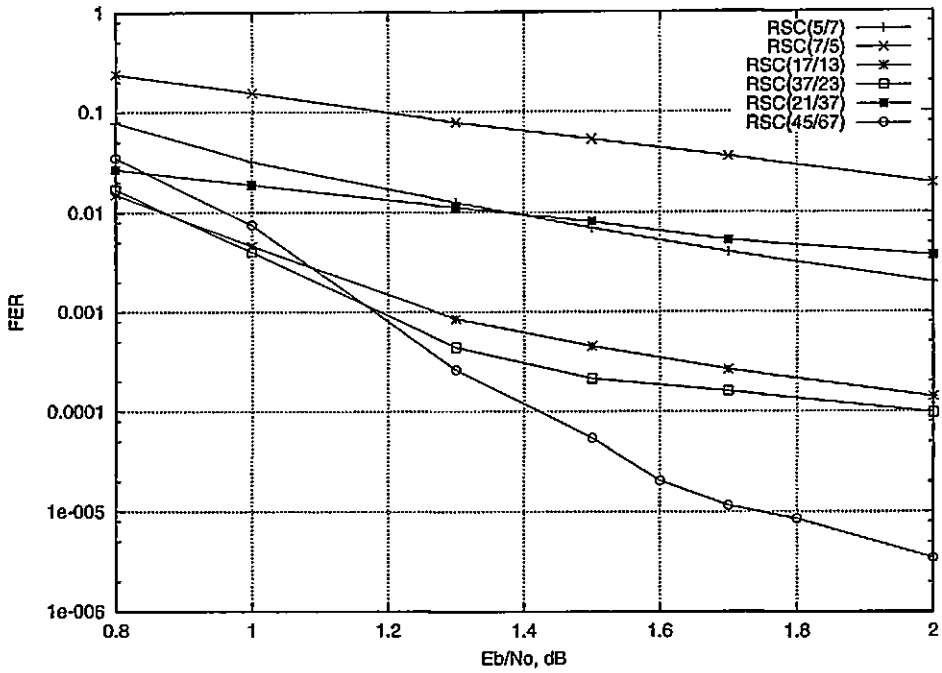
Table 3.7: Optimal/non-optimal codes
Optimal/Non-optimal component codes used in the simulations. The optimal codes are taken from (Benedetto et al., 1998b).

3.3.2 Component code factor

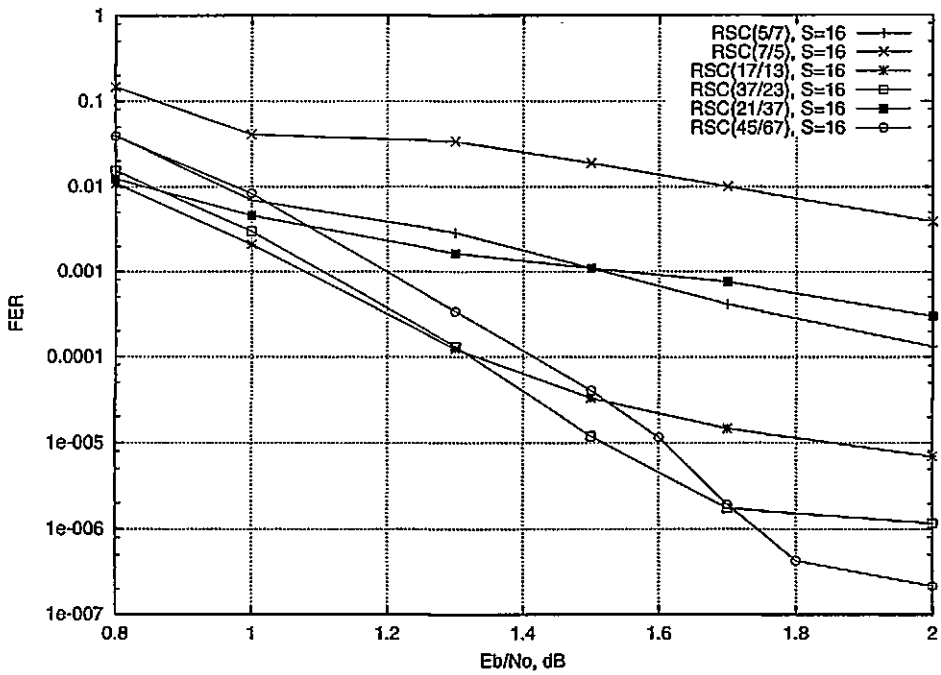
The way to improve the second factor with regards to the first is to improve the parity weight associated with low information weight error events. This is dependent on the code, but there are some general rules which only depend on code memory. This regards primarily $IW = 2$ error events, which are the worst from the point of view of the first factor. The parity weight for $IW = 2$ is maximised simply by choosing a primitive feedback polynomial. In this case, the parity sequence is a Maximum Length sequence with weight $w_T = 2^{m-1}$ and longest period $T = 2^m - 1$, for any non-zero feedforward polynomial f, due to the shift and add property of the Maximum Length sequences (Divsalar and Pollara, 1995b). It was shown in (Divsalar and Pollara, 1995b) that this is the maximum possible value. In (Benedetto et al., 1998b), computer simulations have been used to improve the parity weight associated with higher information weights, in increasing order, for an improved match with the first factor. This is done by choosing the feed forward/primitive feedback pair. Also, their multiplicity has been minimised, as a secondary condition. These codes are called *optimal* from the probabilistic design point of view. A list of optimal/non-optimal codes used in this work is presented in table (3.7).

The limit on w_T and parity weights associated with higher information weight can be increased by increasing code memory. This leads to increased complexity (decoding time and/or memory), and also to negative effects on iterative decoding, as it will be shown in the following. Still, very good results can be obtained with very low memory component codes.

In this work, the codes presented in table (3.7), having memory in the range $M \in$



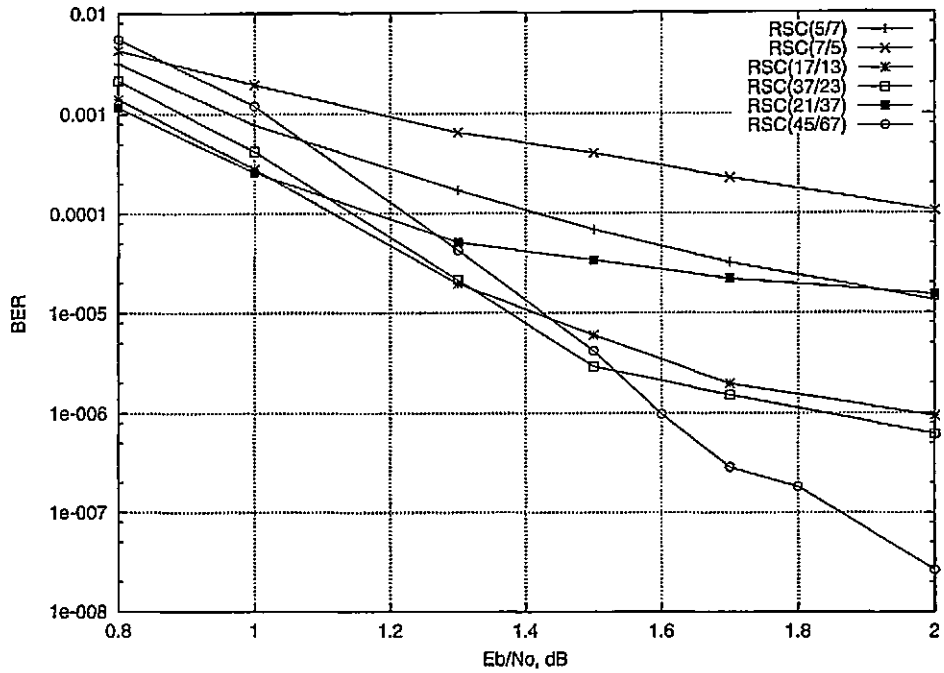
a)



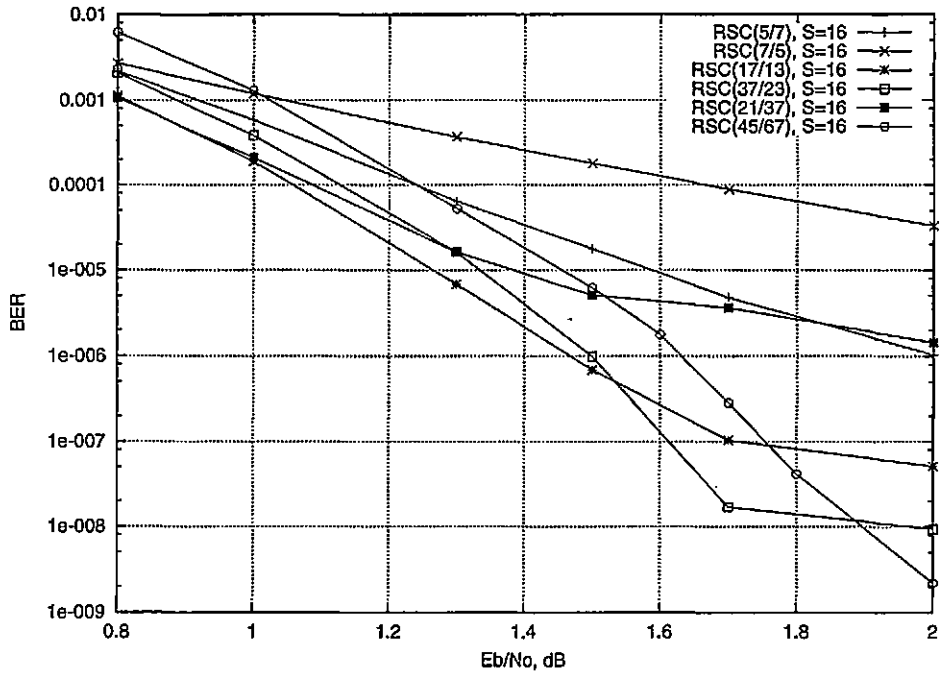
b)

Figure 3.10: Turbo codes FER for N=500

Turbo codes FER for N=500, different component codes and a) randomly chosen interleaver and b) designed (S-type) interleaver



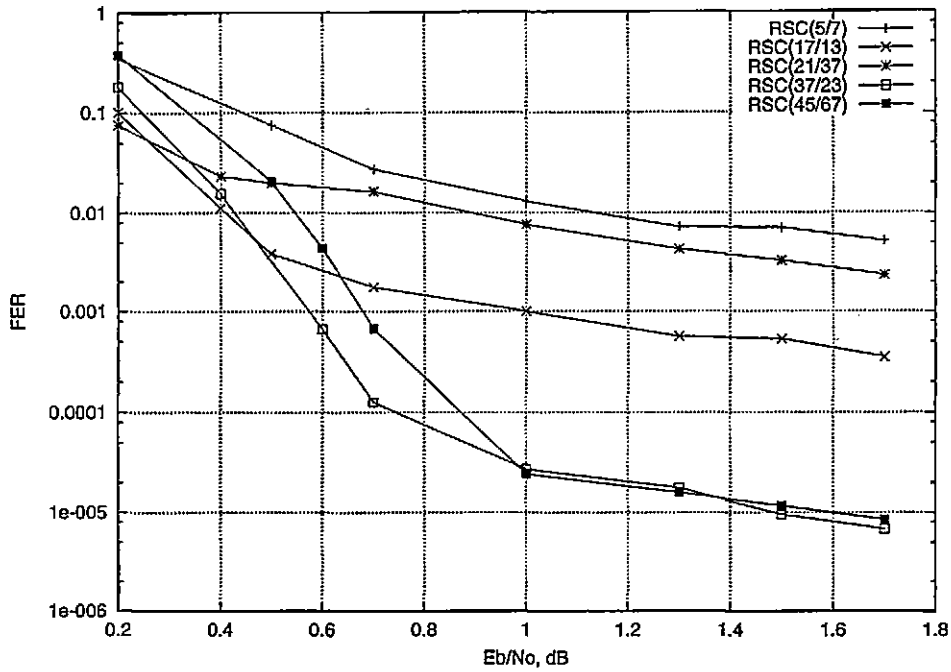
a)



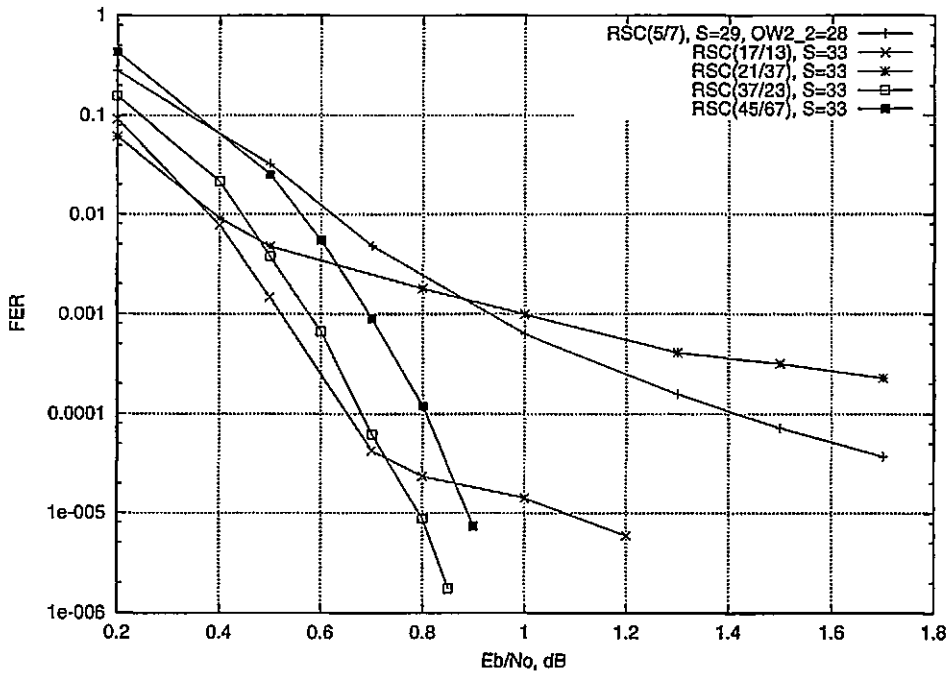
b)

Figure 3.11: Turbo codes BER for N=500

Turbo codes BER for N=500, different component codes and a) randomly chosen interleaver and b) designed (S-type) interleaver



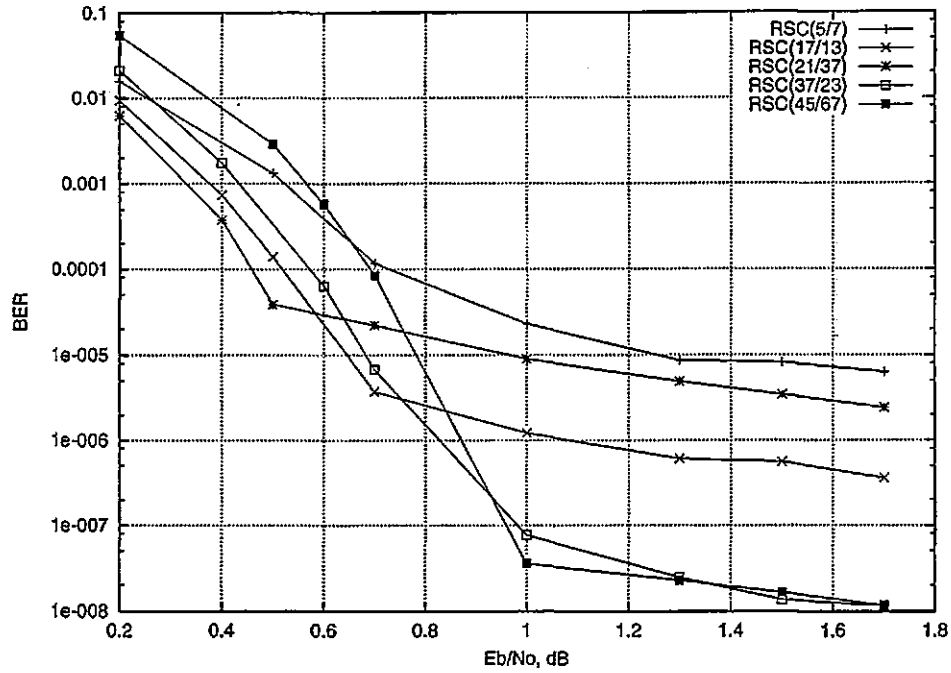
a)



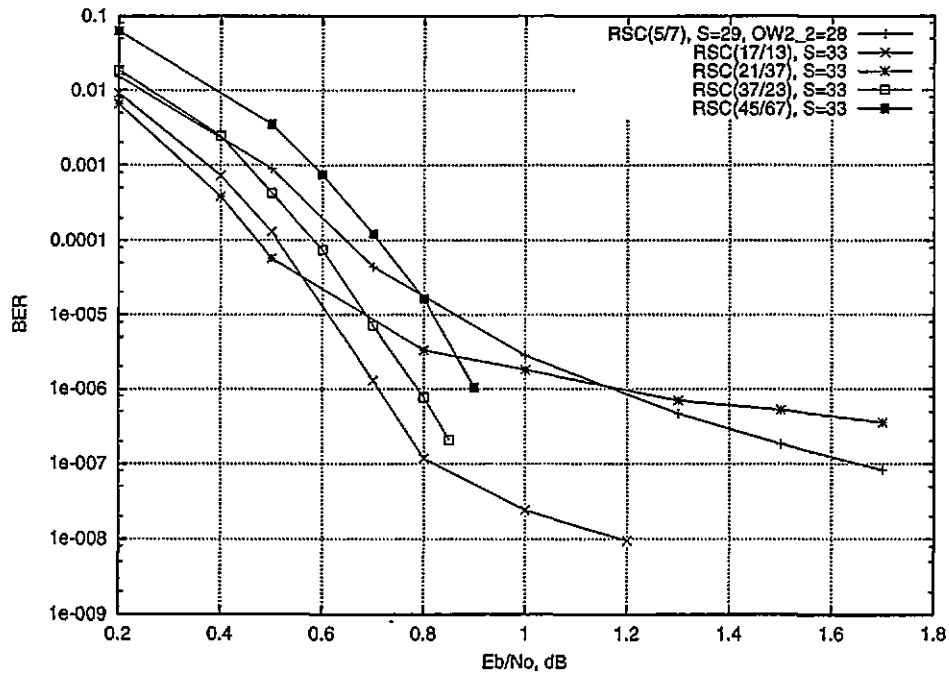
b)

Figure 3.12: Turbo codes FER for N=2000

Turbo codes FER for N=2000, different component codes and a) randomly chosen interleaver and b) designed (S-type) interleaver.



a)



b)

Figure 3.13: Turbo codes BER for N=2000

Turbo codes BER for N=2000, different component codes and a) randomly chosen interleaver and b) designed (S-type) interleaver.

$\{2,..,5\}$ were embedded in turbo code systems and their performance compared for different E_b/N_o values. The simulation results in terms of FER and BER are shown for $N = 500$ in figure (3.10) (FER) and figure (3.11) (BER) for a) a randomly chosen interleaver and b) a designed (S) interleaver and for $N = 2000$ in figure (3.12) (FER) and figure (3.13) (BER) for a) a randomly chosen interleaver and b) a designed (S) interleaver.

The first observation has to be the fact that the performance of the iterative decoder has two components, one that decreases quickly with E_b/N_o, produced by HIWHOW error blocks and one that has a slower decrease with E_b/N_o, produced mainly by LIWLOW error blocks and also by LIWHOW error blocks. The decrease with E_b/N_o of the second component can be correlated with the optimal decoding performance of turbo codes. The different behaviour of the two components with E_b/N_o produces the slope change in the performance curve, the error floor of turbo codes.

It can be observed that there are crossing points in the performance of codes of different memory. Usually, a high memory code is worse at low E_b/N_o than a low memory code and better at higher E_b/N_o. The crossing points become apparent for turbo codes starting with $M = 3$. There is also a crossing point in the performance of optimal/non-optimal codes for memory $M = 4$. The worst performance is that of the $RSC(7/5)$ non-optimal component code, both in terms of FER and BER.

The crossing of the FER curves generally happens at lower E_b/N_o than that of the BER curves. This is due to the fact that the number of errors in a HIWHOW block increases with code memory. Thus, there exist values of E_b/N_o where higher memory codes have less blocks in error, but a higher number of errors in a block, and thus a higher number of bit errors.

The $M = 4$ non-optimal $RSC(21/37)$ turbo code has the best BER at low E_b/N_o of all the codes used in the simulation. Its performance is dominated by a large number of LIWLOW error blocks and a small number of HIWHOW error blocks. This produces a rather high FER, as opposed to the low BER. Attempts to improve its performance by using an S interleaver resulted in a decrease of its FER and BER at high E_b/N_o (the S interleaver eliminates some of the LIWLOW error blocks). At low E_b/N_o the BER is slightly increased when using the S interleaver due to an increase in the number of HIWHOW blocks which compensate the BER reduction due to the smaller number

of LIWLOW blocks. The reduction in LIWLOW error blocks is significant enough to "uncover" the contribution of the HIWHOW error events, resulting in the usual slope change in the BER curve. The FER is improved at low E_b/N_o by using the S interleaver for $N = 500$ and slightly degraded for $N = 2000$. This is again due to the different balance of the two effects of the S interleaver: the reduction in the number of LIWLOW and increase in the number of HIWHOW error blocks. Note that the $RSC(21/37)$ code has been used in the original paper (Berrou et al., 1993b) because of its good performance at low E_b/N_o but it has been determined by simulation in (Andersen, 1999) that there are other codes that have better performance at low E_b/N_o, of which the best code is $RSC(37/25)$.

As the component code memory is increased from $M = 3$ to $M = 5$ the crossing points in performance are separated by around $0.2 - 0.3$dB. They move left in E_b/N_o with interleaver length and also happen at lower BER and FER values. The low E_b/N_o crossings happen for MAP decodings of simple convolutional codes, a process which does not use the iterative algorithm. The sub-optimal codes behave better at low E_b/N_o when used as a single code. This can be part of the reason why non-optimal codes behave better at low E_b/N_o when used in turbo codes.

The LIWLOW error events can be used to determine what error events produce the error floor for each component codes, and how the interleaver design improves the error floor. Figures 3.10(a) and (b) show the improvement obtained by using an $N = 500$, $S = 16$ interleaver instead of a randomly chosen interleaver.

For the $RSC(17/13)$ code, the $(IW = 3, OW = 15)$ error block that produces the d_{free} for the randomly chosen interleaver is too short for the $S = 16$ interleaver and thus its $d_{free} = 22$, caused by an $(IW = 2, OW = 22)$ error block. The lowest $IW = 3$ code weight observed was $OW_3 = 27$. For $N = 2000$, the d_{free} for this component code code is caused by an $(IW = 2, OW = 14)$ error event, whereas for the $S = 33$ interleaver it is caused by an $(IW = 2 + 2, OW = 28)$ error event, resulting in a significant improvement. An attempt to reject the IW_{2+2} error events producing this floor would not be useful, because the value of S needed to increase $OW_2 \geq 32$ can be calculated using (3.8) as $S \geq 35$. Better d_{free} values could be obtained for longer interleavers. A $d_{free} = 40$ is estimated for $N = 5000$, $S = 50$, $RSC(17/13)$ code with the $D = \{6, 13, 20\}$ crossed error events removed, since the S condition for $OW_2 \geq 40$

is $S \geq 49$.

The LIWLOW with lowest code weight for the turbo code using $RSC(37/23)$ and a randomly chosen interleaver is not the $(IW = 2, OW = 22)$ error block that causes its $d_{free-eff}$, but an $(IW = 3, OW = 15)$ error block, caused by the association of an $(IW = 3, OW = 8)$, length 9 error event of the first code with an $(IW = 3, OW = 10)$, length 13 error event of the second code. For the $RSC(37/23)$ turbo code using the $S = 16$ interleaver such an association is not possible. Indeed, in this case the lowest code weight is produced by an $(IW = 3, OW = 29)$ error block caused by the association of an $(IW = 3, OW = 16)$, length 19 error event of the first code with an $(IW = 3, OW = 16)$, length 20 error event of the second code. The $(IW = 2, OW = 22)$ error block cannot occur since $S = 16$ is slightly higher than the period $T = 15$ of the component code, resulting in an $(OW_2)_{min} = 30$. Thus using the $S = 16$ interleaver increases the d_{free} of the $RSC(37/23)$ turbo code from $d_{free} = 15$ to $d_{free} = 29$. For $N = 2000$, the d_{free} is produced by the $(IW = 2, OW = 22)$ and thus $d_{free} = d_{free-eff}$ for the randomly chosen interleaver. The $IW = 3$ error block with the lowest weight observed was $(IW = 3, OW = 23)$. No LIWLOW error block was observed for the $RSC(37/23)$ turbo code using the $S = 33$ interleaver. The lowest possible $(OW_2)_{min}$ can be calculated using equation (3.7) and the characteristics of the code in table (3.7) as $(OW_2)_{min} = \lfloor \frac{33}{15} \rfloor * 8 + 22 = 38$. Since such a mapping is very likely to occur, it is expected that $d_{free} \leq 38$ for this code. A quick computer search for $IW = 2$ error events has shown that the $(IW = 2, OW = 38)$ error event is indeed mapped by the $S = 33$ interleaver with a multiplicity of 5.

The $RSC(45/67)$ error floor is caused by an $IW = 3$ error event: $(IW = 3, OW = 23)$ for the randomly chosen interleaver and $(IW = 3, OW = 27)$ for the $S = 16$ interleaver. Another trial with a different interleaver has shown a $d_{free} = 33$ for the $RSC(45/67)$ turbo code, also caused by an $IW = 3$ error event.

It is interesting to notice that, although the probability of mapping low OW_3 to low OW_3 error events decreases as $1/N$, both $N = 500$ and $N = 2000$ are too short to avoid obtaining a d_{free} caused by such an error event, since both the randomly chosen and the S interleaver do the mapping. Although the $RSC(17/13)$ with S interleaver has $d_{free} = (OW_2)_{min}$, low OW_3 error events could be observed.

The simulations show that increasing the code memory and using a primitive feed-

back produces a large improvement in the error floor, but they lose out at low E_b/N_o most likely due to the iterative algorithm. There is also a complexity/performance balance to consider, since each increase in memory doubles complexity. Increasing memory could be essential for interleaver improvements to be effective. Improving turbo codes by increasing memory relies on providing the interleaver with a smaller number of error events that it could map badly for a given target d_{free}. Higher memory codes increase the $d_{free-eff}$ so much, that even if $IW = 3$ error events are less likely to be mapped (and indeed fewer of them were observed), they usually show up as the d_{free} of the code. This is due to their increased multiplicity, which compensates for their stronger interleaver factor.

This shows the rather weak interleaver gain of turbo codes. Probably increasing the block length to $N = 10000$ would make $IW = 3$ error events very unlikely for a given d_{free}, but they *do* appear even for $N = 2000$.

The conclusion is that since turbo codes have a rather weak interleaver (random) factor, their design relies heavily on the code factor and more carefully chosen interleavers. Code memory $M = 4$ has been chosen as the best compromise for low E_b/N_o performance and complexity against possibility of improvement. It can be observed that for $N = 2000$, and $S = 33$ interleaver, their error floor is outside the simulation range (FER $\leq 10^{-6}$).

Turbo codes using non binary convolutional codes and a special interleaver design suited for these codes have recently been presented in (Berrou and Jezequel, 1999). The new codes make the interleaver design easier.

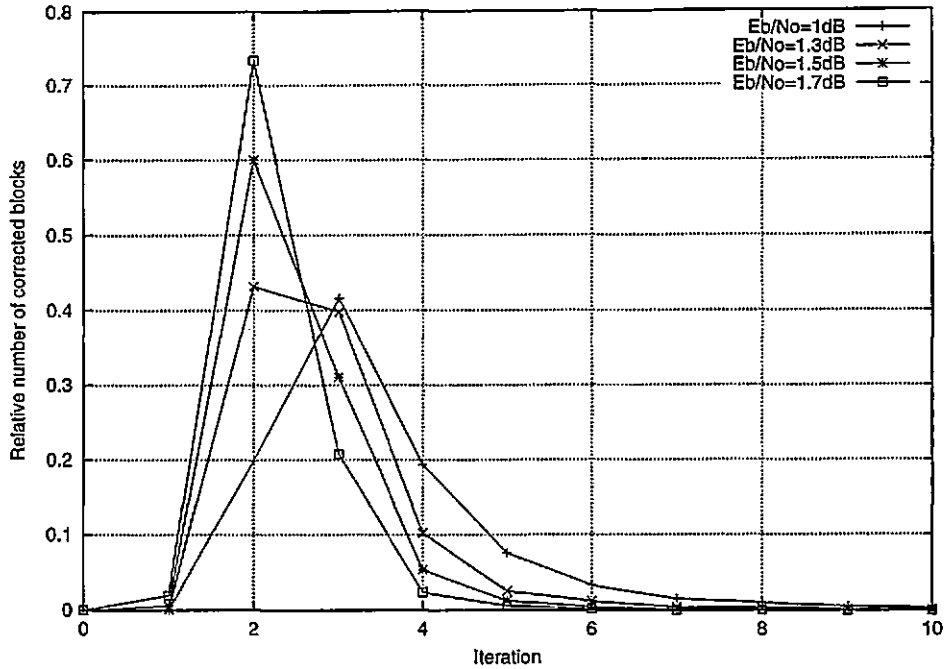
3.3.3 Decoding complexity

A turbo decoder can be implemented in two ways: as a pipeline of decoders, or as a single decoder with feedback. The pipeline decodes the turbo code in the time needed to decode one iteration, but it has to have a fixed number of iterations. An advantage of a single decoder with feedback is that it can allow for a variable number of iterations. The average number of iterations depends on the method used to stop iteration. Several methods are presented in (Hagenauer et al., 1996; Robertson, 1994; Shibutani et al., 1999). If complexity is considered proportional with the number of iterations, a decoder with feedback can reduce complexity at the cost of a bigger input

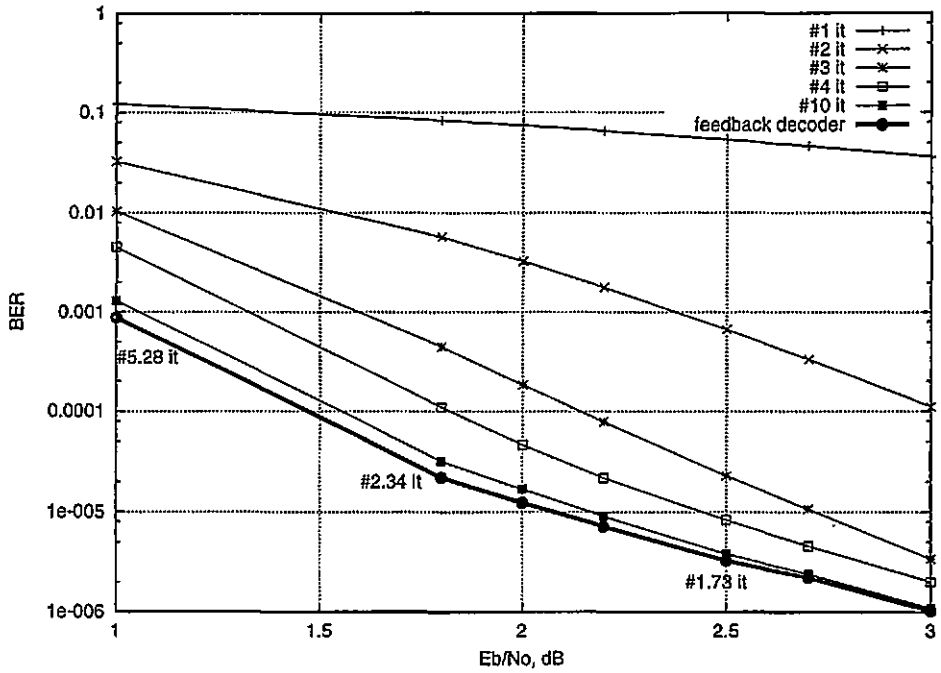
buffer. In the following, the (ideal) stopping criterion considered is: the iteration is stopped when there are no more errors in the block, or a maximum number of iterations (50) has been reached. In figure 3.14(a), the number of blocks corrected for each iteration is presented, relative to the total number of blocks. Several graphs are presented for different E_b/N_o values. It can be observed that the maximum moves to the left (smaller number of iterations) as the E_b/N_o increases, and also the spread of the distribution decreases. Generally, the curve becomes close to zero after about 10 iterations, although some blocks were observed which could be finally decoded, usually without error, after hundreds or even thousands of iterations. It could be assumed that the distribution has a long tail, although it is difficult to tell whether some of the blocks (usually decoded as HIWHOW) would ever converge. The resulting comparisons with the fixed number of iterations are shown in figure 3.14(b). It can be seen that, in conformity with the distribution of the decoded blocks, the most improvement is obtained in the first 3−4 iterations. After that, the improvements are small, converging asymptotically to the feedback decoder curve. Approximately 10 iterations are needed to get close to this curve, with closeness decreasing with E_b/N_o, but insignificantly. The feedback decoder only needs an average of 5.3 iterations at $E_b/N_o = 1$dB down to 1.7 iterations at $E_b/N_o = 2.5$dB.

Investigation of the number of blocks decoded correctly after each iteration can produce interesting results. As an example, a comparison for turbo codes using the $RSC(5/7)$ code and different block lengths at $E_b/N_o = 1.5$dB is shown in figure 3.15(a). They show that turbo codes with small block lengths decode correctly more blocks in the first iterations, but have a larger spread and longer tail of the distribution. Figure 3.15(b) shows the comparison for turbo codes with $RSC(5/7)$ at $E_b/N_o = 1.5$dB using a randomly chosen interleaver as compared to an S interleaver. The S interleaver curve is shifted slightly left, showing that the S interleaver decodes quicker. Since these curves describe the quickness of decoding, they could be used to characterise convergence.

The average number of iterations for several of the turbo codes in the simulations presented in figures (3.10−3.13) are shown in figure (3.16).



a)



b)

Figure 3.14: Correctly decoded blocks vs iteration for different E_b/N_o Turbo code with block length $N = 500$, $RSC(5/7)$ component code: a) Histogram of correctly decoded blocks versus iteration at different Eb/No values b) Performance with/without stop criteria. The numbers under the "feedback decoder" curve represent the average number of iterations of the decoder with feedback using the stop at zero errors criterion.

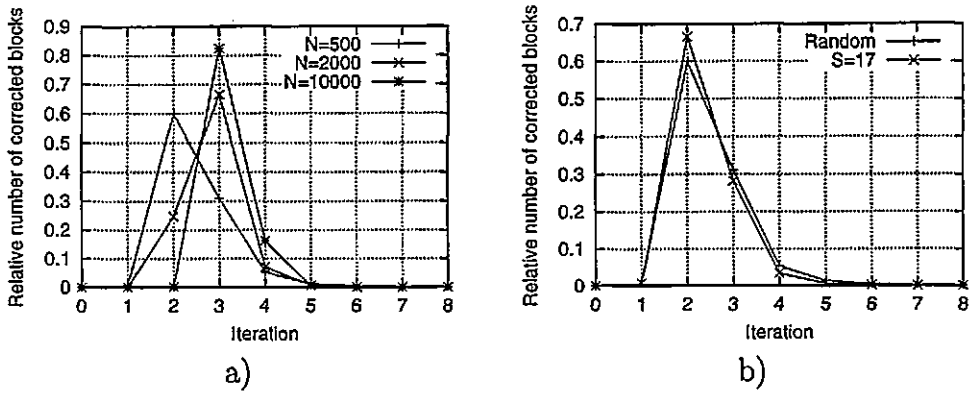


Figure 3.15: Correctly decoded blocks vs iteration for different parameters
The effect of a) increasing interleaver size and b) using the S-class interleaver instead of a randomly chosen interleaver on the number of correctly decoded blocks versus iteration

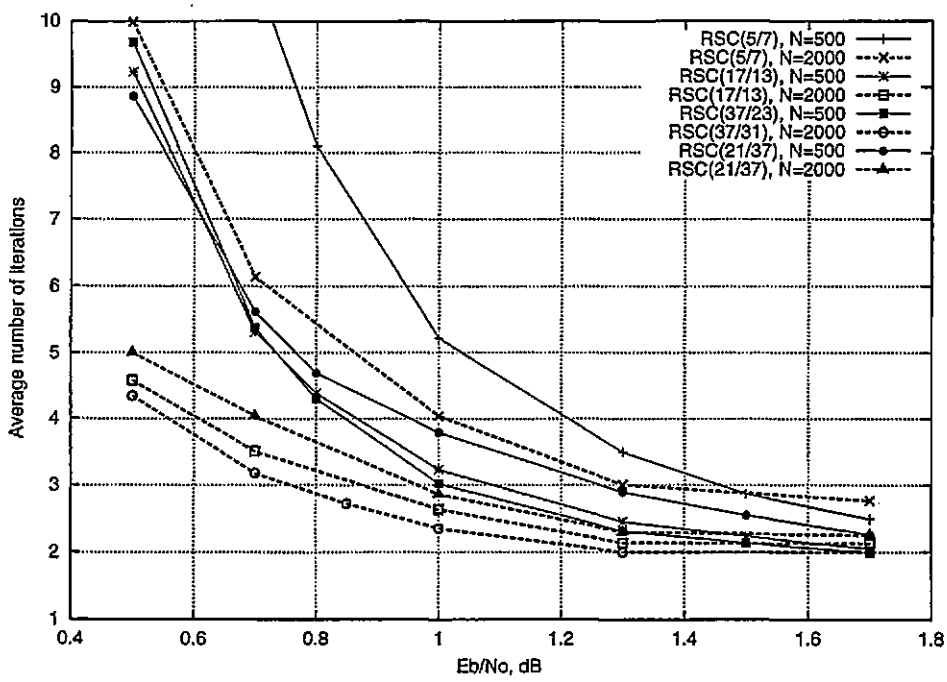


Figure 3.16: Turbo codes average number of iterations
Average number of iterations for different memory/block size turbo codes

3.4 The multiple parallel concatenation

The 3PCCC schemes improve on the interleaver factor, at the price of decreasing code rate. The code rate can be regained either by puncturing or higher rate component codes.

It should be easier to obtain a good code by just picking an interleaver pair at random and there should be a reduced necessity for higher memory codes. In this case, the probability of mapping an $IW = 2$ error event into itself goes down as $1/N$ and $IW = 3$ as $1/N^2$. The d_{free} obtained is more likely to be higher.

3.4.1 Interleaver factor

Similar to turbo codes, the interleaver is designed to increase the total possible length of a 3PCCC error event. In this case, there are two interleavers to design. They could be independently designed, or they could be paired for better performance. The $d_{free-eff}$ definition is readily extended for 3PCCC schemes with randomly chosen interleavers:

$$OW_2 \geq d_{free-eff} = 3w_T + 3w_e + 2 = 3w_T + 8 \tag{3.10}$$

The worst case $IW = 2$ error event for randomly chosen interleavers is presented in figure 3.17(a) and it coincides with the $d_{free-eff}$ of the 3PCCC scheme.

Independent S interleavers

By using two randomly chosen S interleavers, it can be made sure that a short error event in the non-interleaved stream is associated with a long error event in each of the interleaved streams. Also, short error events in any of the interleaved streams, are associated with long ones in the non-interleaved stream. The worst case is presented in figure 3.17(b). A short error event in one of the interleaved streams *could* be associated with a short error event in the other interleaved stream and the two independent S conditions will still be satisfied. This could have an impact on the ML performance, depending on how often this mapping will occur when the two S interleavers are chosen at random.

The minimum code weight associated with an $IW = 2$ error event for this case is

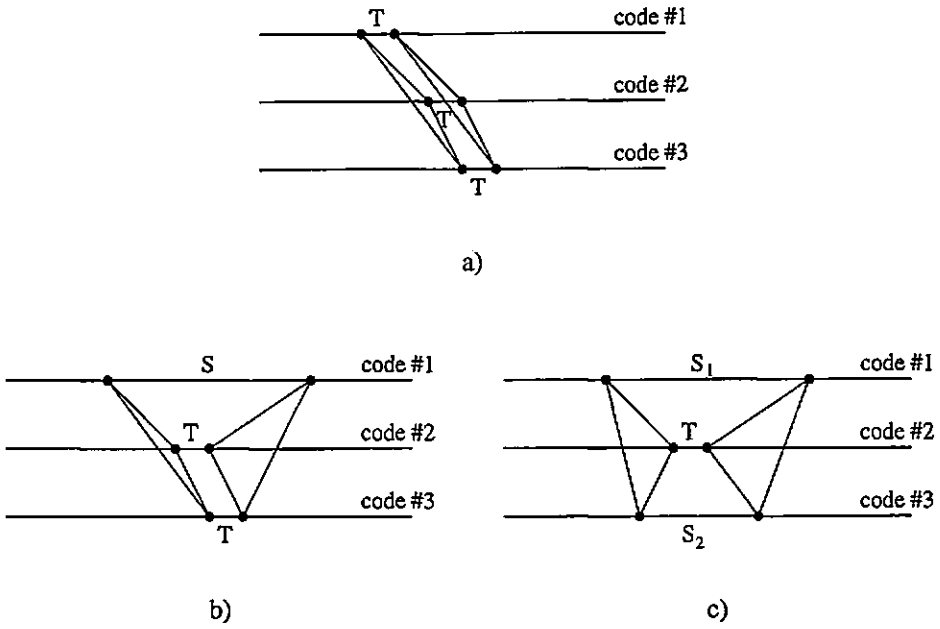


Figure 3.17: 3PCCC worst case $IW = 2$ error events
a) two randomly chosen interleavers ($d_{free-eff}$), b) two independent S interleavers and c) two paired S interleavers. The dots on the code axis represent the two bits of one which cause the error event for each code.

lower bounded by:

$$OW_2 \geq \left(\left\lfloor \frac{S}{T} \right\rfloor + 3 \right) w_T + 8 = \left\lfloor \frac{S}{T} \right\rfloor w_T + d_{free-eff} \tag{3.11}$$

This limit is imposed by the possibility of two "short" error events and a "long" one due to the independent S condition.

Paired S interleavers

Another possibility is to design the two interleavers in reference to each other. This can be accomplished in two ways:

- By simultaneously designing both interleavers. Thus the algorithm starts with both mappings unknown and designs each position alternately. This would be done with the purpose of obtaining a more balanced design.

- Choosing a good S interleaver as the first interleaver and designing the second interleaver as an S interleaver in reference to both the first and second code. The value of S for the first interleaver could also be lowered with the purpose of

$S(I_1, I_2, k, n)$
$\forall i \in \{k-S, k+S\},\ i \neq k,\ \lvert I_2(i) - n\rvert > S$ and $\forall i \in \{n-S, n+S\},\ i \neq n,\ \lvert I_2(I_1^{-1}(i)) - I_2(I_1^{-1}(n))\rvert > S$

Table 3.8: The paired S condition

The paired S condition for interleaver I_2, position k and corresponding interleaved value n. For clarity, the interleaver edge tests have been omitted. Also, in the case of designing the interleaver, the condition is considered satisfied for the values of i for which $I_2(i)$ does not yet exist.

obtaining more balanced S values.

Experiments have shown that the first approach needs a much longer time than the second approach to produce similar results. In the following, the second approach has been used. Attempts to construct a second interleaver using an already designed S interleaver with different values of S_1 revealed that the value of S_2 is not dependent on the value of S_1 but rather characteristic to the fact that the second interleaver is designed under two constraints instead of one.

If the two interleavers are denoted by I_1 and I_2 the double S interleaver condition can be expressed as:

$$\begin{aligned} &\forall \quad i, j \in \{0, .., N-1\}, \quad i \neq j \\ &if \qquad |i-j| < S \\ &then \quad |I_2(i) - I_2(j)| > S \quad and \quad |I_2(I_1^{-1}(i)) - I_2(I_1^{-1}(j))| > S \end{aligned} \tag{3.12}$$

The first part of equation (3.12) ensures that two bits that are close together at the input of the first code are interleaved far away before they enter the third code. The second part ensures that two bits that are close together at the input of the second code are interleaved far away before they enter the third code. A more localised, algorithmic form of equation (3.12) is presented in table (3.8). Figure (3.18) shows the values of parameter S_2 obtained for different interleaver lengths. The algorithm used to determine this value is identical to the algorithm used to determine a single S interleaver, with the S condition in table (3.1) replaced by the paired S condition in table (3.8).

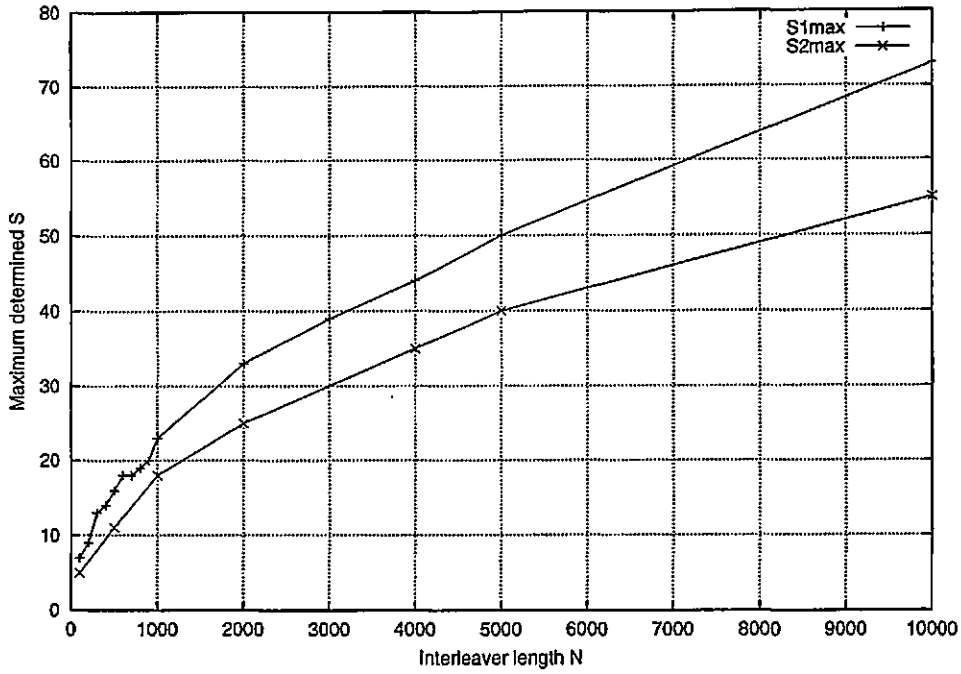


Figure 3.18: Maximum S_2 values for paired S interleavers
Maximum determined value for parameter S_2 for two interleavers in a 3PCCC for different block lengths and comparisons with the value of S for the first interleaver.

The values obtained for the parameter S for the second interleaver, denoted S_2 are presented in figure (3.18), together with the S value for the first interleaver (here denoted S_1), for increasing block sizes. It can be observed that S_2 is significantly smaller than S_1.

The S_1,S_2 paired interleavers guarantee a minimum OW_2 of

$$OW_2 \geq \left(\left\lfloor \frac{S_1}{T} \right\rfloor + \left\lfloor \frac{S_2}{T} \right\rfloor + 3 \right) w_T + 8 = \left(\left\lfloor \frac{S_1}{T} \right\rfloor + \left\lfloor \frac{S_2}{T} \right\rfloor \right) w_T + d_{free-eff} \qquad (3.13)$$

Note that the limit is dependent on both values of S_1 and S_2, and the worst possible case is when a short error event in one of the interleaved streams is mapped into an S_2-long error event into the other interleaved stream and to an S_1-long error event in the non-interleaver stream, as shown in figure 3.17(c). This replaces the worst possible case for the independent S interleavers, where a short error event in one of the interleaved streams is mapped into an S-long error event in the non-interleaved stream, but a short error event in the other interleaved stream. Clearly, the S pair improves on the independent S interleavers.

From turbo codes, it is known that $IW = 2+2$ "crossed" error events are one of the weaknesses of S interleavers. They do not fail to show up in the case of 3PCCC schemes, where their worst case is:

$$OW_{2+2} \geq 2(3w_T + 8) = 2d_{free-eff} \tag{3.14}$$

and is independent of S.

For a 3PCCC with $N = 500$, $RSC(5/7)$ the values are: $d_{free-eff} = 14$, $OW_2 \geq 24$ for independent S interleavers with $S = 15$, $OW_2 \geq 32$ for paired S interleavers with $S_1 = 15$, $S_2 = 12$, and $OW_{2+2} \geq 28$. The third type of error event should have similar probability of occurrence as the others, and is independent of the value of S so it somehow defeats the purpose of using higher values of S, similar to turbo codes. As opposed to turbo codes, the probability of occurrence for such mappings decreases with N this time, so it should be easier to obtain good interleavers. Since the OW_{2+2} event is common to both interleaver types, we might as well use the paired S interleavers rather than the unpaired interleavers, since their OW_2 is higher.

But what *is* the probability of these error events of generating a given minimum weight? Due to the periodicity of the $IW = 2$ error events, fast exhaustive search algorithms can be implemented to obtain an approximate answer. A number of 100,000 randomly chosen interleaver pairs were searched from each of the following groups: a) two $S = 0$ interleavers with $N = 500$ and $N = 2000$, b) two $S = 15$ interleavers ($N = 500$), c) two paired $S_1 = 15$, $S_2 = 12$ interleavers ($N = 500$). The $(OW_2)_{min}$ value was determined for each interleaver pair and the relative number of interleavers versus $(OW_2)_{min}$ is plotted in figure (3.19) for each category. It can be observed that there is a relatively high chance for the designed interleavers to reach their minimum possible distance 24 and 32, justifying the usage of paired S interleavers. Also, they increase the chance of obtaining higher $(OW_2)_{min}$ values. The first category reaches its maximum probability at $(OW_2)_{min} \approx 26$ for $N = 500$ and $(OW_2)_{min} \approx 38$ for $N = 2000$, the second at $(OW_2)_{min} \approx 30$ and the third at $(OW_2)_{min} \approx 36$. Notice that the longer interleaver has a higher most likely $(OW_2)_{min}$, but it also has a larger spread of the distribution.

The $IW = 2+2$ "crossed" error events have also been investigated. The exhaustive algorithm is slower in this case, so only 10,000 randomly chosen interleaver pairs have

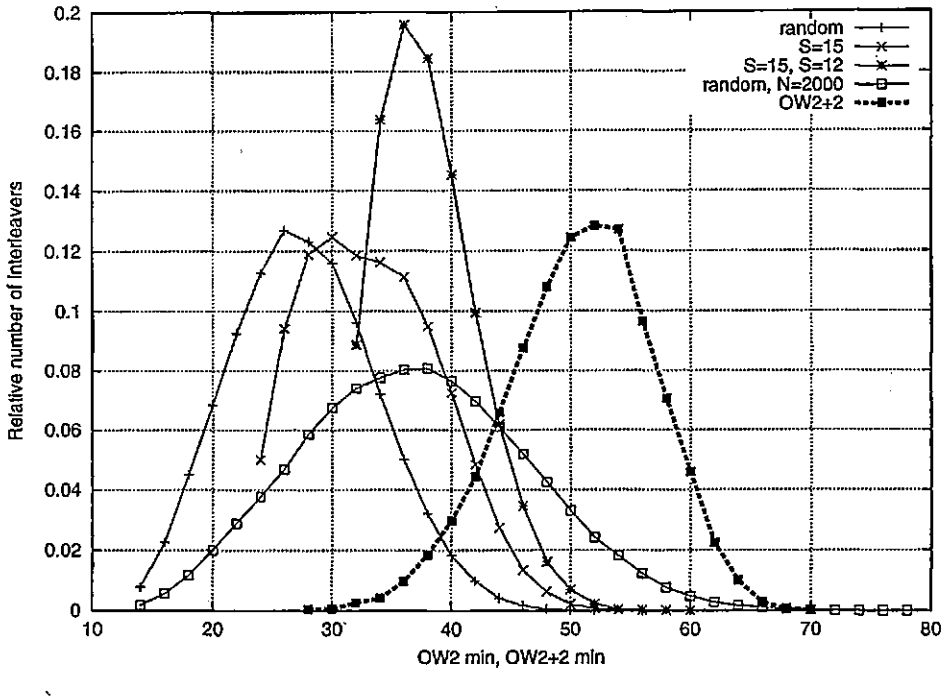
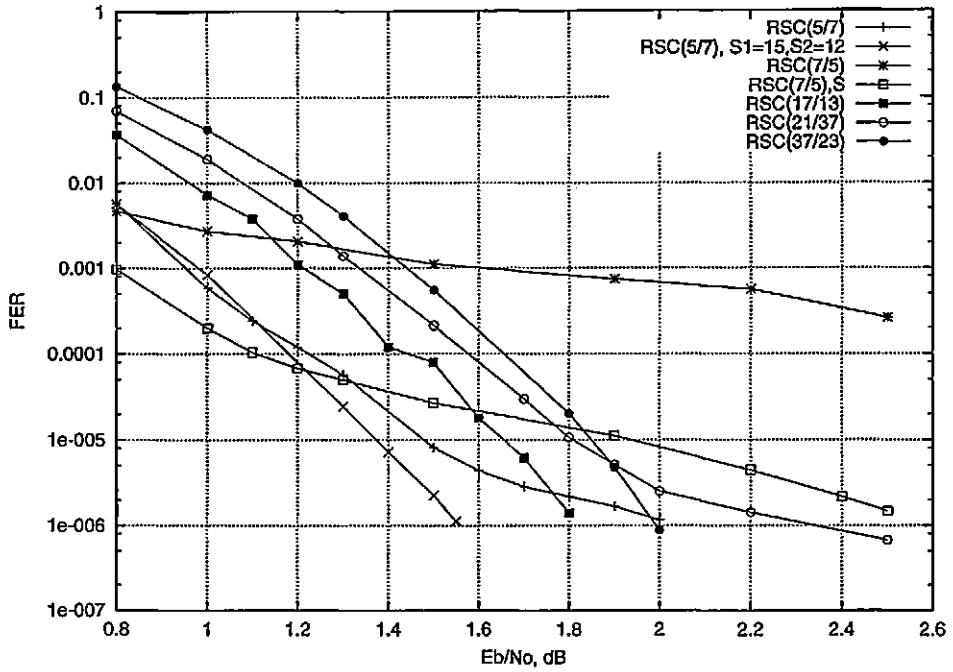


Figure 3.19: 3PCCC $(OW_2)_{min}$ probability distributions
Minimum OW_2, OW_{2+2} probability distributions for 3PCCC using $RSC(5/7)$ and random/S interleaver pair/double S interleavers. The block length is $N = 500$ if not specified.

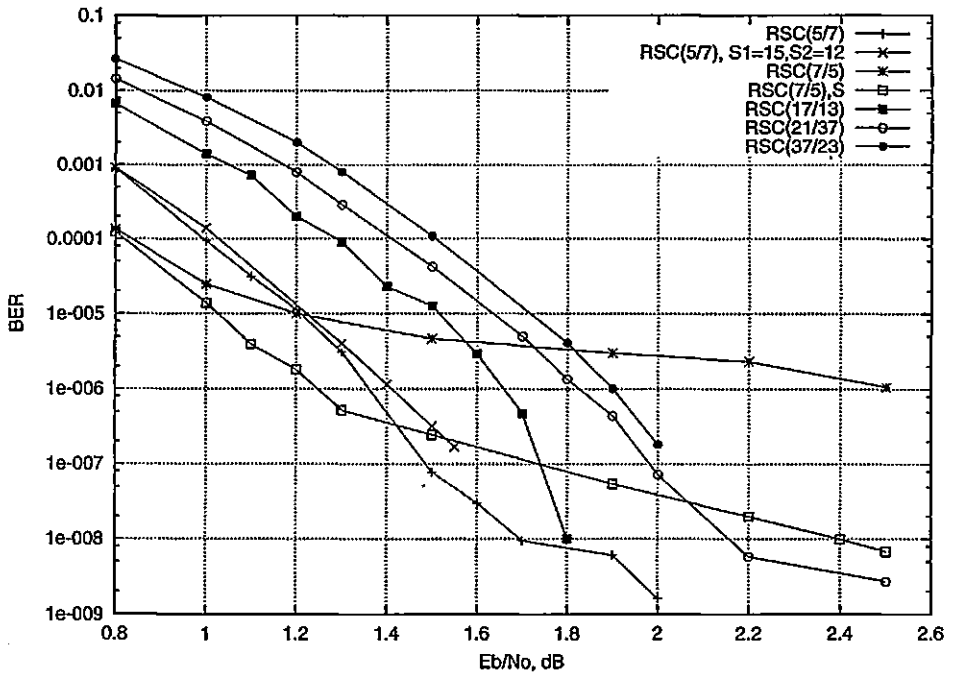
been tested for each group. The results for the random interleaver pair are also shown in figure (3.19). The $N = 2000$ case is missing, since the algorithm becomes very slow (one interleaver pair/minute on a 450 MHz machine). The maximum probability for these error events is obtained around $OW_{2+2} = 54$. They are independent of the interleaver type. It can be observed that the probability of their worst case ($OW_{2+2} = 28$) is actually much lower than that of the worst OW_2 case for any interleaver type. The maximum value of minimum code weight under both OW_2 and OW_{2+2} conditions obtained in this experiment was $OW = 54$.

3.4.2 Component code factor

Since the 3PCCC schemes are straight forward extensions of turbo codes, the component code design rules are similar. Optimal codes for turbo codes are also optimal for any MPCCC scheme. The performance of 3PCCC schemes has been simulated for different parameters, and the results are presented in figure 3.20 for $N = 500$ and figure 3.21 for $N = 2000$. It can be observed that 3PCCC schemes also present crossing



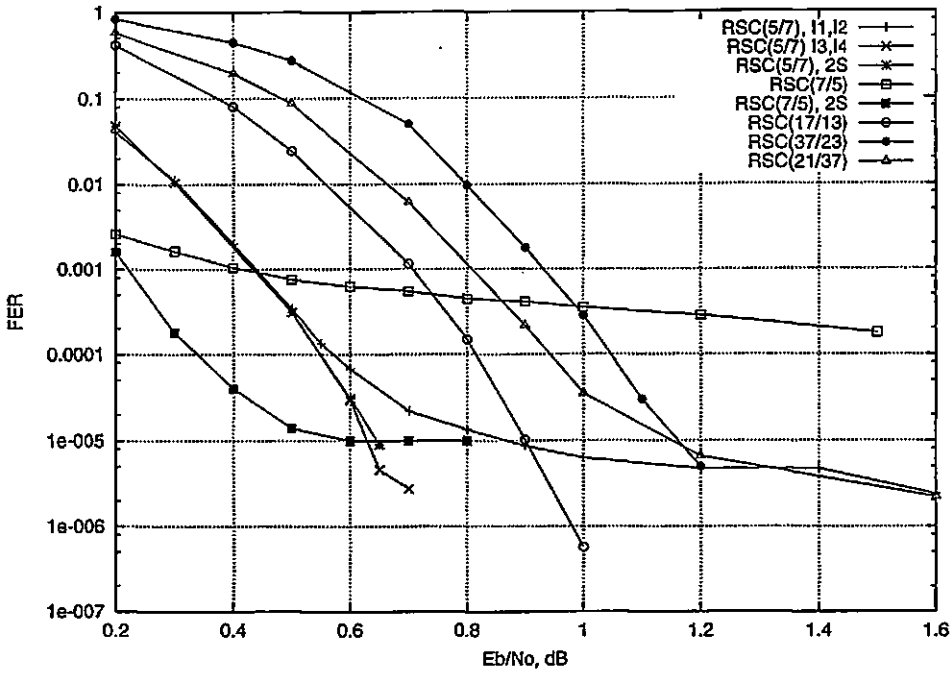
a)



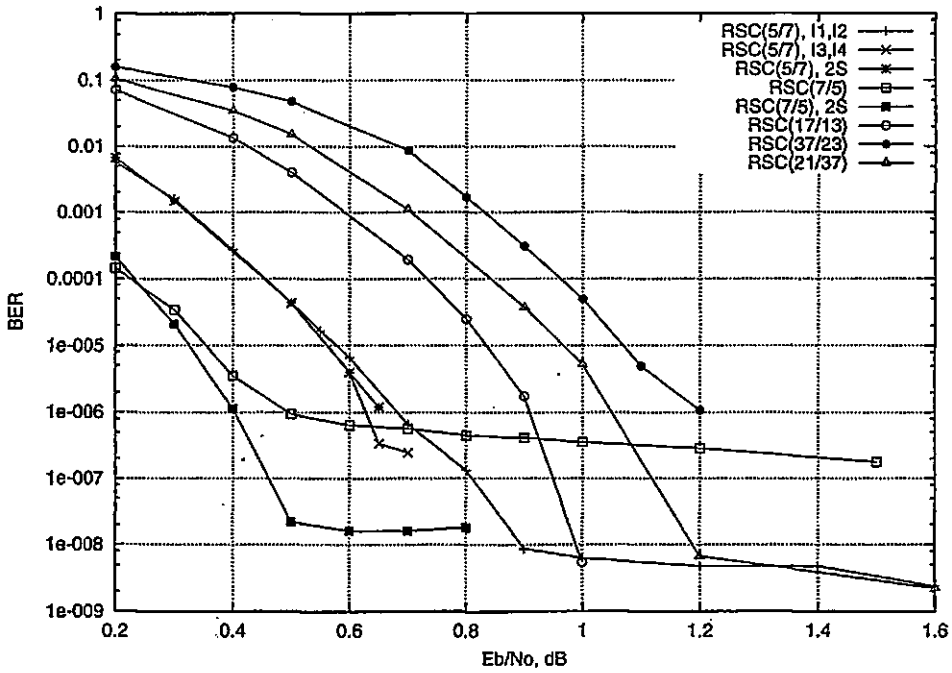
b)

Figure 3.20: 3PCCC performance for N=500

3PCCC with block length $N = 500$ and different component codes, a) FER curves and b) BER curves



a)



b)

Figure 3.21: 3PCCC performance for N=2000

3PCCC with block length $N = 2000$ and different component codes, a) FER curves and b) BER curves. Curves for two randomly chosen interleaver pairs (I_1, I_2 and I_3, I_4) are presented for the $RSC(5/7)$ 3PCCC.

points in performance curves. They also do happen quicker for FER curves than for BER curves, and suboptimal codes outperform optimal codes of the same memory at low E_b/N_o, especially in terms of BER. The crossing points also shift left with increasing N. As opposed to turbo codes, memory $M = 2$ codes can reach a FER $= 10^{-5}$ or lower before the crossing point with $M = 3$ codes, and the crossing points of higher memory (optimal) component codes are out of the simulation range. The performance at low E_b/N_o is also dominated by HIWHOW error events, with information weight increasing with code memory, and generally higher than for turbo codes. Error events with LIWLOW have also been observed at high E_b/N_o, especially for memory $M = 2$ codes, and in a much smaller number than for turbo codes. They dominate the performance of the non-optimal $RSC(7/5)$ code at a lower E_b/N_o than for any other code, resulting in this code having the best performance as the E_b/N_o is decreased. The non-optimal $M = 4$, $RSC(21/37)$ code remains better than the $M = 4$, $RSC(37/23)$ code at low E_b/N_o, but performs worse than the lower memory codes. The $RSC(7/5)$ codes have a rather flat performance curve, caused by a low d_{free}. This can be improved by using $S = 15$, $S = 12$ paired interleavers for $N = 500$ and $S = 33$, $S = 25$ paired interleavers for $N = 2000$, but they still show an error floor in the simulation range. Note that the usage of paired S interleaver produces a higher improvement in FER than in BER. They reduce the number of LIWLOW but do not reduce (and sometimes increase) the number of HIWHOW.

For the $RSC(5/7)$, $N = 500$ code, the observed d_{free} is varying in a large range. By observing the lowest code weight LIWLOW in iterative decoding simulations for 100 randomly chosen interleaver pairs it has been observed to be in the range $16 - 40$, with most of them under 30, producing a visible floor in the simulation range. It was observed that error events were usually $IW = 2$ error events. These observations are confirmed by the interleaver mapping search presented in figure (3.19), where it can be seen that error events with $OW_2 \approx 26$ are most likely to appear when the interleavers are chosen randomly. Using paired S interleavers guarantees a worst case of $OW_{2+2} = 32$, and higher weights with higher probability. Nevertheless, these still produce a visible error floor. The paired $S_1 = 15$, $S_2 = 12$ interleavers with $(OW_2)_{min}$=$(OW_{2+2})_{min}$=54 resulting from searching 10,000 interleaver pairs has been used to lower the error floor. The simulation has not shown any higher information weight LIWLOW. The situation

is significantly improved for $N = 2000$. Although the performance curve can still show an error floor, several trials are enough to produce an interleaver pair that does not, and using paired S interleavers is a straightforward way to avoid bad choices.

3PCCC schemes using $RSC(17/13)$ codes are much easier to choose. Although a first trial has shown an error event $(IW = 2, OW = 36)$ for $N = 500$, a second trial has shown no error floor in the simulation range. The reason can be readily found in figure (3.19). Although the figure refers to $RSC(5/7)$ codes, the shape of the probability distribution is the same for the $RSC(17/13)$ codes (this will be discussed in section 3.5.2). The difference is that the curves are situated at approximately double code weights, since their $w_T(17/13) = 4 = 2 * w_T(5/7)$. Thus the most probable $(OW_2)_{min}$ is approximately $(OW_2)_{min} \approx 50$ for randomly chosen interleavers for $N = 500$ and around $(OW_2)_{min} \approx 70$ for $N = 2000$. Several randomly chosen interleavers produced no observable error floor for the $N = 2000$, $RSC(17/13)$ code. This comes at a cost of several fractions of a dB, but for very low error rate requirements it can be the easiest way to obtain a good code.

The $RSC(37/23)$, $M = 4$ (optimal) code follows with a $0.1 - 0.2$dB gap, and its crossing point with the $RSC(17/13)$ code is outside the simulation range. No LIWLOW error events have been observed for this code.

The average number of iterations for the 3PCCC cases in the simulations presented in figures (3.20) and (3.21) are shown in figure (3.22).

3PCCC schemes have a strong interleaver factor. Memory $M = 3$ codes can be used to obtain very good performance. Also the performance of $M = 2$ codes is improved as compared to turbo codes. This scheme has worse performance at low E_b/N_o, especially for higher memory codes, starting from $M = 3$.

3.4.3 Increasing the number of codes

By increasing the number of codes (interleavers) in an MPCCC schemes, the interleaver factor can be further improved. Unfortunately, increasing the number of codes also leads to a further degradation in performance at low E_b/N_o. A comparative performance for several randomly chosen interleavers is presented in figure (3.23), for the $RSC(7/5)$ component code. It can be seen that, although the performance at higher E_b/N_o is improved, a degradation in performance is shown at low E_b/N_o, even

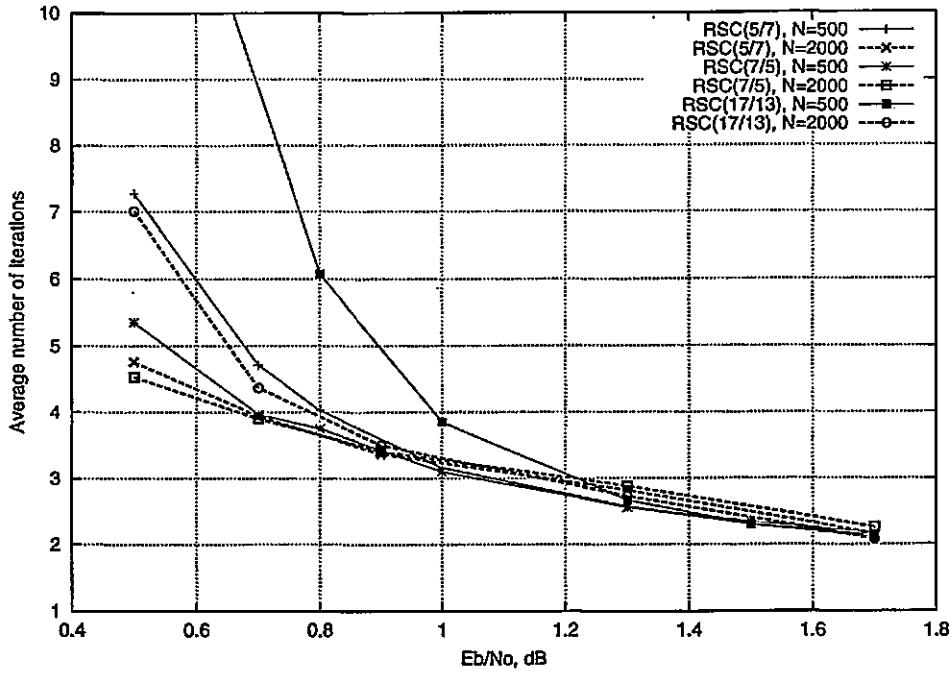


Figure 3.22: 3PCCC average number of iterations
Average number of iterations for different memory/block size 3PCCC. Iteration was stopped at zero errors.

