UNIVERSITY OF MALTA

Faculty of Engineering

Department of Communications and Computer Engineering

M. Phil.

Interleavers for Turbo Codes

by

Johann Briffa

A dissertation submitted in fulfilment of the requirements for the award of

Master of Philosophy of the University of Malta

OCTOBER 1999

Abstract

Since their introduction in 1993, in a seminal paper by Berrou, Glavieux, and Thitimajshima, Turbo codes have changed the way we look at high-performance error-control coding. After some initial skepticism from the coding community, Turbo coding gained rapid acceptance as the original results (Berrou *et al.* claimed a Bit Error Rate of 10^{-5} within 0.7 dB of the Shannon limit) were replicated by a number of independent researchers.

A flexible software Turbo codec and simulation environment is designed and tested. Using this tool, the Turbo code interleaver design problem is considered for large block sizes, where the effect of trellis termination is less marked. The performance of various interleavers with a similar block size are compared, including an implementation of the uniform interleaver. An optimised interleaver design technique based on simulated annealing is proposed – the results obtained show that the error performance may be significantly improved without increasing the delay. Finally, interleaver design for small Turbo codes is considered. In this case it is shown that while correct termination improves the performance for an average interleaver, its effect on Turbo codes with optimised interleavers is negligible.

Using our simulated annealing design technique it is easier to include restrictions which make the interleaver correctly-terminating or odd-even. While the S-random algorithm serves well for specifying interleaver spread, we believe that our algorithm is better suited for more sophisticated design criteria.

Acknowledgements

I would like to take this opportunity to thank my supervisor Dr. Victor Buttigieg, Ph.D. (Manch.), M.Sc. (Manch.), B.Elec.Eng. (Hons), MIEEE, for suggesting this field of study and for all his invaluable assistance.

This thesis finds me once again indebted to my family, particularly my parents, for their patience and support throughout my studies. I never show you how much I appreciate what you do, certainly not as much as you deserve. Thank you!

Johann Briffa

OCTOBER 1999

To my family and to the one I love

There is nothing like looking, if you want to find something.

Thorin Oakenshield

The Hobbit

JOHN RONALD REUEL TOLKIEN, 1937

Contents

List of Figures

List of Tables

Glossary

A_d	The total information weight of all codewords of weight d divided by the number of information bits per codeword
c	A factor depending on the confidence level of a Monte Carlo estimate
D	Delay operator
d	The weight of a codeword (Hamming distance from all zero codeword)
d_{free}	The free distance of a code (smallest codeword weight)
E	The Energy in a Simulated Annealing algorithm
$\frac{E_b}{N_0}$	The signal to noise ratio per bit
E_b	The average energy per data bit
$\mathbf{G}(D)$	Generator matrix of a convolutional code
$g_{i,j}(D)$	Element $i \in [1,k]$, $j \in [1,n]$ of generator matrix $\mathbf{G}(D)$.
$g_i^{(x)}$	Feedback connection for input $x \in [1,k]$ delayed by i time units
K	Number of symbols in source alphabet (possible inputs at time t)
k	Number of input bits in a convolutional code
$\mathrm{L}(x_t)$	Log-Likelihood Ratio of bit x_t
M	Number of states in a convolutional encoder
m	State variable, representing the present state
m'	State variable, representing the next state
N_d	The number of codewords of total weight d
N_{free}	The multiplicity of free distance codewords
N_0	The unilateral noise power spectral density (equal to the average noise energy per modulation period)
N	Number of symbols in encoded alphabet (possible outputs at time t)
n_i	The AWGN component in a communication system

n_{i_i}	In-phase component of the noise
n_{q_i}	Quadrature component of the noise
n	Number of output bits in a convolutional code
P_b	Bit error rate
P_y	$Pr[Y_0^{\tau-1}]$
$p_t(m\|m')$	$Pr[S_{t+1} = m\|S_t = m']$
p	Period of the impulse response of an RSC code
$q_t(X\|m', m)$	$Pr[X_t = X\|S_{t-1} = m'; S_t = m]$
$R(Y_t\|X)$	Pr[having transmitted X if Y_t was received]
$R_t(X)$	Pr[having transmitted X if Y_t was received]
R	Code rate
R_0	Computational cutoff rate for Union Bound
S	Interleaver spread
S_t	Encoder state at time t
s	Number of sets in a Turbo code (number of parallel component codes)
T	Temperature in Simulated Annealing algorithm
t	Time variable, taking the range $0 \leq t < \tau$
$\mathbf{u}(D)$	Input vector to a convolutional encoder
$u_i(D)$	Element $i \in [1, k]$ of input vector $\mathbf{u}(D)$
$\mathbf{v}(D)$	Output vector from a convolutional encoder
$v_j(D)$	Element $j \in [1, n]$ of output vector $\mathbf{v}(D)$
w_d	The average information weight of codewords of total weight d
w_{free}	The average weight of the information sequences causing free distance codewords
w	The information weight of a codeword
$\mathbf{X}_0^{\tau-1}$	Vector containing encoded message
X_t	Encoded symbol transmitted at time t
$\mathbf{x}_0^{\tau-1}$	Vector containing source message
x_t	Source symbol transmitted at time t
$\tilde{\mathbf{x}}_0^{\tau-1}$	Vector containing interleaved source message
$\tilde{x}_t$	Source symbol encoded at time t in the interleaved sequence
$\mathbf{Y}_0^{\tau-1}$	Vector containing received message