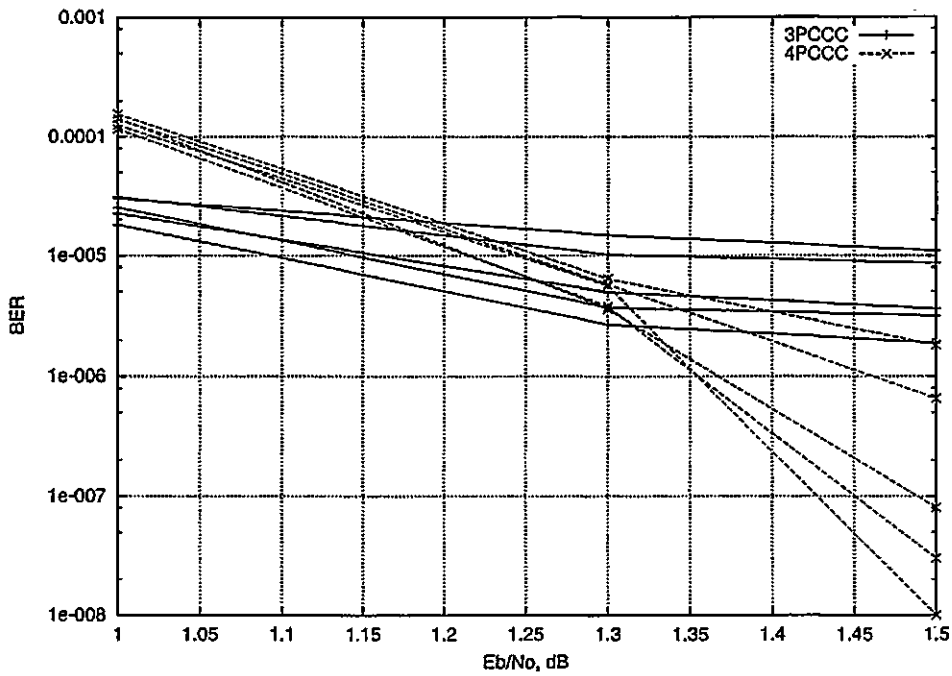


Figure 3.23: 3PCCC/4PCCC performance comparisons
Performance improvement in MPCCC scheme with increasing the number of component codes (interleavers). The performance is determined for the $RSC(7/5)$ non-optimal code, with a block length $N = 500$.

by the non-optimal $RSC(7/5)$ code. Similar to the 3PCCC scheme, a 4PCCC using this component code has the best performance at low E_b/N_o. Higher memory codes show further degradations. The degradation observed is due to HIWHOW error events that appear sooner for this scheme than for the 3PCCC. Also, the HIWHOW error blocks have higher information weight for the same component codes, as compared with 3PCCC. This could be explained by the fact that component codes work at lower equivalent signal to noise ratio, due to decreased code rate, and thus they will produce a higher number of errors with higher probability. Also, the complexity of the extrinsic information exchange is increased.

The conclusion is that it is better to use more carefully designed parameters in 3PCCC schemes than to try to improve performance by a further increase of the number of codes in structure. From this point of view, 3PCCC schemes are seen as a ML/iterative decoding compromise in the MPCCC group.

3.5 On the d_{free} of the MPCCC

For some practical applications, the block error rate (FER) is more important than the bit error rate (BER). Since FER is primarily limited (assuming an optimal decoder) by the d_{free} of the code, it is of interest how this value can be estimated for different MPCCC schemes. In this work it will be considered that the d_{free} of an MPCCC scheme is produced by an $IW = 2$ error event. This is justified by the fact that $IW = 2$ error events are the most likely error events. In this case $d_{free} = (OW_2)_{min}$.

Due to the periodicity of the $IW = 2$ error events for RSC codes, illustrated in figure (3.9) for two particular codes, the code weight can be expressed as a function of the number of periods of the error event. If m is the number of codes in the MPCCC and $n = n_1 + n_2 + ... + n_m$ is the total number of periods of an $IW = 2$ error event of the MPCCC,

$$OW_2 = 2 + \sum_{k=1}^{m}(n_k w_T + w_e) = n w_T + 2m + 2 \tag{3.15}$$

where w_T is the parity weight corresponding to one period and $w_e = 2$ is the edge parity weight, as discussed in the previous sections. The component codes are considered

identical.

3.5.1 Dependence on interleaver length

Figure (3.24) presents the relative number of interleavers producing at least one error event having a given OW_2 for turbo codes and 3PCCC. The component code used was $RSC(5/7)$. The values in this figure were obtain by computer search: a number of $10,000$ interleavers were randomly generated (see Annex A) for each scheme and block length and searched for $IW = 2$ error events using a fast algorithm that takes advantage of the periodicity of the component codes. The number of interleavers having a given OW_2 was counted and divided by $10,000$ to obtain the relative number of interleavers, which can be identified with *the probability of a scheme to produce a given OW_2 when the interleaver is chosen at random.* In figure (3.24), the increase in probability with OW_2 for low OW_2 values can be explained by the multiplicity of error event associations that produces a given OW_2. For example, for turbo codes, the minimum possible $OW_2 = d_{free-eff}$ has $n = 2$ periods and can only be produced by the association of error events of the component codes having $n_1 = n_2 = 1$ period. The next OW_2 has $n = 3$ periods and can be produced in two ways: $n = 3 = (n_1 = 1) + (n_2 = 2) = (n_1 = 2) + (n_2 = 1)$ and so on. Generally, the multiplicity of error events producing a given OW_2 having n periods is $\binom{n-1}{m-1}$, which is just the number of ways n periods can be split between the error events produced by the m codes in the MPCCC structure. The decrease in probability for large OW_2 values is due to the length of the error events that produce these values, which becomes comparable to the length of the interleaver. This is why the decrease happens for higher OW_2 if the interleaver length is increased.

The probability of a given $(OW_2)_{min}$ can be computed in a similar way. Figure (3.25) presents the relative number of interleavers producing a given $(OW_2)_{min}$ for (a) 2PCCC (turbo codes), (b) 3PCCC and (c) 4PCCC for different interleaver lengths. To obtain this result, a number of 100,000 interleavers were searched for each scheme and interleaver length. The component code was $RSC(5/7)$.

Since $OW_2 = d_{free-eff}$ is the minimum possible OW_2 value for any interleaver, it determines $(OW_2)_{min}$ every time it is produced by the interleaver(s). The next higher OW_2, although it has a higher probability, will determine $(OW_2)_{min}$ only when $d_{free-eff}$

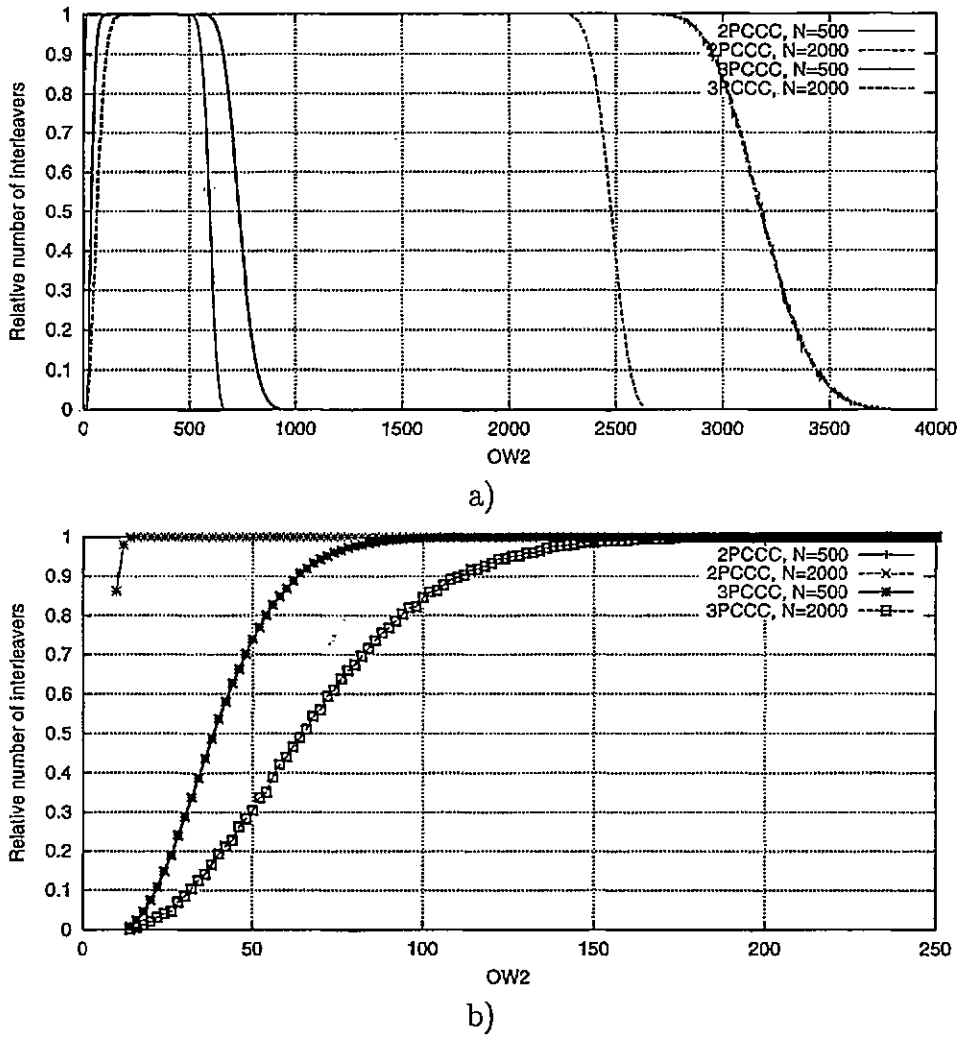


Figure 3.24: OW_2 distribution
a) Relative number of interleavers producing a given OW_2 for different block lengths for turbo codes (2PCCC) and 3PCCC, b) zoomed version of a). The curves are not continuous but take values at the marked points. Graph a) has no marking points for clarity.

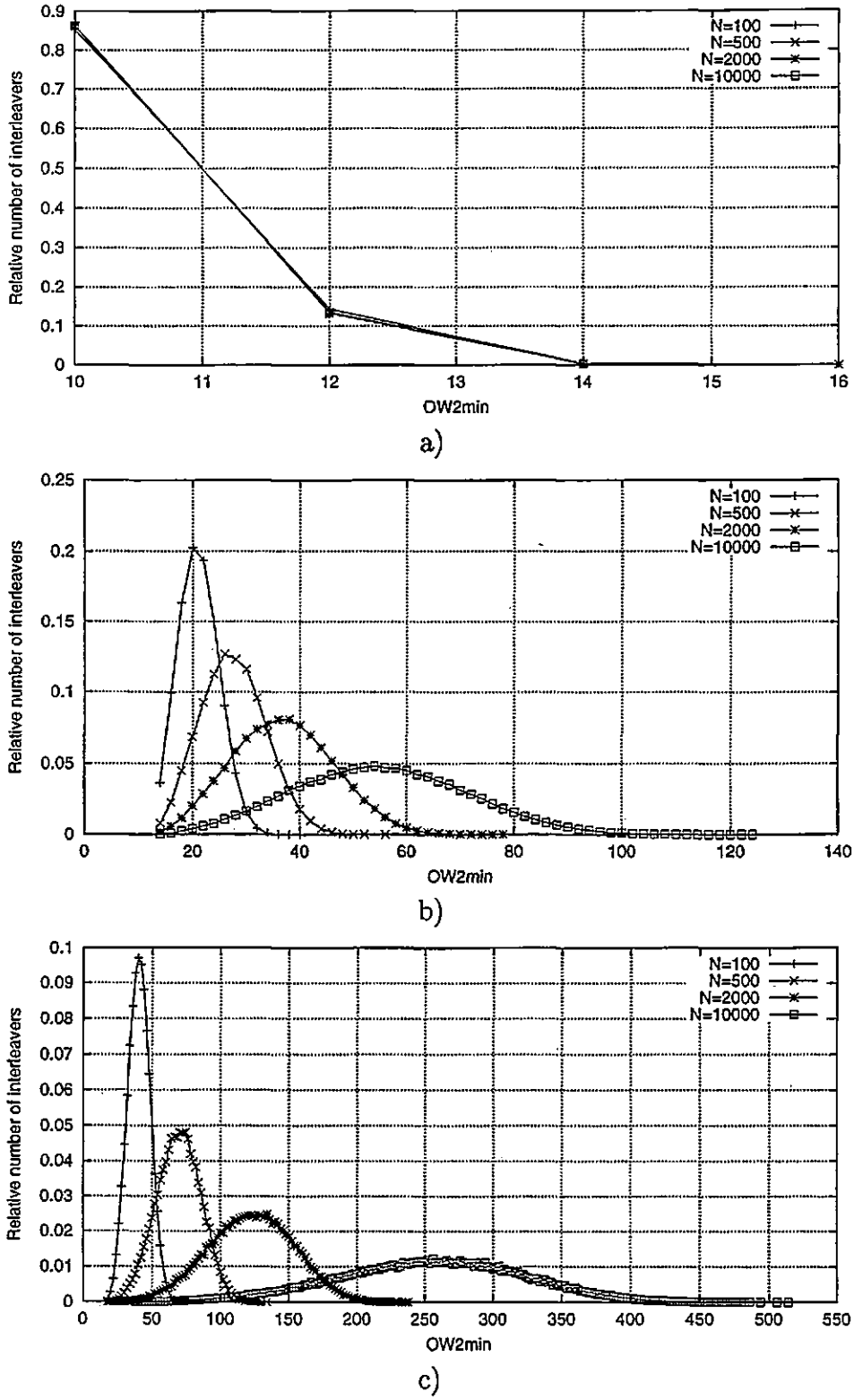


Figure 3.25: Dependence of $(OW_2)_{min}$ on block length
$(OW_2)_{min}$ probabilities for an a) 2PCCC scheme (turbo code) with $R = 1/3$, b) 3PCCC with $R = 1/4$ and c) 4PCCC with $R = 1/5$. The interleaver is chosen at random for each different block length. The component code is $RSC(5/7)$. The curves are not continuous but take values at the marked points.

is not produced and so on. Thus, although higher OW_2 values have higher probabilities, they determine $(OW_2)_{min}$ only if all the lower OW_2 values are not produced. This effect will be referred to as the cumulated masking effect of the lower OW_2 values.

For turbo codes, the $OW_2 = d_{free-eff}$ has a high probability, almost independent of N, giving higher OW_2 values little chance to determine $(OW_2)_{min}$. This is why the d_{free} of turbo codes is usually $d_{free-eff}$, and the performance of turbo codes is so close to the average performance. If the $d_{free-eff}$ is rejected by using an S interleaver, the lowest possible OW_2 for the given value of S will produce $(OW_2)_{min}$ with even higher probability, as shown in figure (3.8).

For a 3PCCC scheme, the probability of $d_{free-eff}$ is much lower, decreasing with the interleaver length N. In this case, higher OW_2 values have a chance to produce $(OW_2)_{min}$ before the cumulated masking effect compensates for their multiplicity, and this chance increases with N. This justifies the existence of a maximum in the probability curves for the 3PCCC schemes, and the shift of this maximum towards higher OW_2 values as N is increased. It also explains the larger spread of the distributions as N is increased. Unfortunately, this means that as the interleaver length is increased, the $(OW_2)_{min}$ can be predicted with decreasing accuracy, until it gets to the point where it could be any value in a large range.

The 4PCCC scheme follows the same pattern, but it has even lower $d_{free-eff}$ probability, decreasing more rapidly with N. In this case the maximum probability can be obtained for higher OW_2 values.

In comparing the d_{free} values produced by each scheme, one should take into account the different code rates. In the following comparisons, d_{free} is identified to $(OW_2)_{min}$. All the comparisons are done for the $RSC(5/7)$ component code. The d_{free} produced with the highest probability by the turbo code is $d_{free} = 10$, which gives approximately the same FER for a rate $R = 1/3$ turbo code as $d_{free} = \frac{4}{3} * 10 \approx 14$ for a rate $R = 1/4$, 3PCCC scheme. It can be observed that the 3PCCC scheme produces a much higher most likely value $d_{free} \approx 26$, and thus behaves much better in terms of FER. A most likely value of $d_{free} \approx 26$ for an $N = 500$, 3PCCC scheme is equivalent to a $d_{free} = \frac{5}{4} * 26 \approx 34$ for a $R = 1/5$, 4PCCC scheme. For $N = 500$, the most probable value is $d_{free} \approx 70$ for a 4PCCC scheme, and thus this scheme improves on FER as compared with 3PCCC.

The experiments presented can be used to verify the probability that single, given error events of the component codes are associated by the interleaver(s) for an MPCCC scheme, by examining the probability of $d_{free-eff}$. This is because $d_{free-eff}$ is generated by only one single error event combination. As shown in figure (3.25)(a) where $d_{free-eff} = 10$ for the $RSC(5/7)$ component code, in the case of turbo codes, $P(d_{free-eff}, N) \approx 0.86$ for any N. In the case of 3PCCC, the probabilities should decrease as $1/N$. As shown in figure (3.25)(b) where $d_{free-eff} = 14$, $P(d_{free-eff}, N = 100) = 3644{*}10^{-5}$ and $P(d_{free-eff}, N = 500) = 794{*}10^{-5}$ differ by a factor of 4.6 which is close to the expected 5, $P(d_{free-eff}, N = 500) = 794 * 10^{-5}$ and $P(d_{free-eff}, 2000) = 179 * 10^{-5}$ by a factor of 4.4 which is also close to the expected 4. For the 3PCCC, the probabilities should decrease as $1/N^2$. As shown in figure (3.25)(c) where $d_{free-eff} = 18$, for $N = 100$, the probability was $P(d_{free-eff}, 100) = 67 * 10^{-5}$ and for $N = 500$, $P(d_{free-eff}, 500) = 3{*}10^{-5}$. The factor is around 22, close to the expected $25 = 5^2$. No $d_{free-eff}$ error events were observed in the experiments for $N = 2000$ and $N = 10000$.

3.5.2 Dependence on code memory

The probability of an MPCCC scheme to have a given $(OW_2)_{min}$ for component codes with increasing memory is shown in figure (3.26) for a) turbo codes (2PCCC) and b) 3PCCC for a block length $N = 500$.

Increasing code memory and using primitive feedback polynomials produces an increase in the $(OW_2)_{min}$ values, shifting the curves to the right. Increasing code memory is a way to obtain higher $(OW_2)_{min}$ values for turbo codes, as opposed to increasing interleaver length. Also, increasing code memory is a better way to obtain a higher $(OW_2)_{min}$ for the 3PCCC scheme than increasing interleaver length, since the maximum probability does not decrease significantly. The larger spread of the distribution is due to the discontinuity of the curves, and to the fact that the points for higher memory codes are situated at longer distances from each other (w_T is increased, see table (3.7)).

Since the only difference (from the point of view of the interleaver(s)) between the error events of an MPCCC scheme having the same number n of periods for different component codes is their length, as long as this length is much shorter than the

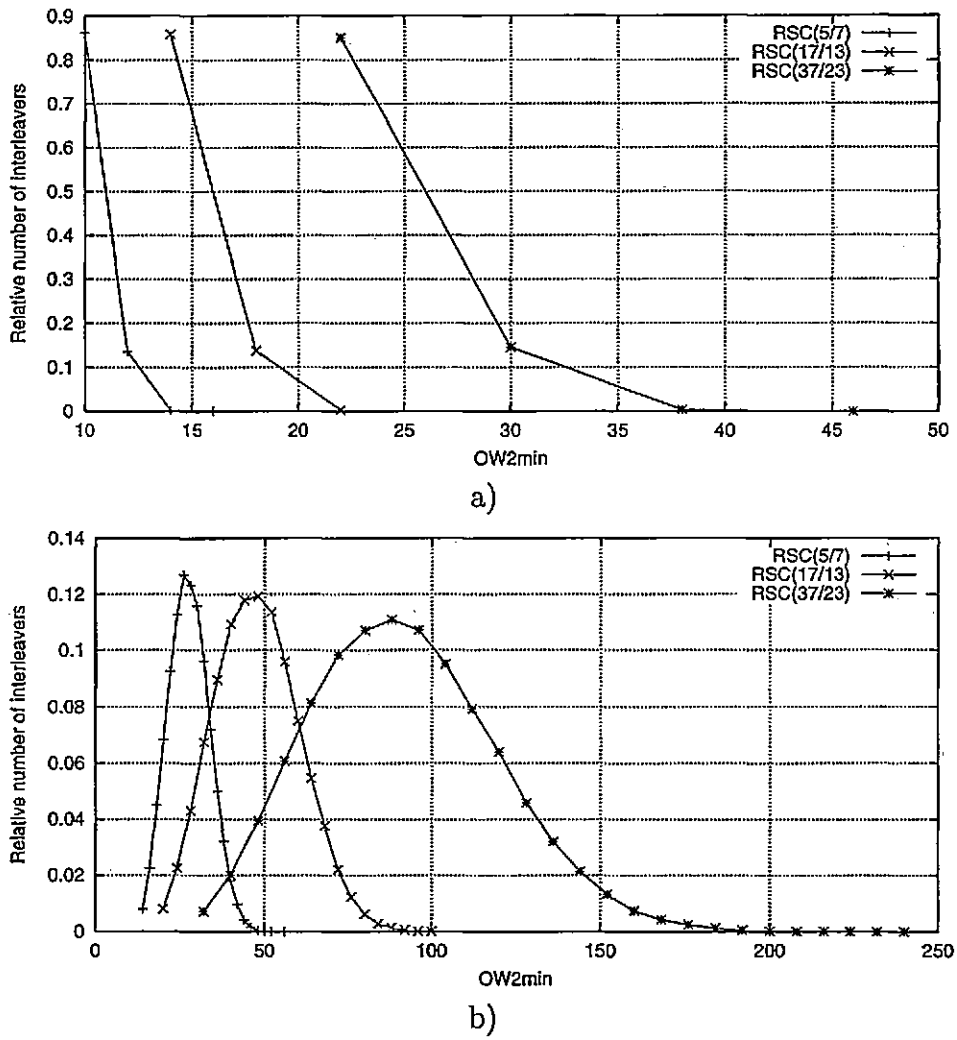


Figure 3.26: Dependence of $(OW_2)_{min}$ on component code $(OW_2)_{min}$ probabilities for an a) 2PCCC scheme (turbo code) with $R = 1/3$, b) 3PCCC with $R = 1/4$. The block length is $N = 500$. The curves are not continuous but take values at the marked points.

interleaver length, they should have (almost) the same probability.

$$P\{(OW_2)_{min} = nw_T^1 + m * 2 + 2\} \approx P\{(OW_2)_{min} = nw_T^2 + m * 2 + 2\} \quad (3.16)$$

where w_T^1 and w_T^2 are w_T values for different, low memory codes. In figure 3.26(a) it can be seen that this relationship holds well for all the $(OW_2)_{min}$ values presented. In the case of the 3PCCC scheme, shown in figure 3.26(b), the relationship holds well up to the $(OW_2)_{min}$ value with maximum probability and then the difference starts increasing more significantly with $(OW_2)_{min}$.

3.6 The serial concatenation

The SCCC scheme has the strongest interleaver factor. In the following experiments, it is very unlikely to observe *any* LIWLOW error event, since the dependence on the interleaver length starts at least with $1/N^2$.

3.6.1 Interleaver factor

The effect of using an S interleaver for SCCC in order to improve its optimal performance is difficult to determine by simulation due to the following reason: the d_{free} of simple SCCC scheme is relatively large (and thus very few LIWLOW have been observed) and the HIWHOW error events dominate their performance over all the simulation observation window.

Still, using an S interleaver would be expected to improve the performance of the codes since the S interleaver tends to transform short error events into long ones. Consider a serial concatenation with component codes $RSC(5/7)$. One of the most likely error events results from associating the $(IW = 2, OW = 6)$ of the outer code with three $(IW = 2, OW = 6)$ error events of the inner code, resulting in a d_{free} as low as $d_{free} = 6 + 6 + 6 = 18$. Also, associating the $(IW = 3, OW = 5)$ error event of the outer code with an $(IW = 2, OW = 6)$ and an $(IW = 3, OW = 5)$ error event of the inner code will produce an even lower $d_{free} = 11$. For block length $N = 500$, the interleaver length is $N_I = 1000$ and $S = 21$ can be used. The $(IW = 2, OW = 6)$ error event is shorter than 21, and thus all the 6 bits of 1 from the outer code will be

interleaved further away than 21 bits. This means that each of the 3 error events will contain more than $n = 7$, $T = 3$ periods and thus each of them will cumulate a code weight of $(7 + 1) * (w_T = 2) + 6 = 22$ resulting in a d_{free} higher than 66. In the case of the $(IW = 3, OW = 5)$ error event of the outer code the S interleaver increases the length of the $(IW = 2, OW = 6)$ and $(IW = 3, OW = 5)$ error events of the inner code, also producing a d_{free} higher than 66.

Unfortunately, "crossed" error events are possible here as well. Two $(IW = 3, OW = 5)$ error events of the $RSC(5/7)$ outer code can be associated with 5 $(IW = 2, OW = 6)$ error events of the $RSC(5/7)$ inner code, resulting in $d_{free} = 30$, independent of S.

All these mappings happen with vanishing probability, reducing to zero quicker or at least as fast as $1/N^{\left\lfloor \frac{d^o_{free}+1}{2} \right\rfloor - 1} = 1/N^2$ for the $RSC(5/7)$ code. The S interleavers can be used to avoid unlikely, bad interleaver choices.

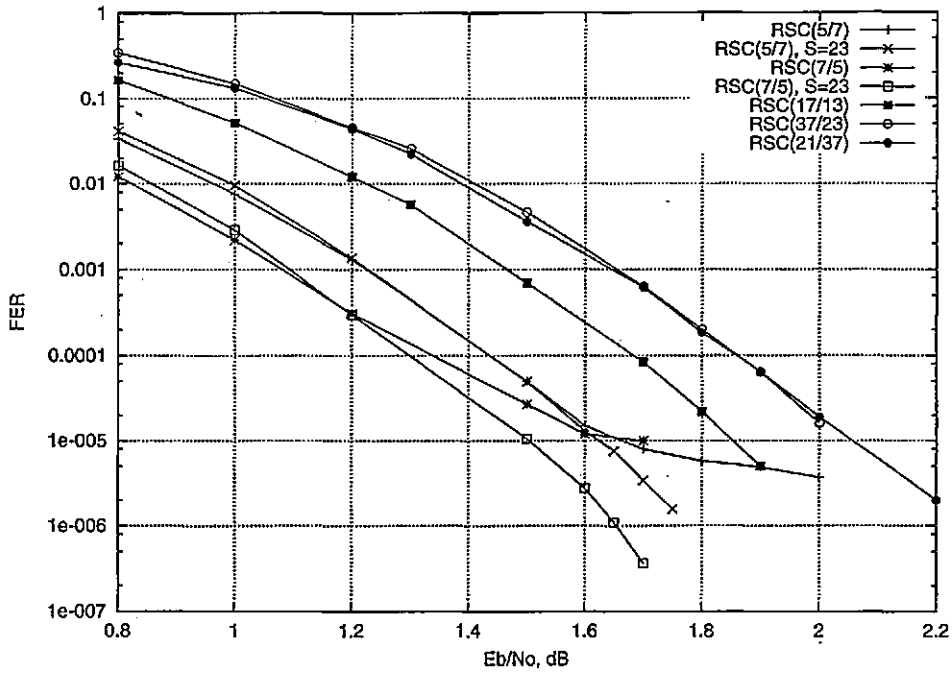
Simulations generally show a slight performance degradation for schemes using S interleavers at low E_b/N_o. This can be observed in figures (3.27) and (3.28).

3.6.2 Component code factor

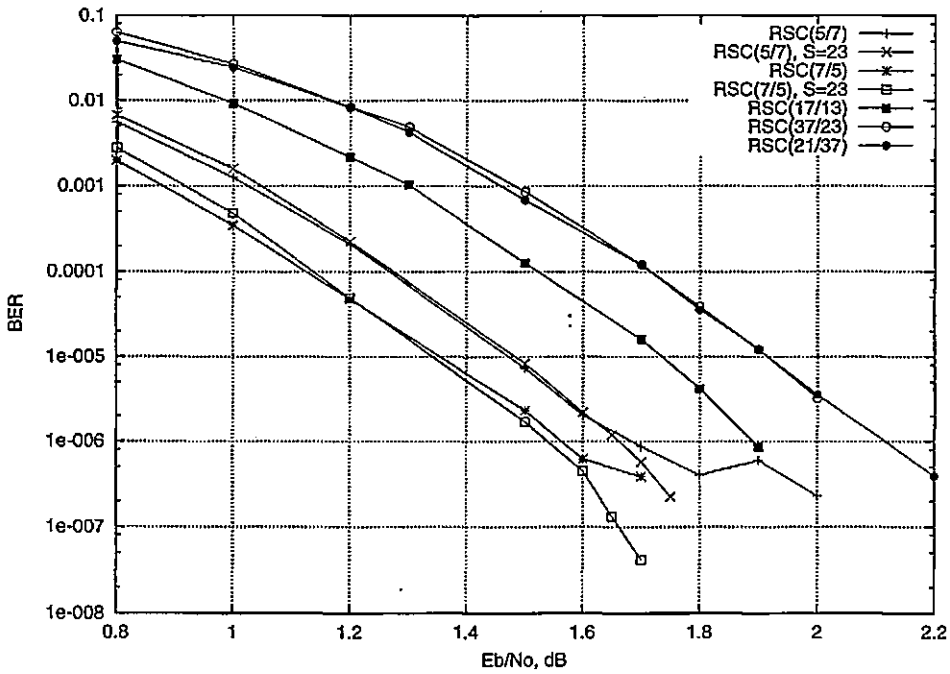
Simulation results for the iterative decoding of SCCC schemes using several component code combinations are presented in figure (3.27) for block length of $N = 500$ (interleaver length $N_I = 1000$) and figure (3.28) for block length $N = 2000$ (interleaver length $N_I = 4000$). The SCCC performance curves have few intersections (if at all) in the simulation observation window. This is because their performance is dominated by HIWHOW error events all over the simulation range.

As for the 3PCCC scheme, the $RSC(7/5)$ component code gives the best performance at low E_b/N_o from all the codes simulated. The difference is that their performance is now also dominated by HIWHOW error events, and they show only a very small number of LIWLOW error events at high E_b/N_o values. The observed d_{free} is usually in the range $30 - 40$. The $RSC(7/5)$ is a non-optimal inner code for the SCCC scheme, which justifies the LIWLOW error events observed.

The $RSC(5/7)$ is an optimal inner code for the SCCC scheme. Simulation has produced a few LIWLOW error events with code weight as high as 70 for a concatenation using an $RSC(5/7)$ code both for the inner and outer code. The number of HIWHOW

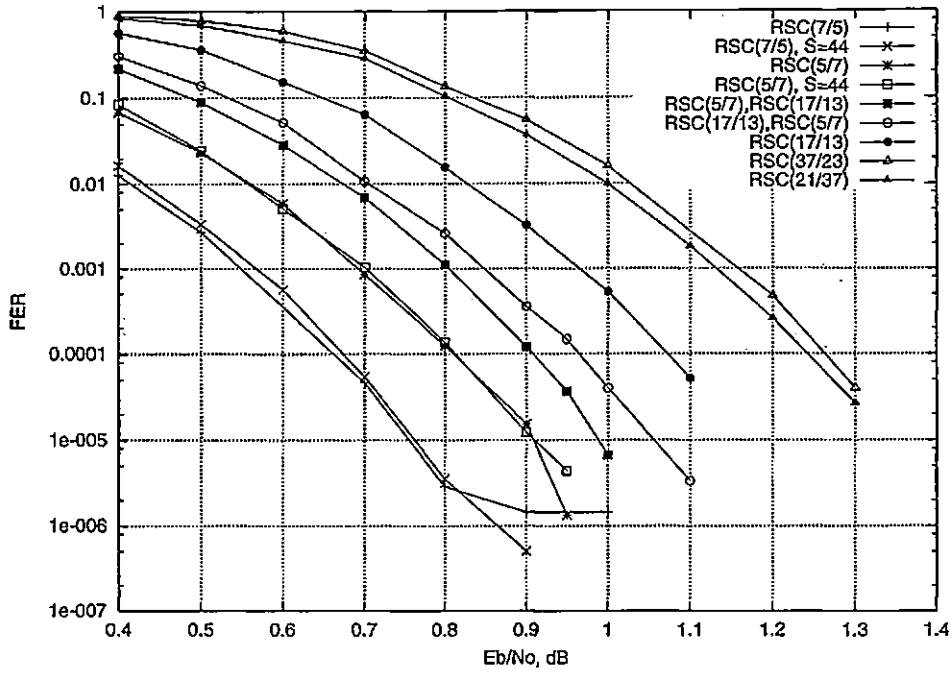


a)

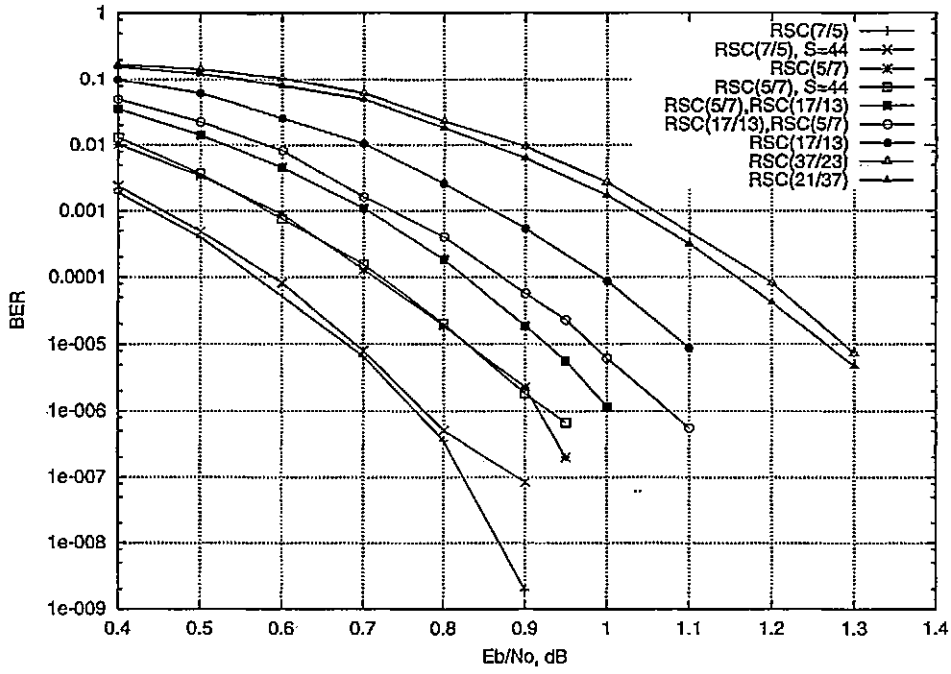


b)

Figure 3.27: SCCC performance for N=500
SCCC with block length $N = 500$ ($N_I = 1000$) and different component codes, a) FER curves and b) BER curves



a)



b)

Figure 3.28: SCCC performance for N=2000
SCCC with block length $N = 2000$ ($N_I = 4000$) and different component codes, a) FER curves and b) BER curves

and their information weight is higher, producing a loss of about 0.1dB as compared to the non-optimal code.

A problem of the SCCC scheme are the limit cycle error events, non-convergent error events with information/code weight varying quasi-periodically in a wide range. The information/code weight associations for these error events cover all the block types, and they are usually observed as LIWHOW blocks. They are visible since they are persistent with increasing E_b/N_o where the HIWHOW error events have a reduced number and produce an oscillating "error floor", different from the d_{free} error floor observed for turbo codes. Since the SCCC schemes have a high d_{free}, it is frustrating that their performance does not follow it. The limit cycles are usually caused by the $(IW = 3, OW = 5)$ error event for the $RSC(5/7)$ code when the 5 code bits of one are mapped into two $IW = 2$ short error events and a single bit, far away from the others. Their persistence at higher E_b/N_o could be explained by the fact that the inner code produces an additional 1 to close a short error event for the wandering bit, which is totally rejected by the outer code, resulting in oscillation. Since the inner code error event has low weight, it is unlikely that the inner code will give it up too quickly with increasing E_b/N_o, which results in the limit cycle error event persistence with E_b/N_o. The limit cycle error events problem reduces with interleaver length, probably with a reducing relative number of the above mappings. These mappings have been also observed for the $RSC(7/5)$ code, but not for higher memory codes. Since they are caused by short error event mappings, they are likely to be avoided when using an S interleaver which will not allow them (provided the outer code error event is short, and the $(IW = 3, OW = 5)$ error event is). This is confirmed by simulation in figure (3.27), where SCCC using an $S = 21$ interleaver do not show the limit cycle error floor, both for schemes using the $RSC(5/7)$ and the $RSC(7/5)$ component code. The fact that these error events have been observed for the 3PCCC scheme as well but corrected by increasing data representation precision suggests that they are, in essence, numerical and not mathematically non-convergent. If infinite precision or different SISO algorithms were used, they could be corrected. The double floating point precision used for the 3PCCC is not enough for the SCCC scheme. Note that these error events appear when each code produces a very likely error event which is mapped into a very unlikely error event for the other code.

Increasing the memory of the component codes produces the usual effect: about $0.1 - 0.2$dB degradation in performance for each increase in the memory of both component codes. The absence of crossing points is due to the absence of LIWLOW error events. They are expected to happen at higher $E_b/N_o$and lower error rates which cannot be simulated. The performance of the non-optimal $RSC(21/37)$ code is better than that of the optimal $RSC(37/23)$, due to a reduced number and information weight of the HIWHOW error events. This is more visible for the $N = 2000$ curves. Note that the non-optimal term refers now only to the inner code.

Asymmetric codes were also simulated and presented in figure (3.28) for block length $N = 2000$. The scheme using $RSC(17/13)$ as outer code and $RSC(5/7)$ as inner code is always better than the scheme that uses $RSC(5/7)$ as outer code and $RSC(17/13)$ as inner code. Both curves are worse than the performance of symmetric SCCC using $RSC(5/7)$ and better than that of the SCCC using $RSC(17/13)$. All observed error events were HIWHOW. Increasing the memory produces the usual degradation at low E_b/N_o (for the SCCC schemes, "low" E_b/N_o means the whole simulation range) but an increase in memory for the inner code produces a bigger degradation than an increase in memory for the outer code.

Generally, it is rather difficult to test the improvement obtained by using designed code parameters on SCCC schemes, since they usually produce an improvement outside the simulation range, and a degradation inside the simulation range. Of course, the performance can be decreased by reducing block length so that the error floors are higher (and thus more accessible), but probabilistic arguments are generally valid for long block lengths. Such an attempt resulted in a high number of error events being observed for different code parameters, difficult to separate in distinct classes.

The average number of iterations for the SCCC cases in the simulations presented in figures (3.27) and (3.28) are shown in figure (3.29).

3.7 Comparisons

The MPCCC and SCCC schemes have been introduced as an attempt to improve the performance of turbo codes. It was shown, based on an assumption of optimal decoding, that they decrease the error rate at the same E_b/N_o for a given block length.

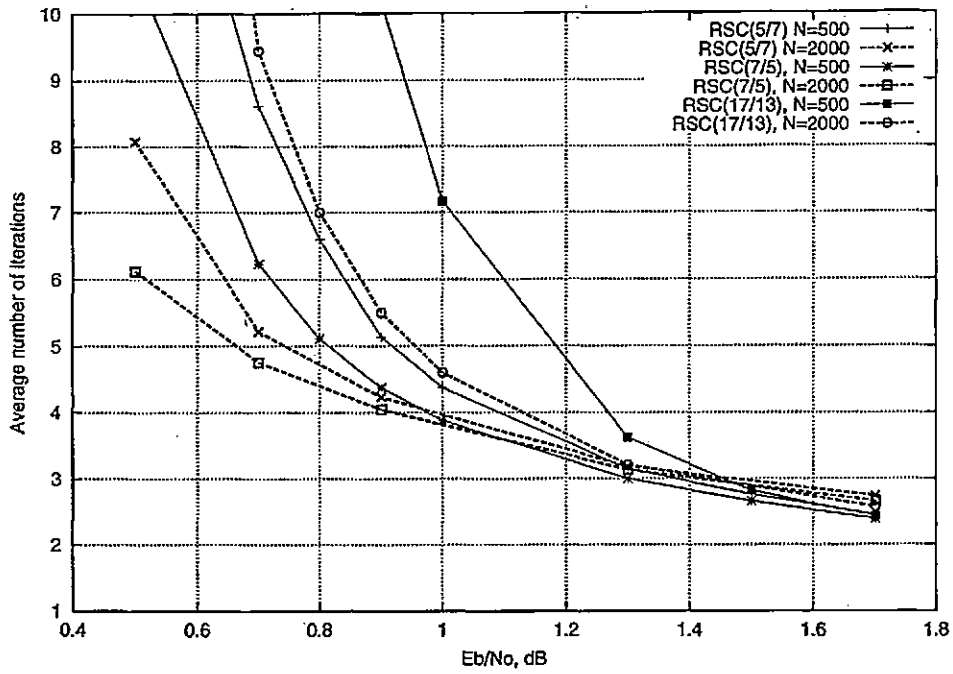
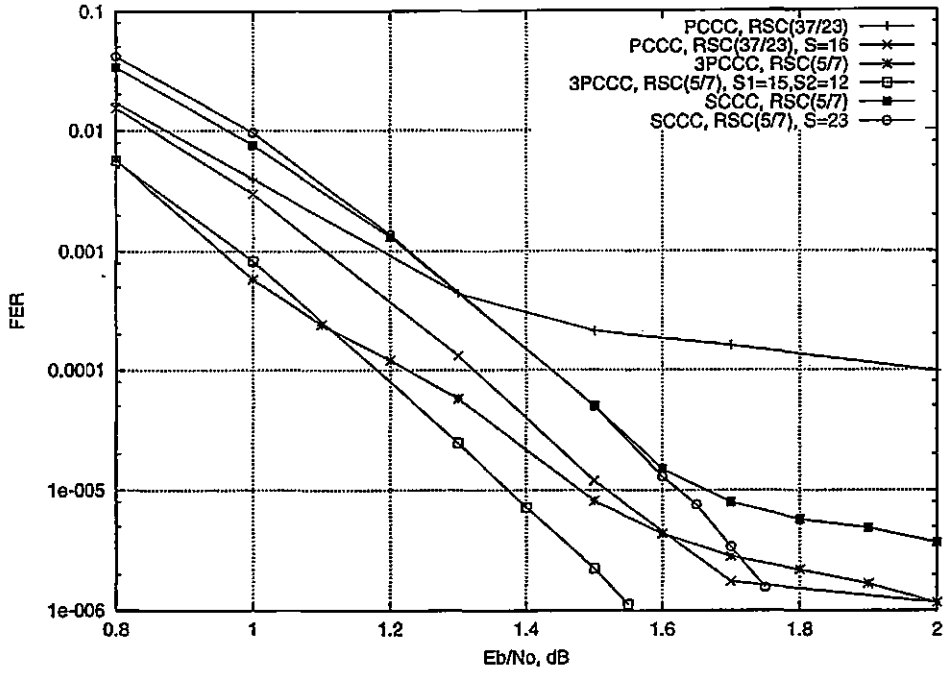


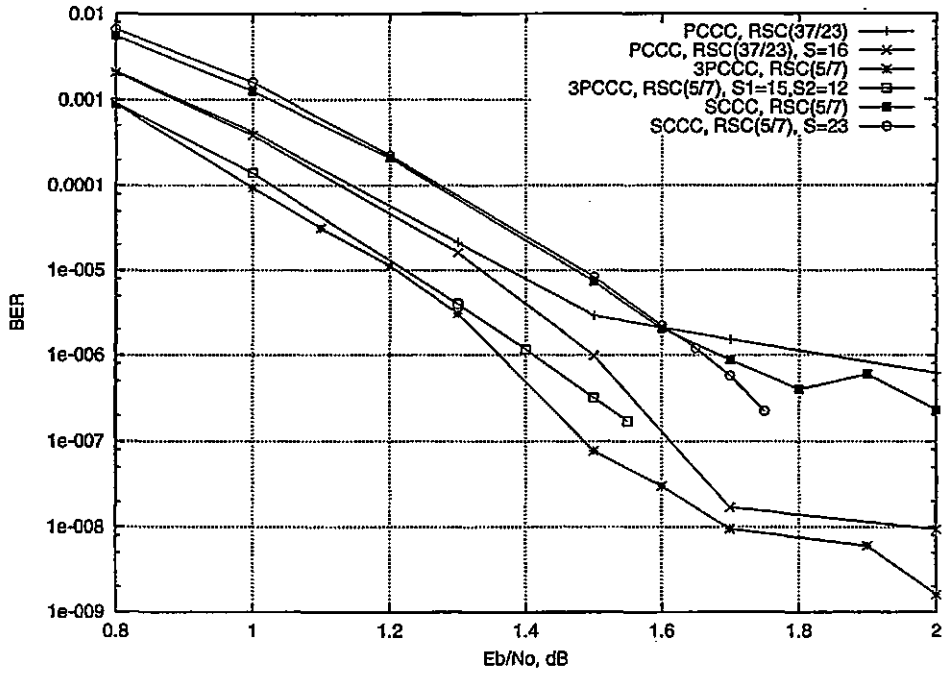
Figure 3.29: SCCC average number of iterations
Average iterations for different memory/block size SCCC. Iteration was stopped at zero errors

A comparison of the three schemes using randomly chosen interleavers as well as S interleavers is presented in figure (3.30) for block length $N = 500$ and in figure (3.31) for $N = 2000$. Since the compared schemes have a different number of codes, and also use component codes of different memory, their decoding complexity is presented in figure (3.32). The definition of "complexity" takes into account the number of codes, the memory of the codes, the block length of the component codes and the average number of iterations, obtained as described in the preceding sections. For turbo codes, the complexity is $2 * M * avgit$, where M is the code memory, and *avgit* is the average number of iterations. The factor 2 appears because there are two decoders. For the 3PCCC schemes, the only difference is that there are 3 decoders, and thus the complexity is $3 * M * avgit$. For the SCCC scheme, the inner decoder has double block length, and thus it also has a factor of 3: $3 * M * avgit$. Although turbo codes have an advantage for the same memory, they lose it due to the need to use $M = 4$ code as the best compromise code, whereas the other schemes use $M = 2$ codes.

The conclusion is that 3PCCC schemes are the best both in terms of complexity and error rate for all E_b/N_o values that can be simulated.

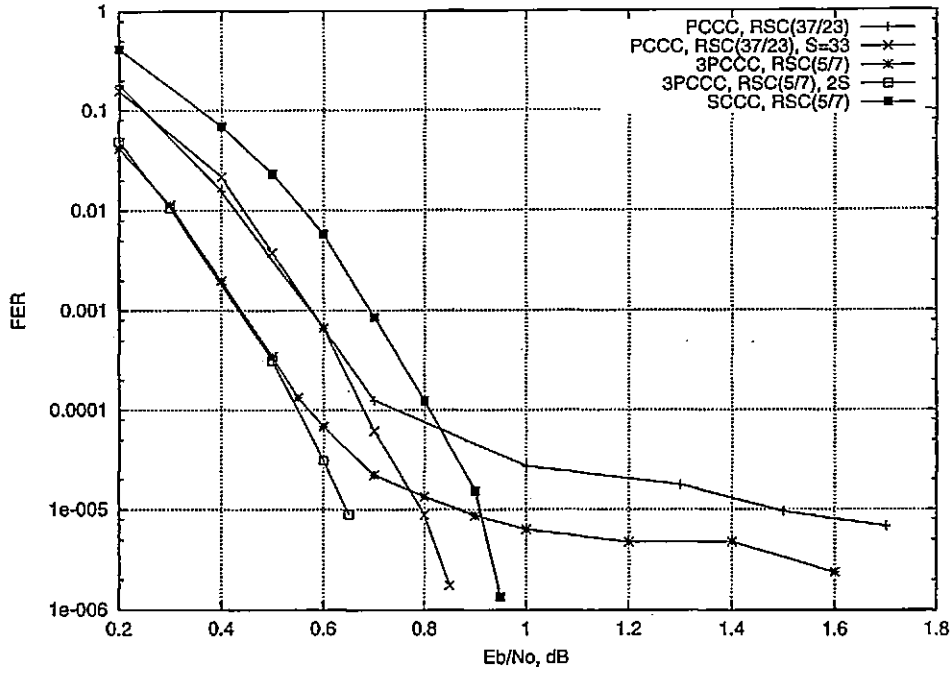


a)

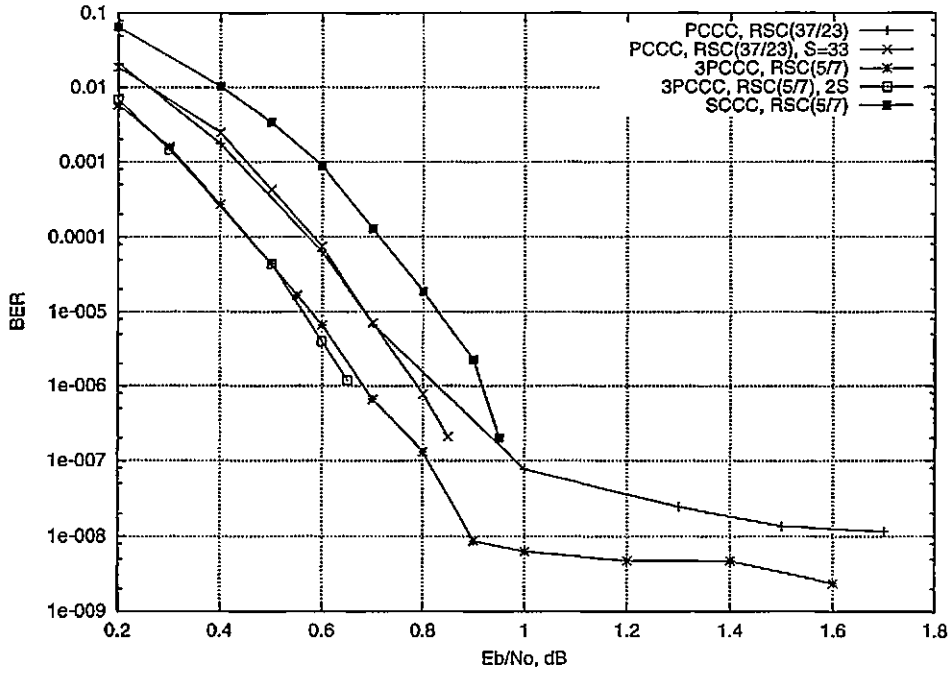


b)

Figure 3.30: Optimal code performance comparison for $N = 500$
Turbo codes/3PCCC/SCCC schemes performance comparisons for optimal codes using randomly chosen and designed (S-type) interleavers. Turbo codes use $RSC(37/23)$ codes and the 3PCCC/SCCC schemes use $RSC(5/7)$ codes.



a)



b)

Figure 3.31: Optimal code performance comparison for N=2000
Turbo codes/3PCCC/SCCC schemes performance comparisons for optimal codes using randomly chosen and designed (S-type) interleavers. Turbo codes use $RSC(37/23)$ codes and the 3PCCC/SCCC schemes use $RSC(5/7)$ codes.

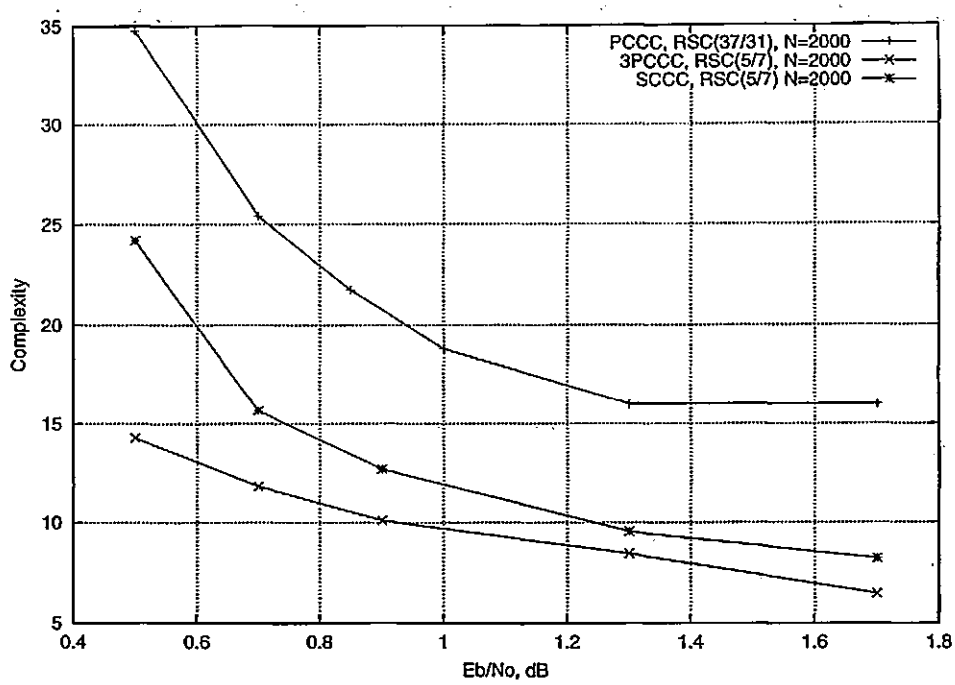
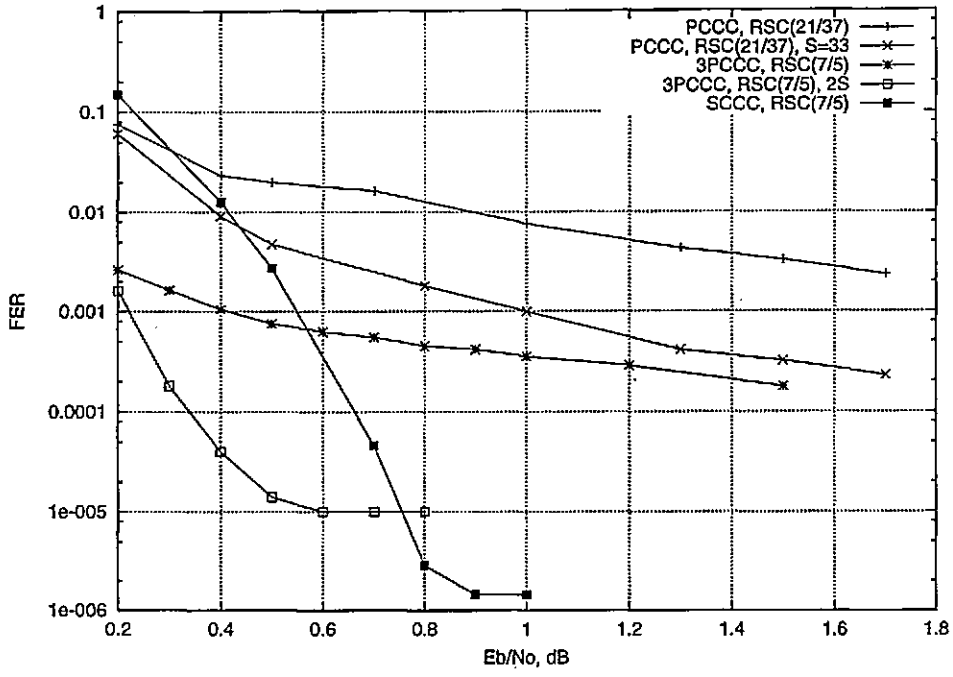
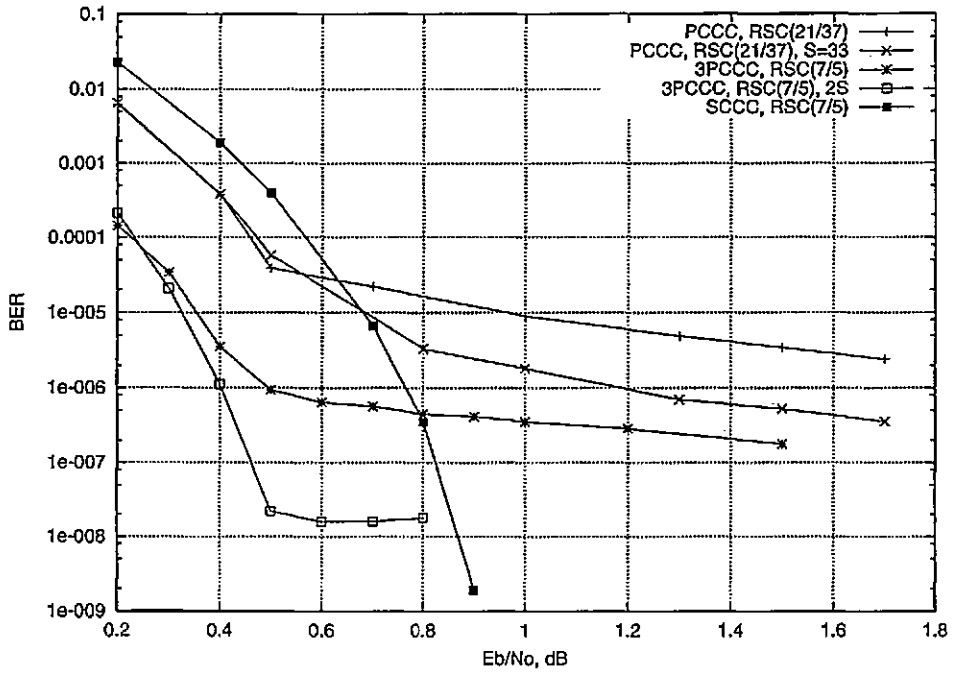


Figure 3.32: Decoding complexity comparisons
Decoding complexity comparisons for turbo codes using the $RSC(37/23)$ code and 3PCCC and SCCC schemes using the $RSC(5/7)$ code. The block length is $N = 500$.

Turbo codes have the highest complexity (as defined above) due to their increased memory. They have better error rate (especially BER) than the SCCC scheme at low E_b/N_o, but have a high error floor at high E_b/N_o where the SCCC have a significantly lower error rate. The situation is improved for turbo codes when using an S interleaver. Their error floor becomes lower than that of both SCCC and 3PCCC schemes using randomly chosen interleavers. The error floor of the SCCC scheme is produced by limit cycle blocks, whereas that of the 3PCCC is produced by a relatively low d_{free}. The simulated SCCC and 3PCCC schemes do not show any error floor when the interleaver is designed. Turbo codes using a designed (S) interleaver still show an error floor for $N = 500$ but their error floor is lowered outside the simulation range for $N = 2000$, similar to that of the other schemes. For $N = 2000$ the SCCC scheme does not show an error floor even with a randomly chosen interleaver. The error floor of turbo codes with S interleaver and $N = 2000$, although outside the simulation range, is easily reachable by using the $(OW_2)_{min}$ search algorithm. For this code, $(OW_2)_{min} = 38$. In the case of the 3PCCC scheme with paired S interleavers and $N = 2000$, the $(OW_2)_{min}$ search resulted in $(OW_2)_{min} = 58$, which is better than the performance of the turbo code,



a)



b)

Figure 3.33: Non-optimal code performance comparison

Non-optimal code performance comparison. Turbo codes use $RSC(21/37)$ and the 3PCCC, SCCC schemes use $RSC(7/5)$. The block length is $N = 2000$.

even if one accounts for the different code rate ($d_{free} = 38$ for $R = 1/3$ is equivalent to $d_{free} = \frac{4}{3} * 38 \approx 42$ for $R = 1/4$). Also, a further search for better interleavers would have more chances to succeed for the 3PCCC scheme. For the turbo code, the chance that $(OW_2)_{min} = 38$ is almost 1 (higher than 0.86), for any $S = 33$ interleaver.

From the above comparisons it is difficult to predict whether there will be an intersection of the 3PCCC and SCCC curves. Given the better *interleaver gain* of the SCCC schemes, it is expected that the SCCC scheme will become better at higher E_b/N_o values, somewhere outside the simulation range.

Simulations results that show the intersection of the SCCC error rate curve with that of the 3PCCC scheme *within* the simulation range are presented in figure (3.33) for a block length $N = 2000$. They are obtained by using the non-optimal $RSC(7/5)$ code for the 3PCCC and SCCC and the non-optimal $RSC(21/37)$ code for turbo codes. These codes have improved performance for each scheme at low E_b/N_oas compared to the performance obtained with optimal component codes, but they produce higher error floors, and this is how the error floor of the SCCC can be compared with that of the 3PCCC. Note that designed interleavers have to be used for the 3PCCC to lower their error floor and move the intersection point to higher E_b/N_o values whereas the SCCC uses a randomly chosen interleaver.

Thus the SCCC scheme can have a lower error floor than the 3PCCC scheme, due to the higher interleaver gain.

The 3PCCC scheme improves on the performance of turbo codes. If the required error rate is low enough, the SCCC scheme can also improve on the performance of turbo codes. If the required error rate is even lower, the SCCC can improve on the performance of the 3PCCC as well.

Due to their weak interleaver gain, turbo codes are improved by increasing component code memory. The 3PCCC and SCCC schemes could also be improved in this way if they were decoded with an optimal decoder. Increasing memory creates problems with the suboptimal, iterative decoder for all schemes, but the problems occur sooner for SCCC and 3PCCC. The limitation in code memory is compensated by the much better interleaver gain of these schemes.

3.8 Conclusions

- The error events of the iterative decoder have been characterised and used to study the performance of the iterative decoders with different parameters for turbo codes, 3PCCC, 4PCCC and SCCC schemes. The way to obtain good performance is investigated for each scheme.

- Detailed algorithms for the S interleaver are presented. The practical values for S are determined and the performance of the S interleaver as compared to a randomly chosen interleaver is studied. The "crossed" error events are presented as a weakness of the interleaver as the $IW = 2$ error events are removed by using high values of S. Formula are derived to determine the $(OW_2)_{min}$ and the value S_{2+2} where the "crossed" error events start dominating performance. Ways to eliminate "crossed" error events are presented and a novel method is used in improving turbo codes.

- The design of the interleaver pairs for the 3PCCC scheme is presented and justified by using the search of $IW = 2$ and $IW = 2 + 2$ error events. Formula are derived for the worst case for each interleaver design, and it is illustrated that the worst case is not the most likely when the interleaver is chosen (almost) at random.

- The way the $IW = 2$ error events produce the $(OW_2)_{min}$ for the MPCCC are determined and illustrated for different interleaver lengths and different component code. The results are obtained by computer search and a qualitative explanation is given. They are also used to verified the interleaver mapping probabilities obtained by combinatorial or average methods, and to compare the d_{free} of the MPCCC as the number of codes is increased. This is a novel, original approach.

- Comparisons between turbo codes, 3PCCC and SCCC are provided, using simulation and $IW = 2$ error event search results. The complexity of decoding is defined based on code memory, number of codes and average number of iterations.

Chapter 4

Turbo code spectra

4.1 Introduction

The iterative decoder is suboptimal and hence it is important to determine how close its output is to the output of an optimal decoder for turbo codes. An optimal decoder for an encoding system, is a decoder that maximizes the probability of a bit sequence or codeword (as in equation (4.1)), or the probability of each information bit separately (as in equation (4.2)), given the received data.

$$P_s = P\{\mathbf{u}_1^N|\mathbf{R}_1^N\} \tag{4.1}$$

$$P_b = P\{u_k|\mathbf{R}_1^N\},\ \forall k \in \{1,2,...,N\} \tag{4.2}$$

where $\mathbf{R}_1^N$ represents the received vector, N is the block length, $\mathbf{u}_1^N$ is the information sequence and u_k is a single information bit. The straightforward ("brute force") way to accomplish this is to compute the probability of each codeword given the received sequence and determine the maximum. This is not practical for long blocks due to the exponential dependence of the number of codewords on block length (2^N for binary codes).

For convolutional codes, optimal decoders exist in the form of the Viterbi algorithm (bit sequence) and the MAP algorithm (bit). They are based on the *trellis* representation of the convolutional codes, which drastically reduces the search alternatives for determining the maximum probability. In this case, the complexity is proportional to 2^k where k is the constraint length of the code.

For block codes, it is more difficult to determine a compact trellis representation. Although it is generally possible to construct a trellis for block codes, the difficulty is finding the *minimal* trellis, e.g. the one that minimizes the search complexity (for example, the number of trellis states). Even if the minimal trellis could be found, it is doubted that, for good codes, its complexity is low enough to allow for practical optimal decoding (Lafourcade and Vardy, 1995). Turbo codes using a block interleaver and terminated component codes are block codes.

4.2 The union bound

The performance of a linear code can be upper bounded by calculating its weight (distance) spectra and using the *union bound* formula. For an AWGN channel with BPSK/QPSK modulation, the union bound is:

$$\text{FER} \leq \frac{1}{2}\sum_{d=d_{free}}^{d_{MAX}} a(d)\text{erfc}\left(\sqrt{R\frac{Eb}{No}d}\right) \tag{4.3}$$

$$\text{BER} \leq \frac{1}{2}\sum_{d=d_{free}}^{d_{MAX}} \frac{w(d)}{N}\text{erfc}\left(\sqrt{R\frac{Eb}{No}d}\right) \tag{4.4}$$

where R is the code rate, d_{free} is the free distance of the code and $\frac{Eb}{No}$ is the bit energy to noise ratio in the AWGN channel. The value d_{MAX} represents the maximum code weight considered, a(d) represents the number of codewords having code weight $ow = d$ and $iw(d)$ is their cumulated information weight. The relationships between $a(d)$ and $w(d)$ and the multiplicity of a given error event mapping $a(iw, ow)$ used in the previous chapters are:

$$a(d) = \sum_{iw} a(iw, ow = d) \tag{4.5}$$

$$w(d) = \sum_{iw} a(iw, ow = d) * iw \tag{4.6}$$

In practice, the union bound sums are computed up to a much lower weight d_{MAX} than the maximum possible, obtaining a truncation which is valid for a given range of E_b/N_o values. The truncation is valid because the erfc() function decreases quickly with distance d. As the E_b/N_o decreases, d_{MAX} has to be increased to keep equations (4.3)

and (4.4) valid. The main difficulty in using the union bound formula for determining the performance of a code consists in determining enough terms in the weight spectra for a given E_b/N_o.

Also, it was found in (Divsalar et al., 1995) that, at least for the average turbo code, the union bound diverges at low E_b/N_o, taking values higher than 1. This is due to a quick increase in the multiplicity of codewords a(d) which compensates for the decrease of the erfc() function with d. This is not a weakness of the code, but of the bound, which is not close enough to the performance of the code. Improved bounds are determined in (Duman and Masoud, 1998; Viterbi and Viterbi, 1998; Divsalar, 1999).

The results in this chapter are based on the union bound, and are justified by the following surmission in (Divsalar et al., 1995): "... *even though the bound diverges, the portion of the bound based only on low-weight input sequences is still a useful predictor of performance*".

4.3 Computing the turbo code spectra

The methods to obtain the weight spectra of concatenated codes with interleavers can be classified based on the way the view the interleaver(s). They can be viewed as a fixed permutation or as a probabilistic device (the *uniform* interleaver in (Benedetto and Montorsi, 1996c)).

4.3.1 Fixed permutation methods

In this case, the spectra is determined for a fixed (real) interleaver.

Limiting the code weight

Since computing the whole spectra of the block code is only feasible for very small block lengths, the spectra is computed up to a maximum code weight d_{MAX}. One possibility is (Seghers, 1995; Daneshgaran and Mondin, 1997b) to consider all codewords of the first code with code weight less than d_{MAX}. Each of these codewords is interleaved, and the overall code weight is computed. The spectra is guaranteed to be complete up to a code weight just higher than d_{MAX}. The codewords of the first code are concatenations of the error events of the convolutional code. The number of error events in a block

increases with block length up to a limit dictated by the value of d_{MAX}, and then remains constant. However, the number of possible positions of these error events in the block increases with block length. Each of these positions has to be tried in order to determine the code weight of the interleaved code.

This produces a dependence on the block length that increases quickly with the maximum weight considered. If this weight is low, rather long blocks can be investigated. This is the case in (Seghers, 1995), where a turbo code with an impressive block length of $256 * 256 = 65536$ has been investigated, but for a maximum weight of $d_{MAX} = 6$. The complexity increases rapidly with d_{MAX}, limiting the block size to $N \approx 100$.

It can be observed that the algorithms presented above have a pronounced asymmetry, since the number of trials is limited only by the first code. This asymmetry is increased if the algorithm is used for parallel concatenations with two interleavers. A more symmetrical method is presented later in this chapter.

Limiting the information/code weight

It is possible that the needed value of d is too high for the algorithm to complete in reasonable time. In this case, an incomplete estimation of the spectra can be obtained by also limiting the information weight. This method has a probabilistic base for turbo codes, since they map lower information weight error events with higher probability, so they are more likely to cause the lower part of the spectra. Searching for $IW = 2$ error events of the concatenated scheme is very fast, due to the possibility of exploiting the periodicity of the convolutional codes. Thus checking the weight of an error event reduces to a simple division. The complexity increases with the maximum information weight considered and block length. In (Divsalar and Pollara, 1995c), a maximum information weight of $IW \leq 3$ is mentioned for a block length $N = 1024$. Since in this case only the information weight was limited, longer block lengths should be achieved by also limiting the code weight. For some schemes (3PCCC,SCCC) with long interleavers, this could be the only method to estimate where the non-zero spectra of the code starts, due to its weaker dependence on d.

4.3.2 Uniform interleaver methods

A uniform interleaver of length N is (Benedetto and Montorsi, 1996c) :

"*A probabilistic device which maps a given input word of weight w into all distinct $\binom{N}{w}$ permutations of it with equal probability $1/\binom{N}{w}$*".

The uniform interleaver does not exist as a real permutation, but the performance of a turbo code using this fictional interleaver is the average of the performances of all real turbo codes with interleaver length N.

It is the uniformity of the interleaver that simplifies the search for the code spectra, making it less dependent on interleaver length. Its simplification consists in the fact that each error event combination of one code does not have to be interleaved and encoded by the second code to determine the overall weight, a process that is strongly dependent on interleaver length. This is because, wherever the error events of the first code are positioned in the block, they determine any possible code weight of the second code with a given, readily determined probability. In this way, high code weights can be achieved, indeed so high that they have produced, in (Divsalar et al., 1995), the divergence of the union bound. Another strength of the probabilistic methods is that they can identify a dependence on the interleaver length (the interleaver gain) without even considering the spectra of the component codes, except for some very general properties. This is more attractive in approaches using the limit as the interleaver length increases towards infinity, rather than fixed (and sometimes short) interleaver lengths.

The weakness of this method is that it does not describe the exact code structure and performance of a turbo code using a real, given interleaver.

4.4 The turbo code tree

Fast methods to determine the weight spectra of a convolutional code (Cedervall and Johannesson, 1989) rely on the tree representation of the codewords. Each node in the tree represents a code *state* and each branch between two nodes a transition from one state to another. The code state represents the *memory* of the code, the link between the previous code bits and the future code bits. Each transition produces a set of code bits, and is generated by one or more information bits. For each transition, the final

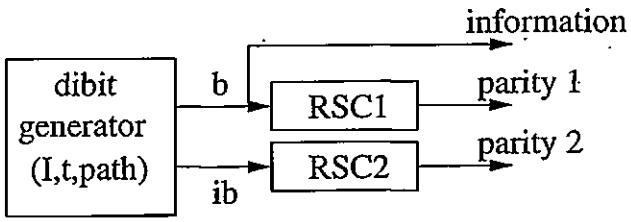


Figure 4.1: Turbo code tree generator

state and the encoded bits depend on the current state and the current information bit(s). A complete path in the tree represents a codeword. Parts of the tree can be dynamically generated and examined without having to examine the rest of the tree. This fact is exploited by sequential algorithms, like the Fano algorithm and the stack algorithm (Michelson and Levesque, 1984).

A modified form of the Fano algorithm can be used to determine the first terms of the weight spectra for a convolutional code. The algorithm simply starts from the root node and sequentially extends every path in the tree, computing its weight at each node. If the weight exceeds a maximum value (which is a parameter of the algorithm), the subtree starting with that node is not examined, since the weight of a path can only increase. Instead, the algorithm backs up one or more stages, and an alternative path is extended.

In order to use this algorithm, it is essential that, at any node in the tree, the algorithm can determine all the possible transitions to the next node. For convolutional codes, this is readily accomplished, since every node in the tree is associated with an encoder state, which represents the only memory of the code. If the encoder takes one input bit at each transition, there are always two possible transitions, one corresponding to an input bit of 0 and one corresponding to an input bit of 1.

A turbo code has more memory than the separate states of its two encoders, due to the presence of the interleaver. In order to use the tree representations of the component codes to generate the turbo code tree, the system can be viewed as a two input / three output bit system, as in figure (4.1). The two input bits are related due to the interleaver. This relationship can be represented as a bit-pair (dibit) generator which produces valid bit pairs based on the memory of the currently extended path, the interleaver constraints and the current depth in the tree. The memory of the whole

depth	interleaver constraint b, ib	branches	valid bit pairs b, ib
1	$(-,-)$	4	00,01,10,11
2	(ib_2, b_2)	2	00,11
3	$(-,-)$	4	00,01,10,11
4	$(-, b_3)$	2	$0b_3$,$1b_3$
5	(ib_3, b_4)	1	ib_3b_4
6	$(-, b_1)$	2	$0b_0$,$1b_0$
7	(ib_1, b_6)	1	ib_1b_6

Table 4.1: Dibit combinations in a turbo code tree
Possible dibit combinations for each depth in a turbo code tree, due to interleaver constraints

system is contained both in the dibit generator and the states of each component code.

To illustrate tree generation, assume that $N = 7$ and the interleaver mapping is given by the following permutation,

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 7 & 2 & 5 & 3 & 4 & 1 & 6 \end{pmatrix} \tag{4.7}$$

i.e. $ib_1 = b_7$, $ib_2 = b_2$, $ib_3 = b_5$ etc. At any node in the tree, the dibit generator checks if the input bits are dependent on previous input bits due to the interleaver, and generates the possible combinations. Table (4.1) shows these combinations for every depth in the tree for the above interleaver.

At depth 1 in the tree, the two input bits are evidently independent (there are no previous bits), so all four dibit combinations are possible, resulting in four branches of the tree at this depth.

At depth 2, due to the fact that $ib_2 = b_2$ there are only two possible input combinations, 00, respective 11 .

At depth 3, since none of the interleaver restrictions refers to previous bits ($ib_3 = b_5$ and $b_3 = ib_4$) the two input bits are independent and all four combinations are possible.

At depth 4, $ib_4 = b_3$ and $b_4 = ib_5$. In this case, b_4 is independent and can have any of the values $\{0, 1\}$, but ib_4 has to be equal to whatever value b_3 has for the currently extended path. In this case, only two input bit combinations are possible, $0b_3$ and $1b_3$.

At depth 5, the interleaver equations are $ib_5 = b_4$ and $b_5 = ib_3$. In this case, both bits have the values already established for b_4 and ib_3 for the current path, and only

one combination is possible, ib_3b_4.

The rest of the table can be interpreted in the same way. The maximum depth of the tree is the interleaver length, in this case $N = 7$. It can be seen that once a branch of the tree has been fully extended, it represents a valid codeword, since the input to the second encoder is an interleaved version of the input of the first encoder. Also, the set of the complete tree branches is identical to the set of codewords of the turbo code. The effect of the interleaver on the layout of the turbo code tree is to vary the number of branches for each depth in the tree. At the same depth in the tree, the number of branches from each node is identical.

A graphical representation of the turbo code tree for the previous example is given in figure (4.2), in which only 8 branches have been fully extended for clarity. The code states associated with each node are represented. Also, the bit-pair that caused the transition is shown on each branch between two nodes. The interleaver constraints are presented at the top of the figure. For clarity, the parity bit values that can be calculated for each transition have not been shown.

The turbo code tree can be used to determine the first terms of the code weight spectra using a modified Fano algorithm. In this case, the metric is the weight of the current path, up to the current node. It is calculated recursively, using the formula

$$M_{k+1} = M_k + dM_k \tag{4.8}$$

Where M_{k+1} , M_k are the weights at depth $k+1$, respective k for the currently extended path, and dM_k is the weight increase due to the transition from depth k to depth $k+1$.

$$dM_k = b_k + p_k^1 + p_k^2 = b_k + p_k \tag{4.9}$$

where $p_k = p_k^1 + p_k^2$. With this recursion equation, the running metric becomes

$$M_k = M_0 + \sum_{n=1}^{k} dM_k = M_0 + \sum_{n=1}^{k} (b_n + p_n) \tag{4.10}$$

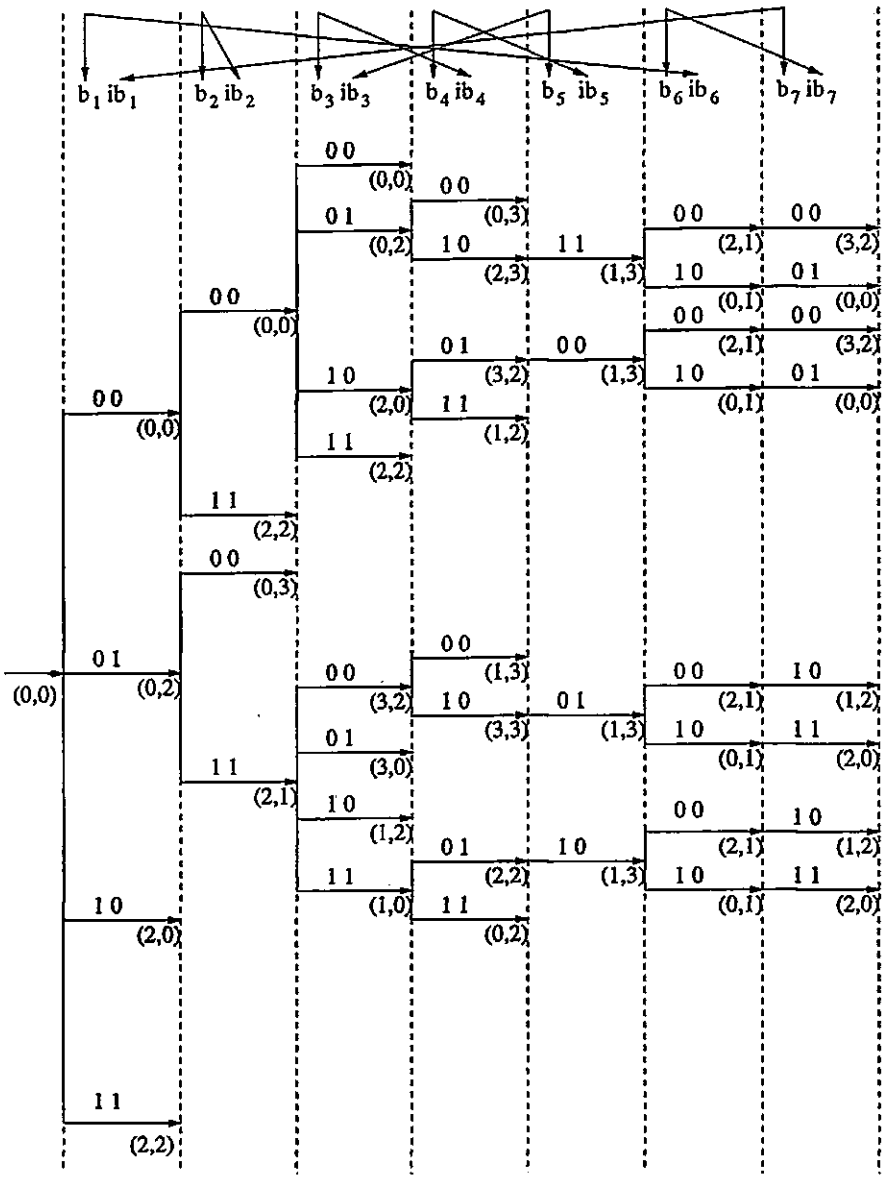


Figure 4.2: Turbo code tree ($N = 7$, $M = 2$ codes)

If a path is fully extended, its final metric is

$$M = M_0 + \sum_{k=1}^{N} dM_k = M_0 + \sum_{k=1}^{N} (b_k + p_k) \tag{4.11}$$

It can be observed that at a given depth k in the tree, more information is available about the full path metric than used in the formula (4.10). This is because of the knowledge of the interleaved bit sequence $ib_1, ..., ib_k$, which could be equal, due to interleaver constraints, to values of the non-interleaved bit b outside the range $b_1, ..., b_k$. In order to use this information, a new metric is defined by the formula

$$M^+_{k+1} = M^+_k + dM^+_k \tag{4.12}$$

where the weight increase dM^+_k considers both the non-interleaved bit b_k as well as the interleaved bit ib_k. The mathematical expression for dM^+_k is

$$dM^+_k = \lambda_{k,I(k)} b_k + \lambda_{k,I^{-1}(k)} ib_k + p_k \tag{4.13}$$

where the coefficient $\lambda_{k,p}$ has been introduced in order to prevent adding the information bit twice.

$$\lambda_{k,p} = \begin{cases} 1 & if \quad k < p \\ \frac{1}{2} & if \quad k = p \\ 0 & if \quad k > p \end{cases} \tag{4.14}$$

With the above definitions, equation (4.12) becomes

$$M^+_k = M_0 + \sum_{n=1}^{k} (\lambda_{n,I(n)} b_n + \lambda_{n,I^{-1}(n)} ib_n + p_n) \tag{4.15}$$

The improved metric M^+ has the following properties:

$$M^+_k \geq M_k \quad , \quad \forall k \in \{1, ..., N\} \tag{4.16}$$

$$M^+_N = M_N \tag{4.17}$$

An example of the basic and improved metric calculations for the interleaver described

depth, k	$I(k)$	$\lambda_{k,I(k)}$	$I^{-1}(k)$	$\lambda_{k,I^{-1}(k)}$	dM	dM^+
1	6	1	7	1	$b_1 + p_1$	$b_1 + ib_1 + p_1 = b_1 + b_7 + p_1$
2	2	0.5	2	0.5	$b_2 + p_2$	$0.5b_2 + 0.5ib_2 + p_2 = b_2 + p_2$
3	4	1	5	1	$b_3 + p_3$	$b_3 + ib_3 + p_3 = b_3 + b_5 + p_3$
4	5	1	3	0	$b_4 + p_4$	$b_4 + p_4$
5	3	0	4	0	$b_5 + p_5$	p_5
6	7	1	1	0	$b_6 + p_6$	$b_6 + p_6$
7	1	0	6	0	$b_7 + p_7$	p_7

a)

depth	b_k	ib_k	state	p_k	dM	dM^+	M	M^+
0	-	-	(0,0)	-	-	-	0	0
1	0	1	(0,2)	1	1	2	1	2
2	1	1	(2,1)	1	2	2	3	4
3	1	1	(1,0)	1	2	3	5	7
4	0	1	(2,2)	1	1	1	6	8
5	1	0	(1,3)	1	2	1	8	9
6	1	0	(0,1)	2	3	3	11	12
7	1	1	(2,0)	2	3	2	14	14

b)

Table 4.2: Basic vs improved metric
Basic vs improved metric changes for an N=7 interleaver

by formula (4.7) is presented in table 4.2(a), and a numeric calculation for a given branch is presented in table 4.2(b). It can be observed that, in table 4.2(a), the final improved metric is just the basic metric calculated in a different order, $M_7^+ = (b_1 + b_7 + p_1) + (b_2 + p_2) + (b_3 + b_5 + p_3) + (b_4 + p_4) + p_5 + (b_6 + p_6) + p_7 = M_7$. This is what equation (4.17) states, and it justifies the possibility of using the improved metric instead of the basic metric, since the two metrics have the same value once a path has been fully extended. This value represents the code weight of the particular codeword.

Equation (4.16) is the reason why M^+ is an "improved" metric. This can be observed for the example in table 4.2(b). Since M^+ is always bigger or equal than M, it usually allows the search algorithm to decide much quicker if a path will be dropped or not, thus reducing the number of visited nodes and increasing the speed of the algorithm. It is difficult to predict the speed improvement, it depends on the interleaver, the component codes and the maximum metric considered. A practical comparison,

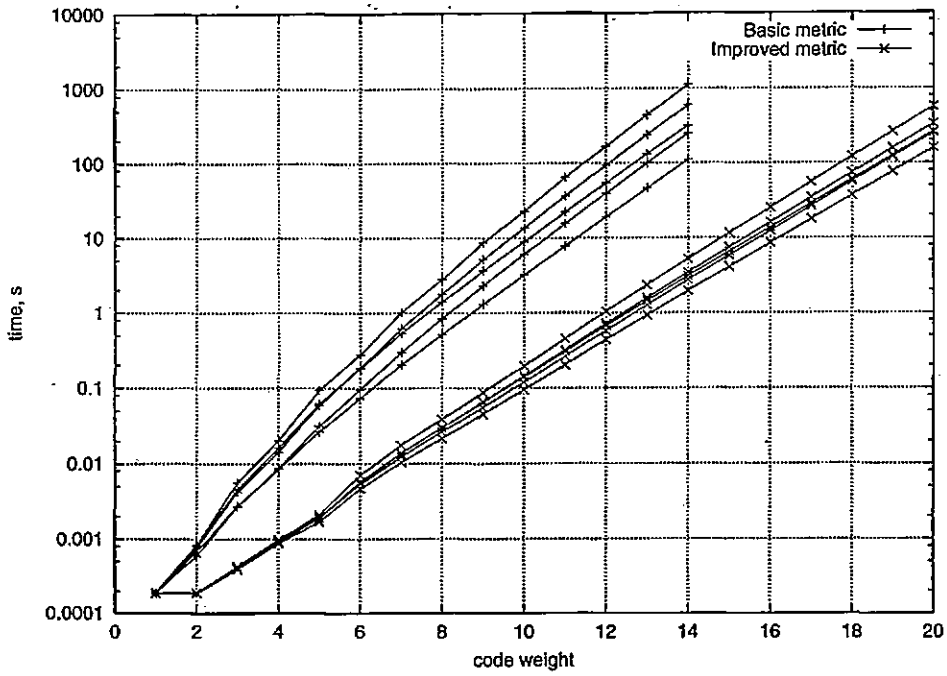


Figure 4.3: Tree search timing comparisons
Tree search timings comparison for algorithms using basic and improved metric, for a turbo code with parameters $N = 100$, $M = 2$ $RSC(5/7)$ component convolutional codes. The machine used was a 450MHz Pentium III.

for an interleaver of length $N = 100$ and component codes $M = 2$, $RSC(5/7)$ is given in figure (4.3).

4.5 The weight spectra of turbo codes

The effect of changing different parameters of the scheme on the code performance can be observed by determining its weight spectra using the tree search method. The effect of increasing block length and code memory, using optimal or non-optimal component codes is discussed below. Also, the trellis termination problem is presented from the optimal decoding point of view, and its effect is studied for different interleavers.

4.5.1 Dependence on block length

The weight distributions for turbo codes with different block lengths are presented in Table (4.3). The interleavers used have been randomly chosen and the component codes are simple memory $M = 2$, $RSC(5/7)$ convolutional codes. It can be observed

	N=50		N=100		N=200		N=500		N=1000	
d	a(d)	w(d)	a(d)	w(d)	a(d)	w(d)	a(d)	w(d)	a(d)	w(d)
7	2	5	-	-	-	-	-	-	-	-
10	3	6	1	2	2	4	2	4	2	4
11	1	3	1	3	1	3	-	-	-	-
12	11	26	2	4	4	8	6	12	5	10
13	12	38	4	12	2	5	1	3	-	-
14	12	35	6	13	10	22	8	16	8	16
15	11	32	9	26	1	3	1	3	1	3
16	26	86	14	36	14	29	8	18	8	16
17	49	204	13	53	8	24	3	9	2	5
18	75	313	31	94	10	29	13	29	14	30
19	138	640	32	136	15	53	6	22		
20	230	1109	58	234	42	148	26	70		
21	420	2231	97	431	27	112				
22	762	4156	163	757	76	306				
23	1196	7051	271	1334	102	459				
24	2337	14435	429	2174						
25	3978	26208	730	3982						

Table 4.3: Dependence of weight spectra on block length
Weight spectra for turbo codes using $M = 2$, $RSC(5/7)$ component convolutional codes, for different interleaver lengths N

that increasing the interleaver length influences the weight spectra of the turbo code in several ways:

1. It does not change the d_{free}=$d_{free-eff}$. This is due to the fact that the probability of a $d_{free-eff}$ error event mapping is almost independent on interleaver length. Note the exception for the short block length $N = 50$, where one of the error events that causes d_{free} is $(IW = 3, OW = 7)$ and the other is a truncated error event. Also note the multiplicity of the $d_{free-eff}$ which is mostly 2. The tree search algorithm has also been used to produce the distribution of the d_{free} of a $RSC(5/7)$ turbo code with $N = 100$ and $N = 500$. The results are presented in figure (4.4). The distribution concentrates around $d_{free} = d_{free-eff} = 10$ as N is increased from $N = 100$ to $N = 500$.

2. It reduces the number of error events for higher weights in the code spectra. For higher code weights, $IW > 2$ information weights produce error events. Since their multiplicity decreases with N, the weight spectra becomes 'thinner', until it is only composed of $IW = 2$ error events. This is the "spectral thinning", presented in (Perez et al., 1996) for a uniform interleaver, and illustrated here for randomly chosen interleavers of increasing length.

3. The actual decrease in the number of error events for a given weight in the weight spectra with the interleaver length, instead of an increase makes possible the *interleaver gain*, presented in the average methods, since the value N divides the weight spectra in the BER union bound formula.

The error rate curves corresponding to the weight distributions in Table (4.3) are presented in figure 4.5(a) for the block error rate FER and (b) for the bit error rate BER. They are calculated by using the union bound formula in (4.3) and (4.4) with code rate $R = 1/3$. The FER curves show an improvement in going from $N = 50$ to $N = 100$ but remain almost constant as N is further increased. Note that the curves are not parallel, and a tentative error floor could be observed, but not as pronounced as that of the iterative decoding performance. This correlates with the spectral thinning theory, and it is possible that the beginning of the error floor is not so visible because the maximum code weight considered is not high enough.

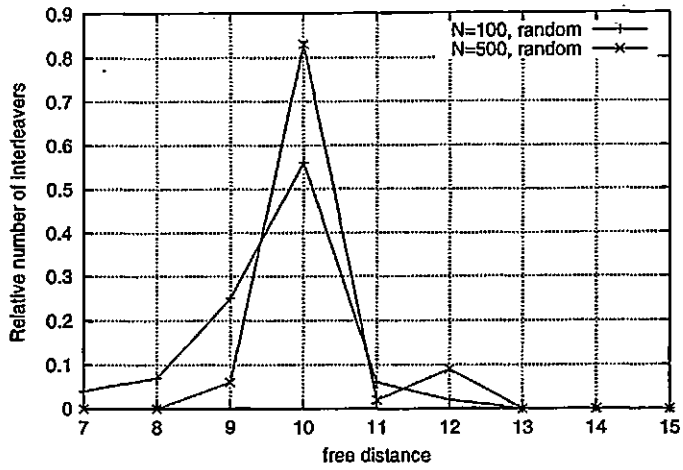


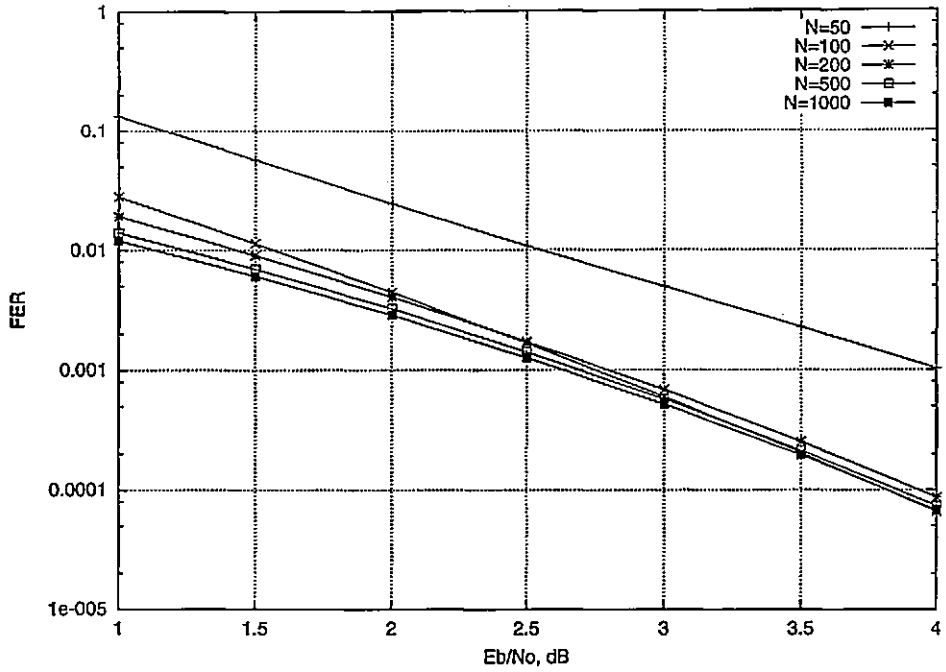
Figure 4.4: Histogram of d_{free} values for turbo codes
Free distance histogram for turbo codes with $N = 100$ and $N = 500$. Turbo codes use $RSC(5/7)$ as component code. The number of experiments was 100 for each interleaver length.

The BER curves obtained by using the weight spectra are compared with the iterative decoder results in figure (4.6), for three different block lengths $N = 100$, $N = 500$ and $N = 1000$. It can be observed that the performance of the iterative decoder is worse that the union bound curves at low E_b/N_o and very close to it at high E_b/N_o. There are two reasons for this difference:

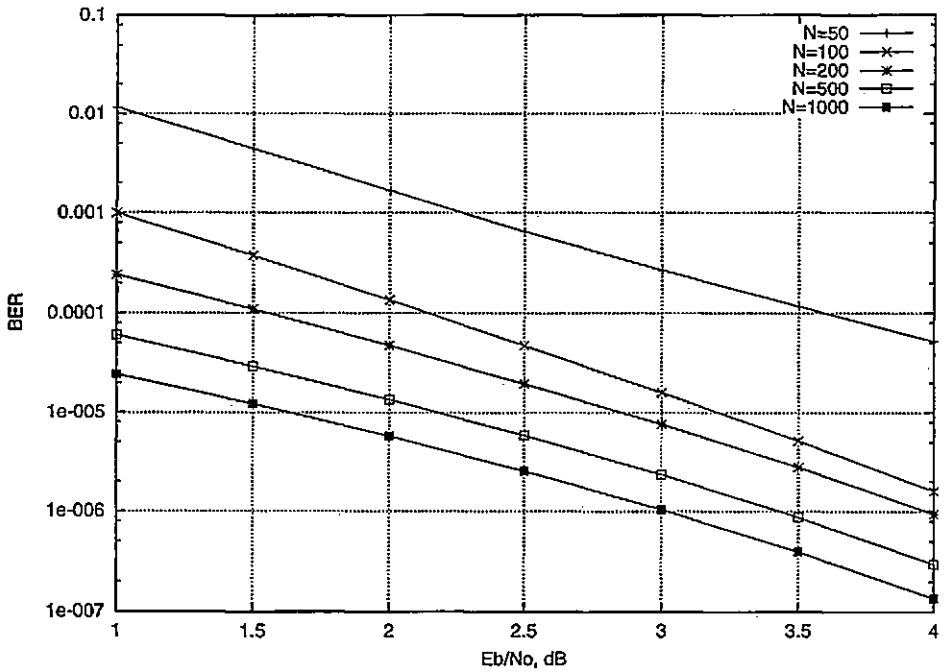
1. At low E_b/N_o the iterative decoder produces the wrong results, e.g. it fails to converge. As the E_b/N_o is increased, the convergence improves, until the nonconvergent blocks disappear completely.

2. The weight spectra of the turbo code is incomplete. It is possible that there are components of the spectra that have been neglected but could have an effect on optimal BER at low E_b/N_o. It is difficult to determine these components since the search time increases exponentially with d_{MAX}.

By determining the information/code weight for the error blocks, it has been observed that the differences at low E_b/N_o are caused by high information/code weight (HIWHOW) error blocks, very unlikely in the optimal decoder case even at $E_b/N_o = 1$dB. This is the reason why the differences at low E_b/N_o are attributed with a higher probability to the iterative decoder's lack of convergence. LIWLOW error events with code weight higher than d_{MAX} have been observed to produce optimal/iterative decoder



a)



b)

Figure 4.5: Union bound turbo code performance for different block lengths Union bound curves for $RSC(5/7)$ turbo codes using randomly chosen interleavers of increasing length a) Frame Error Rate (FER), b) Bit Error Rate (BER)

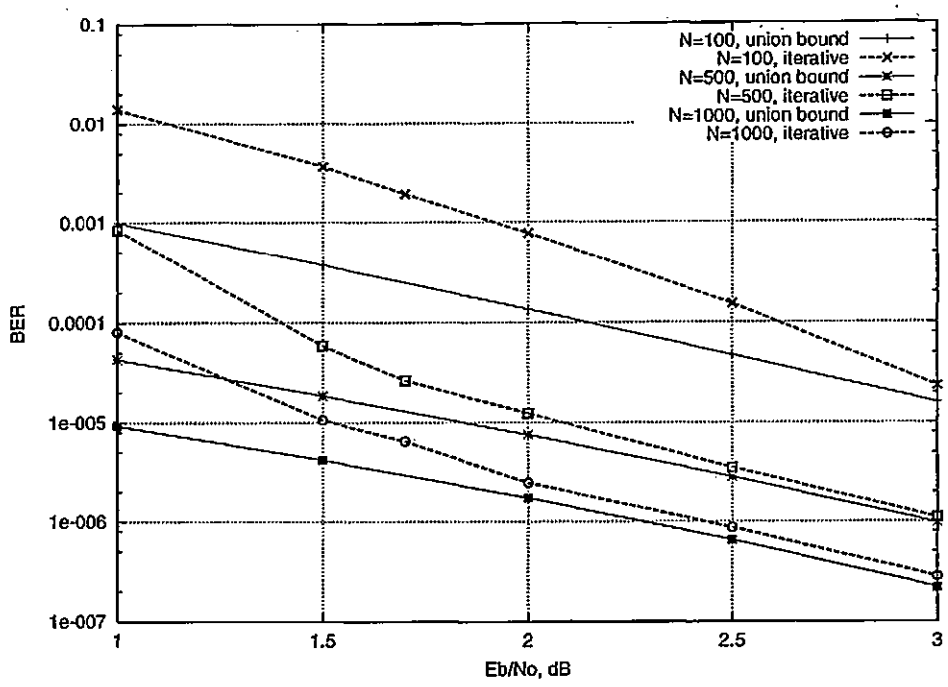


Figure 4.6: Iterative decoding/union bound BER comparison for different N Iterative decoding vs union bound BER comparison for turbo codes using the $RSC(5/7)$ component code and different block lengths

differences for the $N = 1000$ code, at high E_b/N_o. This is to be expected, due to the small number of components of the weight spectra that could be practically determined. This difference also disappears when increasing E_b/N_o as higher distances become insignificant for the optimal decoding performance. Also, it can be observed that as the interleaver length is increased, the iterative curve approaches the optimal decoding curve quicker. This observation, combined with the type of error events that cause the differences suggests that the convergence of the iterative decoder improves with interleaver length.

4.5.2 Dependence on code memory

In order to determine the effect of increasing component code memory, the weight spectra of turbo codes using different memory codes and the same randomly chosen interleaver has been determined and presented in table (4.4). The codes used are the optimal component codes for turbo codes for each memory, as presented in (Benedetto et al., 1998b). It can be observed from the table that there is a significant increase

d	Code							
	M=2		M=3		M=4		M=5	
	a(d)	w(d)	a(d)	w(d)	a(d)	w(d)	a(d)	w(d)
10	2	4	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-
12	6	12	-	-	-	-	-	-
13	1	3	-	-	-	-	-	-
14	8	16	-	-	-	-	-	-
15	1	3	1	3	1	3	-	-
16	8	18	-	-	-	-	-	-
17	3	9	-	-	-	-	-	-
18	13	29	7	15	-	-	-	-
19	6	22	-	-	1	3	-	-
20	26	70	1	2	-	-	-	-
21			3	9	2	6	-	-
22			8	18	3	7	-	-
23							1	3
24							1	4

Table 4.4: Dependence of weight spectra on code memory
Weight spectra for turbo code using the same randomly chosen interleaver and component codes with increasing memory. The block length is $N = 500$.

in d_{free} as the memory is increased. Also, the higher memory codes turbo code has a much 'thinner' spectra, at least for low weights. This comes at the price of higher decoder complexity, with complexity depending exponentially on code memory. The table shows that turbo code performance can be improved by increasing component code memory, but the classical compromise of exponential complexity/performance has to be made. Also, the iterative curves show a degradation in performance at low E_b/N_o as memory is increased.

Note that the d_{free} of the higher memory codes is caused by $IW = 3$ error events rather than $d_{free-eff}$, as it was observed in the previous chapter by analysing the LIWLOW error events. This is because higher memory codes have high $d_{free-eff}$ values and also the block is not long enough to eliminate the higher IW error events. As the block length is increased, these error events will disappear and the performance of the higher memory codes will also be limited by their $d_{free-eff}$. Note that this means an initial increase in d_{free} as it "converges" to $d_{free-eff}$.

d	M=2				M=4			
	optimal		non-optimal		optimal		non-optimal	
	a(d)	w(d)	a(d)	w(d)	a(d)	w(d)	a(d)	w(d)
8	-	-	1	2	-	-	-	-
9	-	-	5	10	-	-	-	-
10	2	4	7	14	-	-	4	8
11	-	-	8	16	-	-	-	-
12	6	12	6	12	-	-	6	12
13	1	3	17	38	-	-	1	2
14	8	16	17	38	-	-	2	4
15	1	3	12	28	1	3	-	-
16	8	18	17	46	-	-	11	22
17	3	9	40	128	-	-	-	-
18	13	29	75	262	-	-	6	12
19	6	22	169	634	1	3	-	-
20	26	70	249	958	-	-	26	84
21					2	6	3	8
22					3	7	54	194

Table 4.5: Optimal/non-optimal code weight spectra
Optimal/non-optimal weight distributions for memory $M = 2$ and $M = 4$ turbo codes with $N = 500$.

4.5.3 Optimal versus non-optimal component codes

Optimal codes have been determined based on averaging turbo code performance over the class of interleavers of length N. Results for a given, randomly chosen interleaver are presented in table (4.5), in comparison with results for non-optimal component codes for the same interleaver. For memory $M = 2$, the optimal code is $RSC(5/7)$ and the non-optimal code $RSC(7/5)$ and for memory $M = 4$, the optimal code is $RSC(37/23)$ and the non-optimal code $RSC(21/37)$. It can be seen that for the same interleaver, there is a significant difference between the two classes of component codes in the d_{free} obtained, as well as in the multiplicity of the error events, leading to a significant improvement in union bound decoding performance for optimal codes. Figure (4.7) presents the union bound BER curves for the two $M = 4$ codes, in comparison to the BER curves obtained with the turbo decoding algorithm. It can be observed that at low E_b/N_o, the difference between the iterative decoder performance and the union bound performance is much bigger for the optimal code than for the non-optimal code. At $E_b/N_o = 1$dB this difference causes the iterative decoder performance to be

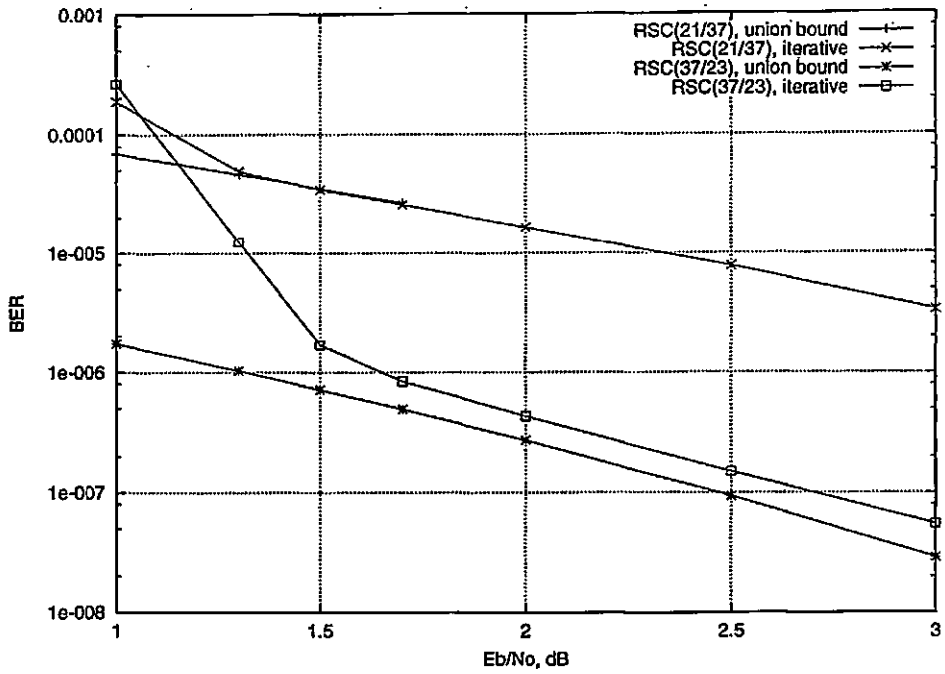


Figure 4.7: Optimal/non-optimal code iterative decoding/union bound BER comparison

worse for the optimal code than for the nonoptimal code, although the union bound performance is much better. Also, the iterative/union bound curves meet quicker for the non-optimal code than for the optimal code. Again, this is due to the presence of high information/code weight (HIWHOW) decoded blocks, which appear much more often for the optimal code at low E_b/N_o. These blocks reduce in number and disappear when the two curves for iterative decoder an union bound performance converge.

This suggests that the difference of the two types of curves is due to the convergence of the iterative decoder, and not to the lack of enough terms in the weight spectra. It also shows that the non-optimal decoder has a positive influence on the decoding process, although it produces a poor weight spectra.

4.5.4 The S interleaver

The S interleaver was presented in previous chapters as a method to improve turbo code performance while keeping the complexity constant, as opposed to increasing code memory. The effect of increasing the parameter S of the interleaver on the weight spectra of the resulting turbo code is presented in Table (4.6). It can be seen that

d	Interleaver					
	S=0		S=8		S=16	
	a(d)	w(d)	a(d)	w(d)	a(d)	w(d)
12	3	6	-	-	-	-
13	1	3	-	-	-	-
14	7	14	7	14	-	-
15	-	-	1	2	-	-
16	15	34	7	14	-	-
17	4	14	-	-	-	-
18	7	17	9	18	-	-
19	5	23	1	2	-	-
20	25	73	18	48	25	71

Table 4.6: Random vs S-class interleaver weight spectra
Weight spectra for $N = 500$, $RSC(5/7)$ turbo code using S interleavers with different values of parameter S.

increasing S does significantly increase the d_{free} of the turbo code, but it does not change the multiplicity of the higher weight error events significantly. This has a good side because it shows that the "crossed" error events discussed in the previous chapter do not increase in number. The distribution of the free distance for turbo codes using the $RSC(5/7)$ component code, $N = 100$ and $S = 0$ and $S = 7$ interleavers is shown in figure (4.8). It can be observed that using the S interleaver shifts the d_{free} distribution towards higher values.

4.5.5 The data tail

The optimal decoding interleaver gain is based on the observation that for recursive encoders, sequences containing a single bit of 1 ($IW = 1$) have theoretically infinite code weight for recursive component codes. In the case of a bit of one occuring close to the end of the block, this assumption is not valid anymore, since only a small part of the infinite error event is actually contributing to the overall code weight. This is the truncation effect of the block interleaver.

Table (4.7) presents the weight distributions for turbo codes using different interleavers, each under three assumptions: 1) there is no restriction on the end state of the codes, 2) the first code has to end in the all zeros state and 3) both codes have to end in the all zeros state. It has been assumed that to force the first code back to

d	Condition					
	-		$S_{1,N}=0$		$S_{1,N}=0 \& S_{2,N}=0$	
	a(d)	w(d)	a(d)	w(d)	a(d)	w(d)
7	1	2	-	-	-	-
8-9	-	-	-	-	-	-
10	3	6	3	6	3	6
11	-	-	-	-	-	-
12	3	6	3	6	3	6
13	1	4	-	-	-	-
14	11	23	11	23	10	20

a)

d	Condition					
	-		$S_{1,N}=0$		$S_{1,N}=0 \& S_{2,N}=0$	
	a(d)	w(d)	a(d)	w(d)	a(d)	w(d)
12	3	6	3	6	3	6
13	1	3	1	3	1	3
14	7	14	25	71	24	68
15	-	-	-	-	-	-
16	15	34	15	34	13	28
17	4	14	4	14	4	14

b)

d	Condition					
	-		$S_{1,N}=0$		$S_{1,N}=0 \& S_{2,N}=0$	
	a(d)	w(d)	a(d)	w(d)	a(d)	w(d)
18	9	19	8	16	8	16
19	-	-	-	-	-	-
20	25	71	25	71	24	68

c)

Table 4.7: The effect of data tail for different interleavers
The data tail problem for different interleavers. Three termination conditions are considered by the tree search algorithm: 1) the final code state can have any value for both codes 2) the final code state is zero for the first code 3) the final code state is zero for both codes. Cases a), b) and c) determine the weight spectra for different, randomly chosen interleavers.

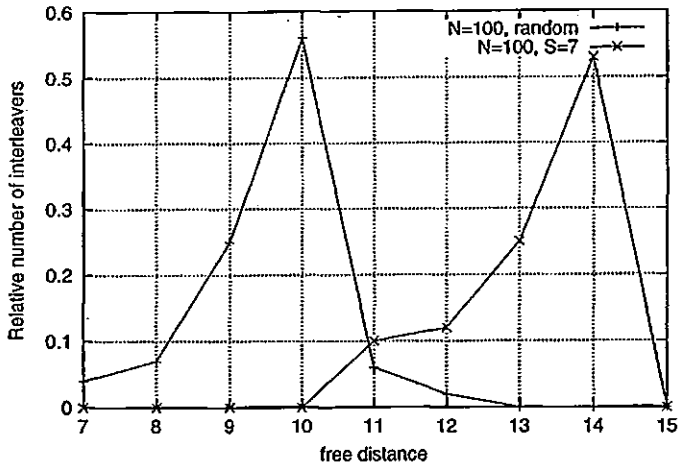


Figure 4.8: Improvement of d_{free} with S
Improvement of d_{free} with S for turbo codes using the $RSC(5/7)$ component code and $N = 100$. The S values are $S = 0$ (randomly chosen interleaver) and $S = 7$.

the all zeros state a data tail of M bits has been appended, and to force both codes to all zero state two data tails, amounting to $2M$ bits had to be appended. This has an impact on code rate and thus on the overall performance, especially for short blocks. It can be seen that using the data tail has a different effect for different interleavers. In case a), it actually improves the code spectra and increases d_{free}, improving the optimal performance whereas in cases b) and c) the weight spectra for low weights is not changed, resulting in a slight performance degradation due to the reduction in code rate. The BER curves corresponding to the weight distributions in Table (4.7) are shown in figure (4.9). This shows that, provided the interleaver is carefully designed, the data tail is not necessary, at least for low memory codes, which are usually employed in turbo codes. In order to design the interleaver for this purpose, it can be observed that a single bit of 1 close to the end of the direct input stream produces a low weight error event for the overall turbo code if it is interleaved also close to the end of the interleaved stream. A simple condition is to require that the last M bits are interleaved far from the end of the interleaved stream. In the design of the S interleavers, this can be included as a modification of the S condition by stating that if a bit is closer than S bits to the end of one of the input streams, it has to be interleaved more than S bits away from the end of the second input stream. An interesting case is that of a row/column interleaver. As established in the previous chapter, this interleaver has

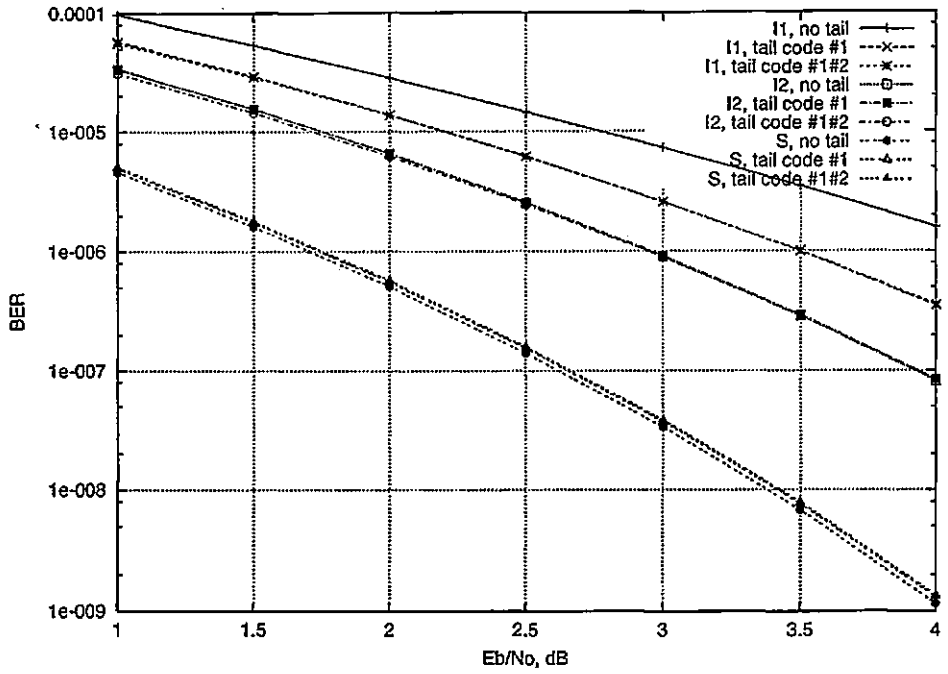


Figure 4.9: Data tail effect on performance

the highest value of S. Also, this type of interleaver always interleaves the last bit in the direct stream into the last bit in the interleaved stream. This results in an error event due to trellis truncation of very low weight ($2-3$, depending on the component code), and thus a very low d_{free}. Also, due to the strong S condition, all the other error events have high code weight (for N=500, usually $ow \geq 18$), since the bits close to the last bit in the direct stream *must* be interleaved more than S bits away from the last bit in the interleaved stream. The problem of the small error event could be easily solved by simply ignoring the last bit, rather than appending an M bit data tail to each code.

The distributions of d_{free} for turbo codes using the $RSC(5/7)$ component code and randomly chosen interleavers with $N = 100$ are shown in figure (4.10). They are determined by using the tree search algorithm an the three possible final code state conditions presented above.

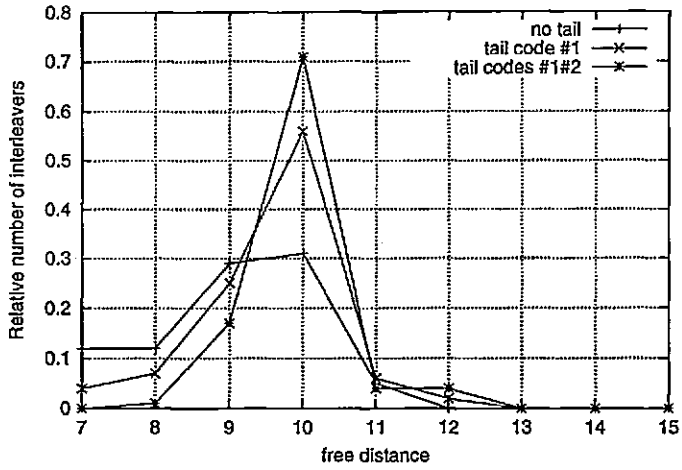


Figure 4.10: Variation of d_{free} with termination scheme
Variation of d_{free} with termination scheme for a turbo code using a randomly chosen interleaver and the $RSC(5/7)$ component code. The block length is $N = 100$.

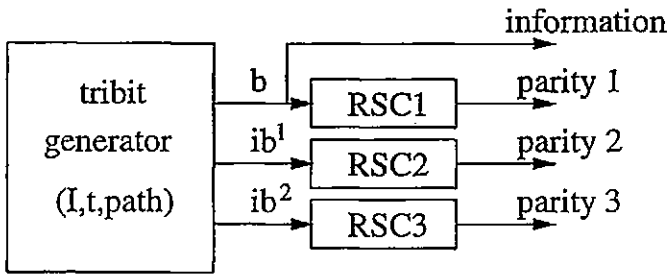


Figure 4.11: 3PCCC tree generator

	PCCC				3PCCC			
	$N = 100$		$N = 200$		$N = 100$		$N = 200$	
d	a(d)	w(d)	a(d)	w(d)	a(d)	w(d)	a(d)	w(d)
8	1	2	4	8	-	-	-	-
9	4	8	3	6	-	-	-	-
10	6	12	3	6	-	-	-	-
11	11	22	8	16	-	-	-	-
12	10	20	12	24	-	-	-	-
13	12	30	7	14	-	-	-	-
14	8	20	19	46	1	2	-	-
15	25	76	12	32	-	-	-	-
16	34	114	31	92	-	-	2	4
17	55	186			3	6	-	-
18	124	450			2	4	2	4
19	181	714			2	4	1	2
20	306	1244			2	4	-	-
21	526	2210			3	6	1	2
22					3	6	3	6
23					4	8		
24					2	6		
25					4	8		
26					4	8		
27					3	10		
28					11	30		

Table 4.8: Turbo code/3PCCC weight spectra
Weight spectra for a turbo code and a 3PCCC scheme using the non-optimal $RSC(7/5)$ code and randomly chosen interleaver(s) with $N = 100$. The turbo code has rate $R = 1/3$ and the 3PCCC code $R = 1/4$.

4.6 Generalisation to MPCCC

The tree generation algorithm can be easily generalized for MPCCC schemes. Figure (4.11) shows the tree generator for the 3PCCC scheme. In the MPCCC case, the number of tree branches for each node will be in the range $n_b \in \{1, ..., 2^n\}$ where n is the number of codes in the scheme. The speed of the tree search algorithm can be improved in a similar way as for turbo codes, by using the interleaved bits available from each code. Experiments have shown that the maximum weight that can be obtained for a given interleaver length in a reasonable amount of time is slightly higher than that for turbo codes. Unfortunately, since the 3PCCC schemes have a lower code rate ($R = 1/4$ for 3PCCC as opposed to $R = 1/3$ for turbo codes) the E_b/N_o range

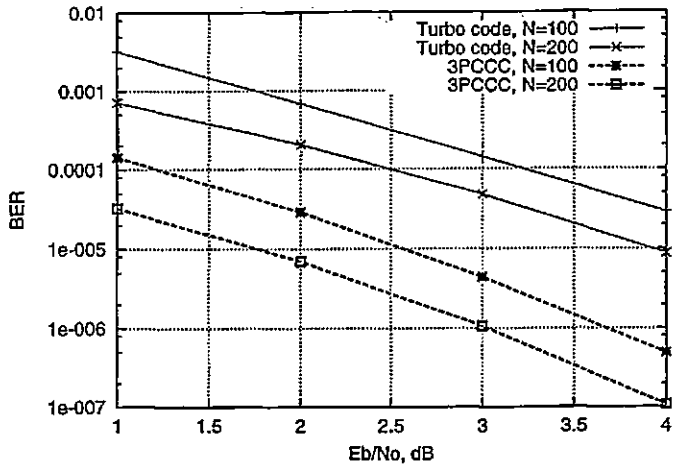


Figure 4.12: Turbo code/3PCCC union bound BER comparison
The component code used are the non-optimal $RSC(7/5)$ codes, and the interleaver length is $N = 100$. The turbo code has rate $R = 1/3$, the 3PCCC has rate $R = 1/4$.

for which the corresponding union bound values are valid is smaller. Also, even for interleaver lengths as small as $N = 200$ the d_{free} can be very close or above the search limit. As an example, for $N = 200$ a $d_{free} = 26$ has been obtained using the $RSC(5/7)$ component code and a randomly chosen pair of interleavers. Not all randomly chosen interleaver pairs produce such a high d_{free}. A value as small as $d_{free} = 14$ has also been observed for the $RSC(5/7)$ component code and $N = 500$, value which corresponds to the $d_{free-eff}$ of this component code for a 3PCCC scheme. An example weight spectra is presented in table (4.8) for the non-optimal component code $RSC(7/5)$. The non-optimal code has been used in order to produce more weight spectra components in the tree search range. The number of weights computed was the maximum possible in a reasonable time (one day on a 450 MHz machine) for the 3PCCC scheme. The weight spectra for the turbo code has been truncated so that

$$d_{MAX}^{TC} = \frac{R^{3PCCC}}{R^{TC}} d_{MAX}^{3PCCC} = \frac{3}{4} d_{MAX}^{3PCCC} \tag{4.18}$$

for a fair comparison. In equation (4.18), d_{MAX}^{TC} is the maximum weight considered in the turbo code spectra, $R^{TC} = 1/3$ is the turbo code rate and d_{MAX}^{3PCCC}, $R^{3PCCC} = 1/4$ are the corresponding values for the 3PCCC scheme. The condition is obtained by requiring that the erfc() function in the union bound formula (4.3) has the same value on d_{MAX} for the two schemes for any given E_b/N_o value. It can be observed that even

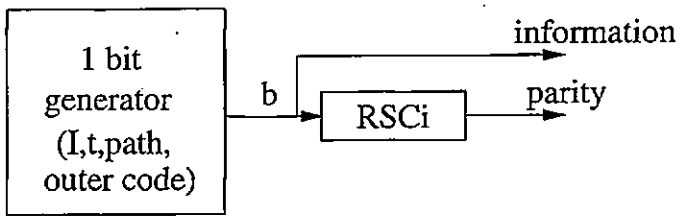


Figure 4.13: SCCC tree generator

in this case the weight spectra is very "thin" for the 3PCCC scheme as compared to the turbo code. The fact that this thinning can overcome the effect of a decreased code rate is shown in figure (4.12), which shows the BER curves obtained by using the union bound. An improvement in performance by more than an order of magnitude can be observed for the 3PCCC scheme.

Due to the fact that the maximum weight for the algorithm is comparable or (usually) smaller than the d_{free} of 3PCCC schemes, its applications in studying these schemes is limited. It can show that the d_{free} for a given scheme is higher than the search limit or it can identify the residual low weight error events. Several code weights can usually be obtained for block lengths $N \leq 200$. A $d_{free} = 26$ has been obtained in *reasonable* time for a 3PCCC scheme using the $RSC(5/7)$ component code and $N = 500$. A result of using the tree search algorithm for the 3PCCC scheme was the observation that, even if the scheme can generate a low d_{free}, the multiplicity of the d_{free} error event and of the immediately following code weights is very small, as compared to the relatively quick increase for turbo codes.

4.7 The tree of the SCCC scheme

A tree generation algorithm for the SCCC concatenation based on a similar idea is presented in figure (4.13). In this case, the bit generator produces one or two valid bit values for each node in the tree. The validity of the current bit value is determined by the previous bits (path), the interleaver (I) and the code structure of the outer code. The condition is that the current set of determined bits should belong to a valid codeword of the outer code. Simplifications can be made based on the limited constraint length of the code. As opposed to MPCCC schemes, the generation of valid

bit values for each node is much slower, which makes the scheme impractical even for short block lengths.

4.8 Non-iterative decoding

The iterative decoding algorithm allows for powerful codes, closer to the Shannon limit than ever, to be decoded with linear complexity. Compared to the iterative decoder, non-iterative decoding schemes for turbo codes (are they still "turbo" then?) are very limited, and can be used successfully only for short block lengths and rather high E_b/N_o values. So what is their attraction? The iterative decoder is suboptimal, and its convergence conditions are not yet known. Although most of the non-iterative methods are also suboptimal, their suboptimality has a different nature, and thus they could give a new dimension to the iterative algorithm. Several suboptimal, non-iterative algorithms are presented in (Narayanan and Stuber, 1998a; Sadowsky, 1997). An optimal non-iterative algorithm based on a turbo code trellis is presented in (Breiling and Hanzo, 1997a; Breiling and Hanzo, 1997b).

The availability of the turbo code tree makes trying sequential decoding algorithms for turbo codes tempting.

4.8.1 Sequential decoding

Sequential algorithms have been a method to decode convolutional codes before the Viterbi algorithm, and are still used for long constraint lengths. A typical example of using the stack algorithm to decode an $N = 100$, $RSC(5/7)$ turbo code is presented in figure (4.14) for $E_b/N_o = 5$dB. Figure 4.14(a) presents the metric evolution for the correct path as opposed to the chosen path, and figures 4.14(b) and (c) the error events for the two component codes. It can be observed that the stack algorithm chooses the wrong path although its final metric is lower than that of the correct path (and hence the wrong path would not have been chosen by an optimal algorithm). The explanation for this situation is that the metric of the chosen path does not decrease under the level where it was higher that the metric of the correct path. As shown in (McEliece, 1977), if this happens, the stack decoder will choose the wrong path. The reason for the slow decrease can be found by studying the error events of the two codes. First, the memory

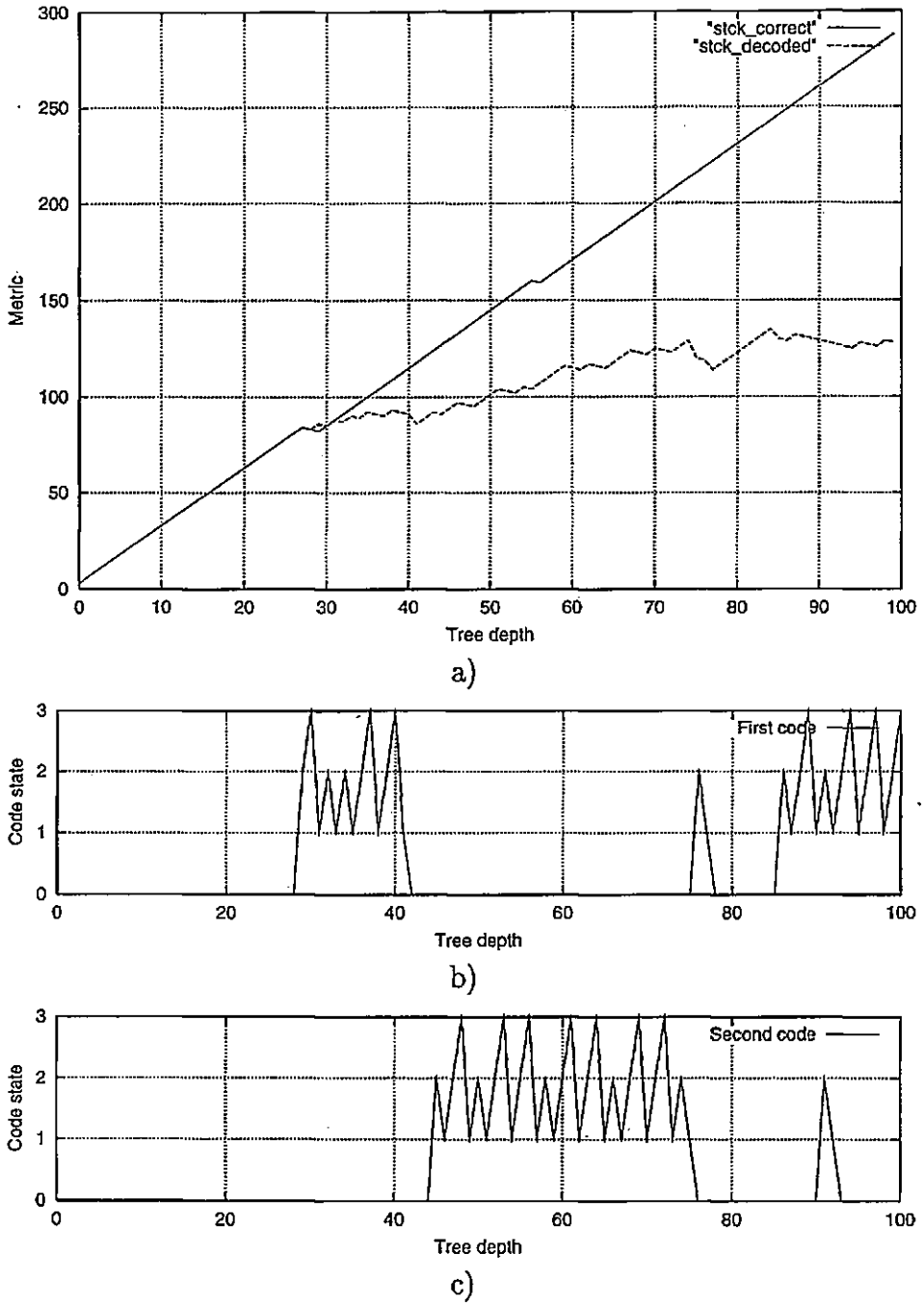
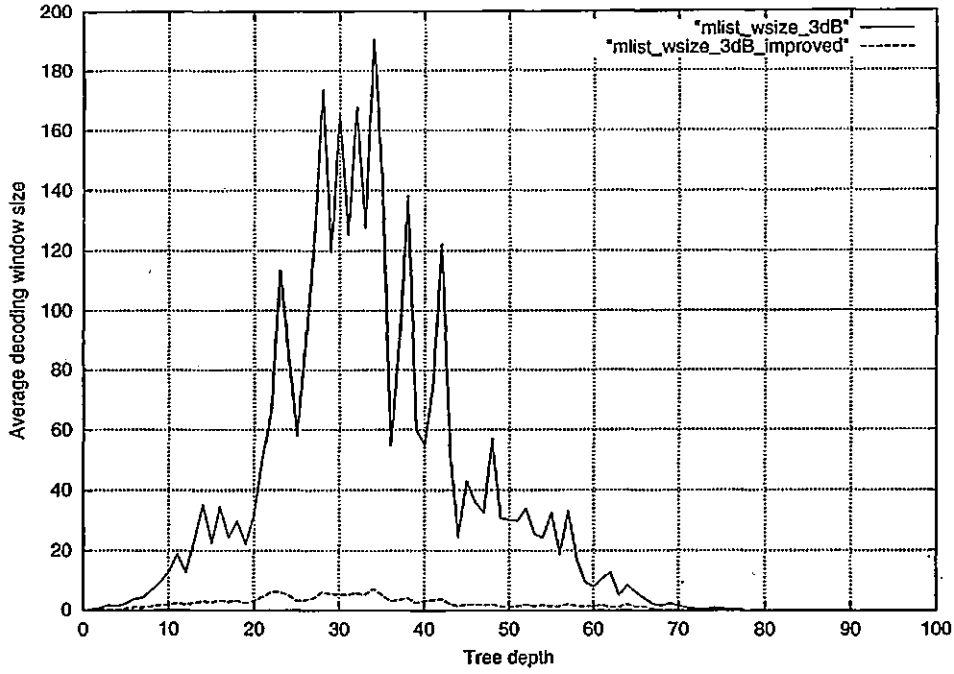


Figure 4.14: Stack decoding results
Stack decoding of an N=100 $RSC(5/7)$ turbo code: Correct/Decoded path metrics (a), decoded error event for the first (b) and second (c) component code.

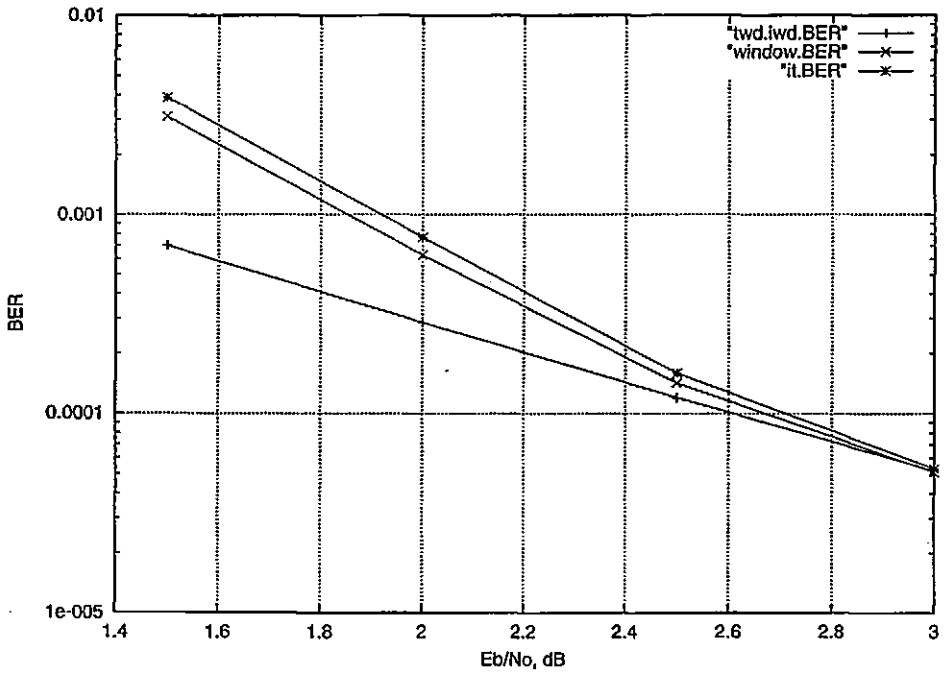
$M = 2$, $RSC(5/7)$ codes are not really suitable for sequential decoding. Second, the decoder chooses a path that generally contains error events from only one code at any given time, which accounts for the slow increase in disagreement between the received data and the chosen path. The "divide and conquer" principle translates to "divide and lose" for sequential decoding of turbo codes. It is possible that a careful, combined code/interleaver design could improve the situation. The advantage will be the possibility to use high memory component codes with no increase in complexity and no convergence problems.

4.8.2 Window decoding

Another alternative, closer to a brute force approach but still using the turbo code tree, is using a 'decoding window' to store the most likely paths at each moment in time. Each path in the window is extended, the resulting paths are sorted in increasing metric order, and the paths with the smallest metrics are discarded in order to keep the number of paths smaller or equal to the size of the window. The metric used is the Euclidean distance. Figure 4.15(a) presents the average window size for each bit, needed to keep the correct path inside the decoding window. Provided the correct path is not discarded, it is usually chosen at the end. A weakness of this method can be observed before the middle of the block, where it needs a large window. Past this point, the window size is small, allowing for quick and correct decoding. The average window size can be reduced by using the interleaved information bits in the distance computation, as presented in the previous sections. The required window size increases with block length, making the algorithm usable only for short blocks ($N < 100$). Also, the window size increases with decreasing E_b/N_o, and a feasible E_b/N_o value for $N = 100$ is $E_b/N_o = 2$dB. Although this algorithm was more successful, it occasionally needs very high window sizes (more than 200000 paths), depending on the noise pattern. These blocks are usually decoded correctly by the iterative algorithm. Figure 4.15(b) shows a BER comparison between the iterative decoding, window decoding and the union bound for an $N = 100$, $RSC(5/7)$ turbo code. Losing the correct path from the decoding window usually produces a high number of errors, which degrade the performance of the window decoder, especially at $E_b/N_o = 1.5$dB.



a)



b)

Figure 4.15: Window decoding results
Turbo code using the $RSC(5/7)$ component code and a randomly chosen interleaver. The block length is $N = 100$. a) Average decoding window size at $E_b/N_o = 3$dB and b) BER comparison with iterative decoding and union bound

$C_I(0)$	$C_I(1)$	$C_I(2)$	$C_I(3)$	$C_I(4)$	$C_I(5)$	$C_I(6)$	$C_I(7)$
$\emptyset$	b_1, ib_1	b_1, ib_1	b_1, ib_1, b_3, ib_3	b_1, ib_1, ib_3, b_4	b_1, ib_1	ib_1, b_6	$\emptyset$

Table 4.9: Interleaver constrained bits

4.9 The turbo code trellis (hypertrellis)

The convolutional code tree is highly redundant and it can be compacted into a trellis. This is based on the observation that, at a given depth into the convolutional code tree, nodes having the same corresponding encoder state will generate identical subtrees. Thus, they can be combined, generating the trellis. The trellis is a suitable representation form for optimal decoding algorithms such as the Viterbi algorithm and the MAP algorithm.