Y_t	Corrupted modulation symbol received at time t
$\alpha_t(m)$	$Pr[S_t = m; Y_0^{t-1}]$
$\beta_t(m)$	$Pr[Y_t^{\tau-1} \vert S_t = m]$
$\gamma_t(m', m)$	$Pr[S_{t+1} = m; Y_t \vert S_t = m']$
$\gamma_t(m', x)$	$Pr[S_{t+1} = m; Y_t \vert S_t = m']$, where $m = f(m', x)$
ε	The Monte Carlo estimate
ζ	Shift distance for a Barrel-Shift interleaver
η	The number of samples in a Monte Carlo experiment
κ	Constraint length of a convolutional code
$\lambda(t)$	The interleaver permuting function
$\lambda_t(m)$	$Pr[S_t = m; Y_0^{\tau-1}]$
μ	The mathematical mean of a sample of results
$\hat{\mu}$	The mathematical mean of the population of results
ν	Length of the tail in symbols
$\Pi_0^{\tau-1}(x)$	Vector containing *A Posteriori* Probabilities
$\Pi_t(x)$	The *a posteriori* probability that $x_t = x$
$\pi_0^{\tau-1}(x)$	Vector containing *A Priori* Probabilities
$\pi_t(x)$	The *a priori* probability that $x_t = x$
ϖ	Side dimension of Berrou-Glavieux interleaver
ρ	The required tolerance in a Monte Carlo estimate (as a fraction)
σ	The standard deviation of a sample of results
$\hat{\sigma}$	The standard deviation of the population of results
$\sigma_t(m', m)$	$Pr[S_{t-1} = m'; S_t = m; Y_0^{\tau-1}]$
$\sigma_t(m', x)$	$Pr[S_{t-1} = m'; S_t = m; Y_0^{\tau-1}]$, where $m = f(m', x)$
τ	Frame length (including tail)
χ	The confidence level of a Monte Carlo estimate (expressed as a fraction)

Abbreviations

AWGN	Additive White Gaussian Noise
BCJR	Bahl-Cocke-Jelinek-Raviv algorithm
BER	Bit Error Rate
BPSK	Binary Phase Shift Keying
DMC	Discrete Memoryless Channel
FER	Frame Error Rate
FSM	Finite State Machine
IODS	Input-Output Distance Spectrum (of an interleaver)
IRWEF	Input-Redundancy Weight Enumerating Function
JPL	Jet Propulsion Lab (NASA)
LAM	Local Area Multicomputer
LLR	Logarithm of Likelihood Ratios
LUT	Look-Up Table
MAP	Maximum *A Posteriori*
MFD	Maximum Free Distance
ML	Maximum Likelihood
MPI	Message Passing Interface
NRC	Non-Recursive Convolutional code
NSC	Non-Systematic Convolutional code
OOP	Object-Oriented Programming
PCBC	Parallel Concatenated Block Code
PCCC	Parallel Concatenated Convolutional Code
RSC	Recursive Systematic Convolutional code
RV	Random Variable

SA	Simulated Annealing
SNR	Signal to Noise Ratio
SOVA	Soft-Output Viterbi Algorithm
TCM	Trellis-Coded Modulation
WEF	Weight Enumerating Function

Part I

Implementation of a Turbo Codec and Simulation Environment

Chapter 1

Introduction

1.1 Fundamentals

In a digital transmission system, error control is achieved by the use of a channel encoder at the transmitter and a corresponding decoder at the receiver, as depicted in Fig. 1.1. The aim is to ensure that the received information is as close as possible to the transmitted information. A well known result from Information Theory is that a randomly chosen code of sufficiently large block length τ is capable of approaching channel capacity [Shannon, 1948]. However, the optimal decoding complexity increases exponentially with τ up to a point where decoding becomes physically unrealisable.

The goal of coding theorists has been to develop codes that have large equivalent block lengths, yet contain enough structure that practical decoding is possible. However, with standard code structures (such as convolutional codes), the decoding complexity still increases at a much faster rate than the achievable gain. For high-performance applications, concatenation of standard codes has proven effective in increasing the effective block length while keeping the complexity manageable [Forney, 1966].

Recently, a new class of error correcting block codes called Turbo codes was introduced [Berrou *et al.*, 1993]. Due to the use of a large pseudo-random interleaver, Turbo codes appear random to the channel, yet possess enough structure that decoding can be physically realised. The size of the interleaver increases the effective block length, significantly improving the performance.

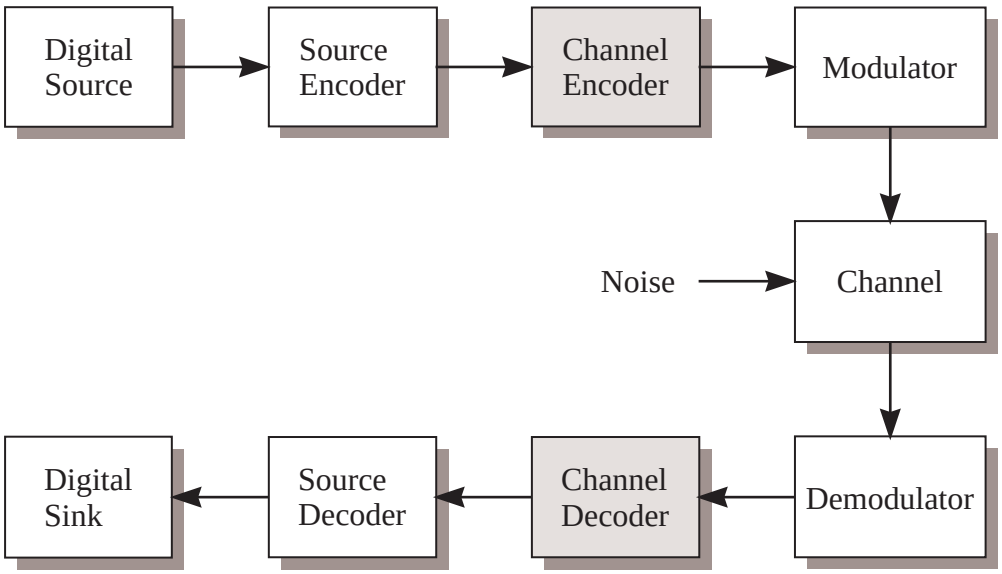


Figure 1.1: A Typical Digital Communication System

1.1.1 Turbo Encoder

A Turbo code is constructed as shown in Fig. 1.2. In the general case, the code consists of two parts: the uncoded information bits and a set of parity sequences generated by passing interleaved versions of the information bits through convolutional encoders. Typically, the encoders used are Recursive Systematic encoders; also, in most Turbo codes the encoders used are the same (making the Turbo code symmetric), and two sets of parity bits are used, one which is generated from the non-interleaved data sequence, and one which is generated from an interleaved sequence. This structure is shown schematically in Fig. 1.3. The parity bits are usually punctured in order to raise the code rate to $\frac{1}{2}$. The data sequence may or may not be terminated, usually depending on the kind of interleaver used.

1.1.2 Turbo Decoder

For the typical two-component Turbo code of Fig. 1.3, the standard iterative decoder is shown in a simplified format in Fig. 1.4. The *a priori* information is usually reset to be equiprobable before starting to decode a frame (unless we know that the information bits have a particular probability pattern). The decoder blocks usually use a modified Bahl-Cocke-Jelinek-Raviv (BCJR) algorithm or a Soft-Output Viterbi Algo-