Figure (4.2) shows four identical code states $(1, 3)$ in a turbo code tree at depth 5. States 1 and 2 can be compacted into one node, since they generate identical subtrees. States 3 and 4 can also be compacted into a single node. Still, the two resulting nodes can not be compacted into a single node, because the subtrees they generate are not identical. This is due to the fact that $b_7 = ib_1$ due to interleaver constraints, and the first 4 paths have $ib_1 = 0$ while the last 4 paths have $ib_1 = 1$. This observation leads to the idea that two nodes in a turbo code tree can be compacted into a single node if the following two statements are true:

1. The two nodes have identical associated component code states
2. The paths leading to the two nodes have identical sets of input bits that will constrain future input bits. The set of constrained input bits at a given depth in the tree depends on the interleaver, and can be defined as

$$C_I(n) = \{b_k | I(k) > n\} \cup \{ib_k | I^{-1}(k) > n\} \quad (4.19)$$
$$\forall k, n \in \{1, ..., N\},\ k < n$$

For the tree in figure (4.2) the set $C_I(n)$ can be graphically identified for each n as being the set of input bits for which the arrows representing the interleaver constraints cross the line marking depth n. They are presented in table (4.9)

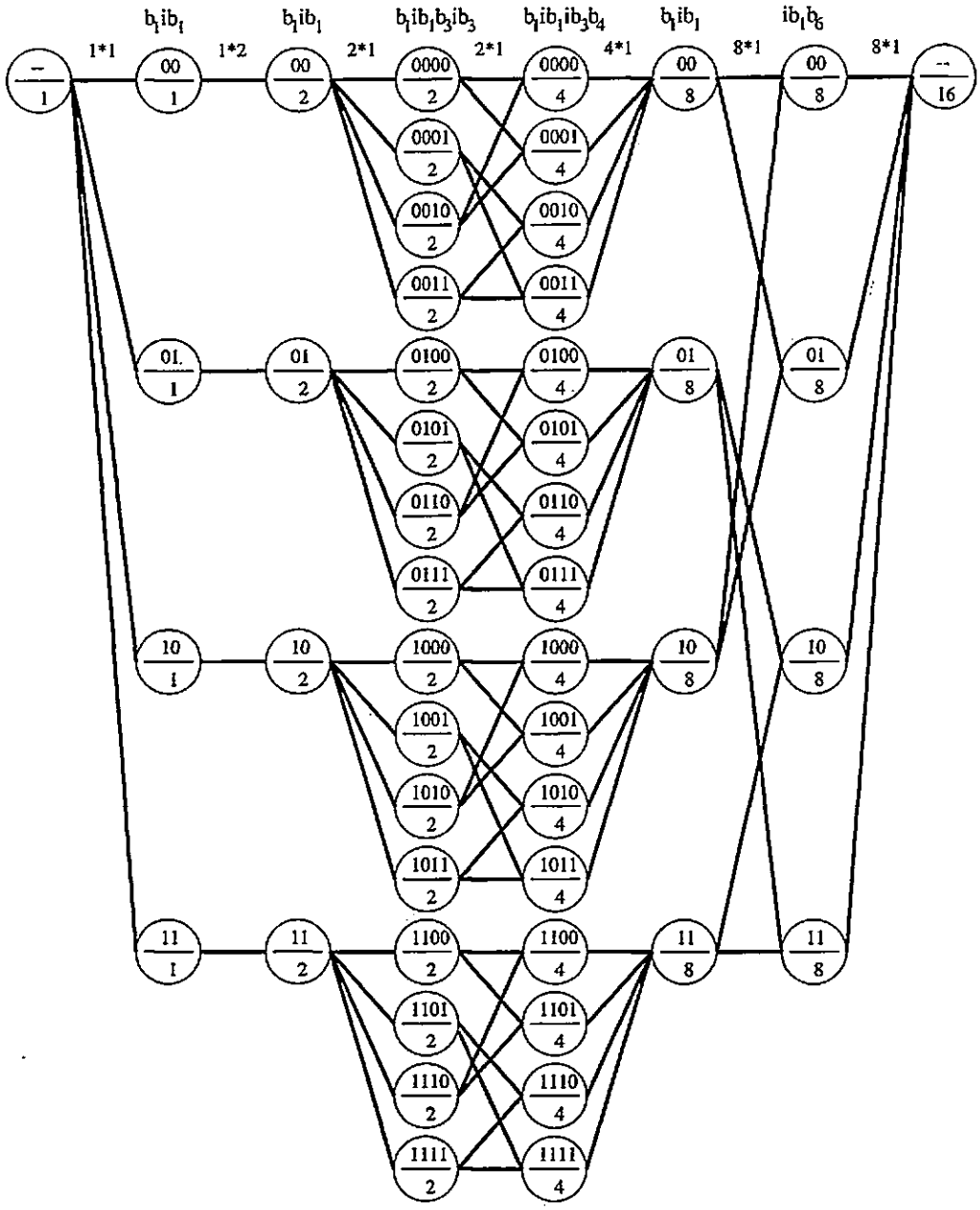


Figure 4.16: Hypertrellis interleaver grouping

Interleaver grouping for the $N = 7$, $M = 2$, $RSC(5/7)$ example turbo code. Each circle represents a group of states that have the same interleaver constrained bits. The number above the line in the circle is the value of the constrained bits. The name of the constrained bits for each stage is shown at the top of the figure. The number below the line in the circle is the actual number of code states in the group for the given turbo code parameters. The two numbers above each transition at the top of the figure are (the number of states)*(number of transitions from each state to another state in a given group). These values are the same for all transitions at the same stage.

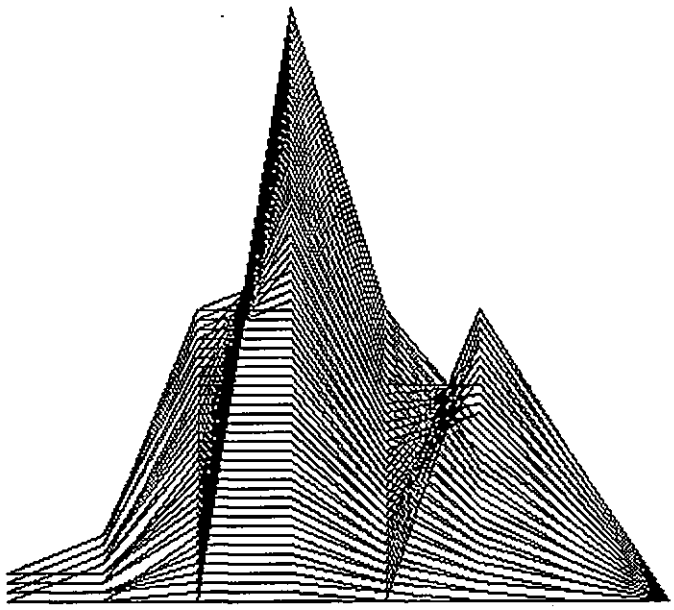


Figure 4.17: Hypertrellis "shape" ($N = 7$)

The nodes at a given depth in a turbo code tree can be grouped depending on their set of constrained bits.

- Two nodes at depth n in the tree will belong to the same group if and only if they have identical sets of constrained bits.
- Two nodes at a given depth in the tree can be compacted into a single node if they belong to the same group and have identical associated code states.
- A group cannot contain more than $m_1 m_2$ different states, where m_1 is the number of states of the first code, respective m_2 is the number of states of the second code.

Denoting $\varphi_I(n)$ the number of elements in $C_I(n)$, the number of groups at depth n is $m_I(n) = 2^{\varphi_I(n)}$. The number $m_I(n)$ can be seen as an expansion of the number of trellis states due to the interleaver. With these definitions, the number of trellis states at stage n, denoted $m_T(n)$ can be bounded by the following expressions

$$m_T(n) \leq m_1 m_2 m_I(n) \tag{4.20}$$

It can be seen from equation (4.20) that the presence of the interleaver causes the turbo code trellis to be time-variant. The grouping for the example interleaver given

by formula (4.7) is shown in figure (4.16). A hypertrellis "shape" is presented in figure (4.17).

Equation (4.20) shows that in this approach, in order to keep the number of states small it is necessary to use local dependence interleavers, interleavers that do not "throw" the bits far away, which could be, for example, a series of small interleavers put together to form a bigger interleaver, or a convolutional interleaver with short constraint length. The results for row/column interleavers are similar to those presented in (Benedetto et al., 1997c). An interesting observation is that a different approach to construct the hypertrellis, presented in (Breiling and Hanzo, 1997a), generated a lower number of trellis states for a row/column interleaver with 3 rows and 330 columns. This interleaver generates a hypertrellis with number of states dependent only on the number of columns, whereas with the presented approach it depends on both dimensions of the row/column interleaver. Also, correlative with (Breiling and Hanzo, 1997a) is the permuted trellis in (Benedetto et al., 1997c) which, as opposed to the nonpermutted version is only dependent on the number of columns.

4.10 Conclusions

- Methods to obtain the turbo code spectra are described.
- A novel tree search algorithm is presented, and used to characterise the performance of turbo code with different parameters. Results from the average theory are verified, and comparisons with the iterative decoding results are performed.
- The tree search method is extended to MPCCC schemes with relative success, and the unsuitability of the method for SCCC schemes is explained.
- Non-iterative decoding algorithms are presented. The window decoding results are compared with iterative decoding results and the union bound.
- A novel method to compact the turbo code tree into a trellis is presented and compared with other methods in the literature.

Chapter 5

Convergence of the iterative decoder

5.1 Introduction

There are two main approaches to estimate the performance of a turbo code system:

- Using computer simulation to determine its BER curve against a range of E_b/N_o values.
- Using the weight spectra of the overall encoder and the union bound (or tighter bounds) to estimate their expected BER, assuming a ML decoder at the receiving side.

The advantage of the first approach is that, in performing the simulations, the actual iterative decoder is used. Consequently, the BER curves closely describe the real system, provided the channel model describes closely the actual channel. The problem with this method is that it is a trial and error method, and it does not offer design criteria for the component codes and the interleaver, in order to improve the performance.

The second approach offers design criteria for component codes and interleavers, assuming that an optimal (ML) decoder is used at the receiving side. This assumption generates the need to compare the optimal decoder with the real, iterative decoder. The output of the two decoders can be different, since the iterative decoder is suboptimal. If the results are different, it is important to determine how big this difference is and

how it can be reduced by designing the component codes and the interleaver. The design criteria may be similar or contradictory to the ML design criteria.

Since the turbo decoder is iterative, the first question to ask is whether or not it converges. Convergence shows if the decoder has reached a stable decision, or it keeps changing the output for each iteration. There are two essential factors that dictate the output of the iterative process: its mathematical tendency to converge or diverge, which is usually estimated using the *fixed point* approach and the data representation errors.

5.2 Non-ML iterative decoder output

It is also necessary to estimate the output of a ML decoder. In order to detect the blocks where the differences occur, it is necessary to perform a blockwise comparison. This is very difficult, since the general ML methods refer to a 'uniform' interleaver as opposed to a particular (randomly chosen) interleaver, and the optimal decoders for a particular interleaver are limited to short block lengths and restricted interleavers.

Although it is difficult to determine which decoding *is* maximum likelihood, it is relatively easy to determine which one *is not*, at least from a binary sequence maximum likelihood point of view. This can be done in the simulations by re-encoding the transmitted information and the decoded information, and determining the Euclidean distances between the two codewords and the received vector. If the distance between the decoded codeword and the received vector is greater than the distance between the transmitted codeword and the received vector, the decoding is not maximum likelihood. If the distance is smaller, then the decoded codeword is more likely than the transmitted codeword, but not necessarily the most likely. It is also of interest to determine whether any of the component codes considers the decoded vector more likely than the encoded one based on its separate (and incomplete) received information.

This approach has the following drawbacks:

- It does not consider the possibility that the decoded sequence maximizes the bit probability, but this is more difficult to test, and even obtaining the entire weight spectra and using the union bound does not accomplish it. The only way to determine that is to use a MAP algorithm on the concatenated scheme as a

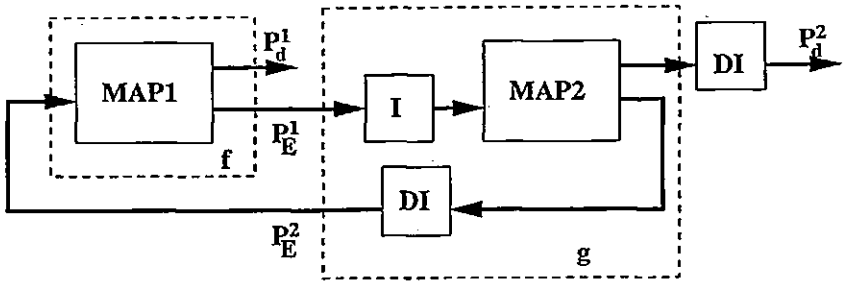


Figure 5.1: Extrinsic information in the turbo decoder

single code.

- It also does not consider the fact that the iterative decoder is not working with binary values, but with floating point values, and its output is not always an exact sequence of zeros and ones, but it has to be thresholded.

Nevertheless, it can be used to obtain new insight into the iterative decoding process, providing a new way to classify the output of the iterative decoder in: *more likely than the encoded sequence* and *not maximum likelihood* decoding.

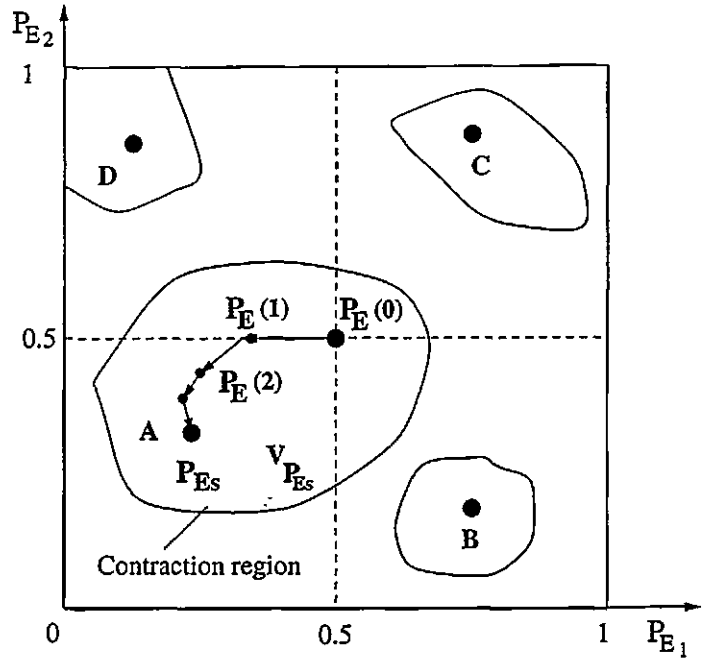
5.3 The *fixed point* interpretation

The usual mathematical method for determining the tendency of an iterative process to converge or diverge is the *fixed point* approach.

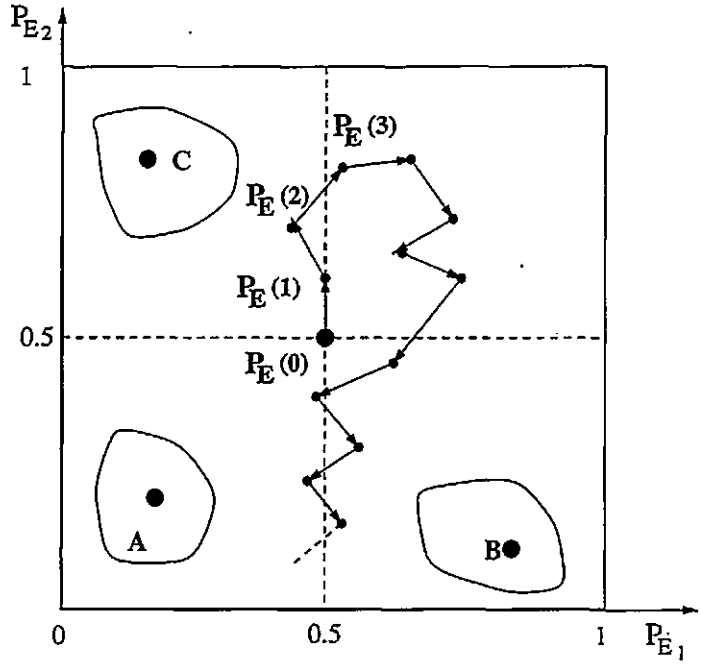
Referring to Fig. (5.1), each MAP decoder can be considered as a function acting on a probability vector $\mathbf{P_E} = (P_{E1}, P_{E2}, ..., P_{EN})$ where N is the interleaver size (block length) and $P_{Ek} = P_E\{u_k = 1\}$, $k = 1, ..., N$. That is, P_{Ek} is the probability of information bit u_k being 1 as computed from the *extrinsic* output of the MAP decoder. Starting from an arbitrary point, $\mathbf{P_E}$ may or may not converge to a solution $\mathbf{P_{Es}}$, depending upon whether or not the initial vector falls within a 'contraction region', Fig. (5.2).

Mathematically, the iterative decoding algorithm can be described as a problem of iteratively solving the equations:

$$\begin{cases} \mathbf{P_E^1} = \mathbf{f}(\mathbf{P_E^2}) \\ \mathbf{P_E^2} = \mathbf{g}(\mathbf{P_E^1}) \end{cases} \tag{5.1}$$



a) Convergent



b) Nonconvergent

Figure 5.2: Visualization of convergence (N=2)

where f and g represent the two N-dimensional MAP functions and g is considered to include the interleaving/deinterleaving process. This problem is equivalent to finding a solution for the equation

$$\mathbf{P_E^1} = \mathbf{f}(\mathbf{g}(\mathbf{P_E^1}) = \mathbf{h}(\mathbf{P_E^1}) \tag{5.2}$$

1. Function h is a *contraction* in a region $V_{\mathbf{P_{Es}}}$ of $\mathbf{P_{Es}}$, i.e. there exists a real positive number $\rho < 1$ such that $||\mathbf{h}(\mathbf{x}), \mathbf{h}(\mathbf{y})|| \leq \rho||\mathbf{x}, \mathbf{y}||$, $\forall \mathbf{x}, \mathbf{y} \in V_{\mathbf{P_{Es}}}$, where x and y are N-dimensional vectors within the contraction region. This implies that h is also N-dimensional.

2. The starting point of the iteration, i.e. the initial value of $\mathbf{P_E^1}$ belongs to $V_{\mathbf{P_{Es}}}$. In practice, this vector is initialized to $\mathbf{P_E^1} = (0.5, ..., 0.5)$.

The above conditions are met if the *norm* of the matrix

$$J_{\mathbf{h}}(\mathbf{x}) = \begin{bmatrix} \frac{\partial h_1}{\partial x_1}(\mathbf{x}) & \frac{\partial h_1}{\partial x_2}(\mathbf{x}) & \dots & \frac{\partial h_1}{\partial x_N}(\mathbf{x}) \\ \frac{\partial h_2}{\partial x_1}(\mathbf{x}) & \frac{\partial h_2}{\partial x_2}(\mathbf{x}) & \dots & \frac{\partial h_2}{\partial x_N}(\mathbf{x}) \\ \dots & \dots & \dots & \dots \\ \frac{\partial h_N}{\partial x_1}(\mathbf{x}) & \frac{\partial h_N}{\partial x_2}(\mathbf{x}) & \dots & \frac{\partial h_N}{\partial x_N}(\mathbf{x}) \end{bmatrix} \tag{5.3}$$

is less than one, $|J_{\mathbf{h}}(\mathbf{x})| < 1$.

This approach could be used to determine design rules for turbo codes in the following way:

1. The function h is determined for generic component codes and interleavers by combining the component MAP functions.

2. The norm of the matrix J is determined in a region of the encoded data vector, assuming a statistical model of the channel.

3. In the generic expression of the norm, the interleaver, component codes and noise contributions are identified, and conditions determined in order to ensure that the starting point is within the convergence region.

These steps are of impractical complexity, even for very small block lengths. It is impossible to determine the MAP functions for generic codes, and even if the parameters were fixed, (transforming the design problem into a convergence study for a

fixed system) the approach is still complex, and impractical for reasonable values of the block length N. A convergence study for values of $N \in \{1, 2, 3\}$ was presented in (McEliece et al., 1995). This study has shown that the iterative decoder may converge to the correct information vector, but also it may diverge, or converge to incorrect data.

Although this approach appears to be too difficult, it gives several qualitative ideas about the behaviour of the iterative decoder. The overall function of the iterative decoder can have several fixed points (or no fixed points). They can be repulsive or attractive. If they are attractive they have a region of attraction, with size and shape dictated by the amplitude and pattern of the noise. The iteration always starts from the center of the space, so the question is whether a contraction region encloses this point or not. If it does, the distance to the attractor will reduce *monotonically* to zero. It is difficult to determine the rate of convergence. This does not always imply that the number of errors or the distance between two successive points should reduce monotonically. If no contraction region includes the central point, then the iteration point will do a 'random' walk. It might stumble over a contraction region, and converge, or it could lock onto a closed path, and never converge.

5.4 The Cauchy criterion for convergence

A more practical approach for a realistic value of N is to consider the decoding process as an infinite array of vectors indexed by the iteration number i.e. $\mathbf{P_E^1}(1), \mathbf{P_E^1}(2), ..., \mathbf{P_E^1}(n), ...$ where

$$\mathbf{P_E^2}(n) = g(\mathbf{P_E^1}(n)) \tag{5.4}$$

The Cauchy criterion states that the array is convergent if and only if for any real positive value δ, a corresponding index n_δ can be found, so that the distance between any two vectors in the subarray starting at n_δ is less than δ,

$$||\mathbf{P_E^1}(n+p), \mathbf{P_E^1}(n)|| < \delta \quad , \quad \forall n, p \geq n_\delta \tag{5.5}$$

The Cauchy criterion is attractive because it does not require the knowledge of the convergence limit. Still, only an approximation of the criterion can be used in practice, since the number of the iterations and the data representation precision are limited. As a practical reformulation, the criterion states that an iterative process has converged when the output vector does not significantly change anymore. The usual practical test for convergence is given by the formula:

$$||\mathbf{P}_{\mathbf{E}}^{\mathbf{1}}(n+1), \mathbf{P}_{\mathbf{E}}^{\mathbf{1}}(n)|| < \delta \tag{5.6}$$

The value of δ has to be chosen for any given iterative process, depending on the expression of the distance, so that the approximation error is not significant.

5.5 Distance choice

The vectors $\mathbf{P}_{\mathbf{E}}(n)$ forming the Cauchy array for the turbo decoder have probability values as components. In defining a distance between two vectors, these values can be considered as probabilities or simply real numbers in the interval $[0, 1]$. In this section, several possible distance choices are presented, and their dependence on the number of iterations and block type is compared.

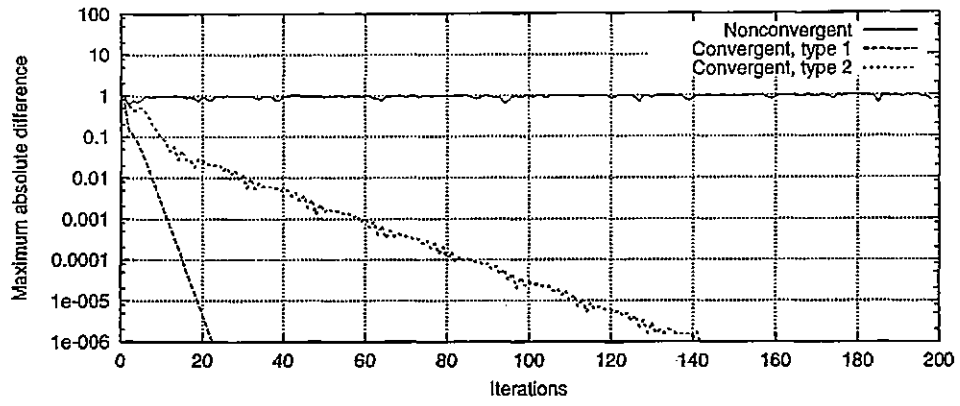
Maximum absolute difference The maximum absolute difference is given by the formula:

$$||\mathbf{P}_{\mathbf{E}}^{\mathbf{1}}(n+1), \mathbf{P}_{\mathbf{E}}^{\mathbf{1}}(n)|| = \max_{k \in \{1,..,N\}} |P_{Ek}(n+1) - P_{Ek}(n)| \tag{5.7}$$

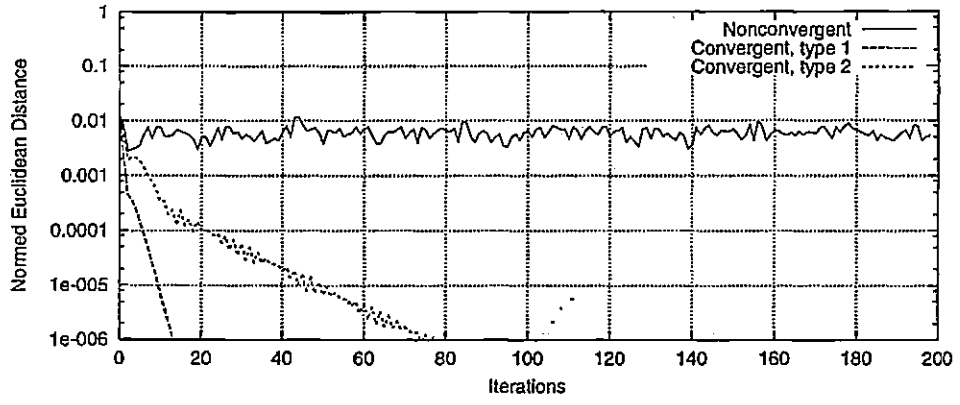
The Euclidean distance The Euclidean distance is given by the formula:

$$||\mathbf{P}_{\mathbf{E}}^{\mathbf{1}}(n+1), \mathbf{P}_{\mathbf{E}}^{\mathbf{1}}(n)|| = \sqrt{\sum_{k=1}^{N} (P_{Ek}(n+1) - P_{Ek}(n))^2} \tag{5.8}$$

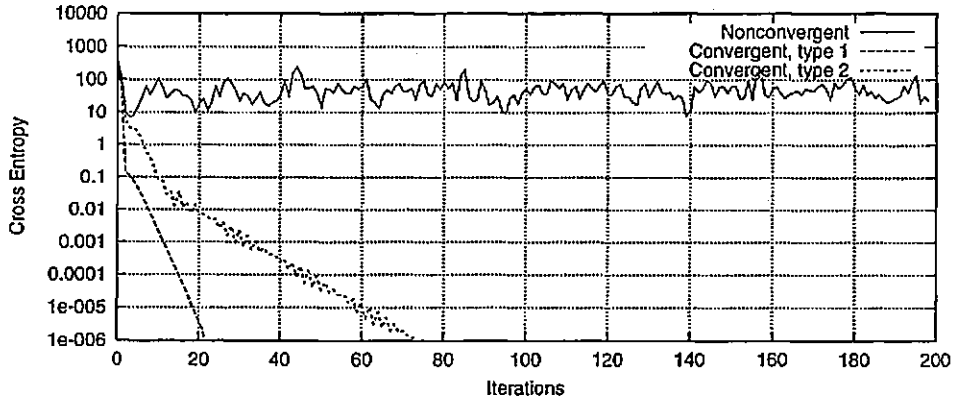
The cross entropy The cross entropy is used in (Hagenauer et al., 1996) to measure the similarity between two consecutive extrinsic information vectors. The formula for



a)



b)



c)

Figure 5.3: Distance choice
Metric dependence on number of iterations for three different type of blocks using: a) maximum absolute difference b) normed squared Euclidean metric and c) cross entropy

the cross entropy is deduced as follows:

$$||\mathbf{P}_{\mathbf{E}}^{\mathbf{1}}(n+1), \mathbf{P}_{\mathbf{E}}^{\mathbf{1}}(n)|| = \mathbf{E}_{\mathbf{P}_{\mathbf{E}}(n)} \left\{ \log \frac{\mathbf{P}_{\mathbf{E}}^{\mathbf{1}}(n)}{\mathbf{P}_{\mathbf{E}}^{\mathbf{1}}(n+1)} \right\} \tag{5.9}$$

and, assuming statistical independence,

$$||\mathbf{P}_{\mathbf{E}}^{\mathbf{1}}(n+1), \mathbf{P}_{\mathbf{E}}^{\mathbf{1}}(n)|| \approx \mathbf{E}_{\mathbf{P}_{\mathbf{E}}(n)} \left\{ \sum_{k=1}^{N} \log \frac{P_{Ek}^{1}(n)}{P_{Ek}^{1}(n+1)} \right\}$$

$$||\mathbf{P}_{\mathbf{E}}^{\mathbf{1}}(n+1), \mathbf{P}_{\mathbf{E}}^{\mathbf{1}}(n)|| \approx \sum_{k=1}^{N} \mathbf{E}_{\mathbf{P}_{\mathbf{E}}(n)} \left\{ \log \frac{P_{Ek}^{1}(n)}{P_{Ek}^{1}(n+1)} \right\}$$

$$\begin{aligned} \mathbf{E}_{\mathbf{P}_{\mathbf{E}}(n)} \left\{ \log \frac{P_{Ek}^{1}(n)}{P_{Ek}^{1}(n+1)} \right\} &= P_{Ek}^{1}(n) \log \frac{P_{Ek}^{1}(n)}{P_{Ek}^{1}(n+1)} + \\ & (1 - P_{Ek}^{1}(n)) \log \frac{(1 - P_{Ek}^{1}(n))}{(1 - P_{Ek}^{1}(n+1))} \end{aligned} \tag{5.10}$$

As opposed to the first two distances, the cross entropy is a probabilistic measure of the similarity between two extrinsic information vectors.

The variation of the three distances against the number of iterations for different types of blocks is shown in figure (5.3). The turbo code uses an $RSC(5/7)$ component encoder with a block length $N = 500$. The comparison shows that the three distances behave in a similar way.

5.6 Convergence evaluation

Using the Cauchy criteria, the performance of the iterative decoder has been split into two parts: a part due to non-convergent blocks and a part due to convergent blocks. For this separation, equation (5.6) has been used, with $\delta = 10^{-5}$ and a maximum number of iterations $nit = 200$. The blocks declared convergent were further checked with $\delta = 10^{-20}$ and $nit = 2000$ maximum iterations. Generally, the high number of iterations is not needed, since the distance quickly reduces to zero. The overall performance is the sum of the two parts.

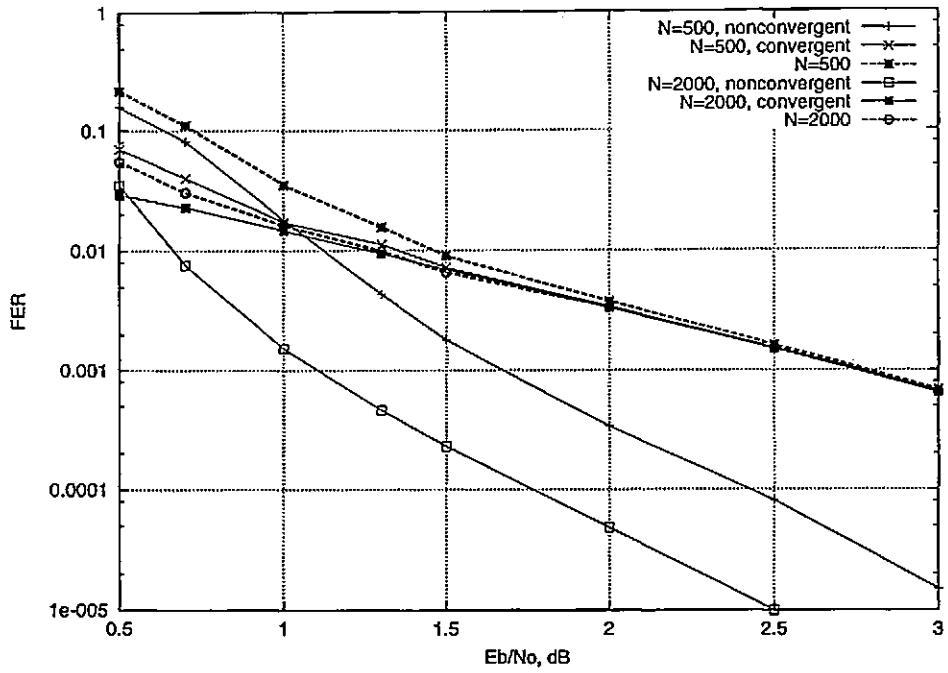


Figure 5.4: Convergence dependence on block length for turbo codes
Dependence of convergent/non-convergent FER on block length for a $RSC(5/7)$ turbo code

5.6.1 Turbo codes

The interleaver

Figure (5.4) shows the effect of increasing the interleaver size upon the two parts of the turbo code performance. The turbo code scheme uses the $RSC(5/7)$ $M = 2$ optimal component code, and the simulations were run for block lengths $N = 500$ and $N = 2000$. It can be observed that the non-convergent curves start by dominating the performance at low E_b/N_o (especially for the short interleaver) and then decrease much quicker than the convergent part. Increasing the interleaver length improves the non-convergent FER, but not the convergent curve. The convergent curve behaves similar to the ML performance for the given component codes which does not improve with interleaver length (the two codes have the same d_{free}, due to high $IW = 2$ mapping probability, independent of block length). It can be observed that, although the convergent curve dominates the high E_b/N_o part of the graph, a small number of non-convergent blocks are still present at $E_b/N_o = 3$dB.

The fact that the interleaver can be designed to reduce the number of non-convergent

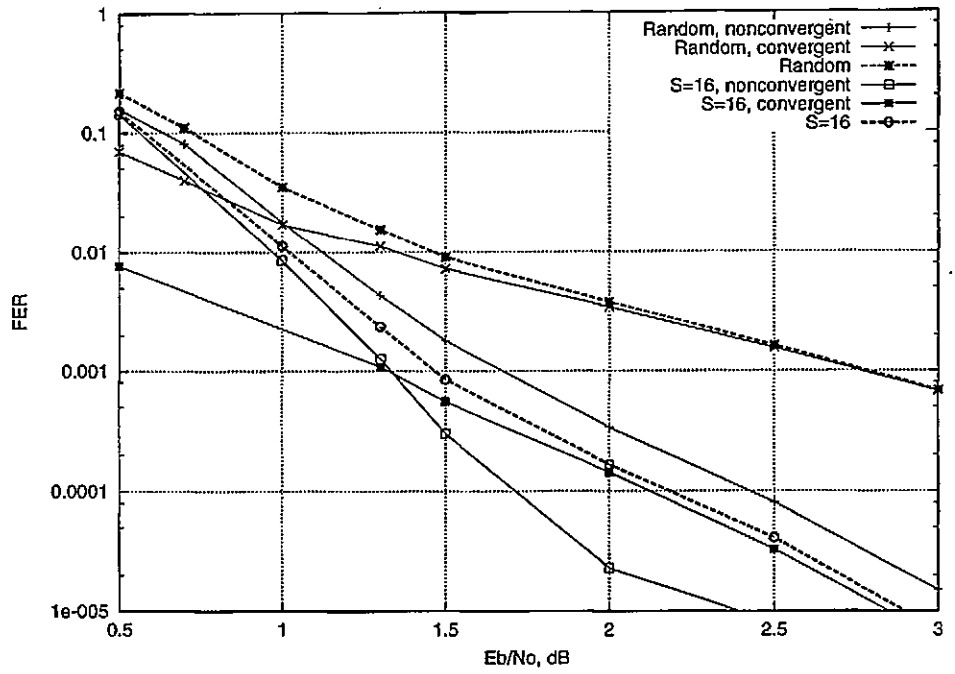


Figure 5.5: Convergence dependence on interleaver type for turbo codes
Dependence of convergent/non-convergent FER on interleaver type for a $RSC(5/7)$, $N = 500$ turbo code

blocks is shown in figure (5.5). The convergent/non-convergent curves are shown for a turbo code using a randomly chosen interleaver and a turbo code using an S-type interleaver with $S = 16$. Both codes have length $N = 500$ and use $RSC(5/7)$ component codes. It can be observed that using the S-type interleaver improves both the convergent and nonconvergent part of the FER curve as compared to the randomly chosen interleaver. This is usually explained by the fact that the S-type interleaver tends to break local correlations better than the randomly chosen interleaver, by interleaving bits in a group of length S further apart. Note that the improvement of the non-convergent curve is more significant as the E_b/N_o is increased: the S interleaver can only break dependencies with length smaller than a certain value.

Code memory

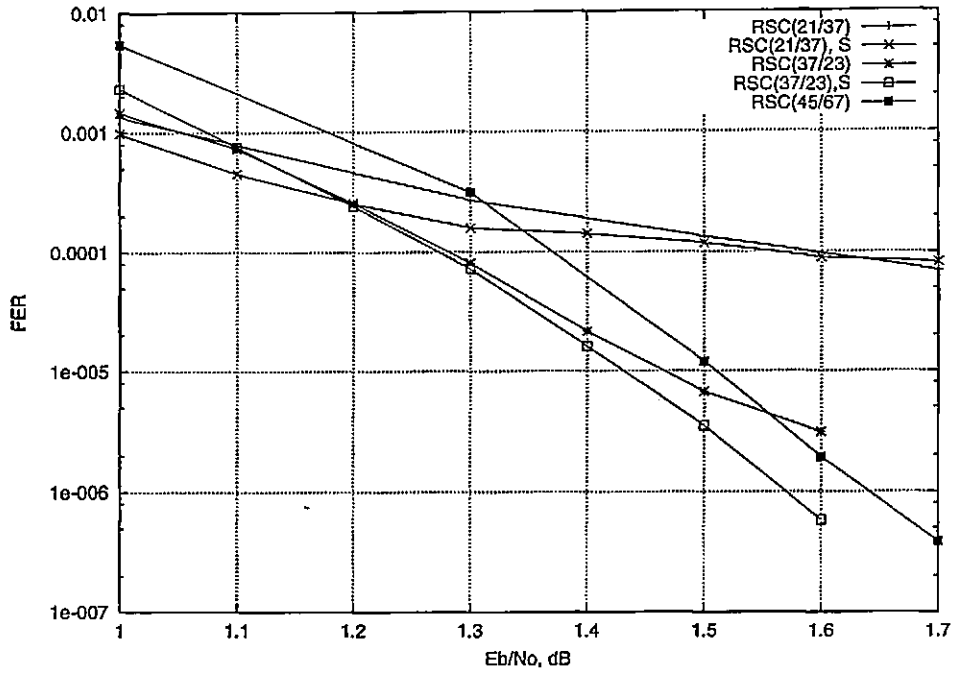
Experiments in previous chapters have shown that increasing the code memory whilst using 'optimal' component codes can drastically lower the error floor characteristic to turbo codes. Nevertheless, at low E_b/N_o, a degradation in performance can be observed as the memory is increased above $M = 3$. Also, non-optimal component

codes of $M = 4$ have been observed to perform better than their optimal opponents at low E_b/N_o, whilst having a pronounced error floor at high E_b/N_o, caused by a low d_{free}.

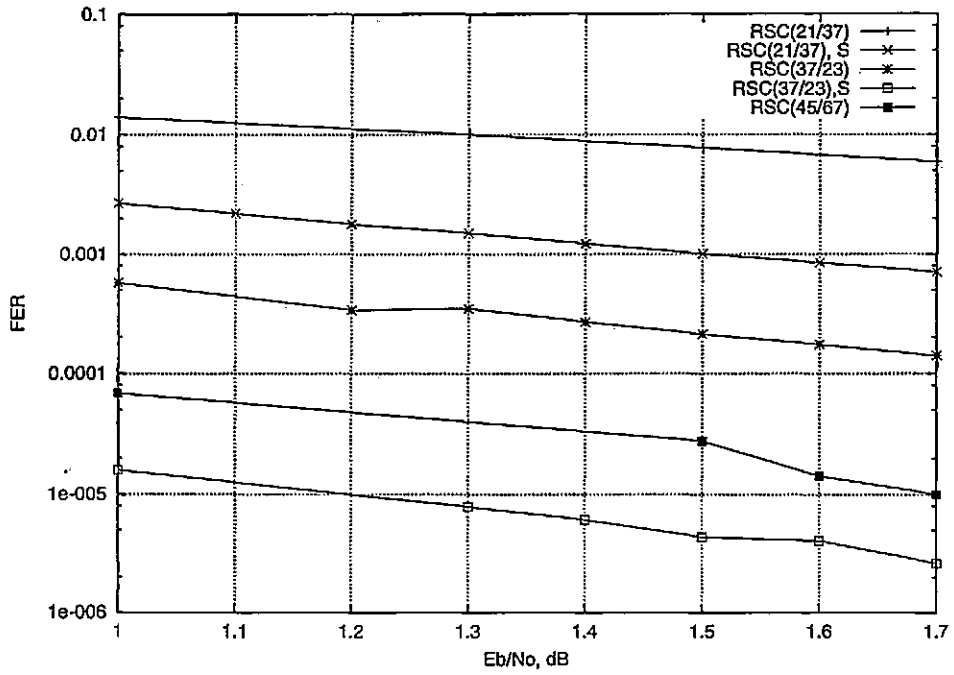
The effect of using different component codes is shown in figure 5.6(a) for non-convergent frame error rate, 5.6(b) for convergent frame error rate. Their corresponding bit error rate curves are shown in figure 5.7(a) (non-convergent) and 5.7 (b) (convergent). It can be observed that the non-convergent curve for non-optimal $M = 4$, $RSC(21/37)$ codes is better than the one for optimal $M = 4$, $RSC(37/23)$ codes, and their performance is dominated by the convergent part. Also, it is rather flat, as opposed to that of the optimal code which intersects it at 1.2dB. The performance of the optimal code is dominated at low E_b/N_o by the non-convergent part, but it drops much quicker than the convergent part, which dominates the high E_b/N_o performance. The non-convergent performance of these codes is not significantly improved by using an S interleaver, as opposed to their convergent part. This can be explained by the fact that they have longer error bursts that cannot be broken by the S interleaver. The performance of a $M = 5$, $RSC(45/67)$ code is also shown. Its performance at low E_b/N_o is also dominated by the non-convergent curve, which is higher than that of the $M = 4$ codes. The convergent performance improves with code memory.

The overall performance of the non-optimal $M = 4$ code is worse than that of the optimal $M = 4$ code in terms of frame error rate. This situation changes in terms of bit error rate. This is due to the fact that the convergent blocks that dominate its performance have low information weight, as opposed to high information weight in the case of the optimal code.

A detailed iterative/union bound comparison is shown in figure 5.8(a) for the $RSC(37/23)$ turbo code and figure 5.8(b) for the $RSC(21/37)$ turbo code. The block length is $N = 500$. In the case of the turbo code using the $RSC(37/23)$ (optimal) component code, the convergent curve is relatively close to the bound, but higher. Since the union bound has been calculated up to $d_{MAX} = 22$, to obtain a better comparison, the error events with $OW \geq 23$ have been eliminated from the convergent curve, obtaining the "convergent, $OW < 23$" curve in figure 5.8(a). This curve is still higher than the union bound. This could be explained by analysing the likelihood of the decoded sequence as opposed to the correct sequence, as presented in section 5.2.

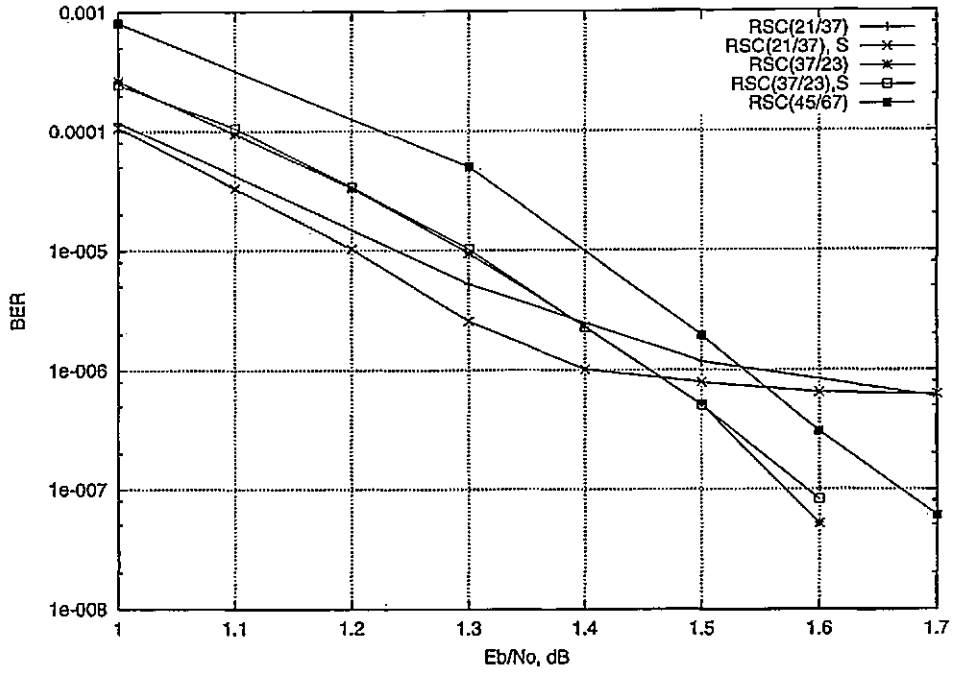


a)

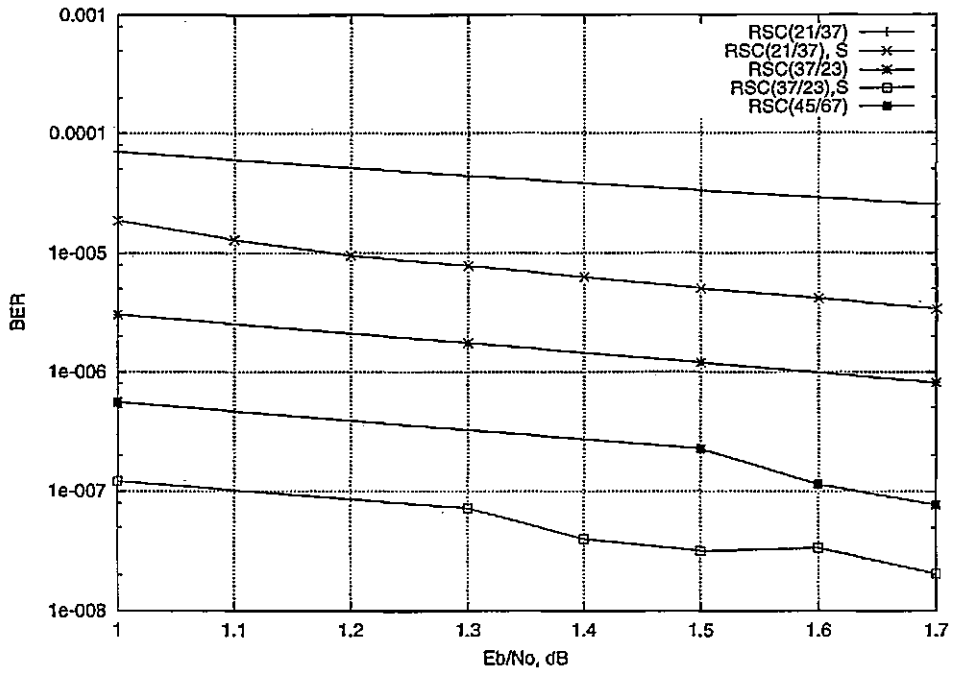


b)

Figure 5.6: FER convergence for turbo codes with different component codes a) non-convergent and b) convergent FER for turbo codes with $N = 500$ and different memory component codes



a)



b)

Figure 5.7: BER convergence for turbo codes with different component codes a) non-convergent and b) convergent BER for turbo codes with $N = 500$ and different memory component codes

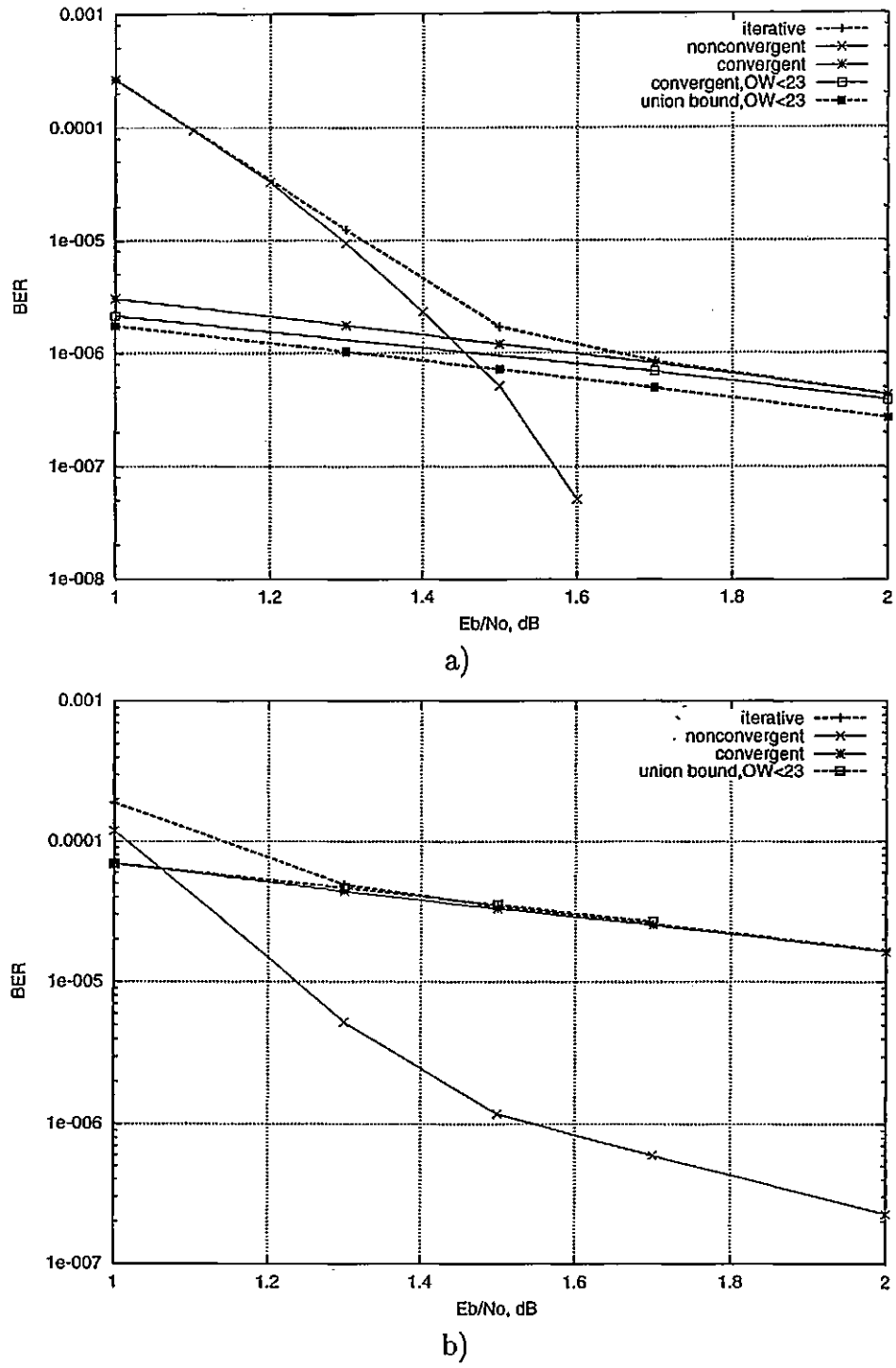


Figure 5.8: Iterative vs union bound performance
Iterative decoding vs union bound comparisons for a) $RSC(37/23)$ and b) $RSC(21/37)$. Block length is $N = 500$.

About half of the convergent decodings are more likely than the correct sequence. The other half are not overall more likely, but they are more likely for one of the component codes. There are more error sequences that are considered more likely only by the first code than error sequences that are considered more likely only by the second code. This can be explained by analysing the $(IW = 3, OW = 15)$ error event that causes most of the errors in the high E_b/N_o region. The first code contributes with a parity of 5 to this error event, whereas the second contributes with a parity of 7 and thus the first code is more likely to make errors than the second code, correlative with the observations. It is just possible that, because a sequence is very likely for one of the codes, this code will "convince" the other code that it is the right sequence, under the condition that the sequence is not very unlikely for the second code. This causes a marginal difference from the union bound curve, which is interesting from the point of view of the iterative decoder.

In the case of the turbo code using the $RSC(21/37)$ (suboptimal) component code, it can be observed that the convergent curve is very close to the bound, and thus the observed difference between the convergent curve and the bound depends on the code structure.

By applying the "more likelihood" argument, it has been observed that the HIWHOW error events which constitute the bulk of the nonconvergent performance are less likely *than the correct sequence, and thus they are* not *maximum likelihood decodings.*

The distribution of the information weight of the nonconvergent blocks for different codes is shown in figure 5.9(a) for E_b/N_o = 1dB and 5.9(b) for E_b/N_o = 1.3dB. Although the number of non-convergent blocks can decrease with code memory, the information weight increases on average for the non-convergent blocks. The number of non-convergent blocks reduces with increasing E_b/N_o from E_b/N_o = 1dB to E_b/N_o = 1.3dB, but their size does not. The information weight distribution of convergent blocks is shown in figure 5.9(c) for E_b/N_o = 1dB and 5.9(d) for E_b/N_o = 1.3dB. The convergent error blocks are low information weight blocks.

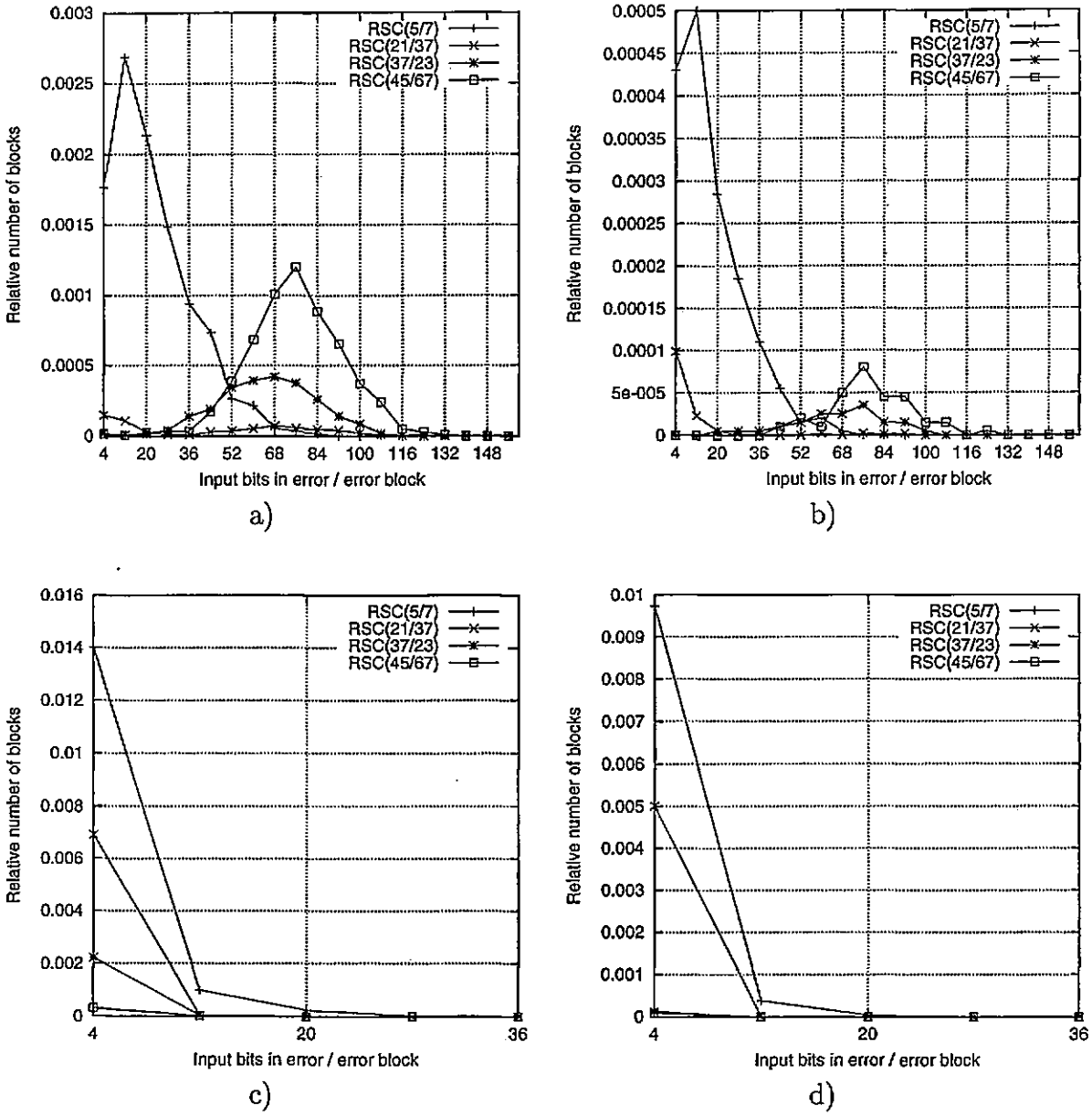


Figure 5.9: Number of errors/block for turbo codes
Distribution of the information weight of the error blocks for component codes with memory $M \in \{2,..,6\}$ for a turbo code using an N=500, S=16 'S' interleaver for a) non-convergent blocks at $E_b/N_o = 1$dB, b) non-convergent blocks at $E_b/N_o = 1.3$dB, c) convergent blocks at $E_b/N_o = 1$dB and d) convergent blocks at $E_b/N_o = 1.3$dB.

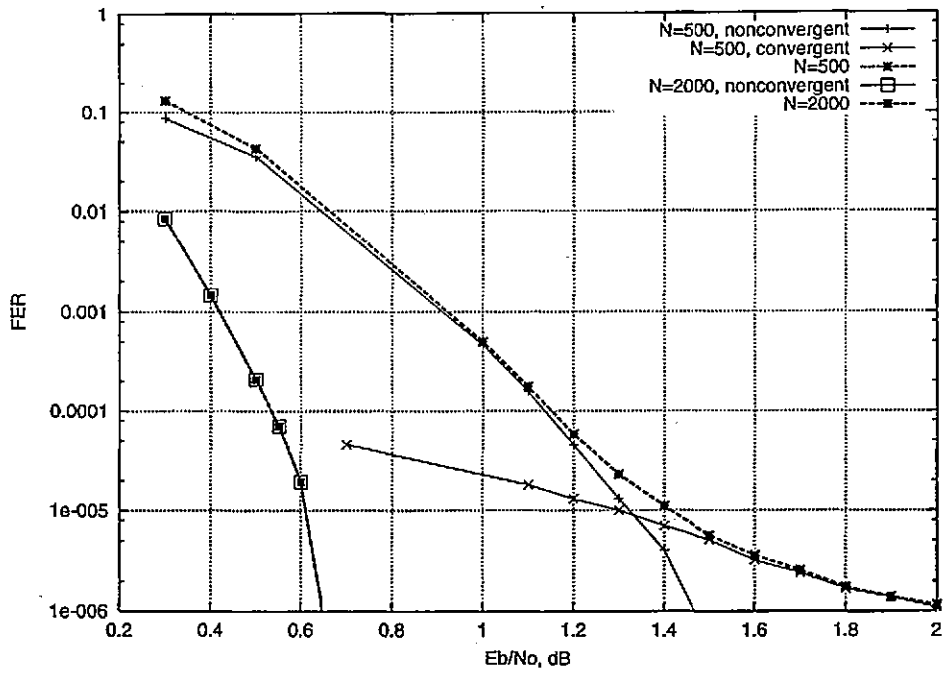


Figure 5.10: Convergence dependence on block length for 3PCCC
Dependence of convergent/non-convergent FER on block length for a $RSC(5/7)$ 3PCCC

5.6.2 Multiple Parallel Concatenation

The interleaver

The number of convergent blocks for the 3PCCC scheme is also improved by increasing the interleaver length, as shown in figure (5.10). The experiments have been performed for 3PCCC schemes employing $M = 2$, $RSC(5/7)$ as component codes. As opposed to turbo codes, the convergent part improves with interleaver length, similar to their ML performance. Also, the performance of 3PCCC schemes is dominated by nonconvergent blocks in the whole simulation range, except for short interleavers that still show an error floor due to a (relatively) low d_{free}.

Figure (5.11) shows the performance graphs for two $N = 500$, $RSC(5/7)$ 3PCCC schemes, one using a randomly chosen interleaver pair and the other using two paired S-type interleavers. It can be observed that using the S-type interleaver could improve the convergent curve, but it slightly degrades the non-convergent curve.

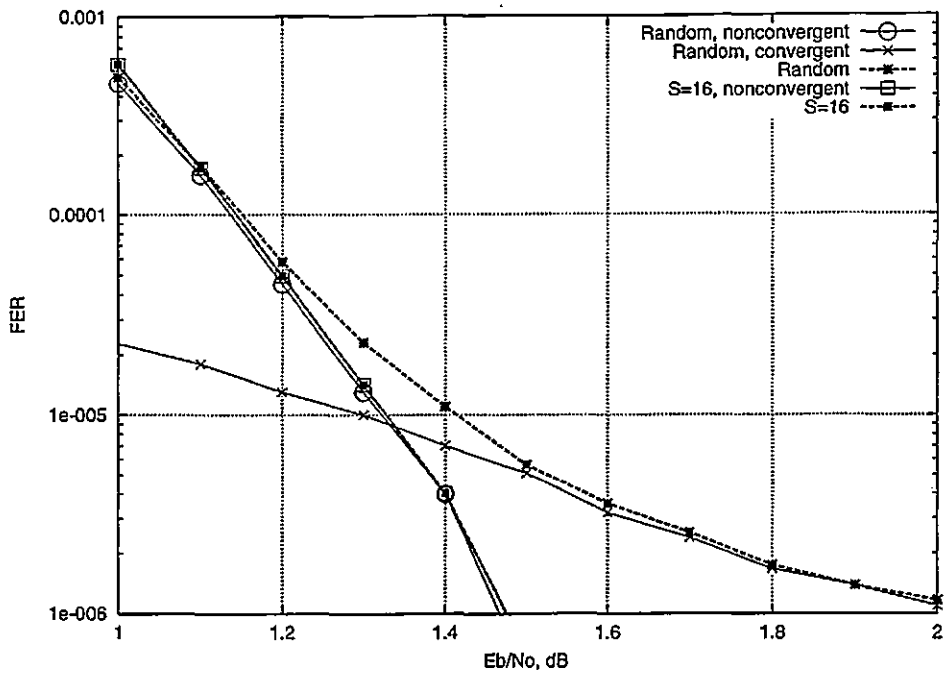


Figure 5.11: Convergence dependence interleaver type for 3PCCC
Dependence of convergent/non-convergent FER on interleaver type for a $RSC(5/7)$ 3PCCC

Code memory

The performance of $N = 500$, 3PCCC schemes using different component codes is shown in figure (5.12) for (a) non-convergent and (b) convergent FER and figure (5.13) for (a) non-convergent and (b) convergent BER. It can be observed that the worse behaviour of the $M = 3$ optimal code is due to non-convergent blocks, which dominate its performance in the simulation range. The performance of the non-optimal, $M = 2$, $RSC(7/5)$ code is better in terms of non-convergent FER (BER) than that of the optimal $M = 2$, $RSC(5/7)$ code, but worse in terms of convergent FER (BER). The non-convergent performance of the non-optimal code can be improved using S-type interleavers.

The distribution of the information weight of the block errors for different $N = 500$, 3PCCC schemes with different component codes is shown in figure (5.14) for (a) non-convergent error blocks at $E_b/N_o = 1$dB, (b) non-convergent error blocks at $E_b/N_o = 1.3$dB, (c) convergent error blocks at $E_b/N_o = 1$dB, (d) convergent error blocks at $E_b/N_o = 1.3$dB. It can be observed that the weight of the non-convergent error blocks increases as compared to turbo codes. It also increases with increasing code memory.

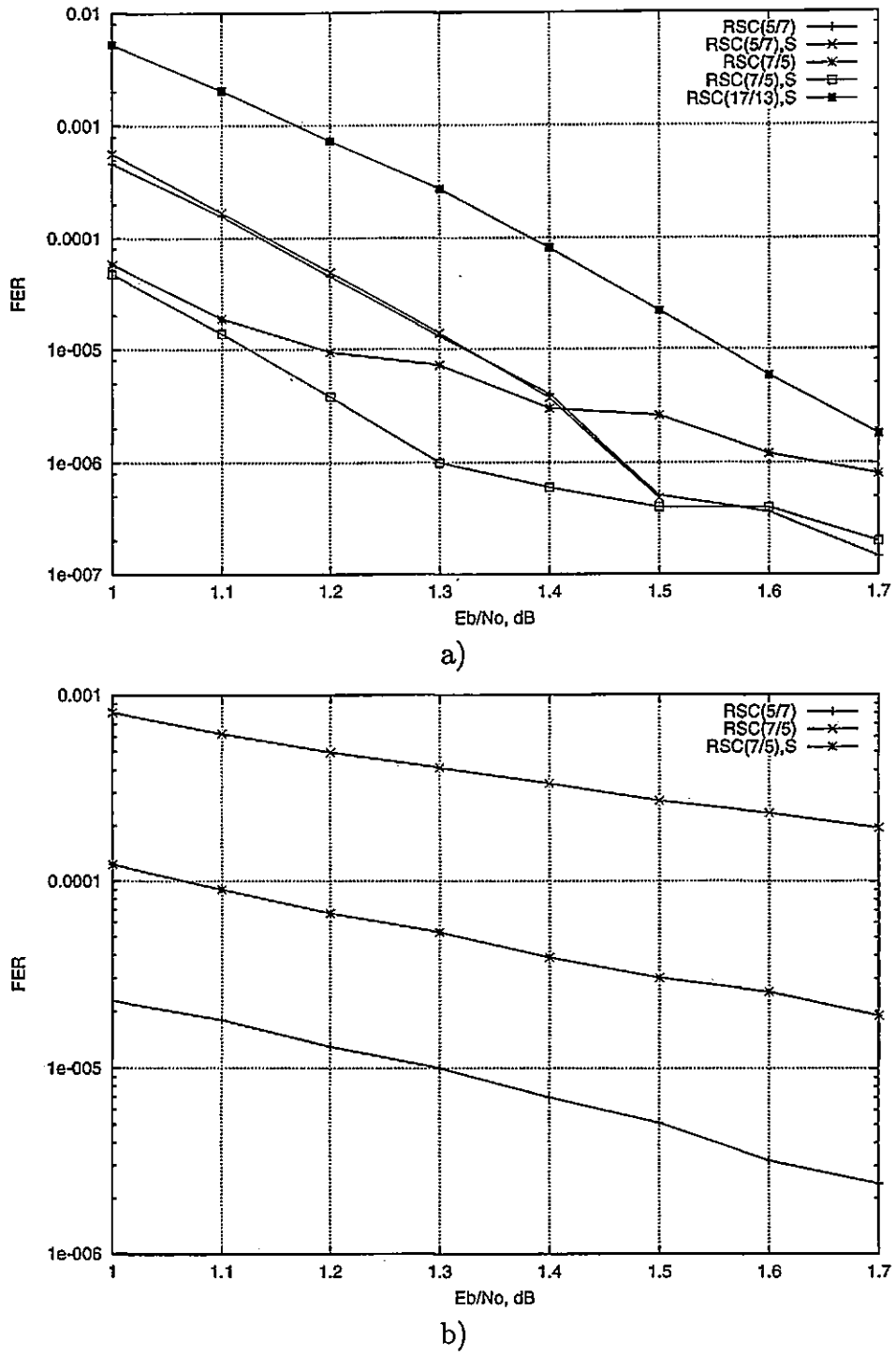


Figure 5.12: FER convergence for 3PCCC with different component codes
a) non-convergent and b) convergent FER for 3PCCC with $N = 500$ and different component codes

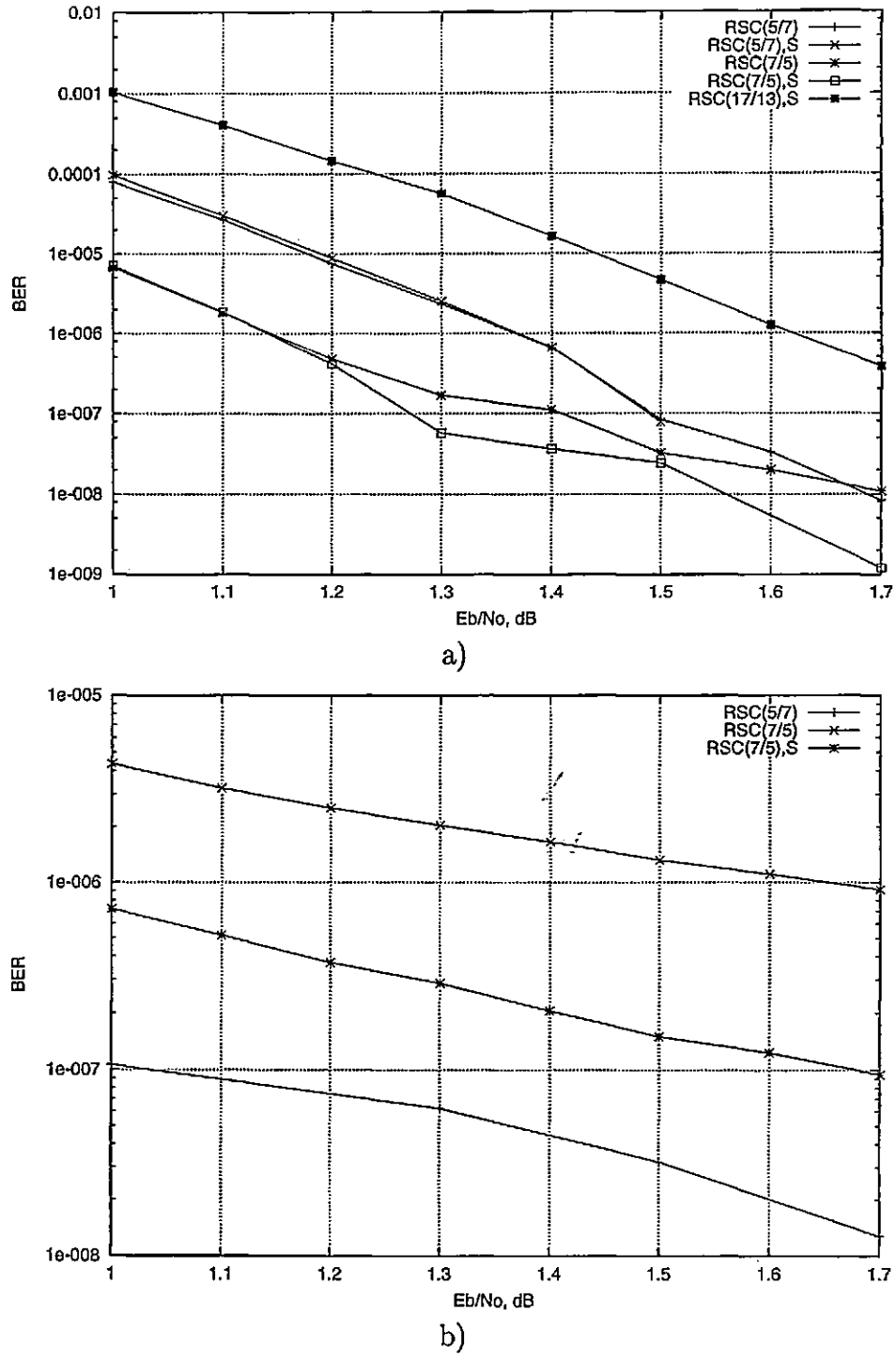


Figure 5.13: BER convergence for 3PCCC with different component codes
a) non-convergent and b) convergent BER for 3PCCC with $N = 500$ and different component codes

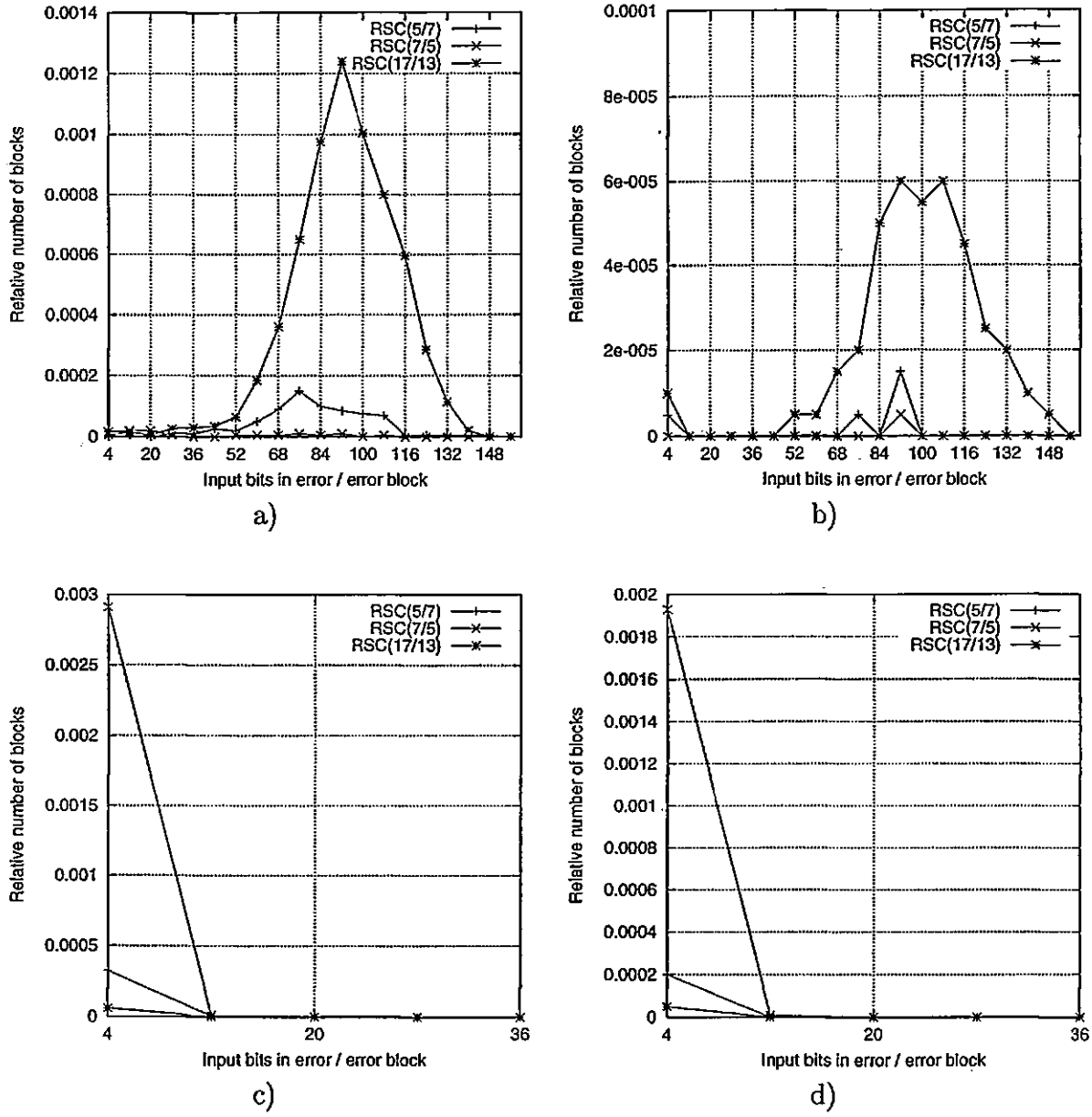


Figure 5.14: Number of errors/block for 3PCCC
Distribution of the information weight of error blocks for a 3PCCC with $N = 500$ and component codes with $M \in \{2, 3\}$ for a) non-convergent blocks at $E_b/N_o = 1$dB, b) non-convergent blocks at $E_b/N_o = 1.3$dB, c) convergent blocks at $E_b/N_o = 1$dB and d) convergent blocks at $E_b/N_o = 1.3$dB.