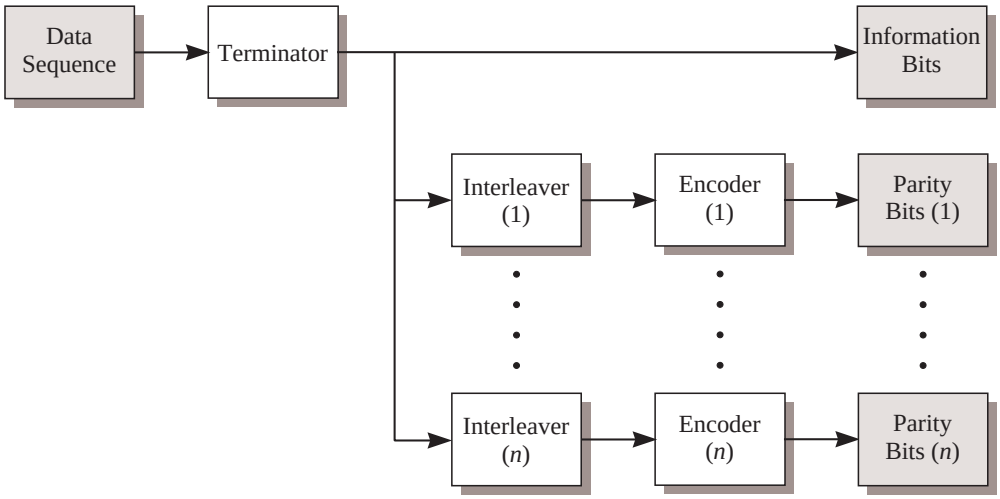


Figure 1.2: Generalised Turbo Encoder

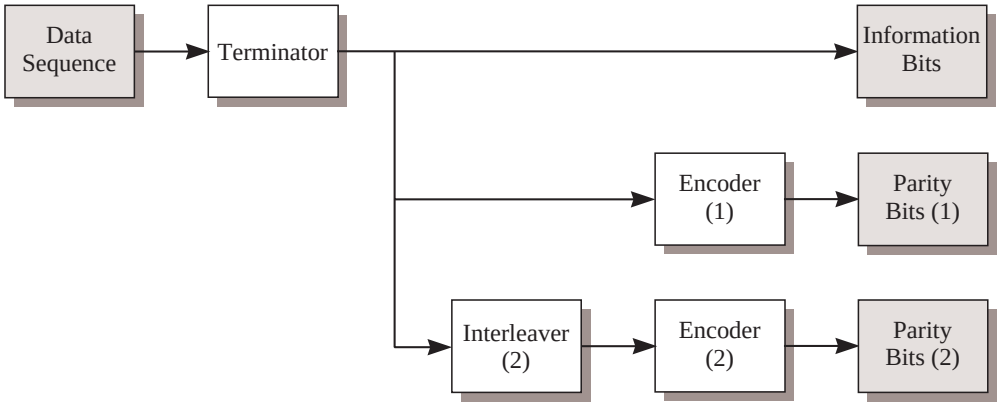


Figure 1.3: Two-Component Turbo Encoder

rithm (SOVA). If the Turbo code consists of more components, it is just a matter of inserting the relevant de-interleaver $\rightarrow$ decoder $\rightarrow$ interleaver blocks. Note that the final decoder in the chain also has a hard-decision block associated with it to output the decoded bits. Successive iterations will use the extrinsic information from this iteration as *a priori* information. Extrinsic information is that information generated by the decoders of this iteration, and is computed by removing the *a priori* information and direct channel information from the iteration's *a posteriori* information. It is essential that only extrinsic information is passed between decoders and between iterations for correct operation of the Turbo decoder.

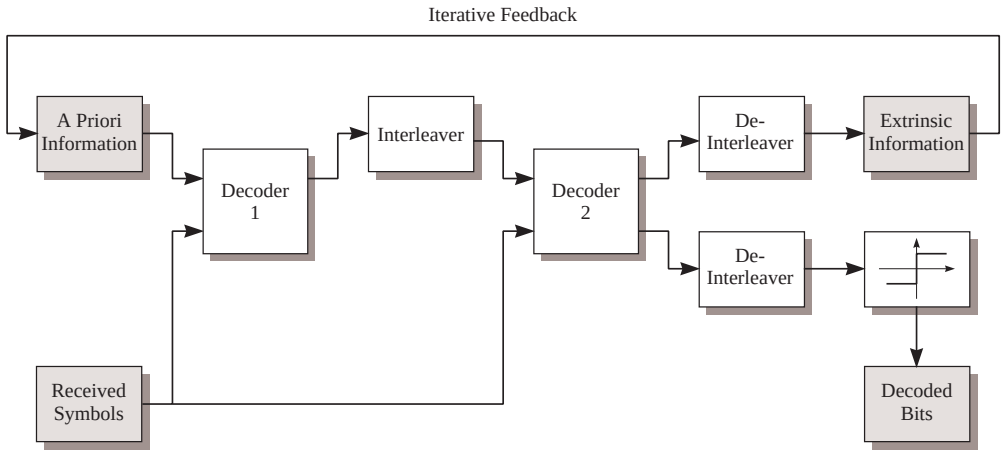


Figure 1.4: Two-Component Turbo Decoder

1.2 Aims and Objectives

The first part of this thesis deals with the software implementation of a flexible Turbo codec and its supporting simulation system. The main objectives of the implementation are speed and flexibility. Speed is a natural requirement since the system is intended for use in a research environment, where several possible schemes will need to be simulated and their results compared in the shortest time possible. The requirement for flexibility is also derived from the research aspect; Turbo coding is a relatively complex scheme, in the sense that there are several parameters that can be varied, and there are also several alternative techniques that can be used at various stages. In order to be able to compare our suggestions with those already proposed in the literature, we must be able to replicate their techniques.

In the second part of this thesis, we investigate the importance of the interleaver in the design of Turbo codes. Various Turbo code parameters related to the interleaver (such as trellis termination, puncturing, and the choice of component codes) are also considered.

1.3 Thesis Outline

Chapter 1 introduces the concept of Turbo coding, giving a brief description of the encoder and decoder. This is followed by the thesis aims and objectives, a thesis outline, and a summary of original contributions.

In Chapter 2 we start by describing in greater detail the operation of a Turbo decoder. Then, we give a review of the development of Turbo coding in the literature, emphasising the use and design of Turbo coding for the Additive White Gaussian Noise (AWGN) channel with Binary Phase Shift Keying (BPSK) modulation.

Chapter 3 opens with a description of the major techniques and algorithms used in our implementation of the Turbo codec and simulation environment. Everything is considered from the mathematical/system viewpoint, with the practical implementation details given in Chapter 4.

The first part ends with Chapter 5, which details the tests done on the system to ensure correct operation and dependability on the results obtained.

The second part opens with Chapter 6, which introduces the interleaver design aspect in the construction of Turbo codes. In this chapter we also consider the performance of Turbo codes and how this is affected by the interleaver design. This leads to Chapter 7, where we summarise the various interleaver design techniques proposed in the literature. Finally, an optimised interleaver design technique based on simulated annealing is proposed.

We next consider in Chapter 8 the interleaver design problem for large block sizes, where the effect of trellis termination is less marked. This is done by comparing the performance of various interleavers with a similar block size; novel interleaver schemes are also used, from which we gather some further insight into the problem.

The design of interleavers for small block sizes is treated in Chapter 9. The effect of termination is also considered, and a comparison of different termination schemes is given.

Finally, in Chapter 10, we summarise our comments and conclusions and suggest some interesting directions for future work.

1.4 Summary of Contributions

This thesis has made some important contributions to the field of Turbo coding. The application of the new ideas proposed in this thesis should enable the transmission of

information at even lower signal to noise ratios than before. We summarise below the original contributions in this dissertation:

- A fair comparison of different tailing schemes for Turbo codes.
- An interleaver design technique based on Simulated Annealing; this is adapted both for tailed and untailed sequences.

Chapter 2

Iterative (Turbo) Decoding

2.1 Introduction

In this chapter we will start by describing in greater detail the operation of a Turbo decoder. Then, we will give a review of the development of Turbo coding in the literature. Emphasis will be made on the use and design of Turbo coding for the Additive White Gaussian Noise (AWGN) channel with Binary Phase Shift Keying (BPSK) modulation.

2.2 Turbo Decoder Operation

2.2.1 Soft Decision Decoding

It is an established result from information theory that soft decision decoding on an AWGN channel gives a better performance for channel coding systems [Hagenauer, 1994]. For this reason, Massey [1984] emphasised that the demodulator should make no decision on the received symbols, but only deliver their relative likelihoods. It follows that in concatenated coding systems, it would be useful if the outer decoder could also provide soft outputs (i.e. the relative likelihood of correct decoding), in order to improve the performance of the inner decoder. However, with many concatenated schemes, this is not simple, because soft decision decoding is more easily applied to convolutional coding systems (due to the simplicity of implementing a soft-decision Viterbi algorithm). For example, soft-decision decoders for powerful block codes like

Reed-Solomon codes are not readily available. An added complication is that the outer decoder needs to provide an estimate to its own reliability in decoding each bit.

Suitable algorithms that provide soft outputs include the BCJR algorithm proposed by Bahl *et al.* [1974], and the SOVA decoder by Hagenauer and Hoeher [1989]. Due to the complexity of these algorithms and the lack of a performance advantage sufficient to justify such an increase in complexity, these algorithms have remained in the shadows for a relatively long time.

The introduction of Turbo codes, however, took concatenation one step further. Now, even the inner[1] decoder passes information to the outer decoder, due to the iterative nature of the Turbo decoding algorithm. Thus, the use of soft metrics in *both* component decoders becomes worthwhile.

2.2.2 Likelihood Ratios

The BCJR algorithm is optimal in the sense that it maximises the bit-error performance by computing the *a posteriori* probability of each bit based on the whole received sequence. In order to simplify the algorithm (without loss of accuracy) for binary codes, the likelihood ratio of each bit is usually computed, rather than the separate probabilities of the bit being zero or one. Furthermore, to reduce the range of values returned, the natural logarithm of the likelihood ratio is usually taken. This metric, called the Logarithm of Likelihood Ratios (LLR) is mathematically represented by:

$$\mathrm{L}(x_t) = \log\left(\frac{Pr[x_t = 1|\mathbf{Y}_0^{\tau-1}]}{Pr[x_t = 0|\mathbf{Y}_0^{\tau-1}]}\right) \tag{2.1}$$

where x_t is the source symbol transmitted at time t and $\mathbf{Y}_0^{\tau-1}$ is the received message (containing τ symbols). Starting with this definition of the final result required at the output of each decoder block, the BCJR algorithm can be modified to reduce its computational complexity [Pietrobon, 1995]. This simplifies its implementation in hardware and consequently improves the decoding speed.

[1] Strictly, Turbo codes do not have an inner and outer code, because the concatenation used is parallel rather than serial. We use this notation, however, to denote the order of decoding in the iterative scheme. Thus, the outer code is the one that is decoded first, while the inner code is decoded second.

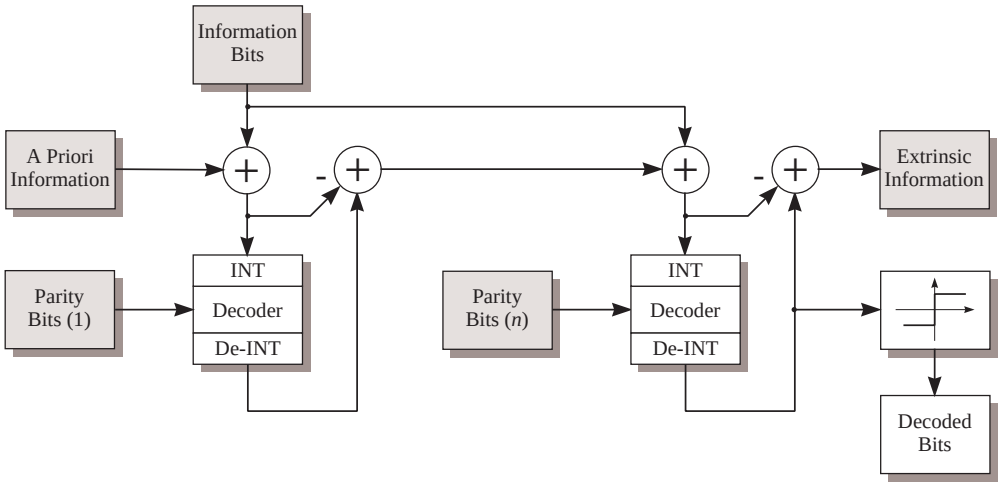


Figure 2.1: Two-Component Turbo Decoder (Detailed)

In our implementation, however, we work directly with the probability values rather than LLR metrics. The reasons behind this decision are:

- Working with probabilities gives us greater flexibility; for example, we can work with non-binary components, and we have a greater range of possible operations on the metrics used.
- The use of probabilities throughout the algorithm helps us better visualise what is happening.
- The performance penalty is not prohibitive. Typically, the penalty is in the order of two times.