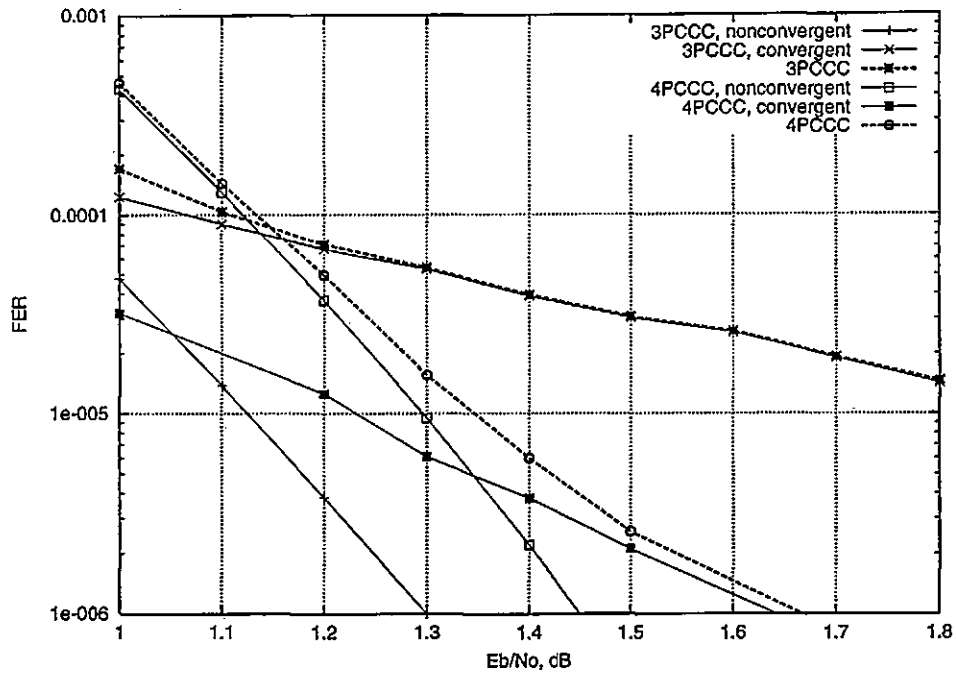


Figure 5.15: 3PCCC/4PCCC convergence comparisons
3PCCC/4PCCC convergent/non-convergent performance comparisons for $N = 500$, $RSC(7/5)$ non-optimal code

The non-optimal $M = 2$, $RSC(7/5)$, 3PCCC has very few high information weight non-convergent error blocks. The convergent performance of all codes is composed of low information/code weight blocks, of which the non-optimal code $M = 2$, $RSC(7/5)$ code has the highest number, and the $M = 3$, $RSC(17/13)$ code the lowest number. These blocks can be associated with ML error events, which show that ML performance improves with increasing memory, but is masked by the presence of non-convergent error events.

Increasing the number of codes

Figure (5.15) presents the FER comparisons for an $N = 500$, $RSC(7/5)$ 3PCCC scheme using an paired S-interleavers and an $N = 500$, $RSC(7/5)$ 4PCCC scheme using randomly chosen interleavers. It can be observed that whilst the performance of the 3PCCC scheme is dominated by the convergent block errors, the 4PCCC scheme has an crossing point, being dominated by non-convergence at E_b/N_o values below 1.3dB, and by convergent block errors over this value. Although the convergent curve for 4PCCC is always more than one order of magnitude better than the one for the

3PCCC scheme, the overall performance is worse at low E_b/N_o due to worse convergence. Thus, although the convergent performance of these schemes is improved as opposed to the 3PCCC schemes, their iterative decoding performance is degraded. This could be explained by the fact that the component codes work at a lower signal to noise ratio in the first iterations, due to the decreased code rate of 4PCCC schemes. The improvement in the convergent performance is to be expected since the convergent curve is associated with the ML performance of the codes.

5.6.3 Serial Concatenation

The performance of the serial concatenated codes is completely dominated by non-convergence. The only convergent error blocks observed are produced by schemes using non-optimal $M = 2$, $RSC(7/5)$ codes, and generally have a high code weight as compared to the 3PCCC schemes employing randomly chosen interleavers. These blocks totally disappear as the block length N is increased from $N = 500$ to $N = 2000$. The performance of the non-optimal codes, although it has a convergent component, is still dominated by the non-convergent blocks, which have a higher information weight than in the case of 3PCCC schemes. The distribution of the information weight of the non-convergent blocks for several component codes is presented in figure 5.16(a) for $E_b/N_o = 1$dB and 5.16(b) for $E_b/N_o = 1.3$dB. Similar to the other schemes, the improvement in non-convergent performance as the E_b/N_o is increased is caused by a reduction in the number of error blocks, rather than in information weight (number of errors/block).

5.6.4 Comparisons

A comparison of the three schemes for $N = 500$ is shown in figure (5.17). The component codes employed are optimal, $M = 4$, $RSC(37/23)$ codes for PCCC and $M = 2$, $RSC(5/7)$ codes for 3PCCC,SCCC. The SCCC scheme has the worse performance due to its lack of convergence, but it intersects the PCCC scheme when it starts showing the characteristic error floor, caused by its convergent component. The performance of the 3PCCC scheme is also dominated by non-convergent blocks, but is better than that of the other schemes (for the $M = 2$ code).

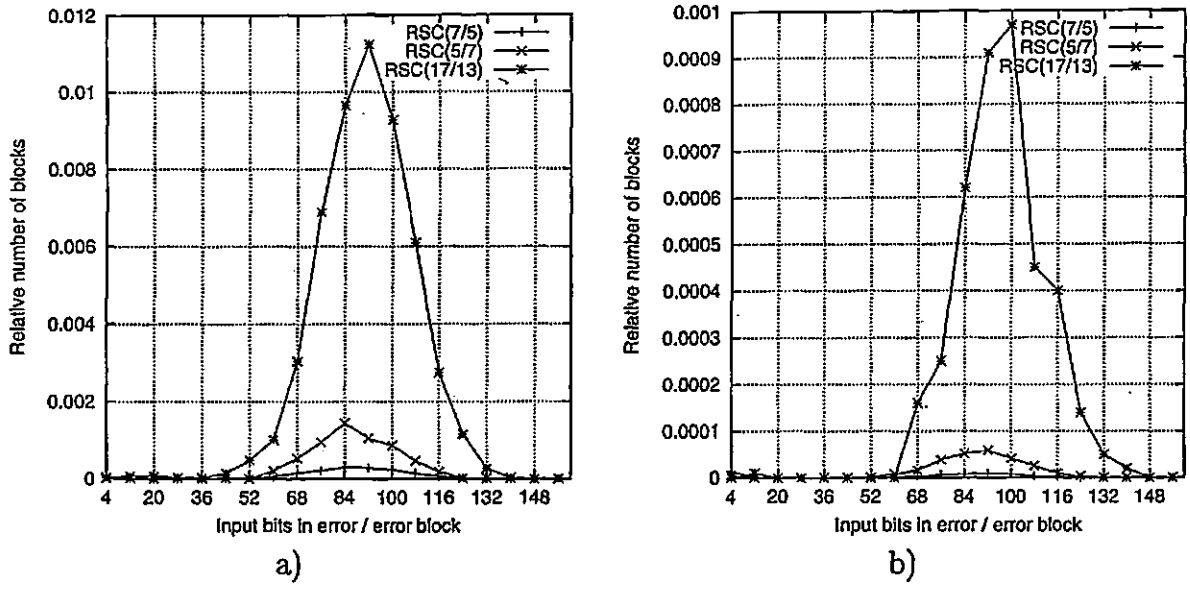


Figure 5.16: Number of errors/block for SCCC
Distribution of the information weight of non-convergent error blocks for component codes with memory $M \in \{2, 3\}$ for an SCCC with $N = 500$ at a) $E_b/N_o = 1$dB and b) $E_b/N_o = 1.3$dB.

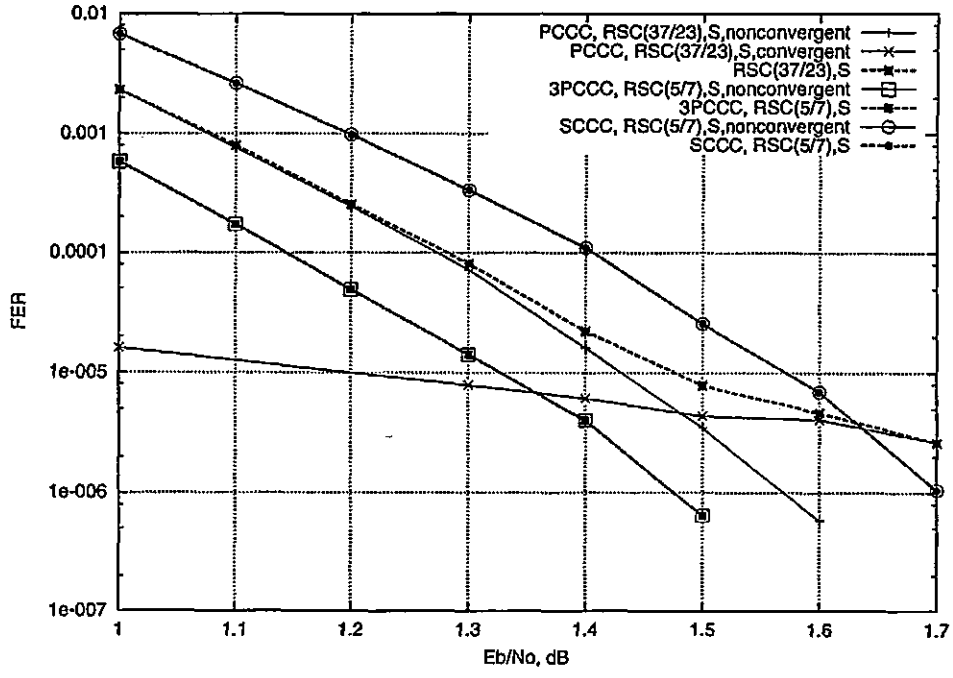


Figure 5.17: Convergence comparisons for different schemes

The 3PCCC and SCCC schemes are more complex schemes, introduced as an alternative to turbo codes, in order to improve their performance with block length, and to decrease their error floor. The arguments for introducing these schemes is based on a probabilistic, union bound approach, which assumes a ML decoder at the receiving end. Due to their increased complexity, their convergence degradation with code memory is much quicker, resulting in the fact that these schemes are not always better than turbo codes in approaching the Shannon limit, although their weight spectra is improved. While the 3PCCC schemes can improve the performance of turbo codes at low E_b/N_o, this is not the case for the SCCC schemes. The situation changes completely at high E_b/N_o, where the convergence of these schemes is improved, and their ML advantage shows up in the large reduction of the error floor.

5.7 Decoded block types

The decoded blocks were classified as convergent or nonconvergent using the criterion in (5.5) and typical distance results are shown in Fig. (5.3). Due to the linearity of the code, simulations can be performed by transmitting the all zeros information sequence, which means that $P_{dk} = 1$ at the decoder output represents a bit error. For any erroneous block, the information weight (number of data errors/block) and the code weight can be calculated, the latter being obtained by re-encoding the decoded data sequence. In this way, any decoded block can be associated with an information weight and code weight. The identification of low code weight blocks is useful for estimating d_{free}, and if the iterative decoder performance is compared with the expected maximum likelihood performance determined by the union bound.

5.7.1 Convergent blocks

The convergent blocks can be further classified in

- Type 1: blocks for which vectors $\mathbf{P}_{\mathbf{Es}}^{\mathbf{1}}$ and $\mathbf{P}_{\mathbf{Es}}^{\mathbf{2}}$ have values close to 0 and 1 (saturation). In this case it can be shown that they are identical.

- Type 2: blocks for which the two limit vectors are non-saturated but stable, as in (5.5). In this case they are generally different.

An example of a Type 1 block is shown in Fig. (5.18) and it represents the limit of the extrinsic information vectors $\mathbf{P}_{\mathbf{E}}^{1}(n)$ and $\mathbf{P}_{\mathbf{E}}^{2}(n)$, for a specified value of δ. Simulation shows that this type of block generally has low information/code weight, similar to what would be expected in ML sequence decoding for a given E_b/N_o. The example shown corresponds to an erroneous block with information weight 2 and code weight 18, and the latter corresponds to the d_{free} of the turbo code used in the simulation. Type 1 error events appear at intermediary and high E_b/N_o. There exists an E_b/N_o threshold under which these blocks become nonconvergent. This limit is dependent mainly on block size. A special case of this type of decoded block is one that decodes with zero error.

An example of a Type 2 decoded block is given in Fig. (5.19) and clearly the probability vectors are not saturated. This particular example corresponds to a block with a decoded information weight of 3 and code weight of 292. The low information weight Type 2 blocks appear at intermediary and high E_b/N_o. They could be associated with *bitwise* ML error blocks. They are nonrepetitive and difficult to identify. The result can be explained by the fact that the MAP decoders inherently minimize the probability of bit error, rather than sequence error. Also, a special kind of low information weight Type 2 errors are limit cycle blocks with limited extrinsic information.

From the above examples, two types of behaviour can be identified for the extrinsic information vector $\mathbf{P}_{\mathbf{E}}$. For Type 1 blocks, the number of decoded bit errors coincides with the number of ones in $\mathbf{P}_{\mathbf{E}}$, whereas for Type 2 blocks there are only 3 bit errors for a relatively erroneous extrinsic vector. For Type 1 blocks, $\mathbf{P}_{\mathbf{E}}$ is decided with high probability and so it dominates the decoding process in the last iterations. For Type 2 blocks, the probability vectors are not saturated and so decoding is a compromise between channel values and extrinsic information values.

5.7.2 Nonconvergent blocks

Aperiodic blocks

The variation of the number of errors for an aperiodic error block with iteration is shown in figure (5.20). The block is nonconvergent at $E_b/N_o = 1\text{dB}$. As the E_b/N_o is increased, the block converges and the number of iterations reduces with E_b/N_o.

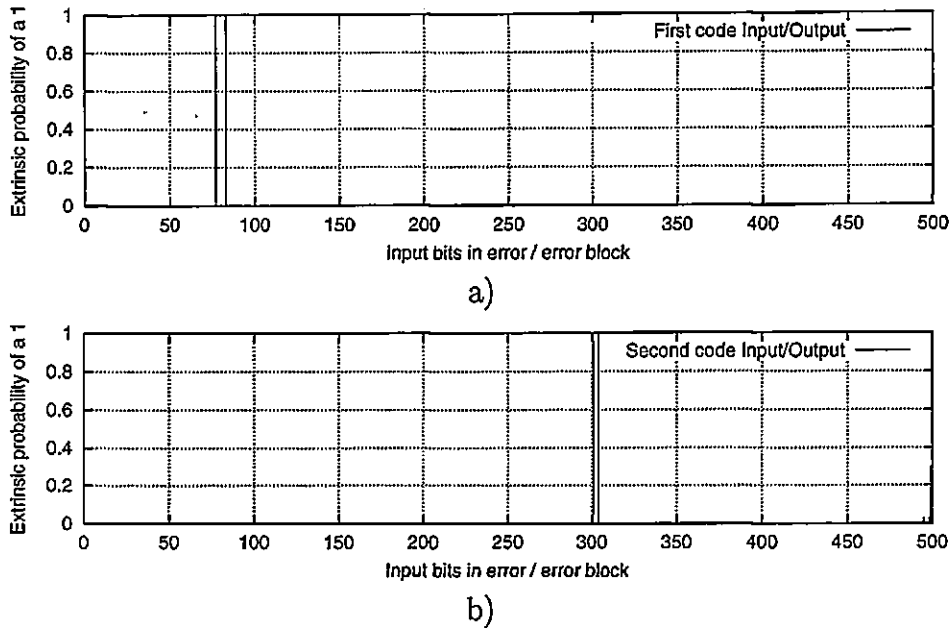


Figure 5.18: Extrinsic information limit for type 1 convergent blocks
Extrinsic information limit for (a) MAP1 and (b) MAP2 (Type 1 decoded block, N=500)

The number of errors produced by a nonconvergent block depends on the component code. The information/code weight of these blocks is usually high (they are HIWHOW error blocks). A small number of nonconvergent blocks with low information/code weight have also been observed. Generally, they are observed at low E_b/N_o, producing the nonconvergent region of the error rate curves. As the E_b/N_o is increased, the number of errors in a block reduces slowly, until it reaches a limit where the block suddenly converges. Aperiodic blocks are sensitive to data precision, and sometimes converge when data precision is increased. Also, they can converge after a long number of iterations, abruptly, a fact that indicates that they have slowly drifted into a convergence region. As shown in previous sections, the interleaver could be chosen to reduce their number at intermediary E_b/N_o. It is believed that the choice of the interleaver does not matter at low E_b/N_o, fact attributed to the impossibility of the interleaver to break 'dependencies' which are too long, due to their limited length. As expected, this improves with interleaver length.

Limit cycle blocks

They can be divided into two types: periodic blocks and quasi-periodic blocks.

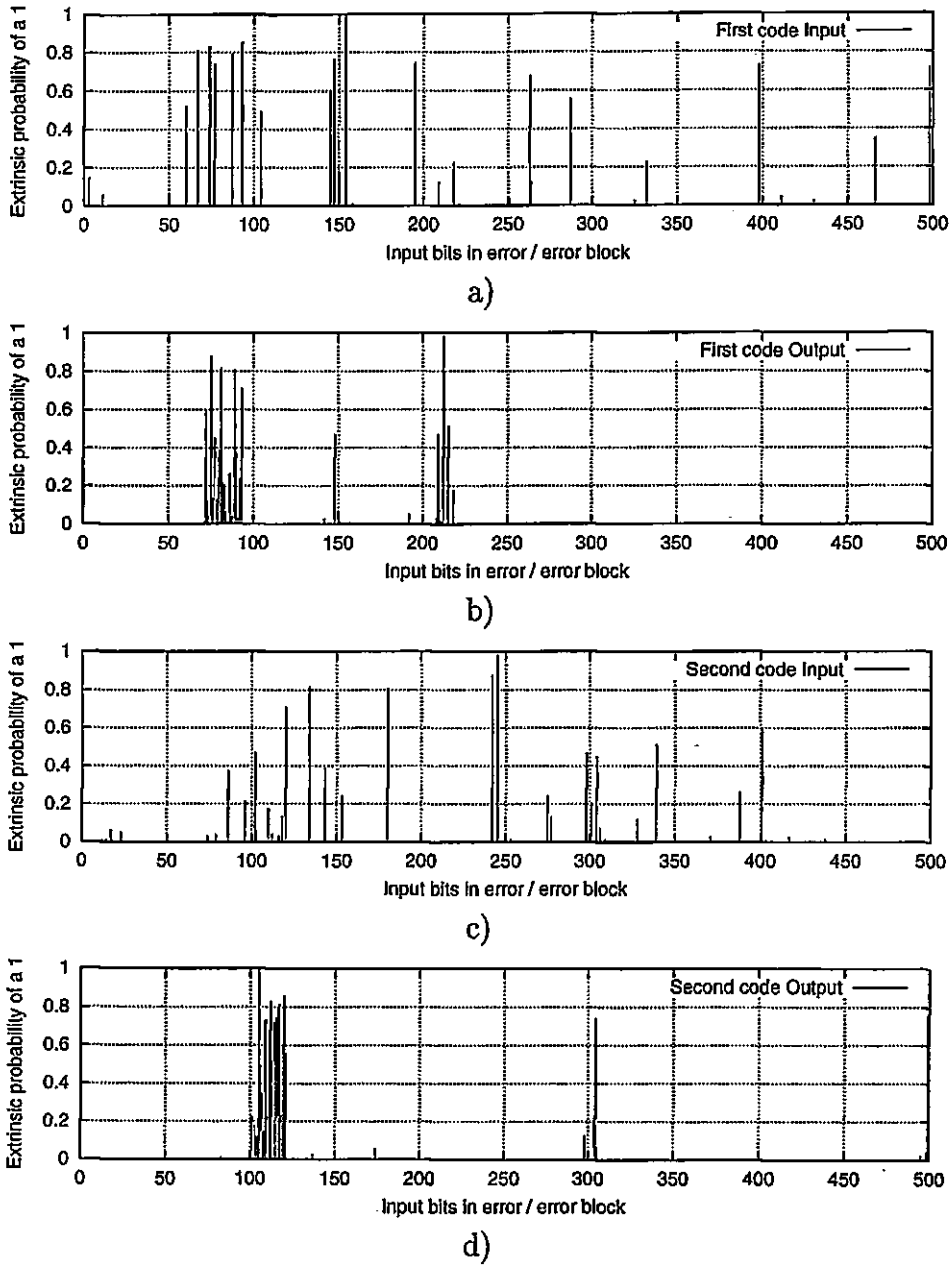


Figure 5.19: Extrinsic information limit for type 2 convergent blocks Extrinsic information limit for MAP1: (a) input and (b) output and MAP2: (c) input and (d) output (Type 2 decoded block, N=500)

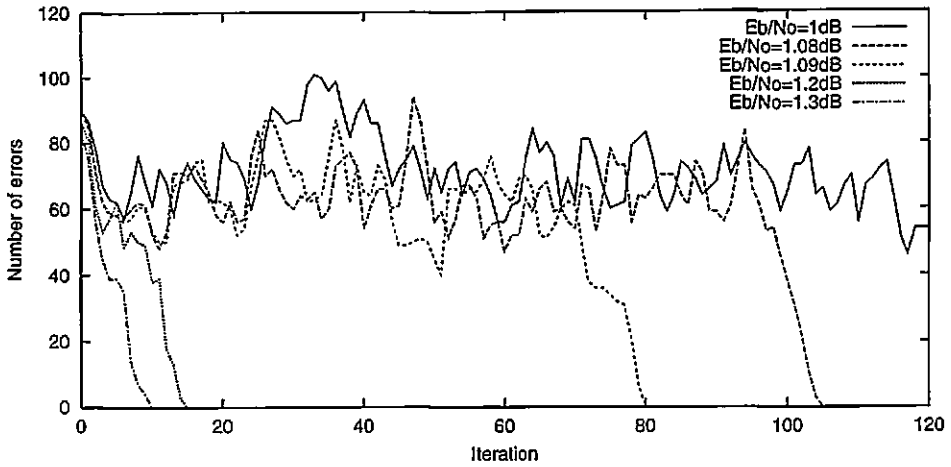


Figure 5.20: Aperiodic block

Behaviour with E_b/N_o. Note the 'random walk' before convergence at intermediary E_b/N_o. The distance between decodings has a similar behaviour. A small decrease in E_b/N_o produces a high increase in number of iterations until 'it takes off'. The $E_b/N_o = 1$dB curve did not converge even with 5000 iterations.

Periodic blocks

Periodic blocks have been generally observed at high E_b/N_o values. The bit error rate for an example periodic block is presented in figure 5.21(a), and the evolution of the Euclidean distance for the first code in figure 5.21(b). In essence, periodic blocks are not so different from Type 2 blocks, excepting the fact that the output of a particular code's decoder does not stabilize to a limit value, but cycles through a finite number of fixed values. Periodic blocks do not appear to be very sensitive to data precision increase from single to double floating point precision.

Quasi-periodic blocks

These blocks are characterised by a large variation in the number of errors with the number of iterations. They are affected drastically by data precision. They appear to be a particular weakness of MPCCC schemes but especially SCCC schemes, degrading their performance at relatively high E_b/N_o.

The finite precision used to to evaluate the iterative algorithm can sometimes lead to a limit cycle in $\mathbf{P_E}$ i.e. a cyclic BER/block as a function of iteration. A typical case is shown in Fig. (5.22). Here the MAP decoder input vector $\mathbf{P_E}(n)$ has two closely spaced errors (a probability of one representing an error) followed by an isolated error.

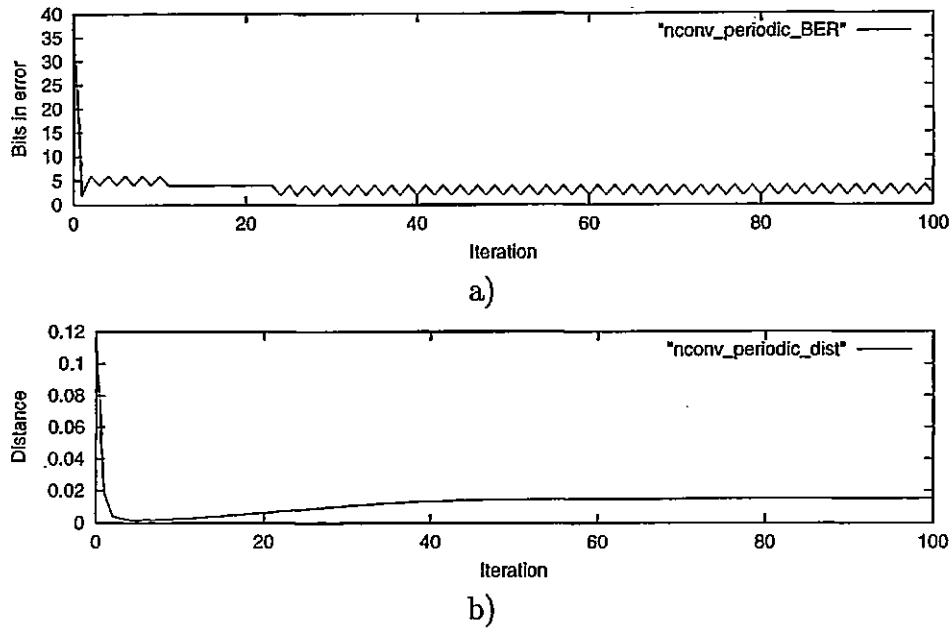


Figure 5.21: Periodic block
The iterative decoder is caught on a closed path and does not converge.

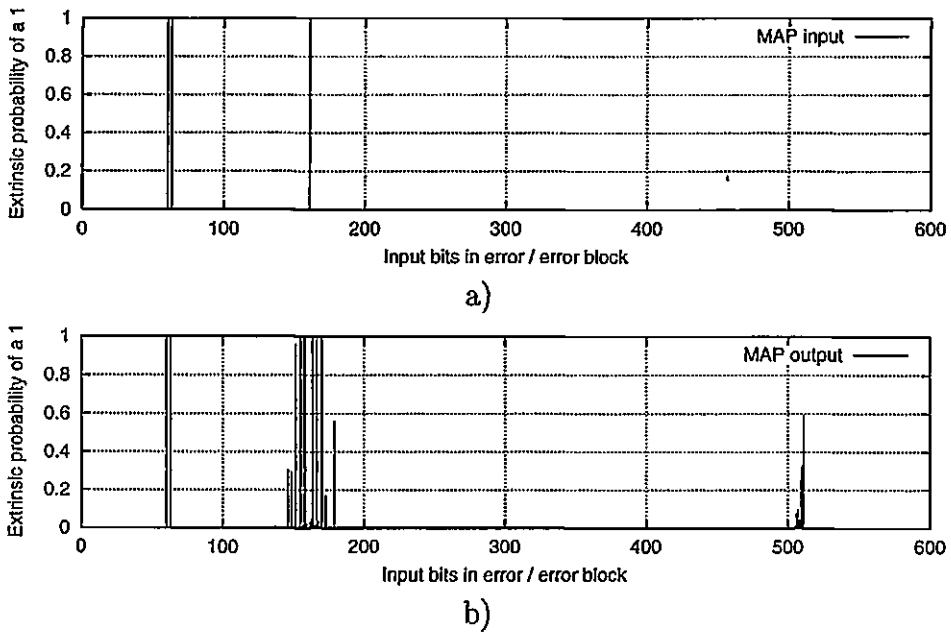
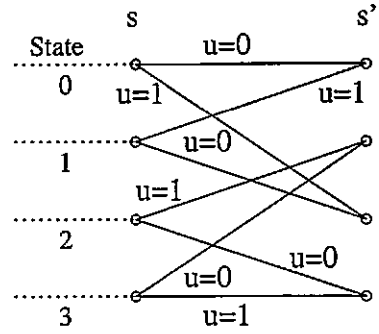
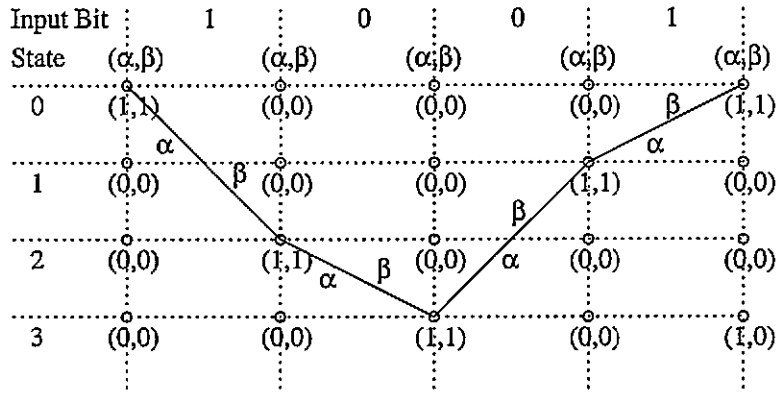


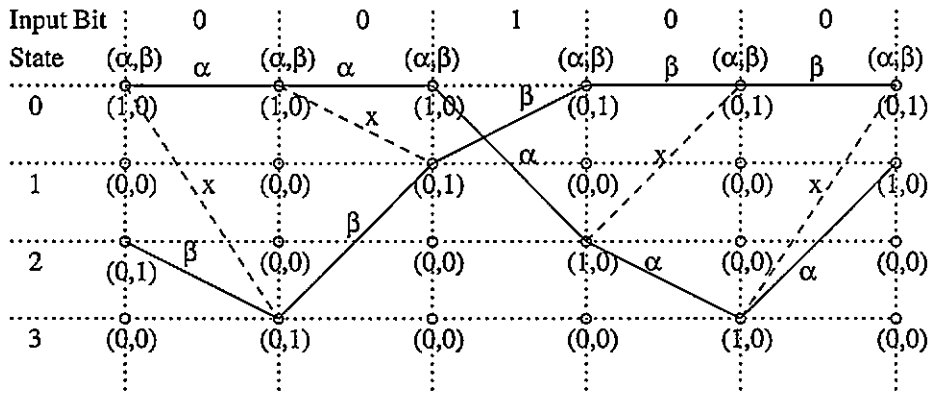
Figure 5.22: Quasi-periodic block extrinsic information
MAP decoder extrinsic information a) MAP input b) MAP output



a)



b)



c)

Figure 5.23: α/β recursions with saturated input
a) Trellis for the $RSC(5/7)$ code and alpha/beta recursions with saturated input values for b) short error event and c) infinite error event

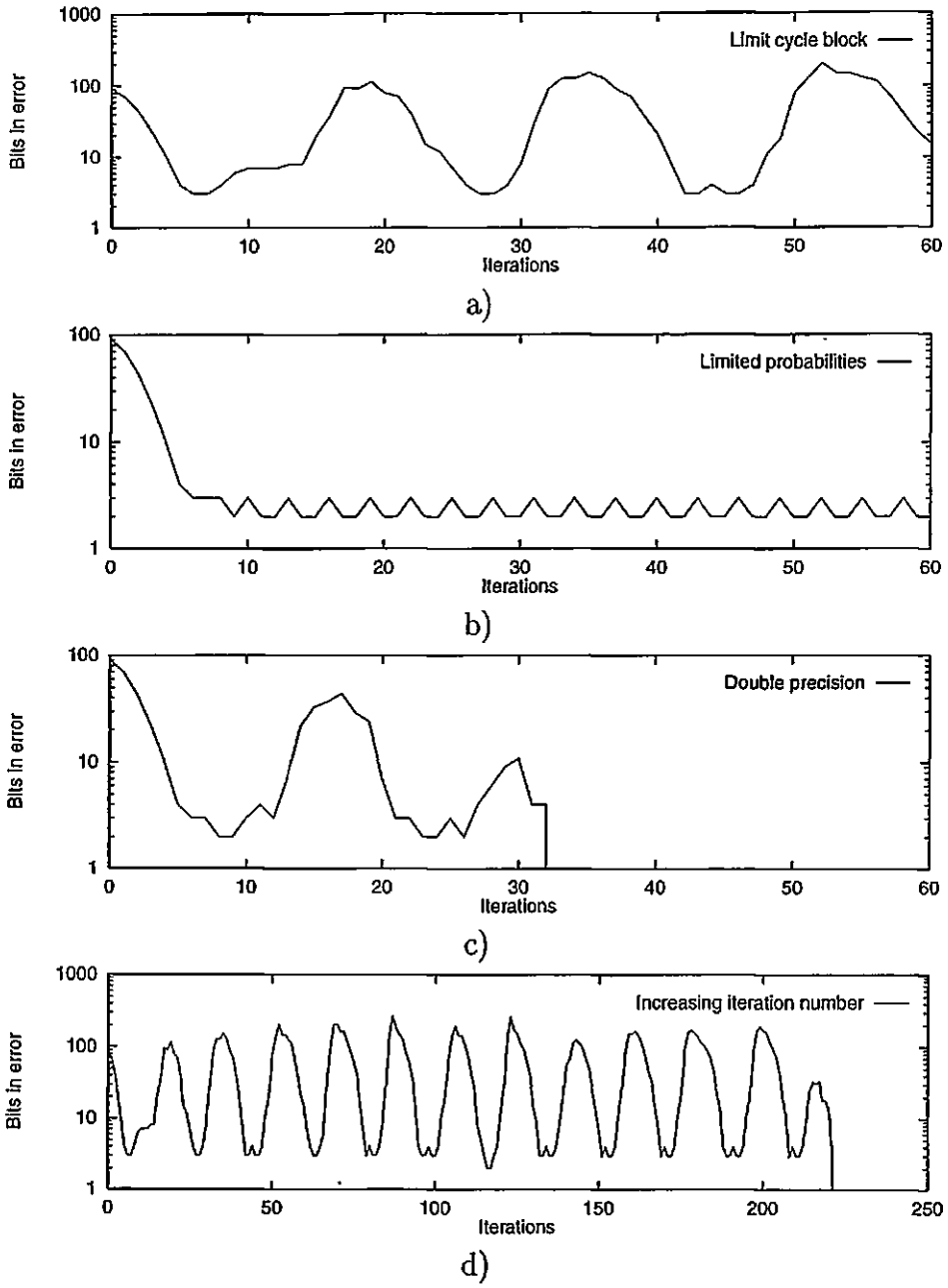


Figure 5.24: Block exhibiting limit cycle effect

The first two errors are separated by only two zeros and, since they are saturated, they force the decoder to follow a short, low weight error event for the $RSC(5/7)$ code used in the simulation. The first two errors are therefore simply translated to the decoder output. This error event is illustrated in Fig. 5.23(b), and the α and β probabilities are used in the usual forward-backward relation

$$P_{Ek}(i) = P_E\{u_k = i\} = \sum_{\{s,s'|u_k=i\}}{}' \alpha_{k-1}(s)\gamma_{Ek}(s,s')\beta_k(s') \quad , \quad i \in \{0,1\} \qquad (5.11)$$

where $\gamma_{Ek}(s,s')$ is the state transition probability from extrinsic information, and both $\alpha_{k-1}(s)$ and $\beta_k(s')$ can be simultaneously large, resulting in a confident decision.

Entirely different results are obtained for the third input error. Fig. 5.22(b) shows that this causes a significant error extension (both before and after the error location), which results in even more errors in the following MAP decoder. On the other hand, since the probabilities are generally non-saturated, and because the function is actually a contraction in that region, the number of errors will again reduce, resulting in a limit cycle effect (Fig. 5.24(a)). This type of behaviour arises since the isolated error is far from the block edges and generates an error event of high code weight that disagrees in many places with the channel values. The nature of this error event is illustrated in Fig. 5.23(c), where it can be seen that the saturated values for α and β correspond to 'invalid' trellis transitions, i.e. the values are no longer 'matched' to yield a high probability when used in (5.11). Error extension then results since the MAP decoder now has to determine the information bits in this region by selecting between two very small probabilities i.e. $P_{Ek}(1), P_{Ek}(0) \ll 1$. The above effects can be reduced in several ways:

- Limit the extrinsic probability P_{Ek} to within a value ε of saturation. Fig. 5.24(b) shows the reduction in cycle amplitude for $\varepsilon = 10^{-7}$. Unfortunately, limiting does also sometimes produce a small number of errors for blocks that would otherwise converge to zero error.

- Increase the machine precision. The effect for a given block is illustrated in Fig. 5.24(c). This does not usually work for SCCC schemes.

- Increase the number of iterations. Due to the chaotic nature of the process, after

several cycles the decoder may converge to the correct sequence, as shown in Fig. 5.24(d).

These blocks could be characteristic to the MAP decoder used. It is possible that they will disappear if the improved log-MAP algorithm is used (Robertson et al., 1997), since it does not involve multiplications or non-linear functions in its implementation.

5.8 Criteria for terminating iteration

Generally speaking, the iterative decoding process is stopped when a maximum number of iterations is reached. However, simulation shows that different blocks need a different number of iterations in order to converge, and the average decoding time can be reduced by terminating the iteration when no improvement is observed. Clearly a good termination criterion is to determine the number of errors for each iteration, and to stop at zero errors by reference to the original data. This has been used in the simulations to determine the absolute minimum for the average number of iterations. In practice, this could be realised by using a powerful cyclic redundancy check to determine if a block has been completely corrected, which means adding redundancy and reducing the code rate.

An alternative approach uses the Cauchy criterion in (5.5) to terminate iteration. Too large a value for δ will increase the BER due to premature termination i.e. before the actual extrinsic limit has been reached, whereas a lower threshold will increase the average number of iterations. Average iteration values and corresponding BER statistics for different thresholds are presented in Table (5.1). It is apparent that, providing $\delta \leq 10^{-3}$, there will be only relatively small variation in BER and iteration number.

From Table (5.1) it can be concluded that the cost of choosing the Cauchy criterion to stop iteration as opposed to a CRC approach is around 1.5-2 iterations on average in order to obtain similar performance. There are two drawbacks to this conclusion:

- The probability of false decision for the CRC has been neglected
- Non-optimal codes (such as $RSC(7/5)$) perform better than optimal codes (such as $RSC(5/7)$) at low E_b/N_o due to the presence of a large number of low information weight quickly converging error events, as opposed to a small number of

Average number of iterations				
E_b/N_o [dB]	Criterion			
	CRC	Cauchy		
	Stop at zero errors	$\delta = 10^{-2}$	$\delta = 10^{-3}$	$\delta = 10^{-6}$
1	3.5	4.4	5.5	6.5
1.5	2.0	3.1	3.7	4.5
2	1.4	2.5	3.1	3.6
Bit Error Rate				
E_b/N_o [dB]	Criterion			
	CRC	Cauchy		
	Stop at zero errors	$\delta = 10^{-2}$	$\delta = 10^{-3}$	$\delta = 10^{-6}$
1	55.41	67.7	57.32	56.9
1.5	1.36	3.1	1.7	1.638
2	0.12	0.59	0.161	0.158

Table 5.1: Average number of iterations and BER for different stopping criteria Average number of iterations and BER statistics for a rate 1/3 turbo decoder with N=500, S=14, $RSC(5/7)$ and different thresholds. All BER values should be multiplied by 10^{-5}.

high information weight non-convergent error events. In this case, the Cauchy criteria provides a quicker stopping condition for these blocks, so a combination of the two criteria will be optimal.

Criteria for terminating iteration in turbo decoders have also been proposed in (Hagenauer et al., 1996), where the metric was cross entropy, and in (Robertson, 1994) where the convergence was determined by estimating a standard deviation for the extrinsic information.

5.9 Evaluation of d_{free} from convergent blocks

The BER for a turbo code can be estimated from the union bound using the code weight spectrum rather than d_{free} alone (Ambroze et al., 1998b). Nonetheless, d_{free} is still an important design parameter, and the convergent blocks can be used to estimate d_{free} even for large block length. It was observed above that convergent blocks of low information weight/low code weight appear for each scheme if the E_b/N_o is high enough. By observing these blocks, one can obtain information about the d_{free} of the concatenated scheme.

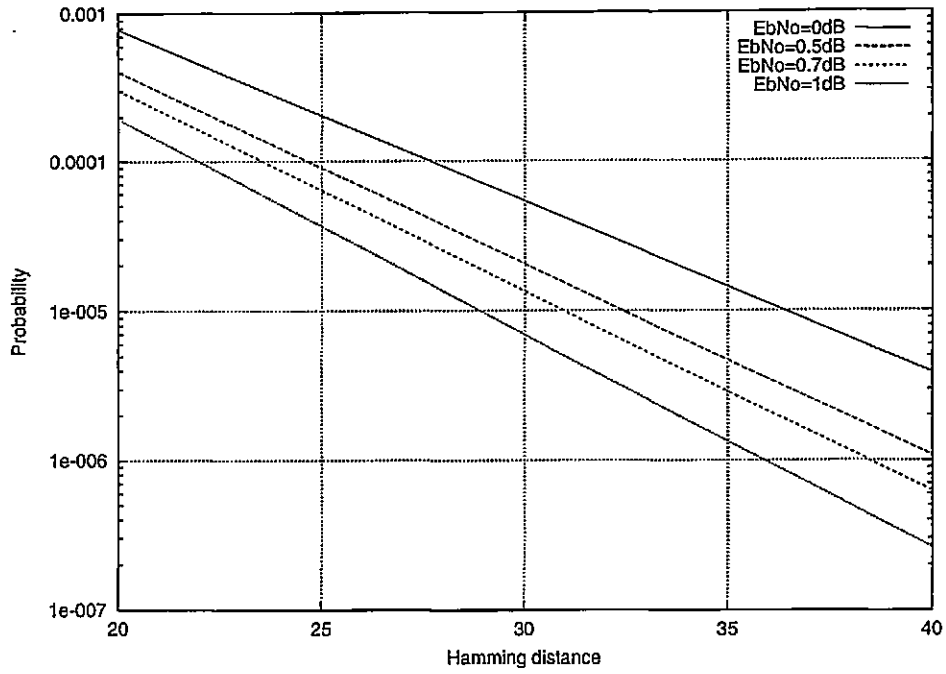


Figure 5.25: Probability of an error event vs Hamming distance
The probability of observing an error event with a given Hamming distance for different E_b/N_o values

As an example, by using the tree search method presented in (Ambroze et al., 1998b), an N=500, $RSC(5/7)$ turbo code using an S=14 interleaver is known to have $d_{free} = 18$ with a multiplicity (number of d_{free} paths) of 9. By applying the union bound for sequence error rate for this code, approximately 12 d_{free} type error events in 200,000 blocks would be expected at an $E_b/N_o = 2$dB. Simulation for 200,000 blocks showed 10 blocks with a code weight of 18 from which it can be deduced that $d_{free} \simeq 18$ for this particular decoder. This implies that d_{free} can be estimated by searching for a converged block with minimum code weight (it is not necessary to explicitly check for convergence). Moreover, this 'block convergence' method can be applied for large N (in contrast to the tree search method) and, if necessary, the number of minimum weight blocks can be increased by decreasing E_b/N_o. Using this approach, the N=2000, S=27, $RSC(5/7)$ turbo code used in the convergence simulations was shown to have $d_{free} \simeq 20$, whereas the N=2000, $RSC(5/7)$, random interleaver turbo code has $d_{free} = 10$.

The tree search algorithm has also been used to determine the weight spectra for 3PCCC schemes having $N = 500$ and $d_{free} \leq 26$ (in this particular case 26 is the

approximate limit of the tree search algorithm). The block convergence method was also applied and the results were confirmed by the tree search algorithm. However, it is relatively easy to find interleaver pairs yielding $d_{free} > 26$, in which case the tree search algorithm simply guarantees that $d_{free} > 26$. For these higher values the block convergence method can be used to estimate d_{free} since there will be a few low code weight convergent blocks even at relatively low E_b/N_o (in general there will also be some convergent blocks with high code weight). As for turbo codes, the minimum code weight blocks should correspond to the d_{free} of the code since this is the most likely error event. As an example, 3 convergent blocks having input weight 2 and code weight 38 have been observed for an $N = 500$, $RSC(5/7)$, 3PCCC scheme using a pair of 'S'-type interleavers. They were the only convergent error blocks at $E_b/N_o = 1$dB in 1200000 blocks (although there were several nonconvergent blocks). For $d_{free} = 30$, the union bound gives about 9 blocks in error in 1200000, for $d_{free} = 33$ the bound gives 3 blocks in error, and for a $d_{free} = 38$ the bound gives about 1 block in error. The 3 convergent blocks of weight 38 observed in the experiment thus suggest a d_{free} in the range 33 to 38.

Figure (5.25) shows the probability of observing a block in error, for a 3PCCC scheme ($R = 1/4$) given its Hamming code weight and the E_b/N_o at which the experiment has been performed, in the assumption of ML decoding. The fact that an error event of a given Hamming weight has not been observed does not necessarily mean that it does not exist: it is possible that not enough blocks have been tested. Since the complexity of iterative decoding increases linearly with block length, for the same E_b/N_o it will be more difficult to simulate enough blocks. This is compensated by the fact that longer codes converge at lower E_b/N_o, so less blocks have to be simulated. The figure can also be used to determine the limits of this method, bearing in mind that about 10^{10} bits can be simulated in reasonable time. For $N = 500$ this means $2 * 10^7$ blocks, allowing for a probability of about 10^{-6} which at $E_b/N_o = 1$dB gives $d_{free} \leq 35$. For $N = 2000$, $5 * 10^6$ blocks can be simulated, allowing for a probability of 10^{-5} which at $E_b/N_o = 0.5$dB also gives a $d_{free} \leq 35$.

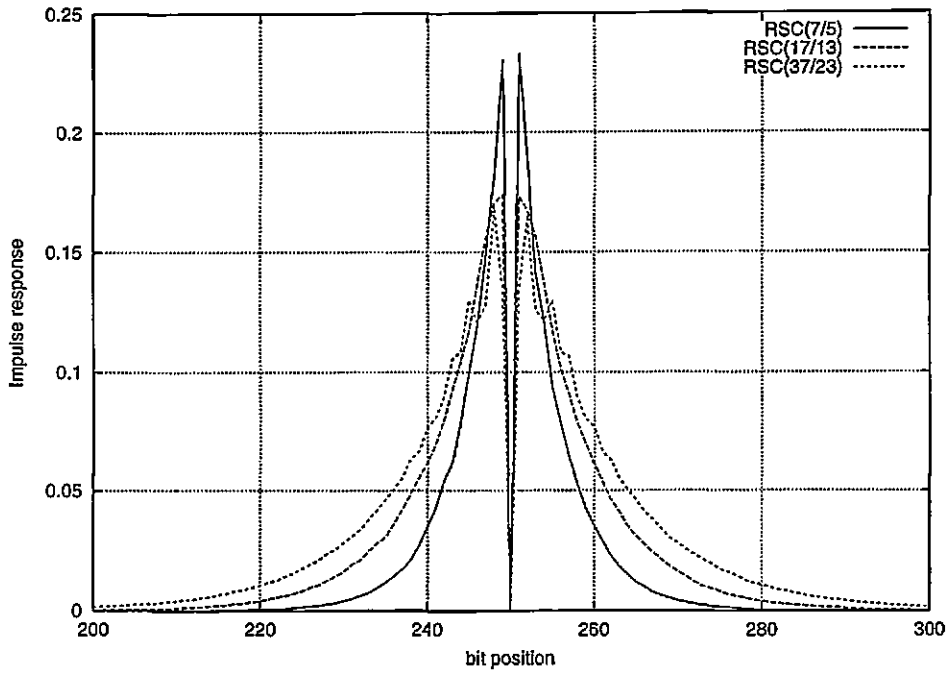


Figure 5.26: Impulse response for different codes
Impulse response for different component codes for input extrinsic bit in position 250

5.10 Correlation and convergence

In (Hagenauer et al., 1996) it is mentioned that improvement of BER with iterations is reduced due to the fact that the extrinsic information becomes correlated. In (Divsalar and Pollara, 1995a) an assumption of independence between the extrinsic outputs in the iterative decoder is used to derive approximate equations for the iterative process. Other papers, such as (Berrou et al., 1993b; Moher, 1998a; Battail, 1997) mention the correlation between the values of extrinsic information at the output of the SISO decoder or between the input and output of the SISO decoder as a problem for the iterative process that has to be dealt with by designing the codes and/or the interleaver (Hokfelt et al., 1999c). Methods to measure the correlation are presented in (Hokfelt and Maseng, 1998; Hokfelt et al., 1999c; Hokfelt et al., 1999e). This section investigate ways to measure the dependence between the extrinsic information values at the input and output of the SISO decoder in an iterative decoding process.

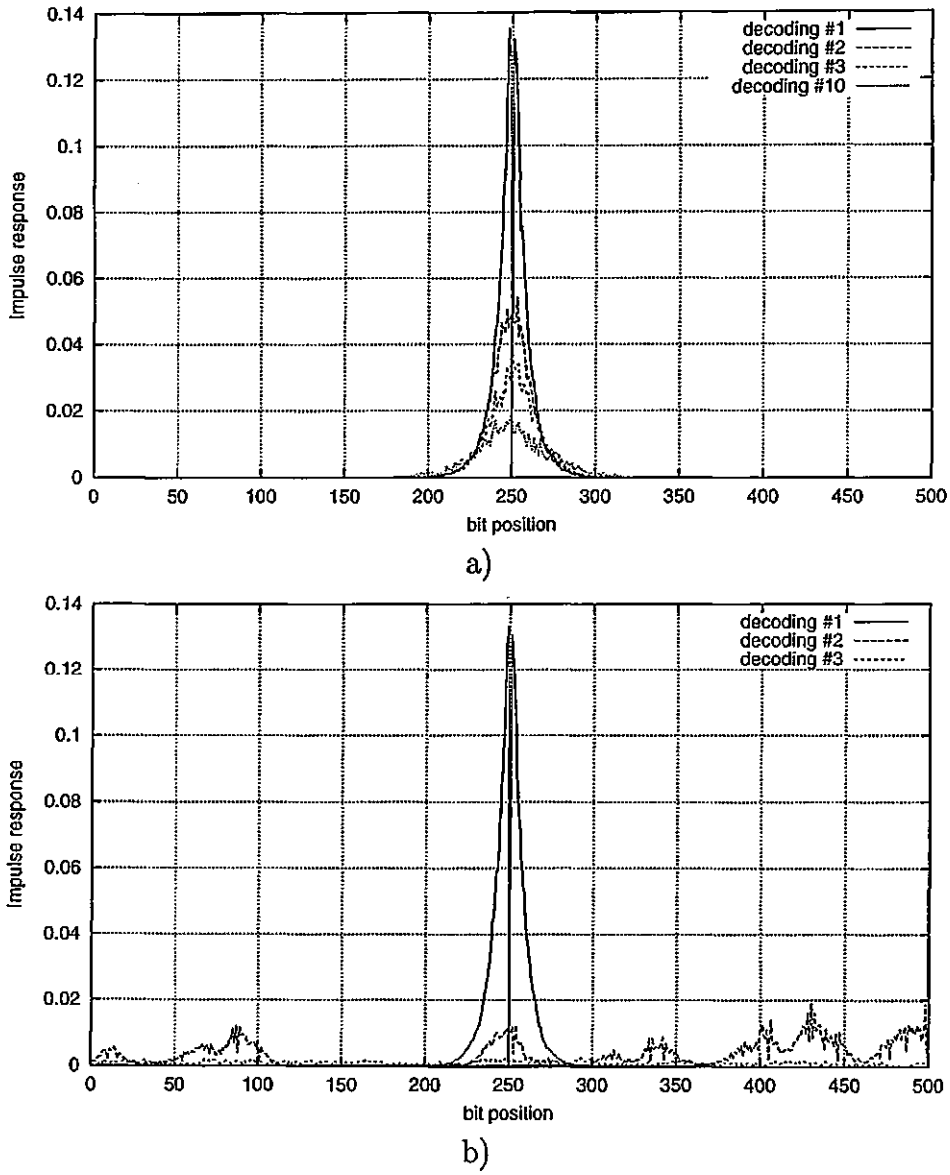


Figure 5.27: Impulse response for iterative decoder
Exchanged information 'dependence' on extrinsic input in position 250 for a turbo code using a) identical interleaver and asymmetric component codes, b) randomly chosen interleaver and symmetric component codes. The block length is $N = 500$.

5.10.1 Impulse response

Consider the turbo decoder presented in figure (2.12) and the iterative algorithm in section (2.2.3). The iterative decoding starts for each received blocks with the extrinsic vector set to **0.5**. The "impulse response" of the iterative decoder is obtained in a very heuristical way by perturbing one component of the starting extrinsic vector whilst keeping the others equal to 0.5 and determining the effect of the perturbation of all components in all the output extrinsic vectors during iteration. Since the effect of the perturbation depends on the received block, the values obtained are averaged over all the received blocks. Thus the "impulse response" is the amplitude of the perturbation of the components of the extrinsic information vector at the output of each SISO decoder as the input extrinsic component in position i is varied between 0 and 1, averaged over all received blocks.

A loose mathematical formulation can be given for the "impulse response" at the output of the first SISO decoder in the iterative algorithm in the case of a MAP algorithm used as SISO decoder. The dependence of the component j of the extrinsic information at the output of a MAP decoder on the component i of the extrinsic information at the input of the MAP decoder can be obtained from the MAP equations as:

$$f_{i,j,k}(x) = \frac{a(i,j,k)x + b(i,j,k)}{c(i,j,k)x + d(i,j,k)} \tag{5.12}$$

where k is the index the received block, x is the value of extrinsic input component i. The functions $a()$, $b()$, $c()$, and $d()$ depend on the received block and the values of i and j. This function is monotonous for $x \in [0,1]$, and thus the variation is

$$\Delta(i,j,k) = |f_{i,j,k}(0) - f_{i,j,k}(1)| = \left| \frac{b(i,j,k)c(i,j,k) - a(i,j,k)d(i,j,k)}{d(i,j,k)(c(i,j,k) + d(i,j,k))} \right| \tag{5.13}$$

The "impulse response" for each bit is calculated as the average over all received blocks:

$$\overline{\Delta(i,j)} = \frac{1}{n} \sum_{k=1}^{n} \Delta(i,j,k) \tag{5.14}$$

The "impulse response" at the output of the first MAP decoder as the input component in position $i = 250$ is varied between 0 and 1 is presented in figure (5.26) for different

component codes. The block size was $N = 500$ and $E_b/N_o = 1$dB. As expected, the output extrinsic component $j = i$ is independent of the input extrinsic component i. Also, the dependence on the immediately close input values is high, and asymptotically decreasing with distance. The "impulse response" has a specific shape for each code. Higher memory codes have smaller maximums, which could explain the reduced number of non-convergent blocks, and a larger span, which could explain the increased number of errors in a block.

The "impulse response" for the next iterations depend on the interleaver used by the turbo code. Figure 5.27 shows the values for a turbo code using a) the identical permutation as interleaver (no interleaving) and b) an interleaver chosen at random. It can be observed that the dependence of the extrinsic information on the initial conditions persists in the first case, and it is spread all over the block in the second case, and quickly disappears with iteration.

As a conclusion, the output of each decoder shows a regional, asymptotically decaying dependence on the input, for each bit position in the block. The role of the extrinsic information is to decorrelate the output from the input in the same position. This exposes the function of the interleaver and the importance of the extrinsic information. The interleaver is used to spread this dependence over the block, breaking local correlations. But the interleaver cannot break the correlation of the output bit with the input value in the same position in the block. This is the role of the extrinsic information and together serve in breaking the correlation and providing uncorrelated information from other positions in the block. A parallel can be drawn between the role of *RSC* codes for ML performance and that of the extrinsic information for iterative decoding, both completing the function of the interleaver.

The combined effect of the interleaver and extrinsic information is illustrated in figure (5.28). Ideally, the extrinsic information in position H should be independent on the extrinsic information in position A. There are two ways for the dependence to propagate: through the output bit in the same position, which is discontinued by using extrinsic information, and through bits in the dependence region, which are not interleaved far enough from the considered bit. This can be reduced by designing the interleaver, and this is the reason why the S interleaver can improve convergence.

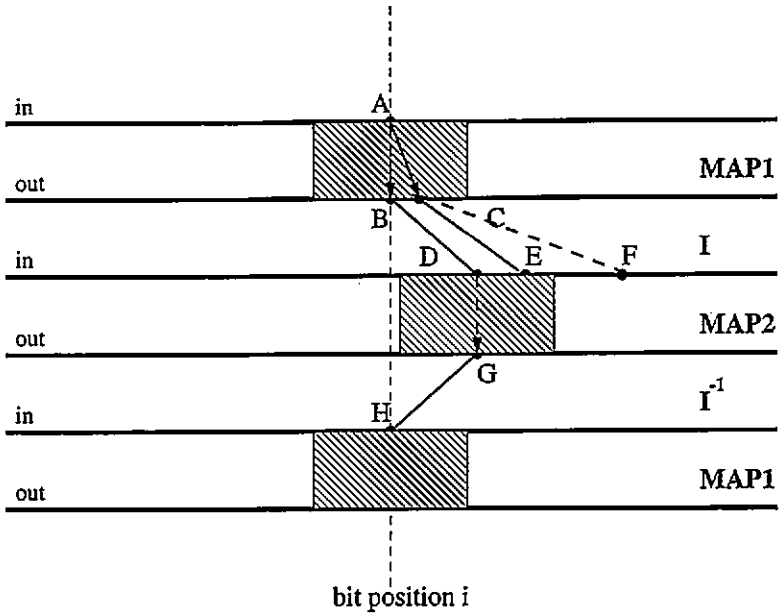


Figure 5.28: Input/output dependence propagation
Dependence propagation for a turbo code. There are two ways to propagate output/input correlation: (direct) output/input values in the same position and (indirect) through the interleaver.

5.10.2 Linear correlation coefficient

Given two distributions x_i and y_i, the linear correlation coefficient is given by (Press and Teukolski, 1993):

$$r = \frac{\sum_i (x_i - \overline{x}) \sum_i (y_i - \overline{y})}{\sqrt{\sum_i (x_i - \overline{x})^2 \sum_i (y_i - \overline{y})^2}} \tag{5.15}$$

Its values belong to $[-1 : 1]$. It can be used as a measure of correlation between the two distributions. The higher the absolute value $|r|$, the higher the correlation between the two distributions.

In the iterative algorithm, there are several vectors for which the correlation coefficient can be calculated: the channel values, the input and output extrinsic information for each SISO decoder. The linear correlation coefficient can be calculated between a component of one of the vectors and a component of another or the same vector, in terms of probabilities or log-likelihood values. The distribution for a component consists of the values this component takes for all simulated blocks.

The practical computation has two stages: computing the average of the distribu-

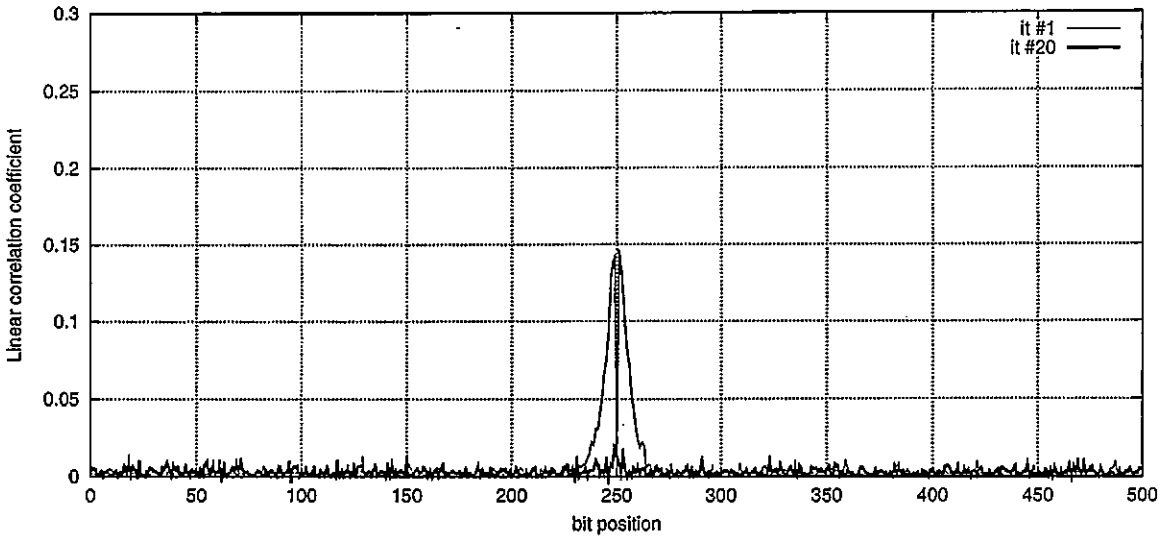
tions (first run of the program) and computing the correlation factor using the precomputed averages (second run). The simulation has to be run using the same encoded data for each block, preferably the all-zeros sequence.

Figure (5.29) shows the correlation coefficient between the output extrinsic information for bit position 250 and the received values corresponding to the systematic bits (a) and parity bits (b) in the whole block for the first and last iterations (20 iterations have been performed). The correlation with the systematic bit is similar with the impulse response curve, and the output bit is not dependent of the systematic bit in the same position. The correlation with the parity bit is similar, except for the fact that it has a strong dependence on the parity bit in the same position. The correlation decreases with the number of iterations, and at the end, the output extrinsic becomes uncorrelated with the channel values, it is *new* information generated in the iterative process to compensate for the missing bit of each decoder. Figure (5.30) shows the correlation factor between the input/output extrinsic information for output value in position 250 and all the input values. The extrinsic values become more and more correlated with the number of iterations. Also, the correlation is spread by the interleaver over the whole block. There are correlation peaks, corresponding to extrinsic information values that are close together both in the direct and interleaved stream.

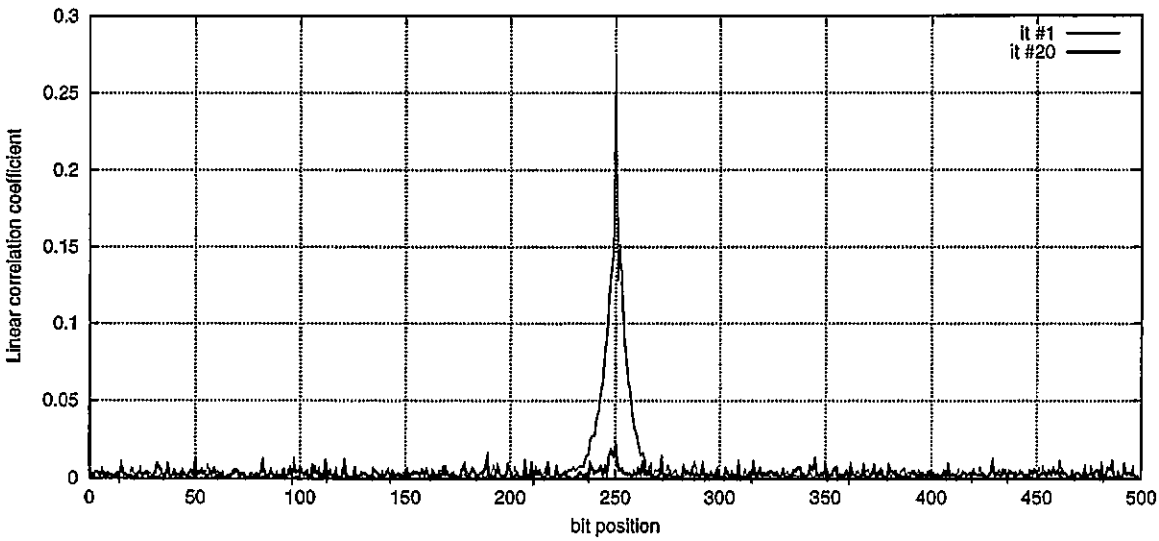
The correlation coefficient between the input and output extrinsic values for the same bit position ($i = 250$) for turbo codes with block length $N = 500$ and several component codes, and also for a 3PCCC scheme with the same block length is presented in figure (5.31). Correlation shows a quick increase in the first iterations, and then an asymptotic increase. Higher memory codes correlate quicker, and non-optimal codes have a slower increasing correlation curve. 3PCCC schemes also correlate quicker.