2.2.3 Turbo Decoder Structure

For the two-component Turbo code shown in Fig. 1.3, the operation of the Turbo decoder is shown in Fig. 2.1. Note that the gray blocks do not represent the actual bits mentioned, but rather their probabilities as derived from the channel (for the information and parity bits) or from previous decoder stages (for the *a priori* and *a posteriori* information). Only the decoded bits block actually represents the said bits. The summing junctions perform a multiplication operation on the probabilities; conversely, a subtracting junction performs division.

The interleaver (INT) and de-interleaver (De-INT) for each component code are shown in the figure grouped with the corresponding decoder. Note how the output of the decoder must be subtracted from its input before passing it to the next stage decoder. This operation effectively computes the *extrinsic* information associated with each information bit, which will constitute the *a priori* information for the next stage. The final hard-decision decoding is performed by selecting the most probable information bit at every position within the frame.

2.2.4 Turbo Code Performance

The typical BER performance[2] to be expected of a Turbo code in the AWGN channel after each iteration is given in Fig. 2.2. One may note that the characteristic consists of three main regions:

- At high SNR, the performance is dominated by the free distance of the Turbo code. In this region, the iterative decoder converges rapidly (after 2–3 iterations), and the performance is very close to that predicted by the Union bound. Also, the performance in this region usually improves only slowly with increasing SNR. It can be shown that this *error floor* depends heavily on the interleaver used (The JPL team show that performance in the error floor region is directly related to the first few terms of the weight distribution [Divsalar *et al.*, 1995] and that the weight spectrum depends on the chosen interleaver [Dolinar & Divsalar, 1995]).
- At low to moderate SNR, there is a sharp drop in BER. In this region, a relatively large number of iterations are required for convergence.
- At very low SNR, increasing iterations do provide a gain in performance; however, the overall code performance is poor, and the BER is certainly beyond the normal operating region for most communication systems.

[2] In this case, 'typical' refers to the shape of the performance curves rather than a quantification of the BER expected at any particular SNR. Naturally, that depends on the size and parameters of the actual code used.