5.11 Conclusions

- The convergence problem of the iterative decoder is qualitatively presented as a fixed point problem.
- A non-ML test has been used to determine which decodings would not have been chosen by an optimal decoder. This test usually qualifies HIWHOW blocks as non-ML.



a)



b)

Figure 5.29: Correlation of extrinsic output with channel values
Correlation between output extrinsic in position 250 and a) systematic received value and b) parity received value in all positions in the block for a turbo decoder with block length $N = 500$, $RSC(17/13)$ component code. Correlation is only computed for the first (non-interleaved) code.

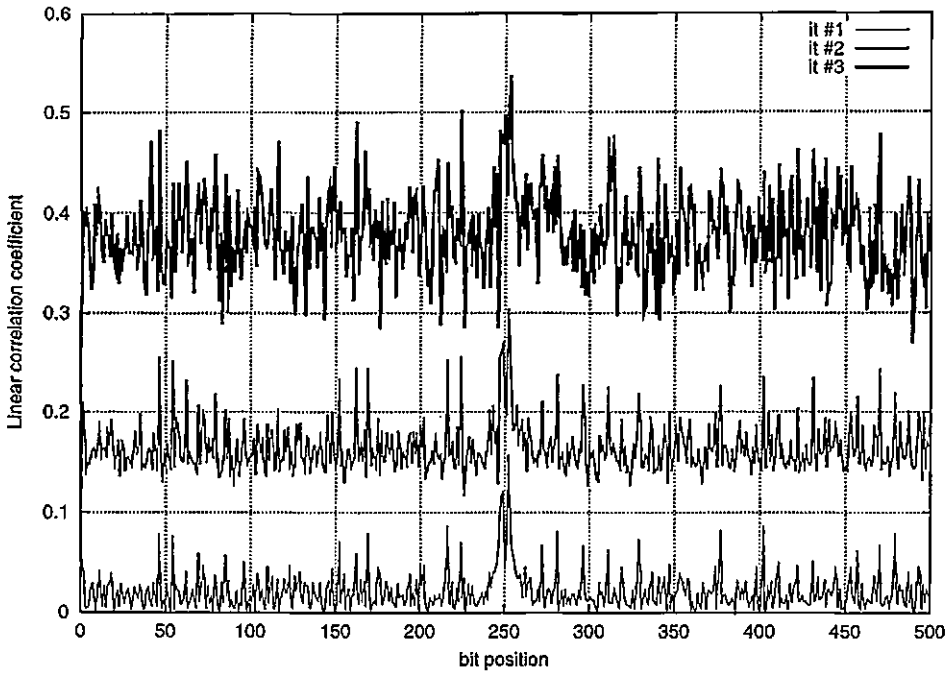


Figure 5.30: Output/input extrinsic correlation vs bit position
Correlation between output extrinsic value in position 250 and input extrinsic in all positions in the block for a turbo decoder with block length $N = 500$, $RSC(17/13)$ component code. The parameter of the curves is the number of iterations. The correlation is only computed for the first (non-interleaved) code.

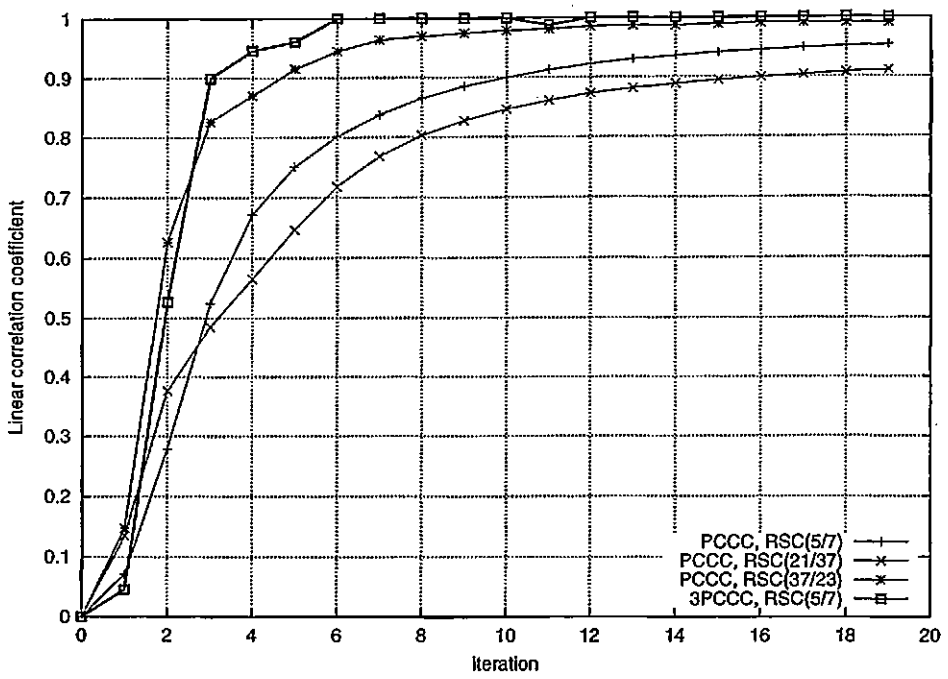


Figure 5.31: Correlation versus iteration
Output/input extrinsic correlation for output/input bit 250 for different component codes and number of codes in an MPCCC concatenation. Correlation is only computed for the first (non-interleaved) code.

- The Cauchy convergence criterion has been used to separate the performance of turbo codes, MPCCC and SCCC into two components: non-convergent performance (usually HIWHOW blocks) and convergent performance. The distribution of information weight for each component has been determined, showing that convergent blocks have generally low information weight. The study has been performed for different parameters of the concatenated schemes, correlating them with the iterative decoder tendency to converge.

- The Cauchy criterion has also been used as an iteration stopping criterion and compared with other stopping criteria.

- The two components of the performance curve for a $M = 4$ component code turbo code have been compared with their union bound performance, obtained by tree search. It has been observed that the union bound curves are close to the convergent performance. The non-optimal $RSC(21/37)$ code has a closer convergent performance to the union bound than the optimal $RSC(37/23)$ code,

for which a slight difference has been observed.

- Two methods to determine the correlation between the input and the output of the MAP decoder have been presented and applied for several code parameters. The combination interleaver/extrinsic information effect for the iterative decoding has been presented as a parallel to the combination RSC codes/interleaver for optimal decoding performance.

Chapter 6

Conclusions

6.1 Contributions to knowledge

- Detailed description of encoding/decoding algorithms for three different concatenated schemes.
- Speed improvement of the S interleaver algorithm, useful comparisons with other algorithms.
- Performance improvement of the S interleaver by rejecting $IW = 2 + 2$ error events or forcing bits. Mathematical formulae derived for the worst case OW_2 given the value of S. The value of S for which the contribution of $IW = 2$ error events to error rate is masked by "crossed" error events calculated.
- Design and justification of the paired S interleavers for 3PCCC schemes, as compared with randomly chosen interleavers or two separately designed interleavers. Mathematical formulae derived for the worst case OW_2 given the values of S for independent and paired S interleavers.
- Analyse of the d_{free} of MPCCC schemes for different interleaver lengths and different component codes for turbo codes (2PCCC), 3PCCC and 4PCCC.
- Detailed theoretical/practical discussion of the random interleaver theory for turbo codes, 3PCCC and SCCC and comparisons.
- Short and clear discussion of the methods to obtain the turbo code weight spectra with their advantages and disadvantages.

- Novel fast tree based algorithm to obtain the weight spectra of turbo codes, with investigations into the actual weight spectra of turbo codes with fixed interleaver. Examples of the effect of interleaver length/design, component code structure and data tail on the weight spectra, with ML/iterative decoding comparisons.

- Non-iterative decoding methods based on the turbo code tree described, applied to short block length ($N \leq 100$) turbo codes and compared with iterative decoding results.

- Turbo code hypertrellis obtained from the turbo code tree based on simple observations and comparisons with other methods from literature.

- Formulating the convergence of the iterative algorithm as a fixed point problem and illustrating its general behaviour.

- Using the Cauchy criterion to separate the performance of iterative decodings into convergent/non-convergent performance, and identifying the information/code weight of the convergent non-convergent blocks, with comparisons for the three schemes.

- The S interleaver has been shown to improve convergence for turbo codes with low memory component codes, but not for higher memory codes. The S interleaver does not improve convergence for 3PCCC and SCCC.

- Comparison of the convergent BER curve of turbo codes with the union bound curves using the weight spectra obtained by the tree search method.

- Several ways to estimate the free distance of turbo codes presented: by observing LIWLOW (convergent) error events (with estimates up to $d_{free} \approx 35$), by searching for $(OW_2)_{min}$ or by searching the turbo code tree.

- Several methods to stop the iteration based on convergence/zero errors presented, with comparisons between fixed/variable number of iteration schemes.

- Description of error blocks observed in the iterative decoding process, with their behaviour with E_b/N_o.

- Introducing/using methods to measure the correlation of extrinsic information as a function of iteration. Justifying the usage of the extrinsic information and interleaver from the iterative point of view, with a parallel with the usage of *RSC* codes and interleaver from the ML point of view.

6.2 Conclusions and future work

This work has analysed the performance of turbo codes and other concatenated schemes, the multiple parallel concatenation (MPCCC) and the serial concatenation (SCCC). The channel considered was the AWGN channel with BPSK/QPSK modulation. There are two components that dictate the performance of these coding systems: the optimal decoding performance and the iterative decoding performance.

The usual method to study the optimal performance of concatenated schemes with interleavers is the *uniform interleaver* method, introduced in(Benedetto and Montorsi, 1996c). This method calculates an average performance over all interleavers of a given length N. The main problem of this method is that when a real interleaver is chosen at random, there is no way to tell how far its performance is going to be from the average. It calculates the average of the performance probability distribution, but not the distribution itself. In this work, this problem has been approached in several ways:

- In the simulations, by observing the LIWLOW error events that generate the error floor for a given, randomly chosen interleaver and given component codes.

- By computer search for the $IW = 2$ and $IW = 2 + 2$ error events, producing the distribution for $(OW_2)_{min}$ and $(OW_{2+2})_{min}$ for the MPCCC schemes. In this way, it was observed that turbo codes (2PCCC) produce an $(OW_2)_{min}$ distribution which has a high peak for $d_{free-eff}$, the minimum code weight that can be produced by the component codes. In this case, the average coincides with the peak and the distribution has a very reduced spread. Turbo codes with randomly chosen interleaver are very close to their average performance, as was observed in (Benedetto and Montorsi, 1996c). The situation changes for 3PCCC and 4PCCC schemes, where the spread of the distribution increases with block length.

- By using a novel tree search algorithm to produce the weight spectra for a given interleaver. The union bound is used to produce the FER and BER of the code.

The second approach was very useful in understanding the way the d_{free} is produced for turbo codes and MPCCC schemes when the interleaver is chosen at random. At the moment, the method is limited to computer search for $IW = 2$ and $IW = 2 + 2$ "crossed" error events and a qualitative explanation of the minimum distance generation. The problem in producing a full combinatorial approach is the need to count interleavers that produce dependent error event mappings only once. A continuation of this method is to analyse higher IW error events and produce the combined contribution to the d_{free} of the codes. It would be interesting to obtain an answer to the question whether the 3PCCC and SCCC schemes are asymptotically good, and how the interleaver(s) are chosen. The uniform interleaver approach shows that the average FER converges to zero as $N \to \infty$ for 3PCCC and SCCC schemes, but since the spread of the curves increases with block length, the problem of picking the right interleaver is non-trivial.

The first and the third method to analyse the weight distribution of a given interleaver are relatively successful for turbo codes with randomly chosen interleavers, with the third method more limited by the interleaver length. However, for turbo codes using designed interleavers and for 3PCCC and SCCC schemes with reasonable block length they are rather problematic to use. The tree search algorithm needs an unreasonably long time and also the simulations have to be performed for a very large number of blocks in order to observe any LIWLOW block. In this case, the fast search for low IW error events proves to be the best method in obtaining an upper limit on the d_{free} of the codes.

Investigation of the hypertrellis generation methods can show ways to simplify tree/trellis generation.

The convergence problem of the iterative decoder was approached in several ways:

- In a qualitative way, as a fixed point problem.

- By separating the error blocks in the iterative decoding simulations based on their information/code weight and observing their contribution to the performance of the schemes at different E_b/N_o values. By calculating their Euclidean distance

to the received vector, it was observed that HIWHOW are always further away from the received vector than the correct sequence, whereas LIWLOW are closer than the correct sequence for at least one component code.

- By determining the convergence of each block using the Cauchy criterion and separating the performance of the iterative decoding in convergent/non-convergent performance. HIWHOW blocks have been found to be mostly non-convergent blocks. The component codes and the interleaver design have been found to affect convergence.

- By computing the correlation of the output values with the input values of the extrinsic information produced by the SISO algorithms.

These approaches correlate the parameters of the concatenated scheme with the tendency of the iterative decoder to converge. It was observed that optimal, higher memory codes produce HIWHOW blocks with higher information weight. This is attributed to a more correlated output of the SISO decoder using these codes. The separation of the performance curves in non-convergent and convergent attribute their change in slope to the convergence of the iterative decoder. There exists another explanation of the slope change, based on the effect of the interleaver, presented under the name of "spectral thinning" in (Perez et al., 1996). This attributes the change in performance to a non-uniform spectra. The answer is probably a combination of the two: the non-uniform spectra produces the non-convergence of the iterative decoder at low E_b/N_o.

A possible look into the problem is the behaviour of the non-optimal as opposed to the optimal codes. The difference is linked with the code structure by (Andersen, 1999) by suggesting that, at least for the SOVA algorithm, the iterative decoder converges better for non-optimal codes due to the smaller steps it has to take for non-optimal codes, which leads to smaller disagreements between the two codes. Investigating the difference between the codewords produced in each iteration could lead to interesting results.

A different approach is to calculate correlations for single, non-convergent blocks. Each bit position in the block generates a distribution along the iterations (and a high number of iterations can be used to obtain enough samples). The correlation between

positions in the block can be calculated by determining the linear correlation coefficient.

Replacing the MAP algorithm with the log-MAP algorithm can identify the block types produced by the numerical problems of the MAP algorithm. It would be interesting to see if quasi-periodic limit cycle error events are characteristic to the MAP algorithm.

Appendix A

Interleaver construction

A.1 Randomly chosen interleaver

Choosing an interleaver at random can be accomplished with the following routine:

1. set designed position $k \leftarrow 1$, reset interleaver $I \leftarrow \mathbf{0}$
2. get random number $1 \leq n \leq N$
3. **if** n not already used ($\exists!\ i < k$ so that $I(i) = n$)

   then $I(k) \leftarrow n$

   else goto 2.
4. **if** $k < N$ **then** $k \leftarrow k + 1$, goto 2.

Whether the interleaver generated in this way is random or not relies heavily on the uniform random number generator used to produce the values of n. In this work, the function ran2() from (Press and Teukolski, 1993) was used. Theoretically, if the random number generator is good, then there should not be any bias for any interleaver position, and the probability of the interleaver is:

$$P_I = \frac{1}{N} * \frac{1}{N-1} * ... * \frac{1}{2} * 1 = \frac{1}{N!} \tag{A.1}$$

The number of trials for each position is likely to increase with the value of k. A quicker way to obtain an interleaver is by using the following algorithm:

1. set designed position $k \leftarrow 1$, reset interleaver $I \leftarrow 0$
2. get random number $1 \leq n \leq N$
3. **if** n not already used ($\exists!\ i < k$ so that $I(i) = n$)

   then $I(k) \leftarrow n$

   else circularly search for an unused value $1 \leq n' \leq N$ starting from n. $I(k) \leftarrow n'$
4. **if** $k < N$ **then** $k \leftarrow k + 1$, goto 2.

Whilst this takes only N trials to complete, it is biased. Consider designing an interleaver of length $N = 7$ and the following situation:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ x & 0 & x & x & x & 0 & x \end{pmatrix} \tag{A.2}$$

where x signifies that the value above it has been used and 0 a spare value. The value $n' = 2$ will be chosen if and only if $n \in \{7, 1, 2\}$ and thus its probability is $P(2) = 3/7$ whereas $P(6) = 4/7$ and thus the choice is correlated with the previous values. This was observed when failing to obtain a higher S value for the S interleaver by using this algorithm to generate unique random numbers in the range $1..N$ quicker.

A.2 The rectangular interleaver

The rectangular (or row/column) interleaver, arranges the input bits in a matrix having L lines and C columns. The bits are written line by line and read column by column. The length of the interleaver is $N = L * C$. The interleaver function can be expressed as (Benedetto et al., 1997c):

$$I(k) = C * (k \bmod L) + \left\lfloor \frac{k}{L} \right\rfloor \tag{A.3}$$

where k is the non-interleaved position, $I(k)$ is the corresponding interleaved position and mod() is the modulo operator. The *square* interleaver is a rectangular interleaver with $L = C = \sqrt{N}$.

Appendix B

The MAP algorithm

The MAP algorithm is the optimal SISO algorithm for bit maximum likelihood decoding of a convolutional code. The equations for the MAP algorithm presented below are based on (Bahl et al., 1974). Central to the MAP decoder is computing the probability of a decoded bit in a block, given the received vector $\mathbf{R}_1^N$:

$$P_d\{d_k = 0\} = P_d\{d_k = 0|\mathbf{R}_1^N\} \tag{B.1}$$

where d_k can represent either an information or a code bit. For a convolutional code, this probability can be computed as the sum of the probability of all transitions that are generated by $d_k = 0$ (if d_k is an information bit) or produce $d_k = 0$ (if d_k is a code bit):

$$P_d\{d_k = 0\} = \sum_{m',m|d_k(m',m)=0} P\{S_k = m, S_{k-1} = m'|\mathbf{R}_1^N\} \tag{B.2}$$

where S_k represents the code state at stage k. By using Bayes' rule (B.3),

$$P\{A, B\} = P\{A|B\}P\{B\} \tag{B.3}$$

equation (B.2) becomes:

$$P_d\{d_k = 0\} = \frac{1}{P\{\mathbf{R}_1^N\}} \sum_{m',m|d_k(m',m)=0} \lambda_k(m', m) \tag{B.4}$$

where

$$\lambda_k(m', m) = P\{S_{k-1} = m', S_k = m, \mathbf{R}_1^N\} \tag{B.5}$$

is the joint probability of $S_k = m$ and $S_{k-1} = m'$. The term $P\{\mathbf{R}_1^N\}$ in (B.4) can be seen as a normalising term. It is not necessary to compute it, since it appears in both $d_k = 0$ and $d_k = 1$ expressions. Based on the fact that $P_d\{d_k = 0\} + P_d\{d_k = 1\} = 1$, we can write:

$$P_d\{d_k = 0\} = \frac{P_d\{d_k = 0\}}{P_d\{d_k = 0\} + P_d\{d_k = 1\}} \tag{B.6}$$

It can be observed that by using equation (B.4) in equation (B.6) the term $\frac{1}{P\{\mathbf{R}_1^N\}}$ cancels out. In (Berrou et al., 1993b) this term is computed, leading to a more complicated formulation of the algorithm. An alternative to equation (B.6) is the log-likelihood value:

$$L_k = \log\left(\frac{P_d\{d_k = 0\}}{P_d\{d_k = 1\}}\right) \tag{B.7}$$

B.1 Computing the joint probability

The value $\lambda_k(m', m)$ can be divided in three terms by using Bayes' formula (B.3) as follows:

$$\begin{aligned} \lambda_k(m', m) &= P\{S_k = m, S_{k-1} = m', \mathbf{R}_1^N\} \\ &= P\{\mathbf{R}_{k+1}^N | S_k = m, S_{k-1} = m', R_1^k\} P\{S_k = m, R_k | S_{k-1} = m', \mathbf{R}_1^{k-1}\} * \\ &\quad P\{S_{k-1} = m', \mathbf{R}_1^{k-1}\} \end{aligned} \tag{B.8}$$

Since the values received after time k, $\mathbf{R}_{k+1}^N$ depend on the previous values $\mathbf{R}_1^k$ only through the constraint of the code, if $S_k = m$ is known, the knowledge of S_{k-1} and $\mathbf{R}_1^k$ is not relevant and thus $P\{\mathbf{R}_{k+1}^N | S_k = m, S_{k-1} = m', R_1^k\} = P\{\mathbf{R}_{k+1}^N | S_k = m\}$. By using a similar argument, $P\{S_k = m, R_k | S_{k-1} = m', \mathbf{R}_1^{k-1}\} = P\{S_k = m, R_k | S_{k-1} = m'\}$.

Equation (B.8) becomes:

$$\begin{aligned}\lambda_k(m',m) &= P\{\mathbf{R}_{k+1}^N|S_k=m\}P\{S_k=m,R_k|S_{k-1}=m'\}* \\ &\quad P\{S_{k-1}=m',\mathbf{R}_1^{k-1}\}\end{aligned} \tag{B.9}$$

With the following notations:

$$\begin{cases}\alpha_k(m) &= P\{S_k=m,\mathbf{R}_1^k\} &, k\in\{0,...,N\} \\ \beta_k(m) &= P\{\mathbf{R}_{k+1}^N|S_k=m\} &, k\in\{0,...,N\} \\ \gamma_k(m',m) &= P\{S_k=m,R_k|S_{k-1}=m'\} &, k\in\{1,...,N\}\end{cases} \tag{B.10}$$

the joint probability becomes:

$$\lambda_k(m',m)=\alpha_{k-1}(m')\gamma_k(m',m)\beta_k(m) \tag{B.11}$$

where α computes the probability of state $S_{k-1}=m'$ based on the values received before time k, $\mathbf{R}_1^{k-1}$, β computes the probability of state $S_k=m$ based on the values received after time k, $\mathbf{R}_{k+1}^N$ and γ is the transition probability, based on the current received value, R_k.

B.2 The α recursion

The values for $\alpha_k(m)$ can be calculated recursively, starting from

$$\alpha_0(m)=\begin{cases}1 & \text{if } m=0 \\ 0 & \text{otherwise}\end{cases} \tag{B.12}$$

which basically states that the encoding process starts from state $S_0=0$ and using the recursion formula obtained below by using Bayes' rule (B.3):

$$\begin{aligned}\alpha_k(m) &= \sum_{m'}P\{S_{k-1}=m',S_k=m,\mathbf{R}_1^{k-1},R_k\} \\ &= \sum_{m'}P\{S_k=m,R_k|S_{k-1}=m',\mathbf{R}_1^{k-1}\}P\{S_{k-1}=m',\mathbf{R}_1^{k-1}\} \\ &= \sum_{m'}P\{S_k=m,R_k|S_{k-1}=m',\mathbf{R}_1^{k-1}\}\alpha_{k-1}(m)\end{aligned} \tag{B.13}$$

Again, the knowledge of R_1^{k-1} does not change the probability of S_k and R_k if S_{k-1} is known and thus

$$\begin{aligned}\alpha_k(m) &= \sum_{m'} P\{S_k = m, R_k | S_{k-1} = m'\}\alpha_{k-1}(m') \\ &= \sum_{m'} \gamma_k(m', m)\alpha_{k-1}(m') \end{aligned} \tag{B.14}$$

Since the α recursion starts from the beginning of the block forward it is also known as the *forward* recursion.

B.3 The β recursion

The values of $\beta_k(m)$ can also be computed recursively, starting with the values of $\beta_N(m)$ and using the recursive relation deduced below by using Bayes' rule (B.3):

$$\begin{aligned}\beta_k(m) &= \frac{P\{\mathbf{R}_{k+1}^N, S_k = m\}}{P\{S_k = m\}} \\ &= \frac{\sum_{m'} P\{\mathbf{R}_{k+1}^N, S_{k+1} = m', S_k = m\}}{P\{S_k = m\}} \\ &= \sum_{m'} P\{\mathbf{R}_{k+2}^N | S_{k+1} = m', S_k = m, \mathbf{R}_{k+1}\} P\{S_{k+1} = m, \mathbf{R}_{k+1} | S_k = m\} \\ &= \sum_{m'} P\{\mathbf{R}_{k+2}^N | S_{k+1} = m', S_k = m, \mathbf{R}_{k+1}\} \gamma_{k+1}(m, m') \end{aligned} \tag{B.15}$$

The knowledge of S_k and $\mathbf{R}_{k+1}$ does not change the probability of $\mathbf{R}_{k+2}^N$ if S_{k+1} is known and thus

$$\begin{aligned}\beta_k(m) &= \sum_{m'} P\{\mathbf{R}_{k+2}^N | S_{k+1} = m'\}\gamma_{k+1}(m, m') \\ &= \sum_{m'} \beta_{k+1}(m')\gamma_{k+1}(m, m') \end{aligned} \tag{B.16}$$

The starting values $\beta_N(m)$ cause the problem of *trellis termination*. If the β recursion is to be started with the values:

$$\beta_N(m) = \begin{cases} 1 & \text{if } m = 0 \\ 0 & \text{otherwise} \end{cases} \tag{B.17}$$

then the final code state has to be $S_N = 0$. In the case the trellis termination is not performed, the β recursion can be initialised in two ways:

- Random start: $\beta_N(m) = 1/2^M$ where M is the memory of the code.
- By using the already computed α values: $\beta_N(m) = \alpha_N(m)$.

Since the β recursion starts from the end of the block backward, it is also known as the *backward* recursion.

B.4 The transition probability

The transition probability at time k can be also determined by using Bayes' rule (B.3):

$$\begin{aligned} \gamma_k(m', m) &= P\{S_k = m, R_k | S_{k-1} = m'\} \\ &= P\{R_k | S_k = m, S_{k-1} = m'\} P\{S_k = m | S_{k-1} = m'\} \end{aligned} \quad \text{(B.18)}$$

The pair S_k, S_{k-1} determine the code bits associated with the R_k values. The statistical description of the channel is used to compute the first term. The second term is 1 if the transition is possible and zero otherwise.

Appendix C

Software

C.1 MPCCC simulation

```
/*
 * File: mpct.cpp
 * Author: A. Ambroze
 * Purpose: MPCCC implementation, using simple 1/2 RSC codes.
 */
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <math.h>
#include <stdlib.h>
#include "mpct.h"
#include "intl.h"

static int sinit(char*,int,int,int**,short*,short*,short*,int*,int*,int*,
 char*,char*,char*,char*,char*,char*);
short rsc(short ff,short fb,short nr_states,short st,short ib,short *parity);
static void normalise(double **buf,int block_length);

double (*distf)(double *dec0,double *dec1,double *decp0,double *decp1,int block_length);
double sq_dist(double *dec0,double *dec1,double *decp0,double *decp1,int block_length);
double abs_dist(double *dec0,double *dec1,double *decp0,double *decp1,int block_length);
double Labs_dist(double *dec0,double *dec1,double *decp0,double *decp1,int block_length);
double ce_dist(double *dec0,double *dec1,double *decp0,double *decp1,int block_length);
double max_dist(double *dec0,double *dec1,double *decp0,double *decp1,int block_length);

mpc::mpc(int block_len,int nrc,int nit,char *inits)
{
 short st,*ptmp;
 int cod,nr_states;

 //error status reset
 error = 0;

 max_nit = nit; block_length = block_len; nr_codes = nrc;

 //code and interleavers interface
 //the function below is in charge of correctly setting:
 //feed_forward, feed_backward, nr_states,do_normalise
 //the interleaver values etc.
 if((error = sinit(inits,block_length,nr_codes,
                   intlp,
                   feed_forward,feed_backward,c_nr_states,&tail_len,&tail_code,
                   &first_code,&beta_start,&use_ext,&do_normalise,&print_e,&print_m,
                   error_msg))) {
   return;
 }
```

```
 //code tables
 nr_states = 0;
 for(cod=0; cod<nrc; cod++) nr_states += c_nr_states[cod];
 if((c_next_st[0][0]=(short*)malloc(4*nr_states*nrc*sizeof(short))) == NULL) {
   sprintf(error_msg,"malloc error (code tables)\n");
   error = -1; return;
 }
 ptmp = c_next_st[0][0];
 for(cod=0; cod<nrc; cod++) {
   c_next_st[0][cod] = ptmp; ptmp += c_nr_states[cod];
   c_next_st[1][cod] = ptmp; ptmp += c_nr_states[cod];
   c_next_p[0][cod] = ptmp; ptmp += c_nr_states[cod];
   c_next_p[1][cod] = ptmp; ptmp += c_nr_states[cod];
 }

 //generate code tables
 for(cod=0; cod<nrc; cod++) {
   for(st=0; st<c_nr_states[cod]; st++) {
     c_next_st[0][cod][st] = rsc(feed_forward[cod],feed_backward[cod],
                                 c_nr_states[cod],st,0,c_next_p[0][cod]+st);
     c_next_st[1][cod][st] = rsc(feed_forward[cod],feed_backward[cod],
                                 c_nr_states[cod],st,1,c_next_p[1][cod]+st);
   }
 }

#ifdef EE_STATS
 ee_init_stats(inits);
#endif
}

static short kxor(short st)
{
 short u=0;
 while(st) {if(st&1) u=1-u; st >>= 1;}
 return u;
}
short rsc(short ff,short fb,short nr_states,short st,short ib,short *parity)
{
 short fbb=kxor(st&fb);
 if(fbb^ib) st |= nr_states; *parity = kxor(st&ff);
 return st>>1;
}

int mpc::hdist(char *info,int *fst0,int *ee_nr,int *ee_l,int *ee_ow)
{
 int  hdist;
 char *intl_info;
 int cod,bit;

  //memory allocation for intl_info
 if((intl_info=(char*)malloc(block_length)) == NULL) return -1;

 hdist = 0;
 for(bit=0; bit<block_length; bit++) hdist += info[bit];
 for(cod=0; cod<nr_codes; cod++) {
   int st,st_prev;
   for(bit=0; bit<block_length; bit++) intl_info[intlp[cod][bit]] = info[bit];
   st = ee_l[cod] = ee_ow[cod] = ee_nr[cod] = 0;
   for(bit=0; bit<block_length; bit++) {
     st_prev = st;
     ee_ow[cod] += c_next_p[intl_info[bit]][cod][st];
     st = c_next_st[intl_info[bit]][cod][st];
     //error event counting
     if(st!=0) {if(st_prev==0) ++ee_nr[cod]; ++ee_l[cod];}
   }
   fst0[cod] = st; hdist += ee_ow[cod];
 }
 free(intl_info);
 return hdist;
}

//ml helper function
```

```
//systematic block is first
//dist[0] contains total distance, dist[cod>=1], distance for each code
void get_ml(double *rec,double *enc,int nr_codes,int block_length,double *dist)
{
 int bit,cod;
 double sys_dist;
 for(cod=0; cod<=nr_codes; cod++) {
   dist[cod] = 0;
   for(bit=0; bit<block_length; bit++) dist[cod] -= (*enc++)*(*rec++);
 }
 //now dist[0] contains systematic, the rest the parity distance for each code
 sys_dist = dist[0];
 //total distance
 for(cod=1; cod<=nr_codes; cod++) {dist[0] += dist[cod];dist[cod] += sys_dist;}
}
//distance to the received sequence
int mpc::is_ml(double *rec,char *info,char *dec,double *info_dist,double *dec_dist)
{
 double *coded;
 //allocate memory for encoded
 if((coded=(double*)malloc((nr_codes+1)*block_length*sizeof(double)))==NULL)
   return -1;
 //encode info
 code(coded,info);
 //determine distance
 get_ml(rec,coded,nr_codes,block_length,info_dist);
 //encode dec
 code(coded,dec);
 //determine distance
 get_ml(rec,coded,nr_codes,block_length,dec_dist);
 //cleanup
 free(coded);
 return dec_dist[0]<=info_dist[0];
}

//encodes: |N sys|N p1|N p2|...
int mpc::code(double *coded,char *info)
{
 int  cod,i;
 short  st;
 double *sys_bit,*p_bit[MAX_NR_CODES];
 char  *intl_info;

 //memory allocation for intl_info
 if((intl_info=(char*)malloc(block_length)) == NULL)
   return -1;

 //the coded channels
 sys_bit = coded;
 for(cod=0; cod<nr_codes; cod++) {coded += block_length; p_bit[cod] = coded;}

 //data tail -- long and problematic, isn't it?
 //(I wonder if it's worth the trouble!)
 //changes info in rather 'random' places
 if(tail_len>0) {
   for(i=0; i<block_length; i++) intl_info[intlp[tail_code][i]] = info[i];
 st = 0;
 for(i=0; i<block_length-tail_len; i++) {
   p_bit[tail_code][i] = c_next_p[intl_info[i]][tail_code][st] ? 1.0:-1.0;
   st = c_next_st[intl_info[i]][tail_code][st];
 }
 for(; i<block_length; i++) {
   if(st==0 || st>c_next_st[0][tail_code][st]) intl_info[i] = 0;
   else                                       intl_info[i] = 1;
   p_bit[tail_code][i] = c_next_p[intl_info[i]][tail_code][st] ? 1.0:-1.0;
   st = c_next_st[intl_info[i]][tail_code][st];
 }
 if(st != 0) {   //check if data tail is working
   sprintf(error_msg,"data tail error (st=%hd!=0)\n",st); return -2;
 }

 //MPCCC encoding
 for(i=0; i<block_length; i++) sys_bit[i] = info[i] ? 1.0:-1.0;
```

```
 for(cod=0; cod<nr_codes; cod++) {
   if(cod != tail_code) {
     for(i=0; i<block_length; i++) intl_info[intlp[cod][i]] = info[i];
     st = 0;
     for(i=0; i<block_length; i++) {
       p_bit[cod][i] = c_next_p[intl_info[i]][cod][st] ? 1.0:-1.0;
       st = c_next_st[intl_info[i]][cod][st];
     }
   }
 }
 free(intl_info);
 return 0;
}

int mpc::map(double **rec,double **ext,double **dec)
{
 int  i,st;
 double *alpha0,*alpha1,sum;
 double *beta0,*beta1,*beta_swap;
 double *rec_sys0=*rec,*rec_sys1=*(rec+1);
 double **rec_p=rec+2;
 double ext_dec0,ext_dec1;
 //code tables
 short *next_st0 = c_next_st[0][current_code];
 short *next_st1 = c_next_st[1][current_code];
 short *next_p0 = c_next_p[0][current_code];
 short *next_p1 = c_next_p[1][current_code];
 int nr_states = c_nr_states[current_code];

 //initialize alpha recursion
 memset(alpha,0,(block_length+1)*nr_states*sizeof(double));
 *alpha = 1; alpha1 = alpha;
 //alpha recursion
 for(i=0; i<block_length; i++) {
   alpha0 = alpha1; alpha1 += nr_states;
   for(st=0; st<nr_states; st++) {
     alpha1[next_st0[st]] += alpha0[st]*rec_sys0[i]*rec_p[next_p0[st]][i];
     alpha1[next_st1[st]] += alpha0[st]*rec_sys1[i]*rec_p[next_p1[st]][i];
   }
   sum = 0;
   for(st=0; st<nr_states; st++) sum += alpha1[st];
   for(st=0; st<nr_states; st++) alpha1[st] /= sum;
 }
 //beta init
 //beta_start=='z' should be used with all zero info
 if((tail_code==current_code)||(beta_start=='z')) {
memset(beta,0,nr_states*sizeof(double)); beta[0] = 1;
 }
 else {
 switch(beta_start) {
 case 'a':
  for(st=0; st<nr_states; st++) beta[st] = alpha1[st];
break;
 default:
 for(st=0; st<nr_states; st++) beta[st] = 1.0/nr_states;
 }
 }
 beta0 = beta; beta1 = beta + nr_states;
 //beta recursion and decoding
 for(i=block_length-1; i>=0; i--) {
   alpha1 -= nr_states;
   sum = 0;
   ext_dec0 = ext_dec1 = 0;
   for(st=0; st<nr_states; st++) {
     ext_dec0 += alpha1[st]*rec_p[next_p0[st]][i]*beta0[next_st0[st]];
     ext_dec1 += alpha1[st]*rec_p[next_p1[st]][i]*beta0[next_st1[st]];

     beta1[st] = beta0[next_st0[st]]*rec_sys0[i]*rec_p[next_p0[st]][i];
     beta1[st] += beta0[next_st1[st]]*rec_sys1[i]*rec_p[next_p1[st]][i];
     sum += beta1[st];
   }
   ext[0][i] = ext_dec0; ext[1][i] = ext_dec1;
   for(st=0; st<nr_states; st++) beta1[st] /= sum;
```

```
   //swap beta pointers
   beta_swap = beta1; beta1 = beta0; beta0 = beta_swap;
 }
 if(dec != NULL) {
   for(i=0; i<block_length; i++) {
     dec[0][i] = ext[0][i]*rec_sys0[i]; dec[1][i] = ext[1][i]*rec_sys1[i];
   }
 }
 return 0;
}

//at the moment, works with no noise for 0,1 as well as -1,+1 received
//added gaussian noise if sigma!=0
static void get_prob(double **prob,double *rec,int block_length,double sigma)
{
 int i;

 if(sigma <= 0) {
   for(i=0; i<block_length; i++) {
     if(*rec++>0) {prob[0][i] = 0; prob[1][i] = 1;}
     else         {prob[0][i] = 1; prob[1][i] = 0;}
   }
 }
 else {
   for(i=0; i<block_length; i++) {
     prob[0][i] = exp(-((*rec+1)*(*rec+1))/(2*sigma*sigma));
     prob[1][i] = exp(-((*rec-1)*(*rec-1))/(2*sigma*sigma));
     ++rec;
   }
 }
}

int mpc::decode(double *rec,char *dec,int *ber_per_map,double sigma,
int qstop,double Mdist)
{
 int cod,it,bit,i;
 double *o_ext[2],*o_dec[2],*tmp;
 double *code_input[MAX_NR_CODES][4];
 double *sys_prob[2];
 int maps,max_nr_states=0,errs,stop=0;
 double *dec_prev=NULL;

#ifdef EE_STATS
 ++blocks;
#endif

 ////////////////////////////
 //POINTER AND MEMORY INIT//
 ////////////////////////////
 //temporary memory allocation
 //max_nr_states
 for(cod=0; cod<nr_codes; cod++)
   if(max_nr_states<c_nr_states[cod]) max_nr_states = c_nr_states[cod];
 //alpha and beta memory (beta is a switched buffer)
 if((beta=(double*)malloc((block_length+3)*max_nr_states*sizeof(double)))
    == NULL) return -1;
 alpha = beta+2*max_nr_states;
 //output extrinsic
 if((o_ext[0]=(double*)malloc((4*nr_codes+6)*block_length*sizeof(double)))
    == NULL) {free(beta); return -1;}
 o_ext[1] = o_ext[0]+block_length; tmp = o_ext[1];
 //memory used so far = 2 blocks
 //output decoded
 tmp += block_length; o_dec[0] = tmp; tmp += block_length; o_dec[1] = tmp;
 //memory used so far = 4 blocks
 //code input (sys*i_ext,parity)
 for(cod=0; cod<nr_codes; cod++)
   for(i=0; i<4; i++) {tmp += block_length; code_input[cod][i] = tmp;}
 //memory used so far = 4*nr_codes+4 blocks
 //systematic channel, which has to be interleaved for each code
 tmp += block_length; sys_prob[0] = tmp; tmp += block_length; sys_prob[1] = tmp;
 //Total memory = 4*nr_codes+6 blocks
 //POINTER init
```

```
 //from received to probabilities (rec --> sys_prob,code_input)
 get_prob(sys_prob,rec,block_length,sigma);
 if(do_normalise) normalise(sys_prob,block_length);
 for(cod=0; cod<nr_codes; cod++) {
#ifdef EE_STATS
 //interleave systematic probs
 for(bit=0; bit<block_length; bit++) {
   code_input[cod][0][intlp[cod][bit]] = 1;
   code_input[cod][1][intlp[cod][bit]] = 1;
 }
#else
 //interleave systematic probs
 for(bit=0; bit<block_length; bit++) {
   code_input[cod][0][intlp[cod][bit]] = sys_prob[0][bit];
   code_input[cod][1][intlp[cod][bit]] = sys_prob[1][bit];
 }
#endif
 //parity probs
 rec += block_length;
 get_prob(code_input[cod]+2,rec,block_length,sigma);
 if(do_normalise) normalise(code_input[cod]+2,block_length);

 ////////////////////
 //The iterative loop//
 ////////////////////
 maps = 0;
 //assume errors; only introduced this for maps=nrc*nit+1
 //this makes it similar to fsc
 if(print_e) printf("\nerrs: ");
 if(print_m) printf("\nmetric: ");
 for(it=0; it<max_nit; it++) {
   current_code = first_code;
   for(cod=0; cod<nr_codes; cod++) {
     int tg_cod;

     //decode <cod>

#ifdef EE_STATS
     if(do_stats) {
       switch(cc_func) {
       //input systematic
       case 's':
       case 'S':
         for(bit=0; bit<block_length; bit++) {
           eed_iext[0][intlp[current_code][bit]] = sys_prob[0][bit];
           eed_iext[1][intlp[current_code][bit]] = sys_prob[1][bit];
         }
         break;
       //input parity
       case 'p':
       case 'P':
         memcpy(eed_iext[0],code_input[current_code][2],
                block_length*sizeof(double));
         memcpy(eed_iext[1],code_input[current_code][3],
                block_length*sizeof(double));
         break;
       default:
       //save input extrinsic
         memcpy(eed_iext[0],code_input[current_code][0],
                block_length*sizeof(double));
         memcpy(eed_iext[1],code_input[current_code][1],
                block_length*sizeof(double));
         if(do_normalise) normalise(eed_iext,block_length);
       }
     }
     //multiply code_input by the systematic input
     for(bit=0; bit<block_length; bit++) {
       code_input[current_code][0][intlp[current_code][bit]]*=sys_prob[0][bit];
       code_input[current_code][1][intlp[current_code][bit]]*=sys_prob[1][bit];
     }
#endif
     if(do_normalise) normalise(code_input[current_code],block_length);
     map(code_input[current_code],o_ext,o_dec);
```

```
    if(do_normalise) normalise(o_ext,block_length);
    ++maps;

#ifdef EE_STATS
    if(do_stats) {
      memcpy(eed_oext[0],o_ext[0],block_length*sizeof(double));
      memcpy(eed_oext[1],o_ext[1],block_length*sizeof(double));
      ee_collect_stats(eed_iext,eed_oext,o_dec,it,current_code);
    }
#endif
    //determine errors
    errs = 0;
    for(bit=0; bit<block_length; bit++) {
      double prob0,probi;
      prob0 = o_dec[0][intlp[current_code][bit]];
      probi = o_dec[1][intlp[current_code][bit]];
      //don't be surprised by the decoding formula,
      //it's justified by a very old bug to do with NaN values
      if((prob0>probi&&dec[bit]==0)||(prob0<probi&&dec[bit]==1));
      else ++errs;
    }
    ber_per_map[maps] += errs;
    if(print_e) printf("%d;",errs);
    //test stop condition
    switch(qstop) {
    case 1:
      if(errs==0) stop = 1;
    break;
    case 2:
      //because this can stop with non-zero errors, ber_per_map calculation
      //assumption of non-zero errors after stop is not true, so do not rely
      //on it; should patch this sometimes;
      if(current_code != first_code) break;
      if(it==1) {
        dec_prev=(double*)malloc(2*block_length*sizeof(double));
        if(dec_prev!=NULL) {
          for(i=0; i<block_length; i++) {
            dec_prev[i] = o_ext[0][i]; dec_prev[block_length+i] = o_ext[1][i];
          }
        }
      }
      else {
        if(dec_prev!=NULL) {
          double dist=distf(o_ext[0],o_ext[1],dec_prev,dec_prev+block_length,block_length);
          memcpy(dec_prev,o_ext[0],block_length*sizeof(double));
          memcpy(dec_prev+block_length,o_ext[1],block_length*sizeof(double));
          if(print_m) printf("%g;",dist);
          if(dist<=Mdist) stop = 1;
        }
      }
      //free memory on convergence or maxit
      if((stop||it==max_nit-1)&&dec_prev!=NULL)
        free(dec_prev); dec_prev = NULL; //just in case
      break;
    }
    if(stop) break;

    //distribute extrinsic
    for(tg_cod=0; tg_cod<nr_codes; tg_cod++) {
      if(tg_cod != current_code) {
        for(bit=0; bit<block_length; bit++) {
          code_input[tg_cod][0][intlp[tg_cod][bit]] *=
            o_ext[0][intlp[current_code][bit]];
          code_input[tg_cod][1][intlp[tg_cod][bit]] *=
            o_ext[1][intlp[current_code][bit]];
        }
      }
    }
#ifdef EE_STATS
    //reset input extrinsic to 1
    for(bit=0; bit<block_length; bit++) {
      code_input[current_code][0][intlp[current_code][bit]] = 1;
      code_input[current_code][1][intlp[current_code][bit]] = 1;
```

```
    }
#else
    //reset input extrinsic to systematic received
    for(bit=0; bit<block_length; bit++) {
      code_input[current_code][0][intlp[current_code][bit]]=sys_prob[0][bit];
      code_input[current_code][1][intlp[current_code][bit]]=sys_prob[1][bit];
    }
#endif
    //increment current code
    if(cod<nr_codes-1) if(++current_code>=nr_codes) current_code = 0;
  }
  if(stop) break;
 }

 /////////////////
 //DECISION TIME//
 /////////////////
 for(bit=0; bit<block_length; bit++)
   if(o_dec[1][intlp[current_code][bit]]>=o_dec[0][intlp[current_code][bit]]) dec[bit] = 1;
   else                                                                       dec[bit] = 0;
 //free memory
 free(beta); free(o_ext[0]);
 return stop ? maps:maps+1;
}

mpc::~mpc() {
 int cod;
 if(!error) {
   free(c_next_st[0][0]); for(cod=0; cod<nr_codes; cod++) free(intlp[cod]);
 }
#ifdef EE_STATS
 if(do_stats) ee_print_stats();
#endif
}

/////////////////
//normalisation//
/////////////////
static void normalise(double **buf,int block_length)
{
  int bit;
  double nval;

  for(bit=0; bit<block_length; bit++) {
    if((nval=buf[0][bit]+buf[1][bit])==0) {
      fprintf(stderr,"normalise: division by zero\n"); exit(2);
    }
    buf[0][bit] /= nval; buf[1][bit] /= nval;
  }
}

////////////////////////////////////
//extrinsic info and decoded stats//
////////////////////////////////////

#ifdef EE_STATS
//helper functions
double*** calloc3(int nit,int nr_codes,int block_length)
{
  double ***ret,*buf;
  int it,cod;
  if((ret=(double***)malloc(nit*sizeof(double**)))==NULL) return NULL;
  if((ret[0]=(double**)malloc(nr_codes*nit*sizeof(double*)))==NULL) {
    free(ret); return NULL;
  }
  for(it=1; it<nit; it++) ret[it] = ret[it-1]+nr_codes;
  if((buf=(double*)calloc(nr_codes*nit*block_length,sizeof(double)))==NULL) {
    free(ret[0]); free(ret);return NULL;
  }
  for(it=0; it<nit; it++) for(cod=0; cod<nr_codes; cod++) {
    ret[it][cod] = buf; buf += block_length;
  }
  return ret;
```

```
}
void free3(double ***ret)
{
  free(ret[0][0]); free(ret[0]); free(ret);
}
//this is biig memory...
double**** calloc4(int nit,int nr_codes,int block_length)
{
  double ****ret,**buf,*big_buf;
  int it,cod,bit;
  if((ret=(double****)malloc(nit*sizeof(double***)))==NULL) return NULL;
  if((ret[0]=(double***)malloc(nr_codes*nit*sizeof(double**)))==NULL) {
    free(ret); return NULL;
  }
  for(it=1; it<nit; it++) ret[it] = ret[it-1]+nr_codes;
  if((buf=(double**)malloc(nr_codes*nit*block_length*sizeof(double*)))==NULL) {
    free(ret[0]); free(ret); return NULL;
  }
  for(it=0; it<nit; it++) for(cod=0; cod<nr_codes; cod++) {
    ret[it][cod] = buf;
    buf += block_length;
  }
  //the big memory is here
  if((big_buf=(double*)calloc(nr_codes*nit*block_length*block_length,sizeof(double)))==NULL) {
    //no big surprise
    free(ret[0][0]); free(ret[0]); free(ret); return NULL;
  }
  for(it=0; it<nit; it++) for(cod=0; cod<nr_codes; cod++)
    for(bit=0; bit<block_length; bit++) {
      ret[it][cod][bit] = big_buf; big_buf += block_length;
    }
  return ret;
}
void free4(double ****ret)
{
  free(ret[0][0][0]); free(ret[0][0]); free(ret[0]); free(ret);
}

void mpc::ee_init_stats(char *inits)
{
  char *s;
  //reset block counter
  blocks = 0;
  //read init string
  if(inits==NULL) {do_stats = 0; return;}
  if((s=strstr(inits,"s="))==NULL) {do_stats = 0; return;}
  do_stats = 1;

  cc_func = s[2]; use_logl = 0;
  if(s[3]=='L') {
    fprintf(stderr,"eed_init_stats: using log likelihoods for correlation\n");
    use_logl = 1;
  }
  //averages
  if((eed_iavg=calloc3(max_nit,nr_codes,block_length))==NULL) {
    error = -1; strcpy(error_msg,"ee_init_stats: calloc3 error (eed_iavg)\n");
    return;
  }
  if((eed_oavg=calloc3(max_nit,nr_codes,block_length))==NULL) {
    error = -1; strcpy(error_msg,"ee_init_stats: calloc3 error (eed_oavg)\n");
    return;
  }
  //if not average calculation, average tables need to be loaded
  if('A'<=cc_func && 'Z'>=cc_func) {
    FILE *f;
    int it,cod,bit;
    char eed_iavg_fname[20]="eed_ _iavg";

    eed_iavg_fname[4]=tolower(cc_func);
    if((f=fopen(eed_iavg_fname,"rt")) == NULL) {
      error = -2;
      strcpy(error_msg,"eed_init_stats: can't open average file ");
      strcat(error_msg,eed_iavg_fname); strcat(error_msg,"\n");
```

```
    return;
  }
  for(it=0; it<max_nit; it++) {
    for(cod=0; cod<nr_codes; cod++) {
    for(bit=0; bit<block_length; bit++) {
      if(fscanf(f,"%lf",&eed_iavg[it][cod][bit])==EOF) {
        error = -2; strcpy(error_msg,"eed_init_stats: unexpected EOF in eed_iavg\n");
        fclose(f); return;
      }
    }
    fprintf(f,"\n");
    }
  }
  fclose(f);

  if((f=fopen("eed_oavg","rt")) == NULL) {
    error = -2; strcpy(error_msg,"eed_init_stats: can't open average file eed_oavg\n");
    return;
  }
  for(it=0; it<max_nit; it++) {
    for(cod=0; cod<nr_codes; cod++) {
    for(bit=0; bit<block_length; bit++) {
      if(fscanf(f,"%lf",&eed_oavg[it][cod][bit])==EOF) {
        error = -2; strcpy(error_msg,"eed_init_stats: unexpected EOF in eed_oavg\n");
        fclose(f); return;
      }
    }
    fprintf(f,"\n");
    }
  }
  fclose(f);
}
//i_extrinsic,o_extrinsic
//o_extrinsic is only allocated because of log_likelihood late implementation
if((eed_iext[0]=(double*)malloc(4*block_length*sizeof(double)))==NULL) {
  error = -1; strcpy(error_msg,"ee_init_stats: malloc error (eed_iext)\n");
  return;
}
eed_iext[1] = eed_iext[0]+block_length;
eed_oext[0] = eed_iext[1]+block_length;
eed_oext[1] = eed_oext[0]+block_length;

switch(cc_func) {
case 'c':
case 'p':
case 's':
case 'x':
  break;
case 'X':
  //correlation
  if((eed_inorm=calloc3(max_nit,nr_codes,block_length))==NULL) {
    error = -1; strcpy(error_msg,"ee_init_stats: calloc3 error (eed_inorm)\n");
    return;
  }
  if((eed_onorm=calloc3(max_nit,nr_codes,block_length))==NULL) {
    error = -1; strcpy(error_msg,"ee_init_stats: calloc3 error (eed_onorm)\n");
    return;
  }
  if((eed_corr=calloc4(max_nit,nr_codes,block_length))==NULL) {
    error = -1; strcpy(error_msg,"ee_init_stats: calloc4 error (eed_corr)\n");
    return;
  }
  break;
case 'C':
case 'P':
case 'S':
  cbit=block_length/2;
  if((eed_inorm=calloc3(max_nit,nr_codes,block_length))==NULL) {
    error = -1; strcpy(error_msg,"ee_init_stats: calloc3 error (eed_inorm)\n");
    return;
  }
  if((eed_cbit_onorm=(double*)calloc(max_nit*nr_codes,sizeof(double)))==NULL) {
    error = -1; strcpy(error_msg,"ee_init_stats: calloc error (eed_cbit_onorm)\n");
```

```
      return;
    }
    if((eed_cbit_corr=calloc3(max_nit,nr_codes,block_length))==NULL) {
      error = -1; strcpy(error_msg,"ee_init_stats: calloc3 error (eed_cbit)\n");
      return;
    }
    break;
  default:
    error = -2; sprintf(error_msg,"ee_init_stats: no such cc_func (%c)\n",cc_func);
    return;
  }
}

void mpc::ee_collect_stats(double **rec,double **ext,double **dec,int it,
   int cod)
{
  int bit,bit2;
  double val;
  //rec and ext are volatile buffers for eed, they can be changed here
  //need to change them for the use_logl option (L)
  if(use_logl) {
    for(bit=0; bit<block_length; bit++) {
      rec[0][bit] = log(rec[0][bit]/(rec[1][bit]+TINY));
      ext[0][bit] = log(ext[0][bit]/(ext[1][bit]+TINY));
    }
  }

  switch(cc_func) {
  case 'c':
  case 'p':
  case 's':
  case 'x':
    //averages
    for(bit=0; bit<block_length; bit++) {
      eed_iavg[it][cod][bit] += rec[0][bit]; eed_oavg[it][cod][bit] += ext[0][bit];
    }
    break;
  case 'X':
    //correlation
    for(bit=0; bit<block_length; bit++) {
      //norm
      val = rec[0][bit]-eed_iavg[it][cod][bit]; eed_inorm[it][cod][bit] += val*val;
      val = ext[0][bit]-eed_oavg[it][cod][bit]; eed_onorm[it][cod][bit] += val*val;
      for(bit2=0; bit2<block_length; bit2++) {
        eed_corr[it][cod][bit][bit2] +=
          (rec[0][bit2]-eed_iavg[it][cod][bit2])*(ext[0][bit]-eed_oavg[it][cod][bit]);
      }
    }
    break;
  case 'C':
  case 'P':
  case 'S':
    val = ext[0][cbit]-eed_oavg[it][cod][cbit]; eed_cbit_onorm[it*nr_codes+cod] += val*val;
    for(bit=0; bit<block_length; bit++) {
      val = rec[0][bit]-eed_iavg[it][cod][bit]; eed_inorm[it][cod][bit] += val*val;
      eed_cbit_corr[it][cod][bit] +=
        (rec[0][bit]-eed_iavg[it][cod][bit])*(ext[0][cbit]-eed_oavg[it][cod][cbit]);
    }
    break;
  }
}

void mpc::ee_print_stats()
{
  FILE *f;
  int it,cod,bit,bit2;
  char fname[100];
  double val;

  switch(cc_func) {
  case 'c':
  case 'p':
  case 's':
```

```
case 'x':
  strcpy(fname,"eed_ _iavg"); fname[4]=cc_func;
  if((f=fopen(fname,"wt")) != NULL) {
    for(it=0; it<max_nit; it++) {
      for(cod=0; cod<nr_codes; cod++) {
        for(bit=0; bit<block_length; bit++) {
          fprintf(f,"%.10f ",eed_iavg[it][cod][bit]/blocks);
        }
        fprintf(f,"\n");
      }
    }
    fclose(f);
  }
  if((f=fopen("eed_oavg","wt")) != NULL) {
    for(it=0; it<max_nit; it++) {
      for(cod=0; cod<nr_codes; cod++) {
        for(bit=0; bit<block_length; bit++) {
          fprintf(f,"%.10f ",eed_oavg[it][cod][bit]/blocks);
        }
        fprintf(f,"\n");
      }
    }
    fclose(f);
  }
break;
case 'X':
  for(it=0; it<max_nit; it++) {
    for(cod=0; cod<nr_codes; cod++) {
      for(bit=0; bit<block_length; bit++) {
        for(bit2=0; bit2<block_length; bit2++) {
          val = eed_inorm[it][cod][bit2]*eed_onorm[it][cod][bit];
          eed_corr[it][cod][bit][bit2] /= (sqrt(val)+TINY);
          if(eed_corr[it][cod][bit][bit2]<0)
            eed_corr[it][cod][bit][bit2] = -eed_corr[it][cod][bit][bit2];
        }
      }
      //write
      sprintf(fname,"eed_corr_i%d_c%d.bin",it,cod);
      if((f=fopen(fname,"wb"))!=NULL) {
        if(!fwrite(eed_corr[it][cod][0],block_length*block_length*sizeof(double),1,f)) {
          fprintf(stderr,"eed_print_stats: could not write data to disk\n");
        }
        fclose(f);
      }
      else fprintf(stderr,"eed_print_stats: could not open file %s\n",fname);
    }
  }

  free3(eed_inorm); free3(eed_onorm); free4(eed_corr);
break;
case 'C':
case 'P':
case 'S':
  sprintf(fname,"eed_%c_corr_%d",(char)tolower(cc_func),cbit);
  if((f=fopen(fname,"wt")) != NULL) {
    for(it=0; it<max_nit; it++) {
      for(cod=0; cod<nr_codes; cod++) {
        for(bit=0; bit<block_length; bit++) {
          val = eed_inorm[it][cod][bit]*eed_cbit_onorm[it*nr_codes+cod];
          eed_cbit_corr[it][cod][bit] /= (sqrt(val)+TINY);
          if(eed_cbit_corr[it][cod][bit]<0)
            eed_cbit_corr[it][cod][bit] = -eed_cbit_corr[it][cod][bit];
          fprintf(f,"%.10f ",eed_cbit_corr[it][cod][bit]);
        }
        fprintf(f,"\n");
      }
    }
    fclose(f);
  }
  else fprintf(stderr,"eed_print_stats: could not open file %s\n",fname);
  free(eed_cbit_onorm); free3(eed_inorm); free3(eed_cbit_corr);
break;
}
```

```
  free(eed_iext[0]); free3(eed_iavg); free3(eed_oavg);
}
#endif

//distances
double abs_dist(double *dec0,double *dec1,double *decp0,double *decp1,int block_length)
{
 double dist=0,dd;
 int i;

 for(i=0; i<block_length; i++) {
  dd=dec0[i]-decp0[i]; if(dd<0) dd=-dd; dist += dd;   //prob of 0
  dd=dec1[i]-decp1[i]; if(dd<0) dd=-dd; dist += dd;   //prob of 1
 }
 dist /= (2*block_length);
 return dist;
}
double Labs_dist(double *dec0,double *dec1,double *decp0,double *decp1,int block_length)
{
 double dist=0,dd,dc,dp;
 int i;

 for(i=0; i<block_length; i++) {
   //log likelihoods
   dc=log(dec0[i]/(dec1[i]+TINY)+TINY); dp=log(decp0[i]/(decp1[i]+TINY)+TINY);
   dd=dc-dp; if(dd<0) dd=-dd; dist += dd;
 }
 dist /= block_length;
 return dist;
}
double sq_dist(double *dec0,double *dec1,double *decp0,double *decp1,int block_length)
{
 double dist=0,dd;
 int i;

 for(i=0; i<block_length; i++) {
   dd=dec0[i]-decp0[i]; dist += dd*dd;    //prob of 0
   dd=dec1[i]-decp1[i]; dist += dd*dd;    //prob of 1
 }
 dist = sqrt(dist)/(2*block_length);
 return dist;
}
//cross entropy
double ce_dist(double *dec0,double *dec1,double *decp0,double *decp1,int block_length)
{
 double dist=0;
 int i;

 for(i=0; i<block_length; i++)
   dist += decp1[i]*log(decp1[i]/(dec1[i]+TINY))+decp0[i]*log(decp0[i]/(dec0[i]+TINY));
   if(dist<0) dist = -dist;
 return dist;
}
//Max difference
double max_dist(double *dec0,double *dec1,double *decp0,double *decp1,int block_length)
{
 double dist=0,dd;
 int i;

 for(i=0; i<block_length; i++) {
   dd=dec0[i]-decp0[i]; if(dd<0) dd = -dd; if(dist<dd) dist=dd;  //prob of 0
   dd=dec1[i]-decp1[i]; if(dd<0) dd = -dd; if(dist<dd) dist=dd;  //prob of 1
 }
 return dist;
}
```