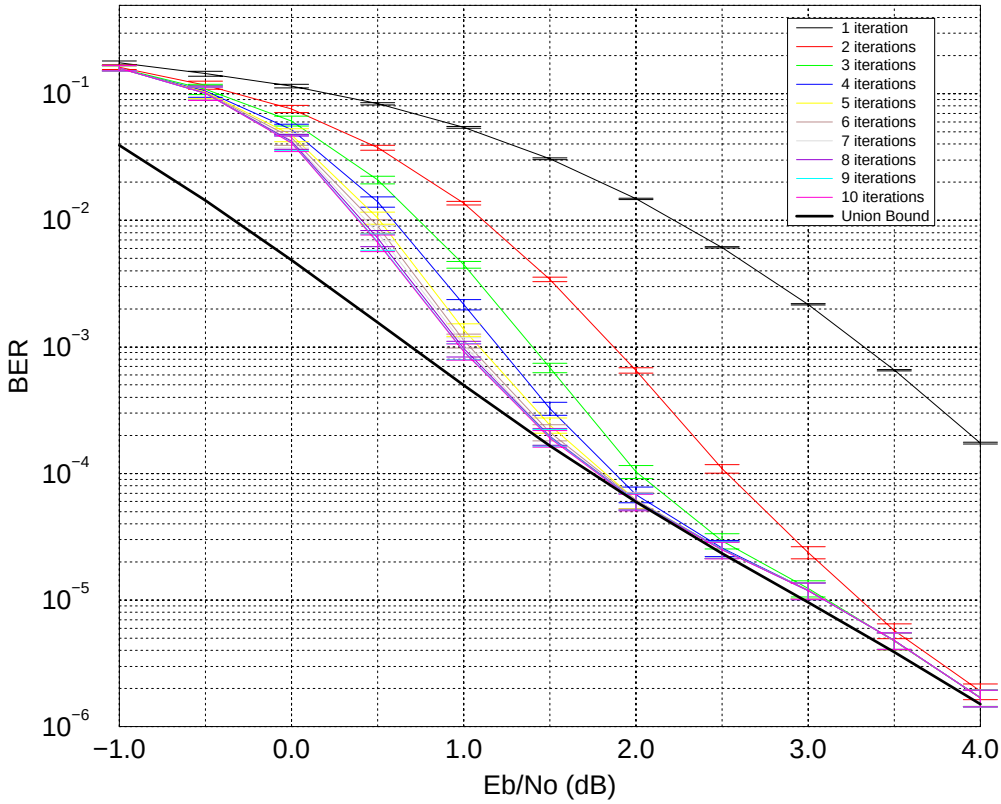


Figure 2.2: Error Performance of a Typical Turbo Code

2.3 Literature Review

2.3.1 Presentation

Turbo codes were originally presented by Berrou *et al.* [1993], claiming that a BER of 10^{-5} is achievable at 0.7 dB for a rate $\frac{1}{2}$ code, which is only 0.7 dB away from the Shannon limit. This paper also introduced Recursive Systematic Convolutional (RSC) codes as part of the Turbo coding system. The authors state that while Non-Systematic Convolutional (NSC) codes perform better than conventional Systematic codes at high SNR, the opposite is usually true at low SNR. Furthermore, they also state that for code rates higher than $\frac{2}{3}$, RSC codes can be better than NSC codes at any SNR. The BCJR algorithm is used because it can supply the *A Posteriori* Probability (APP) for every decoded bit, which is required as an input to the next stage decoder. The BCJR algorithm is modified to cater for RSC codes and to supply the Logarithm of Likelihood Ratios (LLR) as a soft output. Finally, the soft output of the decoder is shown to contain

the sum of the soft input and the extrinsic information[3]; it is this extrinsic information which is fed back for the next iteration. The interleaver used is described as a square matrix, with bits written row by row and read pseudo-randomly. The exact reading pattern, however, is not detailed. This omission was rectified in [Berrou & Glavieux, 1996].

2.3.2 Initial Reception

The quoted performance for Turbo codes was so good that initially, the coding community was skeptical about the results. Initial papers attempted to fill in the missing details (as in [Hagenauer *et al.*, 1994], which addressed the iterative interface between decoder blocks) and to confirm the accuracy of the performance claims (see [Robertson, 1994a] and [Divsalar & Pollara, 1995c]).

In their first paper on Turbo codes, Divsalar and Pollara [1995c], leading the coding research team at NASA's Jet Propulsion Lab (JPL), described a method for terminating the trellis of a RSC code, which was missing in [Berrou *et al.*, 1993]. They also paved the way for the analysis of Turbo codes by considering a small $(80, 16)$ code and computing its weight spectrum for different interleaver patterns. In this way, they concluded that random permutation interleavers result in better weight distribution than structured interleavers. For the same code, the JPL team also compared the iterative decoder with a maximum-likelihood (ML) decoder; as expected, the performance of the Turbo decoder was found to be slightly sub-optimum. In this analysis, however, one should keep in mind that the difference between the Turbo decoder and the ML decoder only applies for this particular code (and interleaver). It is always possible that certain types of code or interleaver structures either improve or degrade the performance of the Turbo decoder. In conclusion, the JPL team compared their rate $\frac{1}{4}$ Turbo codes with the huge Viterbi decoder's performance for a $(\kappa = 15, R = \frac{1}{4})$ convolutional code[4], as used in the Galileo mission. Even at a BER of 5×10^{-3}, the best Turbo code results in a gain of $0.7\,\mathrm{dB}$ relative to the convolutional code. It is also in this paper (and simultaneously by

[3] See Section 1.1.2.

[4] κ is the constraint length of the convolutional code, defined as $(\nu+1)$, where ν is the encoder memory order. R is the code rate.

Robertson [1994b], just two months earlier) that the problem of performance flattening at high SNR was first mentioned.

2.3.3 Analysis of Performance and Design Strategy

The team at JPL continued their research by considering multiple Turbo codes (i.e. Turbo codes with more than two component codes) and their associated decoding techniques and performance analysis [Divsalar & Pollara, 1995a]. Using arguments based on the weight distribution of the Turbo code, Divsalar and Pollara indicate what criteria should be used in the selection of component RSC codes and in the choice of interleaver. Similarly, and assuming a random permutation interleaver, they conclude that weight-2 data sequences are an important factor in the choice of component codes. They also conclude that higher weight sequences have successively decreasing importance, and that increasing the number of component codes has a similar effect in reducing the importance of low-weight sequences. This paper also introduces the semi-random (or S-random) interleaver, which guarantees an interleaver spread[5] S in an otherwise randomly chosen interleaver. Finally, a decoding algorithm appropriate for the parallel decoding of multiple Turbo codes was proposed; at the same code rate ($\frac{1}{4}$) and block size (4096), three-code Turbo codes with $\kappa = 4$ component codes were found to perform better than the Turbo code using multi-rate $\kappa = 5$ codes proposed in [Divsalar & Pollara, 1995c].

The analysis described by Divsalar and Pollara [1995a] is based upon some work done in conjunction with Sam Dolinar (also at JPL) on the weight distributions of Turbo codes. This was subsequently published by Dolinar and Divsalar [1995].

2.3.4 Performance Bounds

The analytical problem of bounding the performance of Turbo codes was first tackled by Benedetto and Montorsi [1995a; 1995b]. This was done by considering the Turbo code as a block code, and using its weight enumerating functions (WEF) to compute a Union Bound, averaging over all possible interleavers. This technique of assuming

[5] See Section 3.3.3.

that all possible permutation interleavers are equally possible, and averaging the performance results over the whole set, is usually termed the *uniform interleaver*. The bit-error and word-error probabilities obtained in this way also assume that the decoding strategy used is ML.