C.2 SCCC simulation

```
/*
 * File: sc.cpp
 * Author: A. Ambroze
 * Purpose: Fast rate 1/4 sccc
 */
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <math.h>
#include "sc.h"
#include "intl.h"

sc::sc(int block_len,char *inits)
{
  int st;
  short *tmp;

  error = 0;
  if(block_len<=0) {
    error = -1; sprintf(error_message,"Block length (%d) should be positive",block_len);
    return;
  }
  block_length = block_len;
  //parse parameter and option string
  parse_inits(inits);
  //generate code tables
  if((ci_next_st0=(short*)malloc(12*(ci_nr_states+co_nr_states)*sizeof(short))) == NULL) {
    error = -1; strcpy(error_message,"malloc error\n");
    return;
  }
  tmp = ci_next_st0; ci_next_st1 = tmp += ci_nr_states;
  ci_next_b0_p1 = tmp += ci_nr_states; ci_next_b1_p1 = tmp += ci_nr_states;
  ci_next_b0_p2 = tmp += ci_nr_states; ci_next_b1_p2 = tmp += ci_nr_states;

  co_next_st0 = tmp += ci_nr_states; co_next_st1 = tmp += co_nr_states;
  co_next_b0_p1 = tmp += co_nr_states; co_next_b1_p1 = tmp += co_nr_states;
  co_next_b0_p2 = tmp += co_nr_states; co_next_b1_p2 = tmp += co_nr_states;

  for(st=0; st<ci_nr_states; st++) {
    ci_next_st0[st] = cc(ci_ff,ci_fb,ci_nr_states,st,0,ci_next_b0_p1+st,ci_next_b0_p2+st);
    ci_next_st1[st] = cc(ci_ff,ci_fb,ci_nr_states,st,1,ci_next_b1_p1+st,ci_next_b1_p2+st);
  }
  for(st=0; st<co_nr_states; st++) {
    co_next_st0[st] = cc(co_ff,co_fb,co_nr_states,st,0,co_next_b0_p1+st,co_next_b0_p2+st);
    co_next_st1[st] = cc(co_ff,co_fb,co_nr_states,st,1,co_next_b1_p1+st,co_next_b1_p2+st);
  }
  //allocate memory for iterative decoder
  //for N=20000 and M=5 --> mem=7Mbytes !!
  if(mem_alloc()<0) return;
}

static short kxor(short st)
{
 short u=0;
 while(st) {if(st&1) u=1-u; st >>= 1;}
 return u;
}

short sc::cc(short *ff,short fb,short nr_states,short st,short ib,short *p1,short *p2)
{
 short fbb=kxor(st&fb);
 if(fbb^ib) st |= nr_states;
 if(ff[0]<=0) *p1 = ib;
 else         *p1 = kxor(st&ff[0]);
 if(ff[1]<=0) *p2 = ib;
 else         *p2 = kxor(st&ff[1]);
 return st>>1;
}

int sc::hdist(char *info,char *fst0)
{
```

```
  int hdist,bit,ret;
  double *coded;

  if((coded=(double*)malloc(4*block_length*sizeof(double))) == NULL) {
    error = -1; strcpy(error_message,"hdist: malloc error");
    return error;
  }
  if((ret=code(coded,info,fst0))<0) return ret;
  hdist = 0;
  for(bit=0; bit<4*block_length; bit++) if(coded[bit]>0) ++hdist;
  free(coded);
  return hdist;
}

int sc::code(double *coded,char *info,char *fst0)
{
  int bit;
  short st;
  short  *ci_next_st[2]={ci_next_st0,ci_next_st1};
  short  *ci_next_p1[2]={ci_next_b0_p1,ci_next_b1_p1};
  short  *ci_next_p2[2]={ci_next_b0_p2,ci_next_b1_p2};
  short  *co_next_st[2]={co_next_st0,co_next_st1};
  short  *co_next_p1[2]={co_next_b0_p1,co_next_b1_p1};
  short  *co_next_p2[2]={co_next_b0_p2,co_next_b1_p2};
  char *co_coded,*ci_info,*co_p;
  double *ci_p;

  if((co_coded=(char*)malloc(4*block_length)) == NULL) {
    error = -1; strcpy(error_message,"code: malloc error");
    return error;
  }
  ci_info = co_coded+2*block_length;

  //code outer
  st = 0; co_p = co_coded;
  for(bit=0; bit<block_length; bit++) {
    *co_p++ = (char)co_next_p1[info[bit]][st]; *co_p++ = (char)co_next_p2[info[bit]][st];
    st = co_next_st[info[bit]][st];
  }
  if(fst0 != NULL) {
    if(st==0) fst0[0] = 1;
    else      fst0[0] = 0;
  }
  //interleave
  for(bit=0; bit<2*block_length; bit++) ci_info[intlp[bit]] = co_coded[bit];
  //code inner
  st = 0;ci_p = coded;
  for(bit=0; bit<2*block_length; bit++) {
    *ci_p++ = ci_next_p1[ci_info[bit]][st] ? 1.0:-1.0;
    *ci_p++ = ci_next_p2[ci_info[bit]][st] ? 1.0:-1.0;

    st = ci_next_st[ci_info[bit]][st];
  }
  if(fst0 != NULL) {
    if(st==0) fst0[1] = 1;
    else      fst0[1] = 0;
  }

  free(co_coded);
  return 0;
}
/*
 * inner code (ci) map
 * inputs: 2 channel probs and 1 extrinsic
 * output: 1 extrinsic
 */

int sc::_ci_map()
{
 int  i,st,ci_block_length = 2*block_length;
 double *alpha0,*alpha1,sum;
 double *beta0,*beta1,*beta_swap;
 double **rec_p1=ci_rec,**rec_p2=ci_rec+2;
```

```
 double *i_ext0 = ci_i_ext[0],*i_ext1 = ci_i_ext[1];
 double *o_ext0=ci_o_ext[0],*o_ext1=ci_o_ext[1];

 //zero output buffers
 memset(o_ext0,0,ci_block_length*sizeof(double));
 memset(o_ext1,0,ci_block_length*sizeof(double));

 //initialize alpha recursion
 memset(alpha,0,(ci_block_length+1)*ci_nr_states*sizeof(double));
 *alpha = 1; alpha1 = alpha;
 //alpha recursion
 for(i=0; i<ci_block_length; i++) {
   alpha0 = alpha1; alpha1 += ci_nr_states;

   for(st=0; st<ci_nr_states; st++) {
     alpha1[ci_next_st0[st]] +=
       alpha0[st]*i_ext0[i]*rec_p1[ci_next_b0_p1[st]][i]*rec_p2[ci_next_b0_p2[st]][i];
     alpha1[ci_next_st1[st]] +=
       alpha0[st]*i_ext1[i]*rec_p1[ci_next_b1_p1[st]][i]*rec_p2[ci_next_b1_p2[st]][i];
   }
   sum = 0; for(st=0; st<ci_nr_states; st++) sum += alpha1[st];
   if(sum==0) {
     sprintf(error_message,"Inner decoder error (alpha)_%d\n",i);
     return -1;
   }
   for(st=0; st<ci_nr_states; st++) alpha1[st] /= sum;
 }
 //beta init
 if(beta_start=='z') {memset(beta,0,ci_nr_states*sizeof(double)); beta[0] = 1;}
 else {
   switch(beta_start) {
   case 'a':
     for(st=0; st<ci_nr_states; st++) beta[st] = alpha1[st];
     break;
   default:
     for(st=0; st<ci_nr_states; st++) beta[st] = 1.0/ci_nr_states;
   }
 }
 beta0 = beta; beta1 = beta + ci_nr_states;
 //beta recursion and decoding
 for(i=ci_block_length-1; i>=0; i--) {
   double tmp_dec;

   alpha1 -= ci_nr_states;
   sum = 0;
   for(st=0; st<ci_nr_states; st++) {
     tmp_dec = rec_p1[ci_next_b0_p1[st]][i]*rec_p2[ci_next_b0_p2[st]][i]*beta0[ci_next_st0[st]];
     o_ext0[i] += alpha1[st]*tmp_dec; beta1[st] = i_ext0[i]*tmp_dec;
     tmp_dec = rec_p1[ci_next_b1_p1[st]][i]*rec_p2[ci_next_b1_p2[st]][i]*beta0[ci_next_st1[st]];
     o_ext1[i] += alpha1[st]*tmp_dec; beta1[st] += i_ext1[i]*tmp_dec;
     sum += beta1[st];
   }

   if(sum==0||(o_ext0[i]==0&&o_ext1[i]==0)) {
     sprintf(error_message,"Inner decoder error (beta or dec)_%d\n",i);
     return -2;
   }
   for(st=0; st<ci_nr_states; st++) beta1[st] /= sum;
   beta_swap = beta1; beta1 = beta0; beta0 = beta_swap;
 }
 return 0;
}

int sc::_co_map()
{
 int  i,st;
 double *alpha0,*alpha1,sum,*beta0,*beta1,*beta_swap;
 double **i_extp1=co_i_ext,**i_extp2=co_i_ext+2;
 double **o_extp1=co_o_ext,**o_extp2=co_o_ext+2;
 double *dec0=co_dec[0],*dec1=co_dec[1];

 //zero output buffers
 memset(o_extp1[0],0,block_length*sizeof(double));
```

```
memset(o_extp1[1],0,block_length*sizeof(double));
memset(o_extp2[0],0,block_length*sizeof(double));
memset(o_extp2[1],0,block_length*sizeof(double));
memset(dec0,0,block_length*sizeof(double));
memset(dec1,0,block_length*sizeof(double));

//initialize alpha recursion
memset(alpha,0,(block_length+1)*co_nr_states*sizeof(double));
*alpha = 1; alpha1 = alpha;
//alpha recursion
for(i=0; i<block_length; i++) {
  alpha0 = alpha1; alpha1 += co_nr_states;
  for(st=0; st<co_nr_states; st++) {
    alpha1[co_next_st0[st]]+=alpha0[st]*i_extp1[co_next_b0_p1[st]][i]*i_extp2[co_next_b0_p2[st]][i];
    alpha1[co_next_st1[st]]+=alpha0[st]*i_extp1[co_next_b1_p1[st]][i]*i_extp2[co_next_b1_p2[st]][i];
  }
  sum = 0; for(st=0; st<co_nr_states; st++) sum += alpha1[st];
  if(sum==0) {
    sprintf(error_message,"Outer decoder error (alpha)_%d\n",i);
    return -1;
  }
  for(st=0; st<co_nr_states; st++) alpha1[st] /= sum;
}
//beta init
if(beta_start=='z') {memset(beta,0,co_nr_states*sizeof(double)); beta[0] = 1;}
else {
  switch(beta_start) {
  case 'a':
    for(st=0; st<co_nr_states; st++) beta[st] = alpha1[st];
    break;
  default:
    for(st=0; st<co_nr_states; st++) beta[st] = 1.0/co_nr_states;
  }
}
beta0 = beta; beta1 = beta + co_nr_states;
//beta recursion and decoding
for(i=block_length-1; i>=0; i--) {
  double tmp_dec;

  alpha1 -= co_nr_states; sum = 0;
  for(st=0; st<co_nr_states; st++) {
    o_extp1[co_next_b0_p1[st]][i] += alpha1[st]*i_extp2[co_next_b0_p2[st]][i]*beta0[co_next_st0[st]];
    o_extp1[co_next_b1_p1[st]][i] += alpha1[st]*i_extp2[co_next_b1_p2[st]][i]*beta0[co_next_st1[st]];
    o_extp2[co_next_b0_p2[st]][i] += alpha1[st]*i_extp1[co_next_b0_p1[st]][i]*beta0[co_next_st0[st]];
    o_extp2[co_next_b1_p2[st]][i] += alpha1[st]*i_extp1[co_next_b1_p1[st]][i]*beta0[co_next_st1[st]];
    tmp_dec=beta0[co_next_st0[st]]*i_extp1[co_next_b0_p1[st]][i]*i_extp2[co_next_b0_p2[st]][i];
    dec0[i] += alpha1[st]*tmp_dec; beta1[st] = tmp_dec;
    tmp_dec=beta0[co_next_st1[st]]*i_extp1[co_next_b1_p1[st]][i]*i_extp2[co_next_b1_p2[st]][i];
    dec1[i] += alpha1[st]*tmp_dec; beta1[st] += tmp_dec;
    sum += beta1[st];
  }
  if(sum==0||(dec0[i]==0&&dec1[i]==0)) {
    sprintf(error_message,"Outer decoder error (beta or dec)_%d\n",i);
    return -1;
  }
  for(st=0; st<co_nr_states; st++) beta1[st] /= sum;
  beta_swap = beta1; beta1 = beta0; beta0 = beta_swap;
}
return 0;
}

int sc::decode(double *rec,char *dec,int *ber_per_it,double sigma,int qstop,double Mdist,int max_nit)
{
  int i,it,errors;
  //used by qstop
  double *dec_prev=NULL;
  ////////////////////
  //determine probs//
  ////////////////////
  if(sigma <= 0) {
    for(i=0; i<2*block_length; i++) {
      if(*rec++>0) {ci_rec[0][i] = 0; ci_rec[1][i] = 1;}
      else         {ci_rec[0][i] = 1; ci_rec[1][i] = 0;}
```

```
    if(*rec++>0) {ci_rec[2][i] = 0; ci_rec[3][i] = 1;}
    else         {ci_rec[2][i] = 1; ci_rec[3][i] = 0;}
  }
}
else {
  for(i=0; i<2*block_length; i++) {
    ci_rec[0][i] = exp(-((*rec+1)*(*rec+1))/(2*sigma*sigma));
    ci_rec[1][i] = exp(-((*rec-1)*(*rec-1))/(2*sigma*sigma)); ++rec;
    ci_rec[2][i] = exp(-((*rec+1)*(*rec+1))/(2*sigma*sigma));
    ci_rec[3][i] = exp(-((*rec-1)*(*rec-1))/(2*sigma*sigma)); ++rec;
  }
}
//init inner code extrinsic
for(i=0; i<2*block_length; i++) ci_i_ext[0][i] = ci_i_ext[1][i] = 1;
//////////////////////
//The iterative loop//
//////////////////////
for(it=1; it<=max_nit; it++) {
  if(_ci_map()<0) return -2;
  //distribute extrinsic
  for(i=0; i<block_length; i++) {
    co_i_ext[0][i]=ci_o_ext[0][intlp[2*i]]; co_i_ext[1][i]=ci_o_ext[1][intlp[2*i]];
    co_i_ext[2][i]=ci_o_ext[0][intlp[2*i+1]]; co_i_ext[3][i]=ci_o_ext[1][intlp[2*i+1]];
  }
  if(_co_map()<0) return -2;
  //stop conditions
  switch(qstop) {
  case 1:
    errors = 0;
    //check for zero errors
    for(i=0; i<block_length; i++)
      if((co_dec[1][i]>co_dec[0][i]&&dec[i]==0)||(co_dec[1][i]<=co_dec[0][i]&&dec[i]==1))
        ++errors;
    ber_per_it[it-1] += errors;
    break;
  case 2:
    errors = 1;
    //allocate memory on first iteration
    if(it==1) { dec_prev=(double*)malloc(2*block_length*sizeof(double));
      if(dec_prev!=NULL) {
        for(i=0; i<block_length; i++) {
          dec_prev[i] = co_dec[0][i]; dec_prev[block_length+i] = co_dec[1][i];
        }
      }
    }
    else {
      if(dec_prev!=NULL) {
        double dist=0,dd;
        for(i=0; i<block_length; i++) {
          if((dd=co_dec[0][i]-dec_prev[i])<0) dd=-dd; dist += dd;
          dec_prev[i] = co_dec[0][i];
          if((dd=co_dec[1][i]-dec_prev[block_length+i])<0) dd=-dd; dist += dd;
          dec_prev[block_length+i] = co_dec[1][i];
        }
        dist /= (2*block_length);
        if(dist<=Mdist)
        errors = 0;
      }
    }

    //free memory on convergence or maxit
    if((!errors||it==max_nit-1)&&dec_prev!=NULL) {free(dec_prev); dec_prev = NULL;}
    break;
  default: errors = 1;
  }
  if(!errors) break;
  //distribute extrinsic
  for(i=0; i<block_length; i++) {
    ci_i_ext[0][intlp[2*i]]=co_o_ext[0][i]; ci_i_ext[1][intlp[2*i]]=co_o_ext[1][i];
    ci_i_ext[0][intlp[2*i+1]]=co_o_ext[2][i]; ci_i_ext[1][intlp[2*i+1]]=co_o_ext[3][i];
  }
}
for(i=0; i<block_length; i++) {  //decision time
```

```
    if(co_dec[0][i]>co_dec[1][i]) dec[i] = 0;
    else                          dec[i] = 1;
  }
  return it;
}

sc::~sc()
{
  free(intlp); free(ci_next_st0); free(alpha);
}
```

C.3 S interleavers

```
/*
 * File: S.cpp
 * Author: A. Ambroze
 * Purpose: S interleaver routines
 */

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include "S.h"
#include "rnd.h"
#include "intl.h"

//#define VERBOSE

void bS(int *intlp,int isize,int S,int runs);

//Quicker implementation of the S condition: an interleaver mapping 'covers' a region
int occupy_region(int *region_check,int isize,int S,int ipos)
{
  int mpos,Mpos,bit,nleft = 0;
  //edges --> nasty edge effects
  mpos=ipos-S; if(mpos<0) mpos=0; Mpos=ipos+S; if(Mpos>=isize) Mpos=isize-1;
  region_check[ipos]++; --nleft;
  for(bit=mpos; bit<=Mpos; bit++) {if(region_check[bit]==0) --nleft; region_check[bit]+=2;}
  return nleft;
}
int free_region(int *region_check,int isize,int S,int ipos)
{
  int mpos,Mpos,bit;
  int nleft = 0;
  //edges --> nasty edge effects
  mpos=ipos-S; if(mpos<0) mpos=0; Mpos=ipos+S; if(Mpos>=isize) Mpos=isize-1;
  for(bit=mpos; bit<=Mpos; bit++) {region_check[bit]-=2; if(region_check[bit]==0) ++nleft;}
  return nleft;
}

//unique random value -- this is where 'region' simplification occurs
int get_uniq_rnd(int *region_check,int isize)
{
 int rnd;
 do {rnd = (int)(isize*randv01());}while(region_check[rnd]);
 return rnd;
}

//performs the swap pair search
int patch_S(int *intlp,int *region_check,int *region_patch,int isize,int S,
    int pos)
{
  int bit,mbit,Mbit,lbit,nleft = 0;
  for(bit=0; bit<pos; bit++) {
    if(region_check[intlp[bit]]<2) { //not covered
      memset(region_patch,0,isize*sizeof(int));
      mbit=bit-S; if(mbit<0) mbit=0;
```

```
      Mbit=bit+S; if(Mbit>=isize) Mbit=isize-1;
      for(lbit=mbit; lbit<=Mbit; lbit++) {
  if(lbit!=bit&&intlp[bit]>=0) {//cover region
    occupy_region(region_patch,isize,S,intlp[lbit]);
  }
      }
      for(lbit=0; lbit<isize; lbit++)
      if((region_check[lbit]&1)==0 && region_patch[lbit]==0) {//found replacement, patch
        ++nleft;
        if(--region_check[intlp[bit]]!=0) {fprintf(stderr,"bckup_check nonzero\n");}
        if(region_check[lbit]++==0) {fprintf(stderr,"region check error\n");}
        intlp[bit] = lbit;
        break;
      }
    }
    if(nleft>0) break;
  }
  return nleft;
}

int getS(int *intlp,int isize,int S)
{
  int nleft=isize,*region_check,*region_patch;
  int pos,ret = 1;

  if((region_check=(int*)calloc(2*isize,sizeof(int)))==NULL) {
    fprintf(stderr,"getS: calloc error\n");
    return -1;
  }
  region_patch = region_check+isize;
  for(pos=0; pos<isize; pos++) intlp[pos] = -1;
  for(pos=0; pos<isize; pos++) {
    if(nleft<=0) {  //Locked, try to patch
#ifdef VERBOSE
      fprintf(stderr,"Locked at %d, trying to patch ... ",pos);
#endif
      if(!(nleft += patch_S(intlp,region_check,region_patch,isize,S,pos))) {
#ifdef VERBOSE
        fprintf(stderr,"failed\n");
#endif
        ret = 0; break;
      }
#ifdef VERBOSE
  fprintf(stderr,"done\n");
#endif
    }
    intlp[pos] = get_uniq_rnd(region_check,isize);
    nleft += occupy_region(region_check,isize,S,intlp[pos]);
    if(pos-S>=0) nleft += free_region(region_check,isize,S,intlp[pos-S]);
  }
  free(region_check);
  return ret;
}

//paired S condition
int verify_region_2S(int *intl1,int *inv_intl1,int *intl2,int isize,int S,int pos,int ipos)
{
  int mpos=intl1[pos]-S,Mpos=intl1[pos]+S;
  int lval=ipos-S,hval=ipos+S;
  int ipos1,ipos2;

  if(mpos<0) mpos = 0;
  if(Mpos>=isize) Mpos = isize-1;
  for(ipos1=mpos; ipos1<intl1[pos]; ipos1++) {
    if((ipos2=intl2[inv_intl1[ipos1]])!=-1) {
      if(ipos2>=lval&&ipos2<=hval) return 0;
    }
  }
  for(ipos1=intl1[pos]+1; ipos1<=Mpos; ipos1++) {
    if((ipos2=intl2[inv_intl1[ipos1]])!=-1) {
      if(ipos2>=lval&&ipos2<=hval) return 0;
    }
  }
```

```
  return 1;
}
//Search for swap pair for 2S
int patch_2S(int *intl1,int *inv_intl1,int *intl2,int *region_check,
     int *region_patch,int isize,int S,int pos)
{
  int bit,mbit,Mbit,lbit,nleft = 0;
  for(bit=0; bit<isize; bit++) {
    //is bit usable?
    if(region_check[intl2[bit]]==1) { //not covered by intl2
      //check if covered by intl1
      if(verify_region_2S(intl1,inv_intl1,intl2,isize,S,pos,intl2[bit])) {
        memset(region_patch,0,isize*sizeof(int));
        mbit=bit-S; if(mbit<0) mbit=0;
        Mbit=bit+S; if(Mbit>=isize) Mbit=isize-1;
        for(lbit=mbit; lbit<=Mbit; lbit++) {
          if(lbit!=bit&&intl2[bit]>=0)  //cover region
            occupy_region(region_patch,isize,S,intl2[lbit]);
  }
        for(lbit=0; lbit<isize; lbit++)
        if((region_check[lbit]&1)==0 && region_patch[lbit]==0) {
          //intl1 constraint
          if(verify_region_2S(intl1,inv_intl1,intl2,isize,S,bit,lbit)) {
            //found replacement, patch
            ++nleft;
            if(--region_check[intl2[bit]]!=0) {fprintf(stderr,"bckup_check nonzero\n");}
            region_check[lbit]++; intl2[bit] = lbit; break;
          }
        }
      }
      if(nleft>0) break;
    }
  }
  return nleft;
}

//Simple S condition
int is_S(int *intlp,int pos,int ipos,int S,int isize)
{
  int Ss,Se,i;
  if((Ss=pos-S)<0) Ss=0;
  if((Se=pos+S)>=isize) Se=isize-1;
  for(i=Ss; i<=Se; i++) {
    if(intlp[i]>ipos-S && intlp[i]<ipos+S) return 0;
  }
  return 1;
}

//Obtaining an S interleaver from row/column interleaver by random swaps
void bS(int *intlp,int isize,int S,int runs)
{
  int rnd1,rnd2,tmp,r1,r2,swaps = 0;

  for(r1=0; r1<runs; r1++) {
    for(r2=0; r2<isize; r2++) {
      rnd1=(int)(isize*randv01());
      do {rnd2=(int)(isize*randv01());}while(rnd1==rnd2);
      if(is_S(intlp,rnd1,intlp[rnd2],S,isize) && is_S(intlp,rnd2,intlp[rnd1],S,isize)) {//swap
        tmp = intlp[rnd1]; intlp[rnd1] = intlp[rnd2]; intlp[rnd2] = tmp;++swaps;
      }
    }
  }
  fprintf(stderr,"swaps=%d\n",swaps);
}

//paired S interleavers
int get2S(int *intl1,int *inv_intl1,int *intl2,int isize,int S)
{
  int nleft=isize,nleft2,*region_check,*region_check2,*region_patch;
  int pos,ret = 1;

  if((region_check=(int*)calloc(3*isize,sizeof(int)))==NULL) {
    fprintf(stderr,"getS: calloc error\n"); return -1;
```

```
  }
  region_check2 = region_check+isize; region_patch = region_check2+isize;
  for(pos=0; pos<isize; pos++) intl2[pos] = -1;
  for(pos=0; pos<isize; pos++) {
    if(nleft<=0) {  //Locked, try to patch
#ifdef VERBOSE
      fprintf(stderr,"Locked at %d, trying to patch ... ",pos);
#endif
      if(!(nleft+=patch_2S(intl1,inv_intl1,intl2,region_check,region_patch,isize,S,pos))) {
#ifdef VERBOSE
        fprintf(stderr,"failed\n");
#endif
        ret = 0; break;
      }
#ifdef VERBOSE
      fprintf(stderr,"done\n");
#endif
    }
    memcpy(region_check2,region_check,isize*sizeof(int)); nleft2 = nleft;
    while(1) {
      if(nleft2<=0) { //Locked
#ifdef VERBOSE
        fprintf(stderr,"Locked(2) at %d, trying to patch ... ",pos);
#endif
        if(!patch_2S(intl1,inv_intl1,intl2,region_check,region_patch,isize,S,pos)) {
#ifdef VERBOSE
          fprintf(stderr,"failed\n");
#endif
          ret = 0; break;
        }
        memcpy(region_check2,region_check,isize*sizeof(int));
        nleft2=nleft;
#ifdef VERBOSE
        fprintf(stderr,"done\n");
#endif
      }
      intl2[pos] = get_uniq_rnd(region_check2,isize);
      region_check2[intl2[pos]]++;
      --nleft2;
      if(verify_region_2S(intl1,inv_intl1,intl2,isize,S,pos,intl2[pos])) break;
    }
    if(ret==0) break;
    nleft += occupy_region(region_check,isize,S,intl2[pos]);
    if(pos-S>=0) nleft += free_region(region_check,isize,S,intl2[pos-S]);
  }
  free(region_check);
  return ret;
}

//verify S
int vfS(int *intlp,int isize,int S)
{
  int nleft = isize,*region_check,pos,ret = 1;

  if((region_check=(int*)calloc(isize,sizeof(int)))==NULL) {
    fprintf(stderr,"getS: calloc error\n"); return -1;
  }
  for(pos=0; pos<isize; pos++) {
    if(nleft<=0) { //Locked
      fprintf(stderr,"Locked at %d\n",pos);
      ret = 0; break;
    }
    nleft += occupy_region(region_check,isize,S,intlp[pos]);
    if(pos-S>=0) nleft += free_region(region_check,isize,S,intlp[pos-S]);
  }
  free(region_check);
  return ret;
}

//determine S
int detS(int *intlp,int isize,int mod)
{
  int lS,minS=isize,maxS=0,avgS=0,mpos,Mpos,mval,Mval,bit,lbit,do_break;
```

```
  for(bit=0; bit<isize; bit++) {
    do_break = 0;
    for(lS=1; lS<isize; lS++) {
      mval=intlp[bit]-lS; Mval=intlp[bit]+lS; mpos=bit-lS; if(mpos<0) mpos=0;
      Mpos=bit+lS; if(Mpos>=isize) Mpos = isize-1;
      for(lbit=mpos; lbit<=Mpos; lbit++) {
        if(lbit!=bit && (intlp[lbit]>=mval&&intlp[lbit]<=Mval)) {
          do_break = 1; break;
        }
      }
      if(do_break) break;
    }
    --lS;
    if(mod) printf("%d\n",lS); if(minS>lS) minS=lS; if(maxS<lS) maxS=lS; avgS += lS;
  }
 fprintf(stderr,"detS: minS/avgS/maxS = %d/%g/%d\n",minS,(float)avgS/isize,maxS);
 return minS;
}
```

C.4 Computing the $(OW_2)_{min}$ and $(OW_{2+2})_{min}$ probability

```
/*
 * FILE: tiw2.cpp
 * Author: A. Ambroze
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "intl.h"
#include "rnd.h"
#include "S.h"

#define MAXI 4
#define MAXW 1000

int tiw22(int nT,int wT,int **intlp,int isize,int ni)
{
  int ow22=ni*isize,l_ow22;
  int i,j,k,l,in;

  for(i=0; i<isize; i++) {
    for(j=i+nT; j<isize; j+=nT) {
      if((l_ow22=j-i)>=ow22) break;
      for(k=j+1;k<isize;k++) {
        l_ow22=j-i;
        for(l=k+nT;l<isize;l+=nT) {
          if((l_ow22+=l-k)>=ow22) break;
          for(in=0; in<ni; in++) {
            int l_nT1,l_nT2,l_nT3,l_nT4;
            if((l_nT1=intlp[in][i]-intlp[in][k])<0) l_nT1 = -l_nT1;
            if((l_nT2=intlp[in][j]-intlp[in][l])<0) l_nT2 = -l_nT2;
            if((l_nT3=intlp[in][j]-intlp[in][k])<0) l_nT3 = -l_nT3;
            if((l_nT4=intlp[in][i]-intlp[in][l])<0) l_nT4 = -l_nT4;
            if(l_nT1%nT==0&&l_nT2%nT==0) {
              if(l_nT3%nT==0&&l_nT4%nT==0) {
                if(l_nT1+l_nT2>l_nT3+l_nT4) {l_ow22 += l_nT3+l_nT4;}
                else                        {l_ow22 += l_nT1+l_nT2;}}
              else {l_ow22 += l_nT1+l_nT2;}
            }
            else {
             if(l_nT3%nT==0&&l_nT4%nT==0) {l_ow22 += l_nT3+l_nT4;}
             else                         break;
```

```
            }
            if(l_ow22 >= ow22) break;
          }
          if(in==ni && ow22>l_ow22) {ow22 = l_ow22;}
        }
      }
    }
  }
  //sanity check
  if(ow22%nT!=0) fprintf(stderr,"tiw22: error: nT=%d does not divide ow22=%d\n",nT,ow22);

  return (ow22/nT)*wT+(ni+2)*4;
}

int* tiw2(int nT,int wT,int **intlp,int isize,int ni,int *nr_ee)
{
  int ow2=(ni+1)*isize,l_ow2;
  int nT0,l_nT[MAXI];
  static int ret[MAXI+2];
  int i,j,in,inT;

  for(i=0; i<isize; i++) {
    int s_ow2;
    for(j=i+nT,s_ow2=wT,nT0=1; j<isize&&s_ow2<ow2; j+=nT,s_ow2+=wT,nT0++) {
      l_ow2 = s_ow2;
      for(in=0; in<ni; in++) {
        if((inT=intlp[in][i]-intlp[in][j])<0) inT = -inT;
        if(inT%nT==0) {l_nT[in] = inT/nT; l_ow2 += l_nT[in]*wT; if(l_ow2 > ow2) break;}
        else          break;
    }
    if(in==ni) {
      if(ow2>l_ow2) {
        ret[1] = nT0; memcpy(ret+2,l_nT,ni*sizeof(int));
        ow2 = l_ow2;
        //reset error event count
        *nr_ee = 1;
      }
      else if(ow2==l_ow2)(*nr_ee)++;
    }
  }

  ret[0] = ow2+2*(ni+2);
  return ret;
}

int main(int argc,char *argv[])
{
  int *intlp[MAXI];
  int nT=3,wT=2,isize=500,ni=0;
  char *s,*hs,idx[10],err[1000];
  int k,nseed,nit,l2=1,l22=0;
  int nr_ee,dfe,*ret;;

  if(argc<2) {
    fprintf(stderr,"usage %s r=<seed,iterations>,l2=<0/1>,l22=<0/1>,nT=<period>,
                    wT=<pw2>,N=<isize>,i1=,i2=,...\n",argv[0]);
    fprintf(stderr,"when iteration is used (r=... is present),i1,i2,.. are specified:\n"\
                    "i1=<S1>,i2=<S2><Spair(1)/doubleS(0)>,i3=<S3>,...\n"\
                    "l2=0/1 specifies whether dmin2 should be calculated\n"\
                    "l22=0/1 specifies whether dmin22 should be calculated\n");
    return 2;
  }

  //parse options
  hs = argv[1];
  if((s=strstr(hs,"l2="))!=NULL) l2=atoi(s+3); //compute OW2 probs
  if((s=strstr(hs,"l22="))!=NULL) l22=atoi(s+4); //compute OW2+2 probs
  if((s=strstr(hs,"nT="))!=NULL) nT=atoi(s+3); //code period
  if((s=strstr(hs,"wT="))!=NULL) wT=atoi(s+3); //parity weight for one period
  if((s=strstr(hs,"N="))!=NULL) isize=atoi(s+2); //block length
  nit = 1;
  if((s=strstr(hs,"r="))!=NULL) sscanf(s+2,"%d,%d",&nseed,&nit);
```

```
  dfe=2*wT+6;
  fprintf(stderr,"#r=%d,%d,nT=%d,wT=%d,N=%d,l2=%d,l22=%d,dfe=%d\n"
                ,nseed,nit,nT,wT,isize,l2,l22,dfe);

  //iterate
  int it,S[MAXI],s2;
  int *inv_intlp;
  int ow2n[MAXW][2],ow2m[2];
  int ow22n[MAXW][2],ow22m[2];
  int ow222m[2];
  int ow22;
  //number of error events causing ow2min
  int *pnr_ee;

  fprintf(stderr,"#S=");
  for(ni=0; ni<MAXI;) {
    sprintf(idx,"i%d=",ni+1);
    if((s=strstr(hs,idx))==NULL) break;
    else {
      //special treatment for second interleaver
      if(ni==1) {sscanf(s+strlen(idx),"%d,%d",&S[ni],&s2); fprintf(stderr,"%d,%d ",S[ni],s2);}
      else      {sscanf(s+strlen(idx),"%d",&S[ni]); fprintf(stderr,"%d ",S[ni]);}
      ++ni;
    }
  }
  fprintf(stderr,"\n#ni=%d\n",ni);
  //allocate interleavers
  if((intlp[0]=(int*)malloc((ni+2)*isize*sizeof(int)))==NULL) {
    fprintf(stderr,"%s: malloc error\n",argv[0]); return 1;
  }
  for(k=1; k<ni; k++) intlp[k] = intlp[k-1]+isize;
  inv_intlp = intlp[ni-1]+isize;
  pnr_ee = inv_intlp+isize;
  memset(pnr_ee,0,isize*sizeof(int));

  if(l2) {
    if(l22) {memset(ow22n,0,sizeof(ow22n)); ow22m[0]=ow222m[0]=0;}
    memset(ow2n,0,sizeof(ow2n)); ow2m[0] = 0;
  }
  else if(l22) {memset(ow22n,0,sizeof(ow22n)); ow22m[0] = 0;}

  srandv01(nseed++,2); //seed random generator
  for(it=0; it<nit; it++) { //for each interleaver
    for(k=0; k<ni; k++) {
      if(k==1&&s2) {
        int bit;

        for(bit=0; bit<isize; bit++) inv_intlp[intlp[0][bit]]=bit;
        while(!get2S(intlp[0],inv_intlp,intlp[k],isize,S[k]));
      }
      else while(!getS(intlp[k],isize,S[k]));
    }
    if(l2) {    //OW2
      ret = tiw2(nT,wT,intlp,isize,ni,&nr_ee);
      if(ret[0]==dfe) pnr_ee[nr_ee]++;
else            pnr_ee[0]++;
      if(ret[0]<MAXW) {++ow2n[ret[0]][0]; ow2n[ret[0]][1] = nseed-ni;}
      if(ret[0]>ow2m[0]) {ow2m[0] = ret[0]; ow2m[1] = nseed-ni;}
    }
    if(l22) {   //OW2+2
      ow22 = tiw22(nT,wT,intlp,isize,ni);
      if(ow22<MAXW) {++ow22n[ow22][0]; ow22n[ow22][1] = nseed-ni;}
      if(ow22>ow22m[0]) {ow22m[0] = ow22; ow22m[1] = nseed-ni;}
      if(l2) {   //OW2+2 and OW2
        if(ret[0]<ow22) {
          if(ret[0]>ow222m[0]) {ow222m[0] = ret[0]; ow222m[1] = nseed-ni;}
        }
        else if(ow22>ow222m[0]) {ow222m[0] = ow22; ow222m[1] = nseed-ni;}
    }
  }
  //report results
  if(l2) {
    fprintf(stderr,"\n#ow2m=%d,nseed=%d\n\n",ow2m[0],ow2m[1]);
```

```
    for(k=0; k<MAXW; k++)
      if(ow2n[k][0]) fprintf(stderr,"%d %d %d\n",k,ow2n[k][0],ow2n[k][1]);
    fprintf(stderr,"#Number of ee causing dfree_effective:\n");
    for(k=0; k<isize; k++)
      if(pnr_ee[k]) fprintf(stderr,"%d %d\n",k,pnr_ee[k]);
  }
  if(122) {
    fprintf(stderr,"\n#ow22m=%d,nseed=%d\n\n",ow22m[0],ow22m[1]);
    for(k=0; k<MAXW; k++)
      if(ow22n[k][0]) fprintf(stderr,"%d %d %d\n",k,ow22n[k][0],ow22n[k][1]);
    if(12) fprintf(stderr,"\n#ow222m=%d,nseed=%d\n\n",ow222m[0],ow222m[1]);
  }

  return 0;
}
```

Bibliography

Acikel, O. and Ryan, W. (1999). Punctured turbo codes for bpsk/qpsk channels. *IEEE Transactions on Communications*, 47(9):1315–1323.

Aitsab, O. and Pyndiah, R. (1996). Performance of Reed-Solomon block turbo code. In *Proc. IEEE GLOBECOM*, pages 121–125.

Ambroze, A., Wade, G., and Serdean, C. (2000a). Turbo code protection of a video watermark channel. Submitted to Proc IEE Vision, Image and Signal Processing.

Ambroze, A., Wade, G., and Tomlinson, M. (1998a). Iterative MAP decoding for serial concatenated convolutional codes. *IEE Proceedings Communications*, 145(2):53–59.

Ambroze, A., Wade, G., and Tomlinson, M. (1998b). Turbo code tree and code performance. *Electronics Letters*, 34(4):353–354.

Ambroze, A., Wade, G., and Tomlinson, M. (2000b). Dependence of d_{free} in mpccc systems. To be submitted to IEE Proceedings Communications.

Ambroze, A., Wade, G., and Tomlinson, M. (2000c). Practical aspects of iterative decoding. *IEE Proceedings Communications*, 147(2):69–74.

Andersen, J. (1996). Turbo codes extended with outer BCH code. *Electronics Letters*, 32:2059–2060.

Andersen, J. (1999). Selection of component codes for turbo codes based on convergence properties. *Annales des telecommunications*, 54(3-4).

Andersen, J. and Zyablov, V. (1997). Interleaver design for turbo coding. In *International Symposium on Turbo Codes*, pages 154–157.

Bahl, L., Cocke, J., Jelinek, F., and Raviv, J. (1974). Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Transactions on Information Theory*, 20:284–287.

Barbulescu, A. (1998). Dynamical system perspective on turbo codes. *Electronics Letters*, 34(8):754–755.

Barbulescu, A., Farrell, W., Gray, P., and Rice, M. (1997). Bandwidth efficient turbo coding for high speed mobile satellite communications. In *International Symposium on Turbo Codes*, pages 119–127.

Barbulescu, A. and Pietrobon, S. (1994). Interleaver design for turbo codes. *Electronics Letters*, 30:2107–2108.

Barbulescu, A. and Pietrobon, S. (1995). Terminating the trellis of turbo codes in the same state. *Electronics Letters*, 31:22–23.

Battail, G. (1997). A conceptual framework for understanding turbo codes. In *International Symposium on Turbo Codes*, pages 55–63.

Benedetto, S., Divsalar, D., Montorsi, G., and Pollara, F. (1995). Bandwidth efficient parallel concatenated coding schemes. *Electronics Letters*, 31(24):2067–2069.

Benedetto, S., Divsalar, D., Montorsi, G., and Pollara, F. (1996). Algorithm for continuous decoding of turbo codes. *Electronics Letters*, 32(4):314–315.

Benedetto, S., Divsalar, D., Montorsi, G., and Pollara, F. (1997a). Serial concatenated trellis coded modulation with iterative decoding: design and performance. In *Proc. IEEE Globecom*, pages 38–43, Phoenix, Arizona, USA.

Benedetto, S., Divsalar, D., Montorsi, G., and Pollara, F. (1997b). A soft-input soft-output APP module for iterative decoding of concatenated codes. *IEEE Communications Letters*, 1(1):22–24.

Benedetto, S., Divsalar, D., Montorsi, G., and Pollara, F. (1998a). Analysis, design and iterative decoding of double serially concatenated codes with interleavers. *IEEE Journal on Selected Areas in Communications*, 16(2):231–244.

Benedetto, S., Garello, R., and Montorsi, G. (1997c). The trellis complexity of turbo codes. In *Proc. IEEE GLOBECOM Communications Miniconference*, pages 60–65.

Benedetto, S., Garello, R., and Montorsi, G. (1998b). A search for good convolutional codes to be used in the construction of turbo codes. *IEEE Transactions on Communications*, 46(9):1101–1105.

Benedetto, S. and Montorsi, G. (1995a). Average performance of parallel concatenated block codes. *Electronics Letters*, 31(3):156–158.

Benedetto, S. and Montorsi, G. (1995b). Performance evaluation of turbo-codes. *Electronics Letters*, 31(3):163–165.

Benedetto, S. and Montorsi, G. (1995c). Role of recursive convolutional codes in turbo codes. *Electronics Letters*, 31(11):858–859.

Benedetto, S. and Montorsi, G. (1996a). Serial concatenation of block and convolutional codes. *Electronics Letters*, 32(10):887–888.

Benedetto, S. and Montorsi, G. (1996b). Serial concatenation of interleaved codes: analytical performance bounds. In *Proc. GLOBECOM*, pages 106–110.

Benedetto, S. and Montorsi, G. (1996c). Unveiling turbo codes: some results on parallel concatenated coding schemes. *IEEE Transactions on Information Theory*, 42(2):409–429.

Benedetto, S. and Montorsi, G. (1997). Performance of continuous and blockwise decoded turbo codes. *IEEE Communications Letters*, 1(3):77–79.

Berrou, C. (1997). Some clinical aspects of turbo codes. In *International Symposium on Turbo Codes*, pages 26–32.

Berrou, C., Adde, P., Ettiboua, A., and Faudeil, S. (1993a). A low complexity soft-output viterbi architecture. In *Proc. IEEE International Conference on Communications*, Geneva, Switzerland.

Berrou, C. and Jezequel, M. (1996). Frame oriented convolutional turbo codes. *Electronics Letters*, 32(15).

Berrou, C. and Jezequel, M. (1999). Non binary convolutional codes for turbo coding. *Electronics Letters*, 35(1):39–40.

Berrou, C., Thitimajshima, P., and Glavieux, A. (1993b). Near Shannon limit error correcting coding and decoding: turbo codes. In *Proc. IEEE International Conference on Communications*, pages 1064–1070, Geneva, Switzerland.

Blackert, W., Hall, E., and Wilson, S. (1995). Turbo code termination and interleaver conditions. *Electronics Letters*, 31:2082–2083.

Breiling, M. and Hanzo, L. (1997a). Non-iterative optimum super-trellis decoding of turbo codes. *Electronics Letters*, 33(10):848–849.

Breiling, M. and Hanzo, L. (1997b). Optimum non-iterative turbo-decoding. In *Proc. of PIMRC*, pages 714–718, Helsinki, Finland.

Burkert, F. and Hagenauer, J. (1997). A serial concatenated coding scheme with iterative turbo and feedback decoding. In *International Symposium on Turbo Codes*, pages 227–231.

Cedervall, M. and Johannesson, R. (1989). A fast algorithm for computing distance spectrum of convolutional codes. *IEEE Transactions on Information Theory*, 35(6):1146–1159.

Daneshgaran, F. and Mondin, M. (1997a). Design of interleavers for turbo codes based on a cost function. In *International Symposium on Turbo Codes*, pages 255–259.

Daneshgaran, F. and Mondin, M. (1997b). An efficient algorithm for obtaining the distance spectrum of the turbo codes. In *International Symposium on Turbo Codes*, pages 251–255.

Divsalar, D. (1999). A simple tight bound on error probability of block codes with application to turbo codes. *JPL TDA Progress Report*, 42-139:1–35.

Divsalar, D., Dolinar, S., Pollara, F., and McEliece, R. (1995). Transfer function bounds on the performance of turbo codes. *JPL TDA Progress Report*, 42-122:44–55.

Divsalar, D. and Pollara, F. (1995a). Multiple turbo codes for deep-space communications. *JPL TDA Progress Report*, 42-121:66–77.

Divsalar, D. and Pollara, F. (1995b). On the design of turbo codes. *JPL TDA Progress Report*, 42-123:99–121.

Divsalar, D. and Pollara, F. (1995c). Turbo codes for deep-space communications. *JPL TDA Progress Report*, 42-120:29–39.

Divsalar, D. and Pollara, F. (1995d). Weight distributions for turbo codes using random and nonrandom permutations. *JPL TDA Progress Report*, 42-122:56–65.

Dolinar, S., Divsalar, D., and Pollara, F. (1998). Code performance as a function of block size. *JPL TDA Progress Report*, 42-133:1–23.

Duman, T. and Masoud, S. (1998). New performance bounds for turbo codes. *IEEE Transactions on Communications*, 46(6):717–723.

Dunscombe, E. and Piper, F. (1989). Optimal interleaving scheme for convolutional coding. *Electronics Letters*, 25(22):1517–1518.

Fei, X. and Ko, T. (1997). Turbo codes used for compressed image transmission over frequency selective fading channel. In *IEEE Globecom.*

Fonseka, J. (1999). Application of turbo codes in satellite mobile systems. *Electronics Letters*, 35(2):114–115.

Forney, G. (1966). *Concatenated codes.* Cambridge, MA: M.I.T. Press.

Fossorier, M., Burkert, F., Lin, S., and Hagenauer, J. (1998). On the equivalence between SOVA and max-log-MAP decodings. *IEEE Communications Letters*, 2(5):137–139.

Franz, V. and Anderson, J. (1998). Concatenated decoding with a reduced-search BCJR algorithm. *IEEE Journal on Selected Areas in Communications*, 16(2):186–195.

Frey, B. and Kschischang, F. (1998). Early detection and trellis splicing: reduced-complexity iterative decoding. *IEEE Journal on Selected Areas in Communications*, 16(2):153–159.

Frey, B. and MacKay, D. (1997). Trellis constrained codes. In *Proc. of the 35 Allerton Conference on Communications, Control and Computing*, Urbana, Illinois.

Frias, J. and Villasenor, J. (1997a). Combining hidden Markov source models and parallel concatenated codes. *IEEE Communications Letters*, 1(4):111–113.

Frias, J. and Villasenor, J. (1997b). Joint source channel decoding of turbo codes. In *International Symposium on Turbo Codes*, pages 259–263.

Gallager, R. (1963). *Low-density parity check codes.* Cambridge, MA: M.I.T. Press.

Gallager, R. (1965). A simple derivation of the coding theorem and some applications. *IEEE Transactions on Information Theory*, pages 3–18.

Goalic, A. and Pyndiah, R. (1997). Real time turbo decoding of product codes on a digital signal processor. In *International Symposium on Turbo Codes*, pages 267–271.

Hagenauer, J. (1995). Source controlled channel coding. *IEEE Transactions on Communications*, 43:2449–2457.

Hagenauer, J. and Hoeher, P. (1989). A viterbi algorithm with soft-decision outputs and its applications. In *Proc. GLOBECOM*, Dallas, Texas.

Hagenauer, J., Offer, E., and Papke, L. (1996). Iterative decoding of binary block and convolutional codes. *IEEE Transactions on Information Theory*, 42(2):429–445.

Hall, E. and Wilson, S. (1997). Turbo codes for noncoherent channels. In *Proc. IEEE GLOBECOM Communications Miniconference*, pages 66–69.

Hall, E. and Wilson, S. (1998a). Convolutional interleavers for stream-oriented parallel concatenated convolutional codes. In *Proc. ISIT.*

Hall, E. and Wilson, S. (1998b). Design and analysis of turbo codes on Rayleigh fading channels. *IEEE Journal on Selected Areas in Communications*, 16(2):160–174.

Hokfelt, J., Edfords, O., and Maseng, T. (1999a). A survey on trellis termination alternatives for turbo codes. In *VTC*, Houston, Texas.

Hokfelt, J., Edfords, O., and Maseng, T. (1999b). Turbo codes: interaction between trellis termination method and interleaver design. In *Radio Science and Communication Conference*, Karlskrona, Sweden.

Hokfelt, J., Edfors, O., and Maseng, T. (1998). Assessing interleaver suitability for turbo codes. In *Northern Radio Symposium*, Saltsjobaden, Sweden.

Hokfelt, J., Edfors, O., and Maseng, T. (1999c). Interleaver design for turbo codes based on the performance of iterative decoding. In *ICC*, Vancouver, Canada.

Hokfelt, J., Edfors, O., and Maseng, T. (1999d). Interleaver structures for turbo codes with reduced storage memory requirement. In *VTC*, pages 212–216, Amsterdam, Holland.

Hokfelt, J., Edfors, O., and Maseng, T. (1999e). Turbo codes: correlated extrinsic information and its impact on iterative decoding performance. In *ITC*, Houston, Texas.

Hokfelt, J. and Maseng, T. (1997). Methodical interleaver design for turbo codes. In *International Symposium on Turbo Codes*, pages 212–216.

Hokfelt, J. and Maseng, T. (1998). On the convergence rate of iterative decoding. In *IEEE GLOBECOM*, Sydney, Australia.

Joerssen, O. and Meyr, H. (1994). Terminating the trellis of turbo codes. *Electronics Letters*, 30(6):1285–1286.

Khandany, A. (1998). Group structure of turbo codes. *Electronics Letters*, 34(2):168–169.

Kiely, A., Dolinar, S., McEliece, R., Ekroot, L., and Lin, W. (1995a). Minimal trellises for linear block codes and their duals. *JPL TDA Progress Report*, 42-121:148–158.

Kiely, A., Dolinar, S., McEliece, R., Ekroot, L., and Lin, W. (1995b). Trellis complexity bounds for decoding linear block codes. *JPL TDA Progress Report*, 42-121:159–172.

Kiely, A., Dolinar, S., McEliece, R., Ekroot, L., and Lin, W. (1996). Trellis decoding complexity of linear block codes. *IEEE Transactions on Information Theory*, 42(6):1687–1697.

Koora, K. and Betzinger, H. (1998). Interleaver design for turbo codes with selected inputs. *Electronics Letters*, 34(7):651–652.

Koora, K. and Finger, A. (1997). A new scheme to terminate all trellis of turbo decoder for variable block length. In *International Symposium on Turbo Codes*, pages 174–180.

Kschischang, F. and Frey, B. (1998). Iterative decoding of compound codes by probability propagation in graphical models. *IEEE Journal on Selected Areas in Communications*, 16(2):219–230.

Lafourcade, A. and Vardy, A. (1995). Asymptotically good codes have infinite trellis complexity. *IEEE Transactions on Information Theory*, 41(2):555–560.

Lazic, D., Beth, T., and Calic, M. (1997). How close are turbo codes to optimal codes? In *International Symposium on Turbo Codes*, pages 192–196.

Lee, B., Bae, S., Kang, S., and Joo, E. (1999). Design of swap interleaver for turbo codes. *Electronics Letters*, 35(22):1939–1940.

Lin, X., Massey, J., Mittelholzer, T., and Rimoldi, B. (1997). Hard decision aided turbo decoding. In *International Symposium on Turbo Codes*, pages 235–239.

MacKay, D. and Neal, R. (1997). Near Shannon limit performance of low density parity check codes. *Electronics Letters*, 33(6):457–458.

Manoukian, H. and Honary, B. (1997). BCJR trellis construction for binary linear codes. *IEE Proceedings Communications*, 144(6):367–371.

McEliece, R. (1977). *The Theory of Information and Coding - A mathematical framework for Communication.* Addison-Wesley publishing company.

McEliece, R., MacKay, D., and Cheng, J.-F. (1998). Turbo decoding as an instance of pearl's belief propagation algorithm. *IEEE Journal on Selected Areas in Communications*, 16(2):140–152.

McEliece, R., Rodemich, E., and Cheng, J.-F. (1995). The turbo decision algorithm. In *33rd Allerton Conference on Communication, Control and Computing.*

Meshkat, P. and Villasenor, J. (1998). New schedules for information processing in turbo codes. In *Proc. ISIT*, Cambridge, MA, USA.

Michelson, A. and Levesque, A. (1984). *Error-control techniques for digital communications.* A Wiley-Interscience publication.

Moher, M. (1998a). Cross entropy and iterative decoding. *IEEE Transactions on Information Theory*, 44(7):3097–3104.

Moher, M. (1998b). An iterative multiuser decoder for near capacity communications. *IEEE Transactions on Communications*, 46(7):870–880.

Narayanan, K. and Stuber, G. (1997). Selective serial concatenation of turbo codes. *IEEE Communications Letters*, pages 136–140.

Narayanan, K. and Stuber, G. (1998a). List decoding of turbo codes. *IEEE Transactions on Communications*, 46(6):754–762.

Narayanan, K. and Stuber, G. (1998b). A serial approach to iterative demodulation and decoding. In *IEEE GLOBECOM*, Sydney, Australia.

Oberg, M. and Siegel, P. (1997). Lowering the error floor for turbo-codes. In *International Symposium on Turbo Codes*, pages 204–208.

Oberg, M., Vityaev, A., and Siegel, P. (1997). The effect of puncturing in turbo encoders. In *International Symposium on Turbo Codes*, pages 184–188.

Ogiwara, H. and Morillo, F. (1997). Applications of turbo codes to TCM. In *International Symposium on Turbo Codes*, pages 200–204.

Perez, L., Seghers, J., and Costello, D. (1996). A distance spectrum interpretation of turbo codes. *IEEE Transactions on Information Theory*, 42(6):1698–1709.

Podemski, R., Holubowicz, W., Berrou, C., and Battail, G. (1995). Hamming distance spectra of turbo-codes. *Annals of Telecommunications*, 50(9-10):790–797.

Press, W. and Teukolski, S. (1993). *Numerical recipes in C.* Cambridge University Press.

Pyndiah, R. (1997). Iterative decoding of product codes: block turbo codes. In *International Symposium on Turbo Codes*, pages 71–80.

Pyndiah, R., Combelles, P., and Adde, P. (1996). A very low complexity block turbo decoder for product codes. In *Proc. IEEE Globecom*, pages 101–105.

Pyndiah, R., Glavieux, A., Picart, A., and Jacq, S. (1994). Near optimum decoding of product codes. In *Proc. IEEE Globecom*, pages 339–343, San Francisco, USA.

Raphaeli, D. and Zarai, Y. (1997). Combined turbo equalization and turbo decoding. In *Proc. IEEE Globecom*, pages 639–643, Phoenix, Arizona, USA.

Reed, M. and Asenstorfer, J. (1997). A novel variance estimator for turbo code decoding. In *Proc. International Conference on Telecommunications*, pages 173–178, Melbourne, Australia.

Reed, M. and Pietrobon, S. (1996). Turbo code termination schemes and a novel alternative for short frames. In *Proc. PIMRC*, pages 354–358, Taipei,Taiwan.

Robertson, P. (1994). Illuminating the structure of code and decoder of parallel concatenated recursive systematic (turbo) codes. In *Proc. IEEE GLOBECOM*, pages 1298–1303.

Robertson, P., Villebrun, E., and Hoeher, P. (1997). Optimal and sub-optimal maximum a posteriori algorithms suitable for turbo decoding. *European Transactions on Telecommunications*, 8.

Robertson, P. and Worz, T. (1995). Coded modulation scheme employing turbo codes. *Electronics Letters*, 31(18):1546–1547.

Sadowsky, J. (1997). A maximum likelihood decoding algorithm for turbo codes. In *Proceedings IEEE GLOBECOM*, Phoenix, Arizona.

Sawyer, W. (1978). *Numerical functional analysis.* Oxford University Press.

Seghers, J. (1995). On the free distance of the TURBO codes and related product codes. Final report, Diploma Project SS 1995, no. 6613, Swiss Federal Institute of Technology, Zurich, Switzerland.

Shannon, C. and Weaver, W. (1949). *The mathematical theory of communication.* University of Illinois Press.

Shibutani, A., Suda, H., and Adachi, F. (1999). Reducing the average number of turbo decoding iterations. *Electronics Letters*, 35(9):701–702.

Summers, T. and Wilson, S. (1998). SNR mismatch and online estimation in turbo decoding. *IEEE Transactions on Communications*, 46(4):421–424.

Svirid, Y. (1995). Weight distributions and bounds for turbo codes. *European Transactions on Telecommunications*, 6(5):543–555.

Takeshita, O., Collins, O., Massey, P., and Costello, D. (1998a). Asymmetric turbo codes. In *ISIT*, Cambridge, MA, USA.

Takeshita, O., Collins, O., Massey, P., and Costello, D. (1998b). On the frame error rate of turbo codes. In *ITW*, Killarney, Ireland.

Viterbi, A. (1998). An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes. *IEEE Journal on Selected Areas in Communications*, 16(2):260–264.

Viterbi, A. and Viterbi, A. (1998). An improved union bound for binary input linear codes on the awgn channel, with applications to turbo decoding. In *Proc. IEEE Information Theory Workshop*, San Diego, California.

Wesel, R. and Cioffi, J. (1997). Joint interleaver and trellis code design. In *IEEE Globecom.*

Wiberg, N. (1997). On the performance of the iterative turbo decoding algorithm. In *International Symposium on Turbo Codes*, pages 223–227.

Wolf, J. (1978). Efficient maximum likelihood decoding of linear block codes using a trellis. *IEEE Transactions on Information Theory*, 24(1):76–80.

Yi, B. (1997). On the synchronization issues of the turbo coded telemetry system. In *International Symposium on Turbo Codes*, pages 275–280.

Appendix D

Publications

Iterative MAP decoding for serial concatenated convolutional codes

1876

A. Ambroze
G. Wade
M. Tomlinson

Indexing terms: Concatenated convolutional codes, Decoders

Abstract: The paper provides detailed computational steps for implementing an iterative concatenated convolutional code (SCCC) decoder. These are based on maximum *a posteriori* probability (MAP) decoding of a single, rate 1/2, recursive, systematic convolutional code, which is reduced to easily implemented equations for forward and backward recursion. In particular, the crucial information exchange between MAP decoders is clarified. Simulation of a rate 1/3 SCCC with memory-2 codes and a coding delay of $N = 1000$ shows a bit error rate of 10^{-6} for $E_b/N_0 = 1.5$dB, and gives a typical interleaver gain of N^{-3}.

1 Introduction

It has been shown that iterative decoding of parallel concatenated convolutional codes (PCCCs or Turbo codes) approaches the theoretical bound for decoded bit error rate (BER) [1–7]. Upper bounds on the BER for PCCCs are presented elsewhere [8], as are bounds for serial concatenated convolutional codes (SCCCs) [9]. A major conclusion from these previous studies is that, for a basic PCCC system employing recursive codes, the BER decreases approximately as N^{-1}, where N is the interleaver length; whereas for an SCCC system, it can typically decrease as N^{-3}. SCCCs are therefore sometimes superior to PCCCs [9–11].

Apart from theoretical work on the upper bound for BER, most published material reports on simulation studies and the general decoding concepts of PCCCs. The decoding principle, but no detail, of an SCCC scheme has been described previously [10]. The objective of this paper is therefore to clarify decoder implementation for an SCCC scheme. The approach is based on simplified forward and backward recursions of the usual maximum *a posteriori* probability (MAP) decoder [2, 5, 12–15]. In particular, we clarify the exchange of information between the two MAP decoders in the SCCC.

IEE Proceedings online no. 19981876

Paper first received 14th April and in revised form 8th December 1997

The authors are with the Satellite Centre, School of Electronic, Communication & Electronic Engineering, University of Plymouth, Plymouth PL4 8AA, Devon, UK

The basic SCCC scheme, based on two recursive, systematic convolutional codes, is shown in Fig. 1. This is a simple unpunctured, rate 1/3 system, and the systematic property enables so-called 'extrinsic' information to be easily extracted. Key features of Fig. 1 are

(i) The input sequence to encoder 2 is a parity sequence as distinct from an information sequence in a PCCC scheme or Turbo code. This means that, with a small modification to Fig. 1, the lowest weight of sequence $v2_i$ can be increased compared with that for a PCCC system, giving increased interleaver gain [10]

(ii) The outer MAP decoder (MAP 1) is effectively fed directly with parity symbol $v2_i$ via channel symbol $r2i$, plus additional information about $v2_i$ derived via $r3_i$ and the structure of MAP 2. This information is in the form of probabilities $P(v2_i = 0)$ (denoted $Pv20_i$). Note that, in order to avoid duplicating information to MAP 1, or feeding back information originally derived from MAP 1, the output of MAP 2 should ideally contain only information derived from $r3_i$, and so this is denoted $Pv20_i(r3)$.

(iii) MAP 1 generates $P(u_i = 0)$ together with an estimate of symbol $v2_i$ in the form of $Pv20_i$. After interleaving, this becomes an estimate $P(w2_i = 0)$, denoted $Pw20_i$, for symbol $w2_i$. The significant point here is that $Pw20_i$ has effectively been derived via $r1_i$ and the structure of the outer code, and is ideally independent of information conveyed via $r2_i$ and $r3_i$ due to the presence of interleaver (or scrambler) I. It therefore acts as additional or 'extrinsic' information for MAP 2, and provides the iterative mechanism. The concept of extrinsic information derived from the use of interleaving has been described elsewhere [2]

2 MAP decoding in an iterative SCCC scheme

Classical MAP decoder theory is outlined in the Appendix. Here we interpret the theory via easily implemented equations, and modify it for the SCCC scheme. For simplicity, both constituent codes (CCS) in Fig. 1 are assumed to have the memory-2 generator in Fig. 2.

Consider first a single MAP decoder for Fig. 2. The forward recursion in eqn. 34 (see Appendix) computes state probability α_i from previous state probabilities α_{i-1}, and γ_k behaves as a transition probability. It can be shown that the α_i can be readily deduced from the trellis in Fig. 2, and it is apparent that each α_i is a sum of just two terms. According to the trellis, the

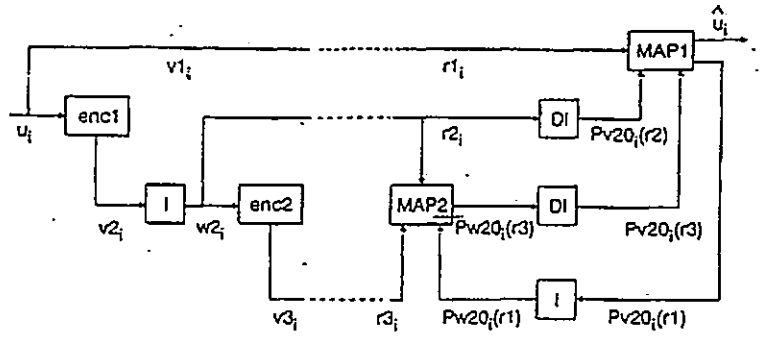


Fig. 1 *Basic R = 1/3 SCCC system*

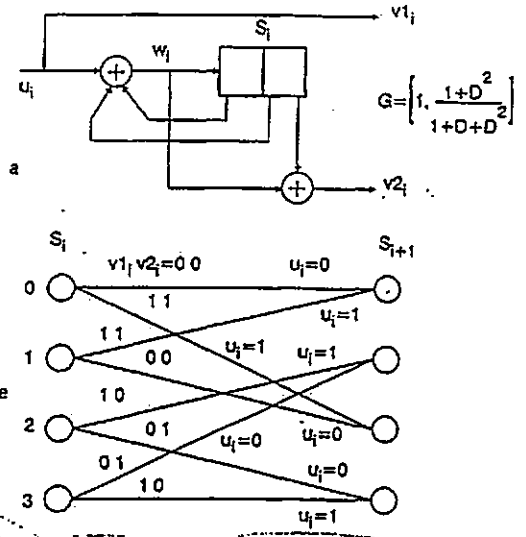


Fig. 2 *(a) memory-2 encoder (b) state transition diagram*

simplified recursions are

$$P(S_{i+1}=0)=p(0,0)P(S_i=0)+p(1,1)P(S_i=1) \quad (1)$$

$$P(S_{i+1}=1)=p(1,0)P(S_i=2)+p(0,1)P(S_i=3) \quad (2)$$

$$P(S_{i+1}=2)=p(1,1)P(S_i=0)+p(0,0)P(S_i=1) \quad (3)$$

$$P(S_{i+1}=3)=p(0,1)P(S_i=2)+p(1,0)P(S_i=3) \quad (4)$$

where the (unnormalised) transition probabilities are given by

$$p(v1,v2)=\exp\left[-\frac{(r1_i-(2v1-1))^2+(r2_i-(2v2-1))^2}{2\sigma^2}\right] \quad (5)$$

Eqn. 5 follows from the assumption that the output of the DMC has a large alphabet, so that the conditional probabilites (as in eqn. 35) tend to Gaussian density functions. Each of the four state probabilities for time $i+1$ must then be normalised by dividing by their sum. Similarly, for the backward probabilities, eqn. 36 is simplified to

$$Pb(S_i=0)=p(0,0)Pb(S_{i+1}=0)+p(1,1)Pb(S_{i+1}=2) \quad (6)$$

$$Pb(S_i=1)=p(1,1)Pb(S_{i+1}=0+p(0,0)Pb(S_{i+1}=2) \quad (7)$$

$$Pb(S_i=2)=p(1,0)Pb(S_{i+1}=1)+p(0,1)Pb(S_{i+1}=3) \quad (8)$$

$$Pb(S_i=3)=p(0,1)Pb(S_{i+1}=1)+p(1,0)Pb(S_{i+1}=3) \quad (9)$$

This again must be normalised. The backward recursion can be initialised by assigning $Pb(S_N=m)=P(S_N=m)$, where m denotes a particular state.

2.1 Modifications for the SCCC scheme

For the basic rate 1/3 scheme in Fig. 1, an additional probability must be incorporated into eqn. 41 to account for the extra information about $v2_i$ generated by MAP2. To deduce this term, we consider a simple change of variables in eqn. 40 in order to obtain the log-likelihood ratio for the inner code:

$$\Lambda(w2_i)=\log\left[\frac{P(r2_i|w2_i=1)}{P(r2_i|w2_i=0)}\frac{\sum_m\sum_n\sum_{l=0}^{1}\gamma_1(r3_i,n,m)\alpha_{i-1}^l(n)\beta_i(m)}{\sum_m\sum_n\sum_{l=0}^{1}\gamma_0(r3_i,n,m)\alpha_{i-1}^l(n)\beta_i(m)}\right] \quad (10)$$

As previously discussed, MAP2 must deliver a probability based on $r3_i$ only, and so, from eqn. 10

$$P(w2_i=0)=Pw20_i(r3)=\sum_m\sum_n\sum_{l=0}^{1}\gamma_0(r3_i,n,m)\alpha_{i-1}^l(n)\beta_i(m) \quad (11)$$

After de-interleaving (DI), this term becomes the additional probability which must be incorporated into eqn. 41 for decoding a rate 1/3 system. Note that both sources of $Pv20_i$ information applied to MAP1 are also used in the forward-backward computations for MAP1.

It is clear from Fig. 1 that MAP1 must generate two probabilities; one given by an enhanced version of eqn. 41 for u_i, and the other for iteration. For iteration, MAP1 must provide a term $P(v2_i=0)$ (denoted $Pv20_i(r1)$ in Fig. 1) derived only from $r1_i$. As explained previously, subsequent interleaving then ensures that information provided about $w2_i$ is (ideally) independent of that provided by $r2_i$ and $r3_i$. By replacing u_i with $v2_i$ in the rate 1/2 analysis given in the Appendix, it can be shown that the required term is given by

$$P(v2_i = 0) = \sum_m \sum_n \sum_{l=0}^{i} \gamma_0(r1_i, n, m)\alpha_{i-1}^{l}(n)\beta_i(m) \quad (12)$$

where γ_0 reduces to either zero or a single probability in $r1_i$ (eqn. 38). The interleaved version of eqn. 12, i.e. $Pw20_i(r1)$, is then used in the forward-backward probability computations for MAP2, and can be regarded as extrinsic information about $w2_i$ for this decoder.

Each MAP decoder performs forward-backward recursions for the complete received sequence, and then the process is repeated for a specified number of iterations. MAP2 decoding is performed first and so, for the first iteration, we set $Pw20_i(r1) = 0.5$ since its extrinsic input is unknown. In addition, as iteration proceeds, the extrinsic output from MAP1 becomes more dependent on the MAP2 output, and so becomes less effective as extrinsic information for MAP2.

3 Implementation of the SCCC scheme

3.1 Outer decoder, MAP1

For an iterative SCCC scheme involving several MAP decoders, it is helpful to write $p(v1, v2) = p(v1)\,p(v2)$ since $p(v2)$ is obtained from a separate de-interleaving process. For example, after decrementing the state index to agree with Fig. 6, eqns. 1 and 2 interpret eqn. 34 as

$$\begin{aligned} &P(S_i = 0) \\ &= P(r1_i|v1_i = 0) \cdot Pv20_i(r2) \cdot P(s_{i-1} = 0) \\ &+ P(r1_i|v1_i = 1) \cdot (1 - Pv20_i(r2)) \cdot P(s_{i-1} = 1) \end{aligned} \quad (13)$$

$$\begin{aligned} &P(S_i = 1) \\ &= P(r1_i|v1_i = 1) \cdot Pv20_i(r2) \cdot P(s_{i-1} = 2) \\ &+ P(r1_i|v1_i = 0) \cdot (1 - Pv20_i(r2)) \cdot P(s_{i-1} = 3) \end{aligned} \quad (14)$$

where

$$p(r1_i|v1_i = 0) = \exp(-(r1_i + 1)^2/\sigma^2) \quad (15)$$

$$P(r1_i|v1_i = 1) = \exp(-(r1_i - 1)^2/\sigma^2) \quad (16)$$

Accounting for the additional information supplied by MAP2, eqns. 13 and 14 become

$$\begin{aligned} &P(S_i = 0) \\ &= P(r1_i|v1_i = 0) \cdot Pv20_i(r2) \\ &\quad \cdot Pv20_i(r3) \cdot P(S_{i-1} = 0) \\ &\quad + P(r1_i|v1_i = 1) \cdot (1 - Pv20_i(r2)) \\ &\quad \cdot (1 - Pv20_i(r3)) \cdot P(S_{i-1} = 1) \end{aligned} \quad (17)$$

$$\begin{aligned} &P(S_i = 1) \\ &= P(r1_i|v1_i = 1) \cdot Pv20_i(r2) \\ &\quad \cdot Pv20_i(r3) \cdot P(S_{i-1} = 2) \\ &\quad + P(r1_i|v1_i = 0) \cdot (1 - Pv20_i(r2)) \\ &\quad \cdot (1 - Pv20_i(r3)) \cdot P(S_{i-1} = 3) \end{aligned} \quad (18)$$

As discussed, once all four probabilities have been computed, they must be normalised. The backward recursion in eqn. 36 can be implemented using eqns. 6–9. For example, accounting for additional information from MAP2, the recursion to state 0 from states 0 and 2, is, from eqn. 6

$$\begin{aligned} &Pb(S_i = 0) \\ &= P(r1_{i+1}|v1_{i+1} = 0) \cdot Pv20_{i+1}(r2) \\ &\quad \cdot Pv20_{i+1}(r3) \cdot Pb(S_{i+1} = 0) \\ &\quad + P(r1_{i+1}|v1_{i+1} = 1) \cdot (1 - Pv20_{i+1}(r2)) \\ &\quad \cdot (1 - Pv20_{i+1}(r3)) \cdot Pb(S_{i+1} = 2) \end{aligned} \quad (19)$$

The two outputs of MAP1 are given by eqn. 12 and an enhanced version of eqn. 41. We note that the first probability in eqn. 41 is already given by eqn. 15. The summation terms in eqn. 41 are then obtained from the trellis in Fig. 2 by noting the transitions corresponding to $u_i = v1_i = 0$, i.e. there are four transitions. Each of the corresponding products must be scaled by the appropriate additional probability associated with $r3$, giving, before normalisation

$$\begin{aligned} &P(u_i = 0) \\ &= P(r1_i|u_i = 0) \\ &\times [Pv20_i(r2) \cdot Pv20_i(r3) \cdot P(S_{i-1} = 0) \cdot Pb(S_i = 0) \\ &+ Pv20_i(r2) \cdot Pv20_i(r3) \cdot P(S_{i-1} = 1) \cdot Pb(S_i = 2) \\ &+ (1 - Pv20_i(r2)) \cdot (1 - Pv20_i(r3)) \\ &\quad \cdot P(S_{i-1} = 2) \cdot Pb(S_i = 3) \\ &+ (1 - Pv20_i(r2)) \cdot (1 - Pv20_i(r3)) \\ &\quad \cdot P(S_{i-1} = 3) \cdot Pb(S_i = 1)] \end{aligned} \quad (20)$$

According to eqn. 12, the feedback output from MAP1 can be deduced from the trellis by summing all terms associated with $v2_i = 0$. Note also that γ_0 must only be associated with $r1_i$ in order to generate true extrinsic information for MAP2. Expanding eqn. 12

$$\begin{aligned} &P(v2_i = 0) = Pv20_i(r1) \\ &= P(r1_i|v1_i = 0) \cdot P(S_{i-1} = 0) \cdot Pb(S_i = 0) \\ &\quad + P(r1_i|v1_i = 0) \cdot P(S_{i-1} = 1) \cdot Pb(S_i = 2) \\ &\quad + P(r1_i|v1_i = 1) \cdot P(S_{i-1} = 2) \cdot Pb(S_i = 1) \\ &\quad + P(r1_i|v1_i = 1) \cdot P(S_{i-1} = 3) \cdot Pb(S_i = 3) \end{aligned} \quad (21)$$

3.2 Inner decoder, MAP2

Since MAP2 uses the same code, the forward and backward recursions are similar to those for MAP1, except that the additional information term is now replaced with the extrinsic information. As an example, the forward recursion to state 0 in eqn. 17 becomes

$$\begin{aligned} P(S_i = 0) = &P(r2_i|w2_i = 0) \cdot Pv30_i(r3) \\ &\cdot Pw20_i(r1) \cdot P(S_{i-1} = 0) \\ &+ P(r2_i|w2_i = 1) \cdot (1 - Pv30_i(r3)) \\ &\cdot (1 - Pw20_i(r1)) \cdot P(S_{i-1} = 1) \end{aligned} \quad (22)$$

where, prior to normalisation,

$$Pv30_i(r3) = \exp(-(r3_i + 1)^2/\sigma^2) \quad (23)$$

and $Pw20_i(r1)$ is the interleaved form of eqn. 21.

As discussed, MAP2 should generate an output which is (ideally) independent of both the extrinsic information and received symbol $r2_i$, as in eqn. 11. Again, eqn. 11 can be implemented with reference to the trellis, giving

$$P(w2_i = 0) = Pw20_i(r3)$$
$$= P(r3_i|v3_i = 0) \cdot P(S_{i-1} = 0) \cdot Pb(S_i = 0)$$
$$+P(r3_i|v3_i = 0) \cdot P(S_{i-1} = 1) \cdot Pb(S_i = 2)$$
$$+P(r3_i|v3_i = 1) \cdot P(S_{i-1} = 2) \cdot Pb(S_i = 3)$$
$$+P(r3_i|v3_i = 1) \cdot P(S_{i-1} = 3) \cdot Pb(S_i = 1) \quad (24)$$

This must then be normalised. In a software implementation of the above equations, the encoder state is usually time-synchronised with the information sequence, i.e. it is in state S_i for input data u_i, as in Fig. 2. As the state in Fig. 6 is time-slipped, it will usually be necessary to increment the state index by 1 in eqns. 13–24.

3.3 Numerical problems

It is worth highlighting several possible numerical problems that can occur during implementation. For example, at the start of MAP decoding (low values of i), both the normalised sums $\Sigma\ \alpha\beta$ in eqn. 31 can be << 1 since $\alpha_i^k(m)$ can be very small for most values of m. This can lead to increased errors at the start of the block; a problem which could largely be removed by using a sliding window MAP algorithm and continuous decoding [16], as in Viterbi decoding. A similar numerical problem can arise when the extrinsic term $Pw20_i(r1)$ is close to 0 or 1, as this can eliminate possible correct paths. One solution is to provide numerical limits to the extrinsic term.

3.4 Improved SCCC scheme

Simulation (Section 4) shows that the basic system in Fig. 1 gives only modest performance relative to what can be achieved with serial concatenation. A reason for this is detailed below.

For a PCCC scheme (Turbo code), it has been shown that the upper bound on the bit error probability depends on interleaver length N approximately as [8]

$$P_b(\epsilon) \ : \ N^{2n_{max}-w_{min}-1} \quad (25)$$

where w_{min} is the minimum information weight in the error events of the individual codes, and $n_{max} = \lfloor w_{min}/2 \rfloor$. In eqn. 25 we have taken only the first term in the bit error probability bound given previously [8], i.e. $w = w_{min}$, since error events tend to be associated with low information weight, at least for large E_b/N_0. For a recursive code $w_{min} = 2$ (e.g. Fig. 2), a polynomial $u(D) = 1 + D^3$ would be divisible by $1 + D + D^2$, giving a finite weight sequence $v2(D)$, which in turn could be considered to be an error event for an all-zeros input. For a PCCC scheme, the interleaver gain therefore goes as N^{-1} [8].

For simplicity, we might then assume that the bit error probability of the SCCC scheme in Fig. 1 is largely determined by error events generated by minimum weight sequences entering encoder 2 (simulation confirms this assumption). From the above discussion, a weight as low as 2 in sequence $v2(D)$ thus generates an error event in decoder 2 relative to the all-zeros sequence. Unfortunately, it is perfectly possible to generate $v2(D)$ of weight 2 for a finite weight input $u(D)$, and so we might conclude that the interleaver gain for the SCCC system in Fig. 1 also goes as N^{-1}.

Fig. 3 shows a straightforward modification of Fig. 1 to increase the minimum weight of sequence $v2(D)$, and thereby improve the interleaver gain. Encoder 2 input now has a minimum weight corresponding to the d_{free} of code 1, and so $w_{min} = d_{free}$. For SCCCs, it is therefore beneficial to choose an outer code with a large d_{free}; in particular, if d^0_{free} is the free distance of the outer code, it has been shown [11] that the largest negative exponent of N is $\lfloor (d^0_{free} + 1)/2 \rfloor$.

Optimal selection of the CCs for SCCC systems is discussed elsewhere [8, 11]. Suppose that both CCs are defined as in Fig. 2, corresponding to $d_{free} = 5$. Puncturing is used to maintain a rate 1/3 and a suitable perforation matrix for encoder 2 is

$$P = \begin{bmatrix} 1111 \\ 1001 \end{bmatrix} \quad (26)$$

This denotes the puncturing of alternate code bits for every two input bits, corresponding to a rate 2/3 inner code. Implementation of Fig. 3 requires small modifications to the equations in Section 3. When decoding u_i, MAP1 must simultaneously use all input information relating to u_i, and so eqn. 20 must be modified accordingly. This means that $P(r1_i|u_i = 0)$ must be multiplied by $P(r3_i|u_i = 0)$.

Now consider the transmission of $w1_i$, which corresponds to $v1_i$ or u_i after interleaving. MAP2 decodes $w1_i$ using $r1_i$, and so the extrinsic term $Pv\ 10_i$ from MAP1 should ideally be independent of $r1_i$. This can be achieved by modifying eqn. 41 to give

$$Pv10_i = \sum_m \sum_n \sum_{l=0}^{1} P(r2_i|u_i = 0, S_{i-1} = n, S_i = m) \cdot \alpha^l_{i-1}(n)\beta_i(m) \quad (27)$$

Accounting for the additional input to MAP1 from MAP2, eqn. 27 is implemented using eqn. 20, but without the $P\ (r1_i|u_i = 0)$ term. A similar modification is required when decoding $w2_i$. In this case, $Pv20_i$ from MAP1 should be independent of $r2_i$, as implemented by eqn. 21. However, MAP1 can now use both inputs

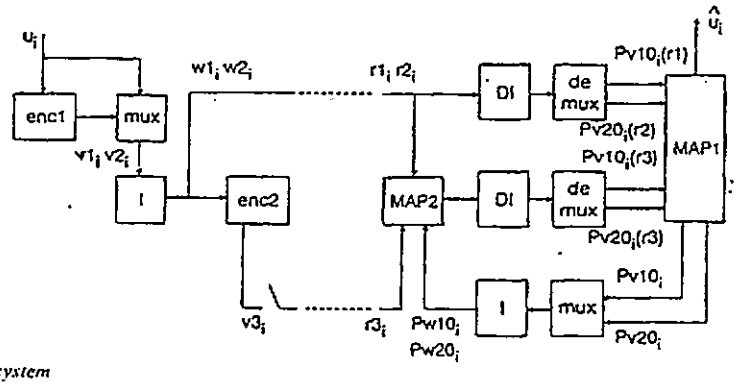


Fig. 3 *Improved R = 1/3 SCCC system*

associated with $v1_i$, giving

$$\begin{aligned} P(v2_i = 0) &= Pv20_i(r1) \\ &= P(r1_i|v1_i = 0) \cdot P(r3_i|v1_i = 0) \\ &\quad \cdot P(S_{i-1} = 0) \cdot Pb(S_i = 0) \\ &\quad + P(r1_i|v1_i = 0) \cdot P(r3_i|v1_i = 0) \\ &\quad \cdot P(S_{i-1} = 1) \cdot Pb(S_i = 2) \\ &\quad + P(r1_i|v1_i = 1) \cdot P(r3_i|v1_i = 1) \\ &\quad \cdot P(S_{i-1} = 2) \cdot Pb(S_i = 1) \\ &\quad + P(r1_i|v1_i = 1) \cdot P(r3_i|v1_i = 1) \\ &\quad \cdot P(S_{i-1} = 3) \cdot Pb(S_i = 3) \end{aligned} \tag{28}$$

Note that the $r3_i$ term used here corresponds to a different time slot to the $r3_i$ used for decoding $w2_i$.

4 Simulation results

For simulation, we need to choose interleaver and noise generation algorithms. Several interleaver designs were tried, although interleaver selection does not appear to be critical [7]. Interleaver optimisation is discussed elsewhere [4, 5]. The selected approach generated random numbers r_i using a linear feedback shift register and primitive polynomial of sufficient order to accommodate the maximum interleaver size. An arbitrary polynomial was selected. In terms of Fig. 1, and with the input bits to the interleaver in order $i = 0$ to $N - 1$, the interleaver function was then simply $w2[r_i] = v2_i$. Random Gaussian noise was simulated using the Rayleigh distribution method. This performed better than the Central Limit theorem method, as the latter requires a summation over a large number of terms for sufficient accuracy.

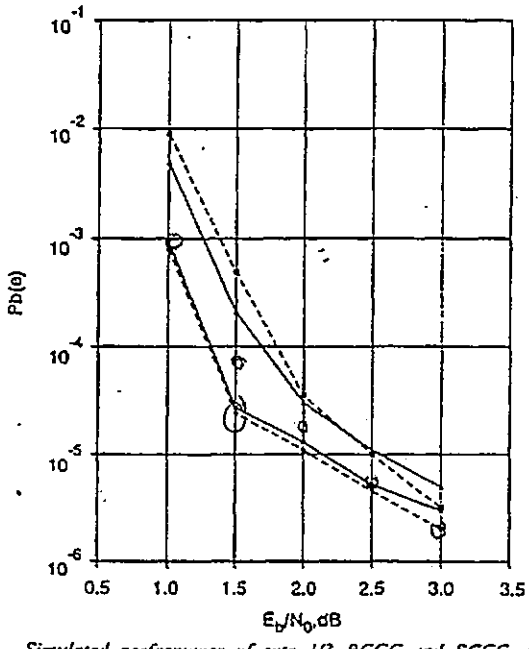


Fig. 4 *Simulated performance of rate 1/3 PCCC and SCCC systems (memory-4, unpunctured)*
■ SCCC, D = 500
▲ PCCC, D = 500
● PCCC, D = 1000
◆ SCCC, D = 1000

Using a simplistic argument, in Section 3.4 we indicated that the performance of the basic SCCC system in Fig. 1 should be approximately the same as that of the corresponding PCCC system. The interleaver gain in each case tends to go as N^{-1}. The simulation in Fig. 4 shows this to be approximately true for $E_b/N_0 >$ 2dB and memory-4 codes, where each CC had generator G_1

$$G_1 = \left[1, \frac{1 + D^4}{1 + D + D^2 + D^3 + D^4}\right]$$

$$G_2 = \left[1, \frac{1 + D + D^2 + D^4}{1 + D^3 + D^4}\right] \tag{29}$$

and a maximum of 20 iterations/block was allowed. In addition, both systems tend to exhibit the relatively high error floor characteristic of Turbo codes.

Fig. 5 shows simulation results for the SCCC scheme in Fig. 3. For low BER this required simulation runs of up to 10^8 bits. For *a–c* in Fig. 5 each CC was defined as in Fig. 2 ($d_{free} = 5$), with puncturing defined by eqn. 26. It is apparent that the interleaver gain is typically N^{-3}, as predicted by theory (note that a delay D now corresponds to an interleaver length $N = 2D$). *b* in Fig. 5 also indicates the mean number of iterations before convergence (zero error/block). At low E_b/N_0 some blocks fail to converge, irrespective of the number of iterations. The Turbo code corresponding to *b* has a BER of 10^{-4} at 1.5dB [10] showing a clear advantage of the SCCC scheme.

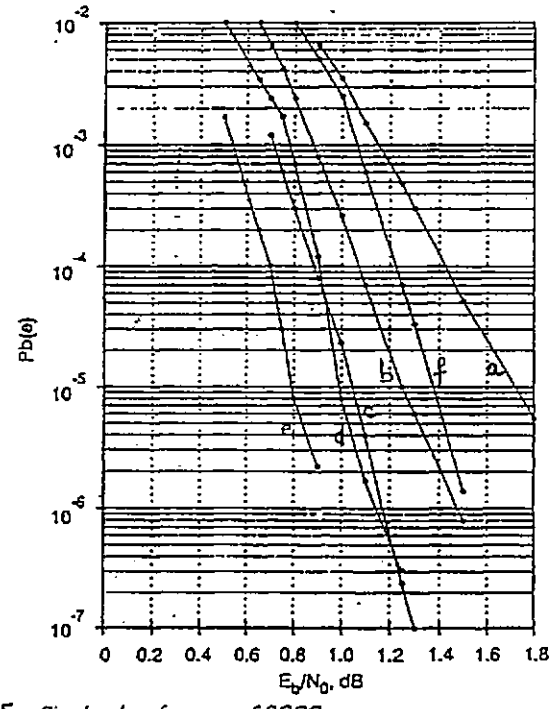


Fig. 5 *Simulated performance of SCCC systems*
a Rate 1/3, memory-2, D = 500
b Rate 1/3, memory-2, D = 1000
c Rate 1/3, memory-2, D = 2000
d Rate 1/4, memory-2, D = 1000
e Rate 1/4, memory-2, D = 2000
f Rate 1/3, memory-4 outer, memory-2 inner, D = 1000

d and *e* in Fig. 5 show rate 1/4 simulations for memory-2 codes (Fig. 2). *f* shows a rate 1/3 SCCC with memory-2 inner code (Fig. 2), and memory-4 outer code corresponding to G_2 in eqn. 29. This outer code has $d_{free} = 7$, giving a theoretical interleaver gain of N^{-4}. As discussed elsewhere [11], making the outer code the more powerful code is beneficial for large E_b/N_0, although this is difficult to show in simulation. However, for D = 1000, Fig. 5 indicates that this procedure is beneficial for $E_b/N_0 >$ 1.5dB, approximately.

5 Conclusions

Detailed computational steps for implementing an iterative SCCC decoder have been presented. These are based on MAP decoding of a single, rate 1/2, recursive, systematic convolutional code, which has been reduced to easily implemented equations for forward and backward recursion. In addition, the essential exchange of information between two MAP decoders has been clarified. For memory-4 codes, the basic SCCC has about the same performance as the corresponding Turbo code. Small modifications to the basic SCCC scheme have been discussed, resulting in a simulated BER for rate 1/3 memory-2 codes of 10^{-6} for E_b/N_0 = 1.5dB, a delay of 1000 bits, and an average of 2.2 iterations/block. For an outer code with d_{free} = 5, the results confirm the theoretical interleaver gain of N^{-1}. For a delay of 1000, increasing the power of the outer code to d_{free} = 7 gives improved performance for E_b/N_0 > 1.5dB.

6 Acknowledgment

This work is part of a programme funded by UK EPSRC Grant GR/K39578.

7 References

1 BERROU, C., THITIMAJSHIMA, P., and GLAVIEUX, A.: 'Les turbo-codes'. Ecole Nationale Supeneure des Telecommunications de Bretagne, France, 1992
2 BERROU, C., GLAVIEUX, A., and THITIMAJSHIMA, P.: 'Near Shannon limit error-correcting and decoding. Turbo-codes (1)'. Proceedings of IEEE international conference on *Communication*, ICC '93, May 1993, Vol. 2/3, pp. 1064–1071
3 ANDERSON, J.: 'Turbo coding for deep space applications'. Proceedings of IEEE international symposium on *Information theory*, Whistler, Canada, 1995, pp. 36
4 JUNG, P., and NASSHAN, M.: 'Dependence of the error performance of turbo codes on the interleaver structure in short frame transmission systems'. *Electron. Lett.*, 1994, 30, (4), pp. 287–288
5 ROBERTSON, P.: 'Illuminating the structure of code and decoder of parallel concatenated recursive systematic, Turbo, codes'. Proceedings of IEEE Globcom conference, San Francisco, California, December 1994, pp. 1298–1303
6 BENEDETTO, S., MONTORSI, G., DIVSALAR, D., and POLLARA, F.: 'Soft-output decoding algorithms in iterative decoding of Turbo codes'. TDA Progress Report pp.42–124, February 1996, http://tda.jpl.nasa.gov/tda/progress_report/42–124
7 BENEDETTO, S., and MONTORSI, G.: 'Unveiling Turbo codes: some results on parallel concatenated coding schemes'. *IEEE Trans. Inf. Theory*, 1996, 42, (2), pp. 409–428
8 BENEDETTO, S., and MONTORSI, G.: 'Design of parallel concatenated convolutional codes', *IEEE Trans. Commun.*, 1996, 44, (5), pp. 591–600
9 BENEDETTO, S., and MONTORSI, G.: 'Serial concatenation of interleaved codes: analytical performance bounds'. Proceedings of GLOBECOM '96, IEEE *Global telecommunications* conference, 18–22 November 1996, Vol. 1, pp. 106–110
10 BENEDETTO, S., and MONTORSI, G.: 'Iterative decoding of serially concatenated convolutional codes', *Electron. Lett.*, 1996, 32, (13), pp. 1186–1188
11 BENEDETTO, S., MONTORSI, G., DIVSALAR, D. and POLLARA, F.: 'Serial concatenation of interleaved codes: performance analysis, design and iterative decoding'. TDA Progress Report 42–126, August 1996, http://tda.jpl.nasa.gov/tda/progress_report/42 – 126
12 BAHL, L.R., COCKE, J., JELINEK, F., and RAVIV, J.: 'Optimal decoding of linear codes for minimising symbol error rate', *IEEE Trans. Inf. Theory*, 1974, 2, pp. 284–287
13 HAGENAUER, J., OFFER, E., and PAPKE, L.: 'Iterative decoding of binary block and convolutional codes'. *IEEE Trans. Inf. Theory*, 1996, 42, (2), pp. 429–445
14 HAGENAUER, J., PAPKE, L., and ROBERTSON, P.: 'Iterative, Turbo, decoding of systematic convolutional codes with the MAP and SOVA algorithms'. Proceedings of ITG conference on *Source and channel coding*, Munich, Germany, October 1994, pp. 21–29
15 ROBERTSON, P., VILLEBRUN, E., and HOEHER, P.: 'A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain'. Proceedings of IEEE international conference on *Communications*, Seattle, Washington, June 1995, pp. 1009–1013
16 BENEDETTO, S., MONTORSI, G., DIVSALAR, D., and POLLARA, F.: 'Algorithm for continuous decoding of turbo codes'. *Electron. Lett.*, 1996, 32, (4), pp. 314–315

8 Appendix: MAP decoding for a rate 1/2 code

The approach outlined here uses the concept of forward and backward recursion introduced previously [12]. Consider the decoding of the rate 1/2 recursive, systematic code generated by Fig. 2*a*. We assume that code symbols $v1_i$ and $v2_i$ are translated to the set {−1, +1} before transmission, and that $\mathbf{r}_i = (r1_i, r2_i)$ is the output of a discrete memoryless channel (DMC) disturbed by AWGN of standard deviation σ. Simple block-mode processing is assumed, whereby a complete block of data must be received before decoding commences. The inherent assumptions are that the decoding delay and memory requirements are acceptable. A block-mode MAP decoder will operate on the set of received symbols $\mathbf{r}_1$ to $\mathbf{r}_N$, denoted here as $\mathbf{r}_1^N$.

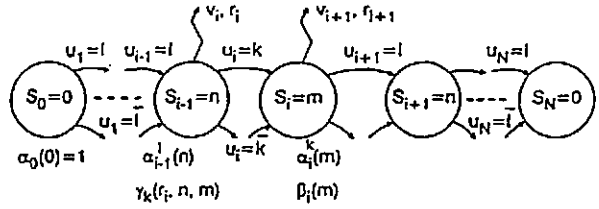


Fig. 6 *General state transition diagram*

For a formal analysis, we define the encoder state transition diagram as in Fig. 6, i.e. state S_{i-1} generates code vector v_i. The encoder commences in state zero for information bit u_1 and ends in state zero after receiving u_N (through the use of a data tail). The log-likelihood ratio of data bit u_i is

$$\Lambda(u_i) = \log \frac{P(u_i = 1|\mathbf{r}_1^N)}{P(u_i = 0|\mathbf{r}_1^N)} \tag{30}$$

Using forward and backward parameters α and β, respectively, this can be written as a summation over all possible states m

$$\Lambda(u_i) = \log \frac{\sum_m \alpha_i^1(m) \cdot \beta_i(m)}{\sum_m \alpha_i^0(m) \cdot \beta_i(m)} \tag{31}$$

where

$$\alpha_i^k(m) = P(u_i = k, S_i = m, \mathbf{r}_1^i) \tag{32}$$

$$\beta_i(m) = P(\mathbf{r}_{i+1}^N | S_i = m) \tag{33}$$

Using Fig. 6, the forward recursion can be shown to be

$$\alpha_i^k(m) = \sum_n \sum_{l=0}^{1} \gamma_k(\mathbf{r}_i, n, m) \cdot \alpha_{i-1}^l(n) \tag{34}$$

where

$$\gamma_k(\mathbf{r}_i, n, m) = P(u_i = k, S_i = m, \mathbf{r}_i | S_{i-1} = n) \tag{35}$$

Similarly, for the backward recursion we have

$$\beta_i(m) = \sum_n \sum_{l=0}^{1} \gamma_l(\mathbf{r}_{i+1}, m, n) \cdot \beta_{i+1}(n) \tag{36}$$

where

$$\gamma_l(\mathbf{r}_{i+1}, m, n) = P(u_{i-1} = l, S_{i+1} = n, \mathbf{r}_{i+1} | S_i = m) \tag{37}$$

In order to evaluate γ_i we make assumptions of independence and expand eqn. 35 as

$$\begin{aligned}\gamma_k(r_i, n, m) &= P(r_i|u_i = k, S_i = m, S_{i-1} = n) \\ &\cdot P(u_i = k|S_i = m, S_{i-1} = n) \\ &\cdot P(S_i = m|S_{i-1} = n) \end{aligned} \quad (38)$$

As shown in Fig. 6, the transition from state $S_{i-1} = n$ can go to one of two states, depending on the (random) input data u_i, and so the last term in eqn. 38 is set to 0.5. The middle term is simply either 1 or 0, and is accounted for when considering practical implementation via the encoder trellis. The systematic property of the code enables the first term in eqn. 38 to be expanded as

$$\begin{aligned}\gamma(r_i) &= P(r1_i|u_i = k, S_{i-1} = n, S_i = m) \\ &\cdot P(r2_i|u_i = k, S_{i-1} = n, S_i = m) \\ &= P(r1_i|u_i = k) \\ &\cdot P(r2_i|u_i = k, S_{i-1} = n, S_i = m) \end{aligned} \quad (39)$$

Finally, using eqn. 39 and inserting the recursion for α into eqn. 31 gives

$$\Lambda(u_i) = \log\left[\frac{P(r1_i|u_i = 1)}{P(r1_i|u_i = 0)} \cdot \frac{\sum_m \sum_n \sum_{l=0}^{1} \gamma_1(r2_i, n, m)\alpha_{i-1}^{l}(n)\beta_i(m)}{\sum_m \sum_n \sum_{l=0}^{1} \gamma_0(r2_i, n, m)\alpha_{i-1}^{l}(n)\beta_i(m)}\right] \quad (40)$$

For practical implementation, we could simply compute the probability

$$\begin{aligned}P(u_i = 0) &= P(r1_i|u_i = 0) \\ &\cdot \sum_m \sum_n \sum_{l=0}^{1} \gamma_0(r2_i, n, m)\alpha_{i-1}^{l}(n)\beta_i(m)\end{aligned} \quad (41)$$

Turbo code tree and code performance

A. Ambroze, G. Wade and M. Tomlinson

A code tree for a rate 1/3, memory-2 turbo code is developed and shown to have a nonuniform branch structure due to the effect of the interleaver. The tree is used to compute the weight spectrum, and the number of terms required for an accurate upper bound to the bit error rate are identified.

Introduction: It is well-known that the tree structure of a convolutional code is highly redundant and can be condensed into a trellis. This is illustrated in Fig. 1 for a systematic recursive code (the type of constituent code (CC) used in turbo codes). The encoder states are shown in brackets, and the branch labels denote the information bits. It is apparent that there are two identical subtrees corresponding to state 1, and it is this type of redundancy which enables the tree to be condensed into the usual trellis. We could say that the information bit generator has no memory; both 0 and 1 are valid values for any state, irrespective of the input sequence which led to that state. This is not true for turbo codes, due to the interleaver.

Turbo code tree: An encoder for a rate $R = 1/3$ turbo code is shown in Fig. 2*a*, where I denotes a random interleaver of length N, and $C1$ and $C2$ generate parity bits using the memory-2 recursive circuit in Fig. 1. Clearly, N information bits must be generated before encoding can commence, corresponding to a decoding delay of N, as indicated.

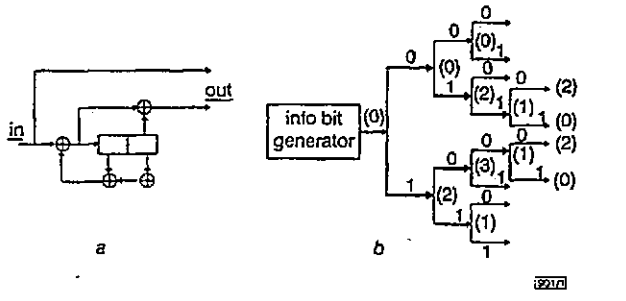


Fig. 1 *Constituent code and code tree*

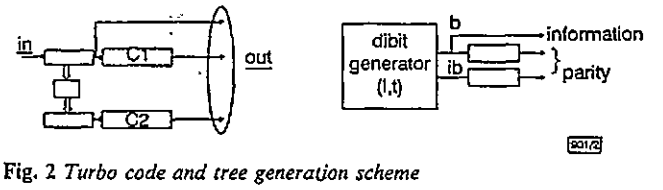


Fig. 2 *Turbo code and tree generation scheme*

We can regard this scheme as a single equivalent, rate 2/3 block code preceded by a bit-pair generator, Fig. 2*b*. If $C1$ has n_1 states and $C2$ has n_2 states, the equivalent code has n_1n_2 states, and the maximum depth of the corresponding tree will be N. The input bit pair arriving at $C1$ and $C2$ will now be constrained by the interleaver, and so we could regard the dibit generator in Fig. 2*b* as having memory. In general, valid bit-pairs will be generated, based on previous bit-pairs.

To illustrate tree generation, assume that $N = 7$ and the interleaver mapping is (0123456) → (6142305), i.e. $ib_0 = b_6$, $ib_1 = b_1$, $ib_2 = b_4$ etc. Part of the resulting tree is given in Fig. 3, where only four complete paths are drawn for clarity. At any node, the dibit generator checks to see if a particular bit depends upon a previous bit. Clearly, at $t = 0$, all four bit pairs are possible at node or state (0, 0), resulting in states (0, 0) (0, 2) (2, 0) and (2,2). At $t = 1$ the interleaver mapping forces $ib_1 = b_1$, resulting in only two possible transitions from states (0, 2) and (2, 0). At $t = 2$ neither b_2 nor ib_2 has been constrained and so there are four possible transitions for every state. At $t = 3$, b_3 is unconstrained, but ib_3 is constrained to b_2, giving just two possible transitions for any state. At $t = 4$, both b_4 and ib_4 are constrained to ib_2 and b_3, respectively, so there is only one possible transition from each state. The tree is completed in a similar manner, and has depth $N = 7$. The total number of codewords is $4{\cdot}2{\cdot}4{\cdot}2{\cdot}1{\cdot}2{\cdot}1 = 2^7$.

Clearly, the effect of the interleaver is to give a nonuniform distribution of branches at different depths of the tree. For a long, random interleaver, the levels with four branches tend to be located near the root of the tree ($t = 0$), whereas those with only one branch are close to the leaves of the tree. Also, the tree is nonredundant and so cannot be compacted into a trellis. For example, at $t = 3$, there are two identical states (1, 2) but these generate two different subtrees.

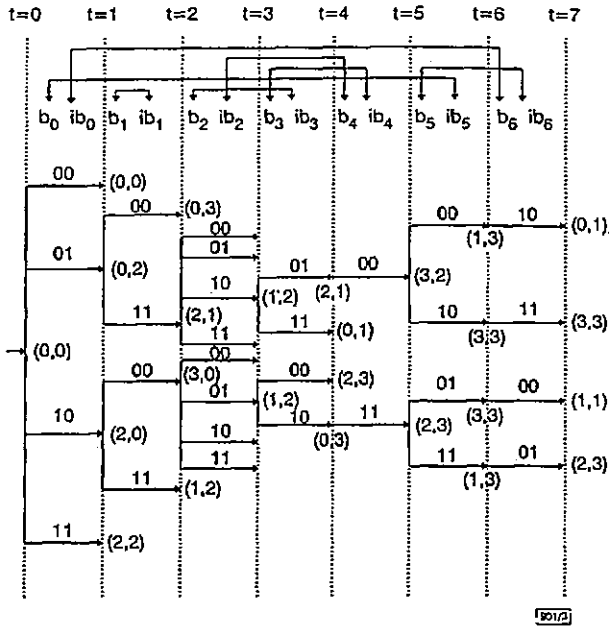


Fig. 3 *Turbo code tree for $N = 7$*

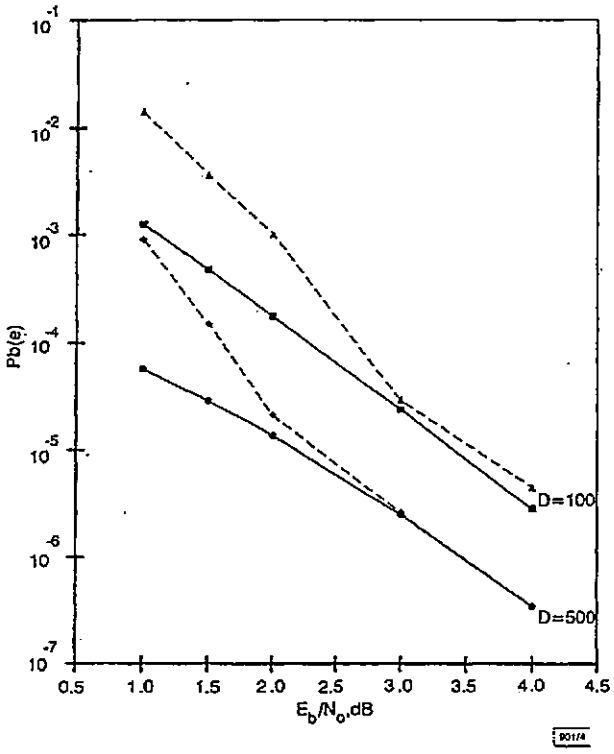


Fig. 4 *Upper bounds and simulated performance*

—■— upper bound, $D = 100$
– – ▲ – – turbo decoder, $D = 100$
—●— upper bound, $D = 500$
– – ◆ – – turbo decoder, $D = 500$

Weight spectrum and error rate bound: For a delay N, the total number of codewords will be 2^N and an efficient tree search is required. One approach is to use an algorithm which dynamically creates only some parts of the tree rather than the full tree. Suppose we search for all codewords with a Hamming weight up to w_{max}. If at any state the cumulative weight exceeds w_{max}, the subtree that starts in that state need not be searched, and an alternative path is selected.

Table 1: Turbo code weight spectrum

	$N = 100$		$N = 500$	
d	$a(d)$	$w(d)$	$a(d)$	$w(d)$
10	3	7	3	6
11	2	5	0	0
12	6	12	3	6
13	1	3	0	0
14	10	21	11	23
15	4	11	0	0
16	17	49	9	22
17	20	64	6	16
18	34	120	13	31
19	42	171	–	–
20	95	404	–	–
21	112	513	–	–
22	220	1006	–	–
23	288	1439	–	–
24	509	2677	–	–
25	822	4580	–	–
26	1374	7745	–	–

Table 1 shows the weight spectra of the code generated by Fig. 2*a* for randomly selected interleavers, and with *C*1 and *C*2 parity outputs defined as in Fig. 1. Here, $a(d)$ is the number of codewords of distance d from the all-zeros codeword, and $w(d)$ is the total information weight (sequence b in Fig. 2*b*) associated with all paths of distance d from the all-zeros codeword. Note that the effective free distance of the turbo code is $d_{free} = 10$. The union bound [2] for the bit error rate is

$$BER < \frac{1}{2}\sum_{d=d_{free}}^{w_{max}} \frac{w(d)}{N}\mathrm{erfc}\left(\sqrt{R\frac{E_b}{N_0}d}\right) \qquad (1)$$

and this is computed for $N = 100$ and $N = 500$ in Fig. 4. In Fig. 4, all available weights in Table 1 have been used, giving close agreement with simulation above 3dB for $N = 100$ and above 2dB for $N = 500$. The deviation from the bound below 2dB is attributed to poor convergence of the iterative algorithm, and too few terms in the upper bound. A non-iterative, modified stack algorithm (the *M*-algorithm [1]) gave a smaller deviation from the upper bound at low E_b/N_0. Fig. 5 illustrates the sensitivity of the upper bound to the number of spectrum elements used.

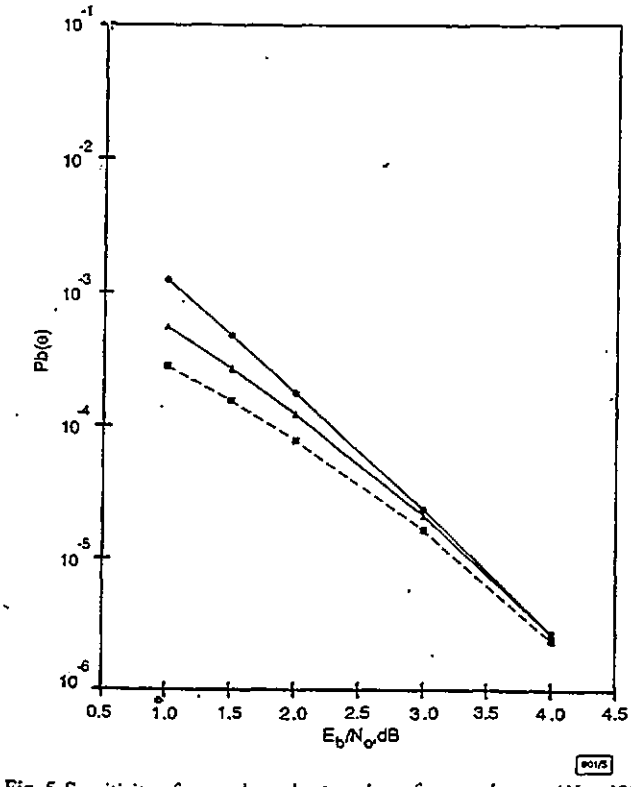


Fig. 5 *Sensitivity of upper bound to number of spectral terms ($N = 100$)*

– – ■ – – 3 terms
—▲— 9 terms
—●— 17 terms

Conclusion: Derivation of the code tree and corresponding weight spectrum for the basic rate 1/3, memory-2 turbo code is feasible for interleavers up to $N = 500$. The effect of the interleaver is to give a nonuniform distribution of branches at different depths of the tree. Nodes with a high number of branches tend to occur near the root of the tree. For $E_b/N_0 \geq 3$dB, the error rate bound is within a factor of two of the simulation for just three spectrum terms, indicating that a large weight spectrum is not required. However, for $E_b/N_0 \simeq 1$dB at least 20 terms are required.

15 December 1997
Electronics Letters Online No: 19980251

A. Ambroze, G. Wade and M. Tomlinson (*Satellite Centre, School of Electronic, Communication, and Electrical Engineering, University of Plymouth, Plymouth, Devon PL4 8AA, United Kingdom*)

Corresponding author: G. Wade

E-mail: j.wade@plymouth.ac.uk

References

1 HASHIMOTO, T.: 'A list type reduced-constraint generalization of the Viterbi algorithm', *IEEE Trans. Inf. Theory*, 1987, IT-33, (6), pp. 866–876

2 VITERBI, A., and OMURA, J.: 'Principles of digital communication and coding' (McGraw-Hill, Los Angeles, 1979)

Weakness in the Helsinki protocol

Gwoboa Horng and Chi-Kuo Hsu

The authors outline an attack on the Helsinki protocol for entity-authentication and authenticated key exchange, which was proposed for standardisation within the ISO/IEC CD 11770-3 standard draft in 1995.

Introduction: There have been many attempts to provide practical protocols for entity-authentication and authenticated key exchange. Several such protocols derived from [1] were surveyed in [2]. One of them, the Helsinki protocol, was proposed for standardisation within the ISO/IEC CD 11770-3 standard draft [3] in 1995. In this Letter, we propose an active attack on it.

Helsinki protocol: The objectives of entity-authentication and authenticated key exchange protocols are as follows [2]:

(1) Mutual authentication between two parties A and B, and
(2) Establishment of a common key K_{AB} between A and B, where:
  (i) each party provides the other with a partial key
  (ii) each party believes that the key was retrieved by the other party correctly
  (iii) each party believes that the partial key it was provided with was actually provided by the other identified party
  (iv) only those authenticated parties are able to construct the final key K_{AB}.

The Helsinki protocol is based on public key cryptography in order to meet the following objectives: messages are encrypted under other parties' public keys, and random numbers serve as message validators and answers. It proceeds as follows [2]:

Step 1: A constructs a block consisting of its identifier 'A', its partial key K_A, and a randomly chosen number r_A, and encrypts it with B's public key.

$$-A \rightarrow B : KT_{A1} = E_B(A, K_A, r_A)$$

Step 2: B decrypts KT_{A1} using its secret key, and verifies the message through the identifier 'A'. If the verification is successful, B constructs a block consisting of its partial key K_B, the extracted r_A, and a randomly chosen number r_B, and encrypts it with A's public key.

$$-B \rightarrow A : KT_B = E_A(K_B, r_A, r_B)$$

Step 3: A decrypts KT_B using its secret key and verifies that r_A is consistent with the original random number sent. If the verification

Practical aspects of iterative decoding

A.Ambroze, G.Wade and M.Tomlinson

Abstract: The convergence problem of iterative, block-mode, turbo decoders is discussed and the performance of a practical convergence criterion is presented. A fixed-point approach is used, whereby the saturation and stability characteristics of the extrinsic-probability vector for each MAP decoder are determined by simulation and used to terminate iteration. If these vectors are saturated and identical, or non-saturated and stable, the decoder has converged to a fixed point. The paper also examines the effect of interleaver design and machine precision effects on convergence. Sometimes, finite precision can lead to a limit-cycle effect, and practical solutions are discussed. Once convergence has been established, it can also be used to determine with high confidence the effective d_{free} of the decoder, even for large block lengths.

1 Introduction

This paper examines the convergence problem of block-mode iterative decoders and illustrates the use of a practical convergence criterion. The approach is largely via extensive simulation, since theoretical analysis is difficult for practical values of interleaver size. Discussion is based on the basic turbo decoder but in general the results also apply to multiple parallel (MPCCC) and serial (SCCC) iterative structures. It is the lack of convergence, and the type of convergence, that result in the finite decoded BER. The paper therefore investigates whether the iterative algorithm converges, and if so, whether it converges to the code's ML performance, as computed from the union bound. In addition, it is well known that interleaver design affects the theoretical performance of turbo codes, and so the effect of interleaver selection on the convergence of the iterative decoder is also examined.

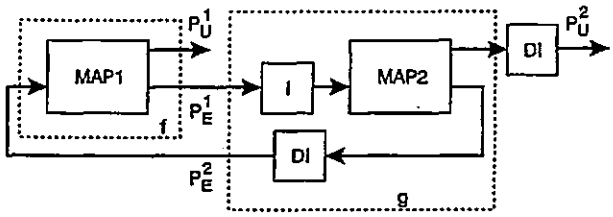


Fig. 1 *Extrinsic information in the turbo decoder*

A fixed-point approach to iterative decoding is presented, and so for each received block a check is made as to whether or not it converges and the properties (saturation and code weight) of the convergence point are determined. Referring to Fig. 1, each MAP decoder can be considered as a function acting on a probability vector $P_E = (P_{E1}, P_{E2}, ..., P_{EN})$ where N is the interleaver size (block length) and $P_{Ek} = P_E\{u_k = 1\}$, $k = 1, ..., N$; i.e. P_{Ek} is the probability of information bit u_k being 1 as computed from the extrinsic output of the MAP decoder. Starting from an arbitrary point, P_E may or may not converge to a solution P_{Es}, depending on whether or not the initial vector falls within a 'contraction region' (Fig. 2). In particular, the decoder is said to have converged to a fixed point if both extrinsic vectors in Fig. 1 have values close to 0 or 1 (and are identical), or if they are non-saturated but stable. For each case, the vector could still have errors even though it represents a

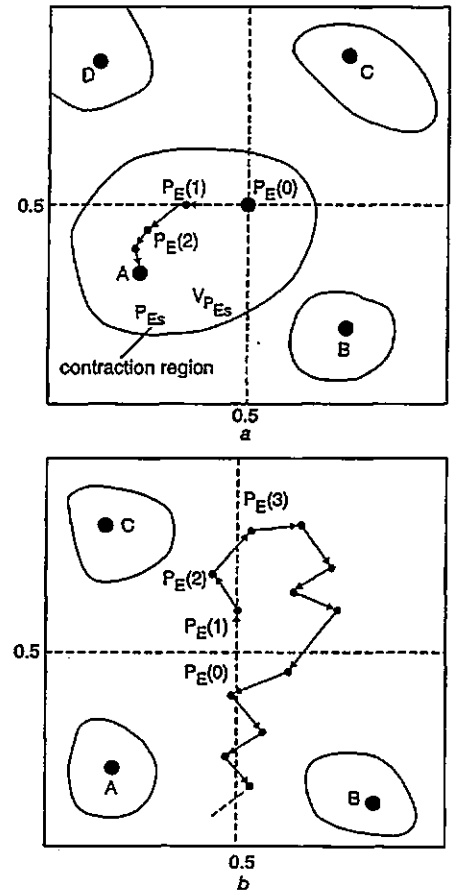


Fig. 2 *Visualisation of convergence ($N = 2$)*
a Convergent
b Nonconvergent
A, B, C, D represent fixed points for function h

IEE Proceedings online no. 20000151
DOI: 10.1049/ip-com:20000151
Paper first received 2nd March and in revised form 24th September 1999

The authors are with the Satellite Centre, SECEE, University of Plymouth, Plymouth, PL4 8AA, UK

italic

fixed point. Mathematically, the iterative decoding algorithm can be described as a problem of iteratively solving the equations

$$\begin{cases} P_E^1 = f(P_E^2) \\ P_E^2 = g(P_E^1) \end{cases} \quad (1)$$

where f and g represent the two N-dimensional MAP functions and g is considered to include the interleaving/deinterleaving process. This problem is equivalent to finding a solution for the equation

$$P_E^1 = f\{g(P_E^1)\} = h(P_E^1) \quad (2)$$

A vector that satisfies eqn. 2 is called a fixed point for function h. An iterative algorithm will converge to a solution P_{Es} for eqn. 2 if the following conditions are fulfilled [1]:

(i) Function h is a contraction in a region $V_{P_{Es}}$ of P_{Es}, i.e. there exists a real positive number $\rho < 1$ such that $\|h(x), h(y)\| \leq \rho\|x, y\|$, $\forall x, y \in V_{P_{Es}}$, where x and y are N-dimensional vectors within the contraction region. This implies that h is also N-dimensional.

(ii) The starting point of the iteration, i.e. the initial value of P_E^1, belongs to $V_{P_{Es}}$, as in Fig. 2. In practice, this vector is initialised to $P_E^1 = (0.5, ..., 0.5)$.

To determine whether the iterative decoder converges, it is necessary to find out whether the starting point lies in a contraction region for the N-dimensional function $h = (h_1, h_2, ..., h_N)$. This is accomplished if the norm of the matrix

$$J_h(x) = \begin{bmatrix} \frac{\partial h_1}{\partial x_1}(x) & \frac{\partial h_1}{\partial x_2}(x) & \cdots & \frac{\partial h_1}{\partial x_N}(x) \\ \frac{\partial h_2}{\partial x_1}(x) & \frac{\partial h_2}{\partial x_2}(x) & \cdots & \frac{\partial h_2}{\partial x_N}(x) \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial h_N}{\partial x_1}(x) & \frac{\partial h_N}{\partial x_2}(x) & \cdots & \frac{\partial h_N}{\partial x_N}(x) \end{bmatrix} \quad (3)$$

is less than 1 in a vicinity of the starting point. This approach is prohibitively complex, since function h does not have a simple analytical expression. Even so, it gives an idea about the algorithm's possible behaviour, as described in [2] for $N \in \{1, 2, 3\}$.

1.1 The Cauchy criterion

A more practical approach for a realistic value of N is to consider the decoding process as an infinite array of vectors indexed by the iteration number, i.e. $P_E^1(1)$, $P_E^1(2)$, ..., $P_E^1(n)$, ... where

$$P_E^2(n) = g\{P_E^1(n)\} \quad (4)$$

The Cauchy criterion [1] is then applied to determine whether or not the arrays are convergent and to stop iteration. Essentially, the criterion states that an array converges if and only if the amplitude of changes (as measured by a defined distance metric) tends to zero as the number of iterations increases. A small threshold δ (typically 10^{-3}) is therefore established and iteration is continued until

$$\left\|P_E^1(n+1), P_E^1(n)\right\| < \delta \quad (5)$$

Blocks failing to satisfy eqn. 5 for a given maximum number of iterations (typically 50) are deemed nonconvergent. Blocks that satisfy eqn. 5 are further checked with lower thresholds, the lower limit of δ being determined by machine precision. For simulation the squared Euclidean distance is used, normalised by the length of the interleaver, i.e.

$$\|x, y\|^2 = \frac{\sum_{k=1}^{N}(x_k - y_k)^2}{N} \quad (6)$$

Normalisation permits uniform thresholds to be used for different interleaver sizes.

2 Decoded block types

Decoded blocks have been classified as convergent or nonconvergent using the criterion in eqn. 5 and typical distance results are shown in Fig. 3. Owing to the linearity of the code, simulations can be performed by transmitting the all zeros information sequence, which means that $P_{Uk} = 1$ at the decoder output represents a bit error. For any erroneous block, the information weight (number of data errors/block) and the code weight can be calculated, the latter being obtained by re-encoding the decoded data sequence.

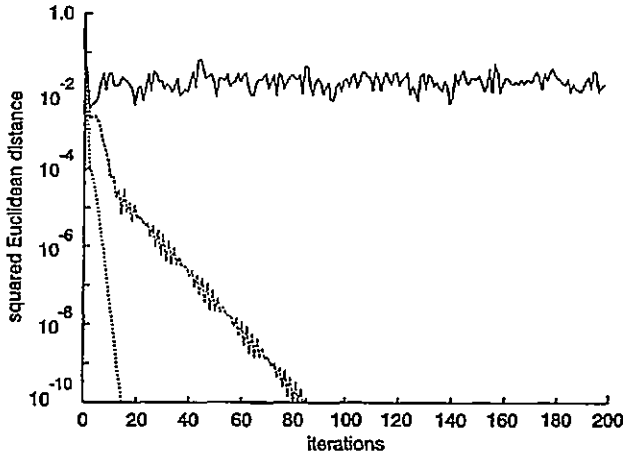


Fig.3 *Convergence for three different types of block*
nonconvergent
convergent
convergent

In this way, any decoded block can be associated with an information weight and code weight. The identification of low-code-weight blocks is useful for estimating d_{free} and if an attempt is made to compare the iterative decoder performance with the expected maximum-likelihood performance determined by the union bound. The convergent blocks can be further classified as:

- type 1: blocks for which vectors P_{Es}^1 and P_{Es}^2 have values close to 0 and 1 (saturation). In this case it can be shown that they are identical.
- type 2: blocks for which the two limit vectors are nonsaturated but stable, as in eqn. 5. In this case they are generally different.

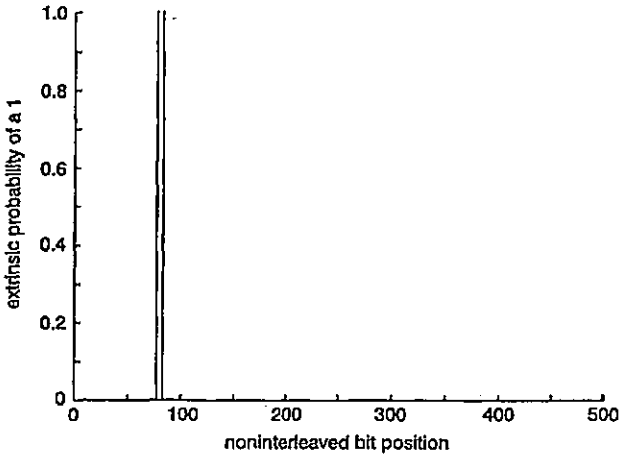


Fig.4 *Extrinsic information vector limit for MAP1 and MAP2 (type 1 decoded block, N = 500)*

An example of a type 1 block is shown in Fig. 4 and it represents the limit of the extrinsic information vectors $P_E^1(n)$ and $P_E^2(n)$, for a specified value of δ. Simulation shows that this type of block generally has low information/code

weight, similar to what would be expected in ML sequence decoding for a given E_b/N_o. The example shown corresponds to an erroneous block with information weight 2 and code weight 18, and the latter corresponds to the d_{free} of the turbo code used in the simulation. A special case of this type of decoded block is one that decodes with zero error. An example of a type 2 decoded block is given in Fig. 5 and, clearly, the probability vectors are not saturated. This particular example corresponds to a block with a decoded information weight of 3 and code weight of 292. In general, the information weight of type 2 blocks is low (in the range 2 – 10 for an $N = 500$). This leads us to associate these errors with bitwise-ML error blocks. They are nonrepetitive and difficult to identify. The result can be explained by the fact that the MAP decoders inherently minimise the probability of bit error, rather than sequence error.

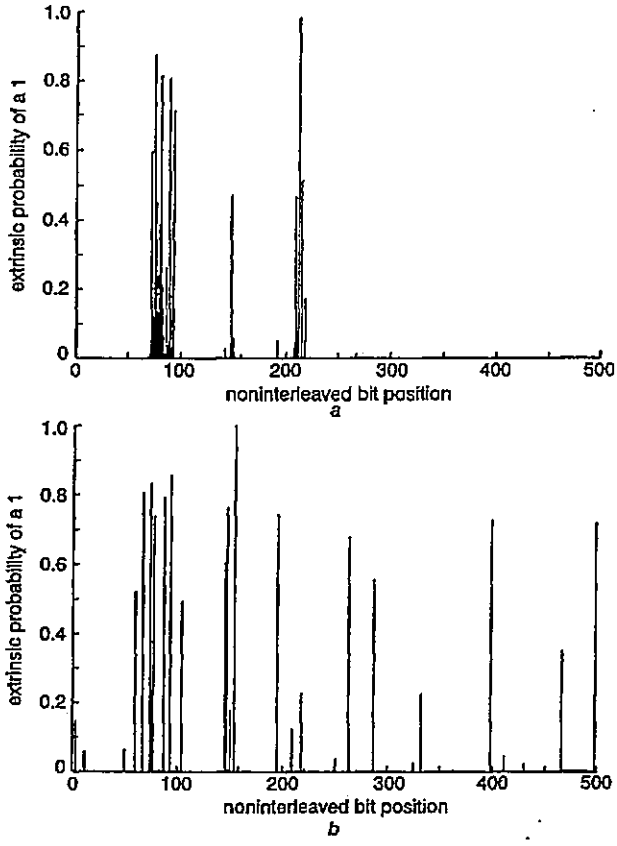


Fig. 5 *Extrinsic information vector limit for MAP1 and MAP2*
Type 2 decoded block, N = 500
a MAP1
b MAP2

From the above examples, two types of behaviour can be identified for the extrinsic information vector P_E. For type 1 blocks, the number of decoded bit errors coincides with the number of ones in P_E, whereas for type 2 blocks there are only three bit errors for a relatively erroneous extrinsic vector. For type 1 blocks, P_E is decided with high probability and so it dominates the decoding process in the last iterations. For type 2 blocks, the probability vectors are not saturated and so decoding is a compromise between channel values and extrinsic information values.

3 Convergence tests

Extensive simulations have been performed to study the convergence problem for a rate-1/3 (unpunctured) turbo decoder based on the constituent RSC(7,5) code. Simulations were performed for interleaver lengths of 500 and 2000 and a total of 200 000 blocks was used for each value of E_b/N_o. Tables 1 and 2 show only those convergent blocks that decoded in error, where convergence satisfied eqn. 5. The remaining blocks converged with zero error. Two interleaver designs have been tested: a randomly selected interleaver and a pseudorandom 'S' interleaver as described in [3]. The 'S' interleaver is designed so that bits that are less than S bits apart in the direct stream become more than S bits apart in the interleaved stream. For $N = 500$ $S = 14$ was used and for $N = 2000$ $S = 27$ was used. From the simulations the following conclusions can be drawn:

- Nonconvergence dominates the block-error rate at low E_b/N_o. As the E_b/N_o increases, nonconvergence decreases, and the convergent error events dominate the block error rate.
- The interleaver can be designed to improve convergence significantly, as well as improve ML performance. The 'S' interleaver is a good example.
- Convergence improves with interleaver size.

Table 1: Convergent/nonconvergent blocks for $N = 500$

I	E_b/N_o (dB)	1	1.3	1.5	2	3
Random	Convergent	3783	1909	1323	477	112
	Nonconvergent	4329	1002	438	53	2
$S = 14$	Convergent	1037	355	223	52	14
	Nonconvergent	2008	321	92	4	0

Table 2: Convergent/nonconvergent blocks for $N = 2000$

I	E_b/N_o (dB)	0.5	0.7	1	1.3
Random	Convergent	8600	5700	3608	2212
	Nonconvergent	8020	1360	284	88
$S = 27$	Convergent	2140	920	392	140
	Nonconvergent	4700	680	32	12

4 Criteria for terminating iteration

Generally, the iterative decoding process is stopped when a maximum number of iterations is reached. However, simulation shows that different blocks need different numbers of iterations to converge, and the average decoding time can be reduced by terminating the iteration when no improvement is observed. Clearly, a good termination criterion is to determine the number of errors for each iteration, and to stop at zero errors by reference to the original data. This has been used in the simulations to determine the absolute minimum for the average number of iterations. In practice, this could be realised by using a powerful cyclic redundancy check to determine whether a block has been completely corrected, which means adding redundancy and reducing the code rate.

An alternative approach uses the Cauchy criterion in eqn. 5 to terminate iteration. Too large a value for δ will increase the BER due to premature termination, i.e. before the actual extrinsic limit has been reached, whereas a lower threshold will increase the average number of iterations. Average iteration values and corresponding BER statistics for different thresholds are presented in Table 3. It is apparent that, provided that $\delta \leq 10^{-3}$, there will be only relatively small variation in BER and iteration number.

Criteria for terminating iteration in turbo decoders have also been proposed in [4], where the metric was cross

entropy, and in [5] where the convergence was determined by estimating a standard deviation for the extrinsic information.

Table 3: Average number of iterations and BER statistics for a rate-1/3 turbo decoder with $N = 500$, $S = 14$, RSC(7, 5) and different thresholds. All BER values should be multiplied by 10^{-5}

Average number of iterations				
	Criterion			
Eb/No (dB)	CRC stop at zero errors	Cauchy		
		$\delta = 10^{-2}$	$\delta = 10^{-3}$	$\delta = 10^{-4}$
1	3.5	4.4	5.5	6.5
1.5	2.0	3.1	3.7	4.5
2	1.4	2.5	3.1	3.6
Bit-error rate				
	Criterion			
Eb/No (dB)	CRC stop at zero errors	Cauchy		
		$\delta = 10^{-2}$	$\delta = 10^{-3}$	$\delta = 10^{-4}$
1	55.41	67.7	57.32	56.9
1.5	1.36	3.1	1.7	1.638
2	0.12	0.59	0.161	0.158

5 Evaluation of d_{free} from convergent blocks

The BER for a turbo code can be estimated from the union bound using the code-weight spectrum rather than d_{free} alone [6]. Nonetheless, d_{free} is still an important performance indicator, and the iterative algorithm can be used to estimate d_{free} even for large block lengths.

As an example, by using the tree search method presented in [6], an $N = 500$, RSC(7, 5) turbo code using an $S = 14$ interleaver is known to have $d_{free} = 18$ with a multiplicity (number of d_{free} paths) of 9. By applying the union bound for sequence-error rate for this code, one would expect approximately 12 d_{free}-type-error events in 200 000 blocks at an $E_b/N_o = 2$dB. Simulation for 200 000 blocks showed 10 blocks with a code weight of 18 from which it can be deduced that $d_{free} \simeq 18$ for this particular decoder. This implies that one can estimate d_{free} by searching for a converged block with minimum code weight (it is not necessary to check explicitly for convergence). Moreover, this 'block-convergence' method can be applied for large N (in contrast to the tree-search method) and, if necessary, the number of minimum weight blocks can be increased by decreasing E_b/N_o. Using this approach, the $N = 2000$, $S = 27$, RSC(7, 5) turbo code used in the convergence simulations was shown to have $d_{free} \simeq 20$, whereas the $N = 2000$, RSC(7, 5), random-interleaver turbo code has $d_{free} = 10$.

The tree search algorithm has also been used to determine the weight spectra for 3PCCC schemes having $N = 500$ and $d_{free} \leq 26$ (in this particular case 26 is the approximate limit of the tree-search algorithm). The block-convergence method was also applied and the results were confirmed by the tree-search algorithm. However, it is relatively easy to find interleaver pairs yielding $d_{free} > 26$, in which case the tree-search algorithm simply guarantees that $d_{free} > 26$. For these higher values the block convergence method can be used to estimate d_{free} since there will be a few low-code-weight convergent blocks even at relatively low E_b/N_o (in general there will also be some convergent blocks with high code weight). As for turbo codes, the minimum code-weight blocks should correspond to the d_{free} of the code since this is the most likely error event. As an example, three convergent blocks having input weight 2 and code weight 38 have been observed for an $N = 500$, RSC(7, 5), 3PCCC scheme using a pair of 'S'-type interleavers. They were the only convergent error blocks at $E_b/N_o = 1$dB in 1 200 000 blocks (although there were several nonconvergent blocks). For $d_{free} = 30$, the union bound gives about nine blocks in error in 1 200 000, for $d_{free} = 33$ the bound gives three blocks in error, and for a $d_{free} = 38$ the bound gives about one block in error. The three convergent blocks of weight 38 observed in the experiment thus suggest a d_{free} in the range 33 to 38.

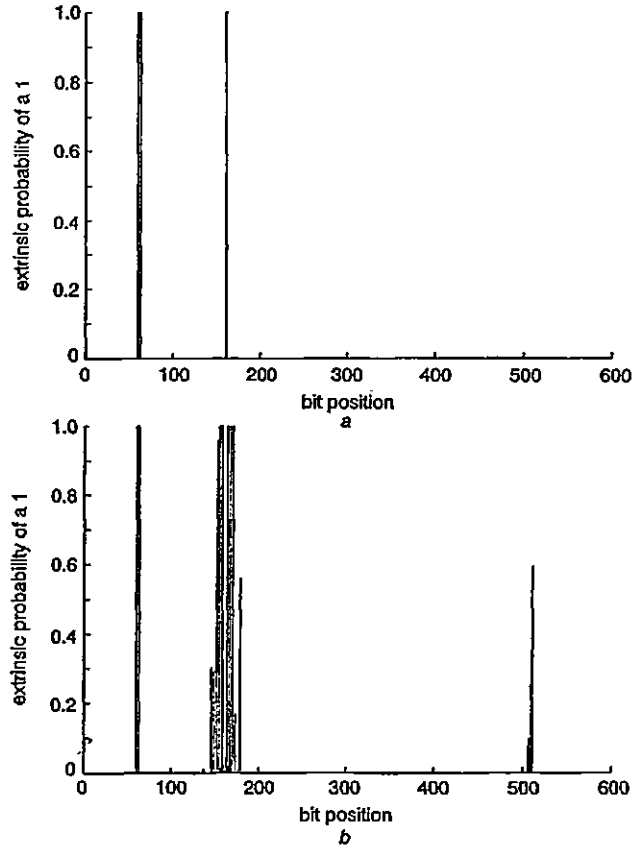


Fig. 6 *MAP decoder extrinsic information*
a MAP input
b MAP output

6 Machine precision effects

The finite precision used to to evaluate the iterative algorithm can sometimes lead to a limit cycle in P_E, i.e. a cyclic BER/block as a function of iteration. A typical case is shown in Fig. 6. Here the MAP decoder input vector $P_E(n)$ has two closely spaced errors (a probability of 1 representing an error) followed by an isolated error. The first two errors are separated by only two zeros and, since they are saturated, they force the decoder to follow a short, low-weight error event for the RSC(7, 5) code used in the simulation. The first two errors are therefore simply translated to the decoder output. This error event is illustrated in Fig. 7*b*, and the α and β probabilities are used in the usual forward–backward relation [7]

$$P_{Ek}(i) = P_E\{u_k = i\} = \sum_{\{s,s' | u_k = i\}} \alpha_{k-1}(s)\gamma_{Ek}(s, s')\beta_k(s') \quad i \in \{0, 1\} \quad (7)$$

where $\gamma_{Ek}(s, s')$ is the state-transition probability from extrinsic information, and both $\alpha_{k-1}(s)$ and $\beta_k(s')$ can be simultaneously large, resulting in a onfident decision.

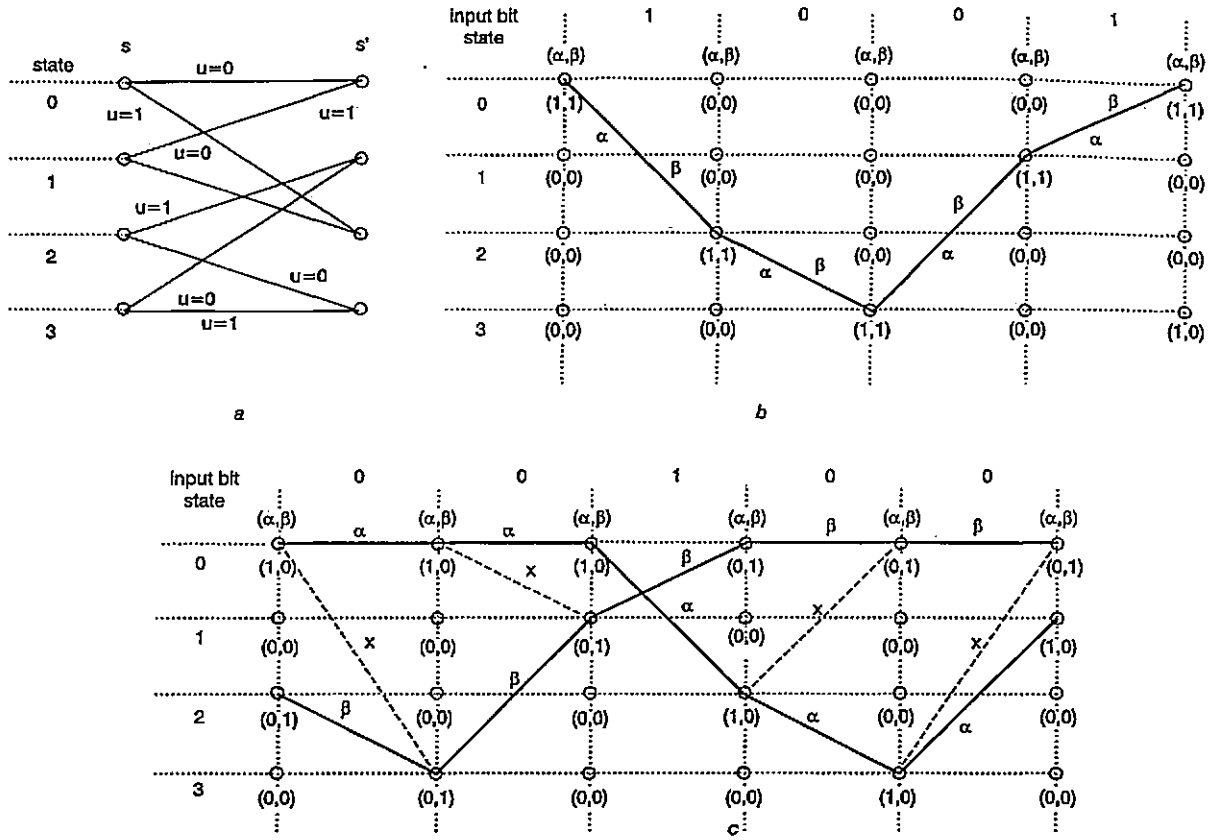


Fig. 7 *Trellis for the RSC(7, 5) code and alpha/beta recursions with saturated input values*
a Trellis
b Short error event
c Infinite error event

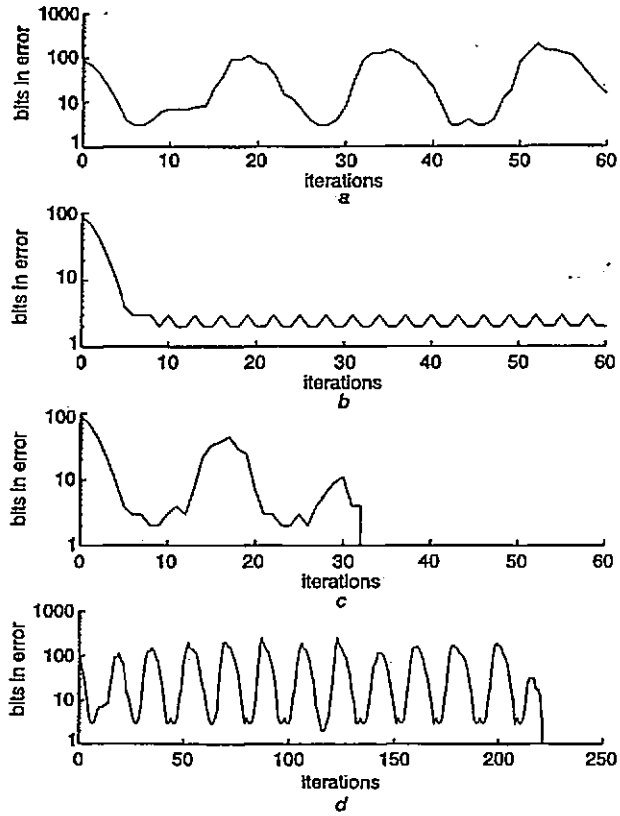


Fig. 8 *Block exhibiting limit cycle effect*
a Limit cycle block
b Limited probabilities
c Double machine precision
d Increasing iteration number

Entirely different results are obtained for the third input error. Fig. 6*b* shows that this causes a significant error extension (both before and after the error location), which results in even more errors in the following MAP decoder. On the other hand, since the probabilities are generally non-saturated, and because the function is actually a contraction in that region, the number of errors will again reduce, resulting in a limit cycle effect (Fig. 8*a*). This type of behaviour arises since the isolated error is far from the block edges and generates an error event of high code weight that disagrees in many places with the channel values. The nature of this error event is illustrated in Fig. 7*c*, where it can be seen that the saturated values for α and β correspond to 'invalid' trellis transitions, i.e. the values are no longer 'matched' to yield a high probability when used in eqn. 7. Error extension then results, since the MAP decoder now has to determine the information bits in this region by selecting between two very small probabilities, i.e. $P_{Ek}(1)$, $P_{Ek}(0) << 1$. The above effects can be reduced in several ways:

- Limit the extrinsic probability P_{Ek} to within a value ε of saturation. Fig. 8*b* shows the reduction in cycle amplitude for $\varepsilon = 10^{-7}$. Unfortunately, limiting sometimes also produces a small number of errors for blocks that would otherwise converge to zero error. Nevertheless, this approach has been used for most simulations.
- Increase the machine precision. The effect for a given block is illustrated in Fig. 8*c*.
- Increase the number of iterations. Owing to the chaotic nature of the process, after several cycles the decoder may converge to the correct sequence, as shown in Fig. 8*d*.

7 Conclusions

The highly complex fixed-point solution to iterative decoding has been reduced to a practical form using the Cauchy criterion. This approach uses a threshold to terminate iteration and so yields a suboptimal solution for the extrinsic

vector P_E in the fixed-point equation. For the 1/3 RSC(7, 5) turbo decoder, a suitable threshold is $\delta = 10^{-3}$.

Decoded blocks are classified as convergent or nonconvergent (in general both yield decoder errors), and the convergence properties are studied for different interleavers. It is shown that an 'S' interleaver can improve convergence compared with a random interleaver, as well as improve ML performance. Two types of convergent block are identified (both with low information weight) depending on whether P_E is saturated or nonsaturated. Saturated blocks have low code weight and correspond to ML sequence decoding, whilst non-saturated blocks have high code weight and can be likened to ML bitwise decoding. The most probable error in saturated blocks corresponds to a d_{free}-type error event, and this fact has been used to estimate d_{free} with high confidence. The technique is more suitable than tree-search methods for large interleavers and has been used to determine a d_{free} value as high as 38.

Some blocks exhibit a limit cycle effect on convergence, whereby the decoded BER is cyclic with iteration. Several solutions are suggested, and a good practical approach is to apply limits to the saturation values of P_E.

8 References

1 SAWYER, W.: 'Numerical functional analysis' (Oxford University Press, 1978)

2 McELIECE, R., RODEMICH, E., and CHENG, J.F.: 'The turbo decision algorithm'. Proceedings of 33rd Allerton conference on *Communication, control and computing*, 1995,

3 DIVSALAR, D., and POLLARA, F.: 'Multiple turbo codes for deep-space communications'. JPLTDA progress report, 1995, vol. 42-121, pp. 66–77

4 HAGUENAUER, J., OFFER, E., and PAPKE, L.: 'Iterative decoding of binary block and convolutional codes', *IEEE Trans. Inf. Theory*, 1996, 42, (2), pp. 429–445

5 ROBERTSON, P.: 'Illuminating the structure of code and decoder of parallel concatenated recursive systematic (turbo) codes'. Proceedings of IEEE GLOBECOM, 1994, pp. 1298–1303

6 AMBROZE, A., WADE, G., and TOMLINSON, M.: 'Turbo code tree and code performance', *Electron. Lett.*, 1998, 34, (4), pp. 353–354

7 AMBROZE, A., WADE, G., and TOMLINSON, M.: 'Iterative MAP decoding for serial concatenated convolutional codes', *IEE Proc., Commun.*, 1998, 145, (2), pp. 53–59