The JPL team also issued a paper on the same subject [Divsalar *et al.*, 1995], in which they focused on the computational properties of the bound and on its comparison with simulated results. Above the computational cutoff rate R_0, the bounds were found to lie close to the simulated performance of Turbo codes; thus, the bounds can be used as an estimate for performance at the error-floor region.

Meanwhile, Benedetto and Montorsi [1996b] were preparing a more complete analysis of the performance of Parallel Concatenated Block and Convolutional Codes (PCBC and PCCC respectively), again based on performance bounds with a uniform interleaver. In this paper, Benedetto and Montorsi argue that the choice of interleaver is not critical for medium-high BER, but a more careful choice pays off for lower BER values because of the resultant code's free distance. They also state that the simulated performance of the iterative decoding algorithm converges with the ML bounds.

Perez *et al.* [1996] present a performance evaluation of Turbo codes based on the code's distance spectrum. Particularly, it is shown that the error floor is a consequence of the relatively low free distance of Turbo codes, and that it can be lowered by increasing the block size. Alternatively, by increasing the free distance (using component codes with primitive feedback polynomials), the slope of the error floor is increased. The excellent performance of Turbo codes at low SNR is shown to be a consequence of a process called 'spectral thinning', which results in the free distance asymptote being the dominant factor affecting performance for low SNR.

2.3.5 Design Technique

The first approach to a more complete design of Turbo codes was published by the JPL team [Divsalar & Pollara, 1995b]. This is primarily an extension of published results, including the best choice of component codes. Rate $\frac{1}{n}$ codes were already considered by Benedetto and Montorsi [1996a]. This paper tackles the choice of rate $\frac{b}{b+1}$ codes,

and uses the same technique to handle punctured rate $\frac{1}{2}$ codes (using odd-even puncturing). It is also shown that for rate $\frac{1}{2}$ codes with odd-even puncturing, the minimum-weight codewords corresponding to a weight-2 input sequence cannot reach the upper bound that is possible for rate $\frac{2}{3}$ codes. Also, this bound cannot be achieved if the feedback polynomial (for any code rate) is non-primitive. This agrees with the suggestion given by Benedetto and Montorsi [1996a] to use codes with primitive feedback polynomials. Simulation results for various Turbo code types are also given.

2.3.6 Trellis Termination

When Turbo codes were presented by Berrou *et al.* [1993], the issue of trellis termination was not considered. The trellises for the interleaved and non-interleaved sequences were not terminated, and the decoder worked without any prior knowledge of the final state. While this may not be considered to be a major problem with the large (65536-bit) frame sizes used, the effect of trellis termination on smaller block sizes as used for instance with speech cannot be ignored [Jung & Naßhan, 1994b].

A termination scheme was suggested by Jung and Naßhan [1994b], where only the second encoder is terminated – it is stated that this scheme gives better error performance than terminating the first encoder. The intuitive argument given as reason for this improvement is that terminating the second sequence results in better extrinsic information as feedback for subsequent iterations.

In [Robertson, 1994b], only the first decoder is terminated. However, the second decoder works with some knowledge of the final state, since the backward recursion equation is initialised with final state information given by the forward recursion. This technique suggested the use of no tailing for either encoder, with the final state being estimated within the decoder from the forward recursion equation [Reed & Pietrobon, 1996]. We do not agree with the use of this technique because this places a bias on the results of the forward recursion equation – the BCJR algorithm [Bahl *et al.*, 1974] clearly states that the initial values for the forward and backward recursion should depend only on *a priori* information. The correct technique to be used when the final state is unknown is to initialise the backward recursion with equal probability for all states;

this technique was also proposed by Reed and Pietrobon [1996] as a novel termination scheme, where it is stated that this gives improved performance.

Divsalar and Pollara [1995c] propose an alternative technique where both sequences are terminated (with different tails), and only the tail bits for the non-interleaved sequence are transmitted. The decoder for the interleaved sequence acts without prior knowledge of the tail bits. We assume, though, that the parity bits in the tail region are transmitted for both encoders.

The problem of terminating the parallel encoders in a Turbo code is that the interleaver causes the two trellises to finish in a different state, and hence different tail sequences are required. Barbulescu and Pietrobon [1995] propose a restriction on the interleaver pattern which forces the two trellises to end in the same state. This allows the use of a single tail sequence for both encoders. This type of interleaver is called a 'simile' interleaver.

2.3.7 Interleaver Design

While the effect of the interleaver on Turbo code performance was recognised early [Divsalar & Pollara, 1995c], the sheer size of the problem of choosing an optimum interleaver meant that a comprehensize solution remained elusive. It was already shown, however, that randomly chosen interleavers perform better than structured interleavers. The actual importance of randomisation was shown by Divsalar and Pollara [1995a] and Perez *et al.* [1996], by considering the effect of the interleaver on critical low weight input patterns. Divsalar and Pollara [1995a] also proposed the S-random interleaver, which is a randomly created interleaver with some restrictions, designed to break low-weight input patterns. Other semi-random interleaver designs include the original one used by Berrou and Glavieux [1996].

Other early attempts at interleaver design include those by Jung and Naßhan [1994a]. In this paper, interleavers are chosen from a large set of random interleavers based on the low-weight-input distance properties of the resulting Turbo code. It was stated that a regular block interleaver performs better than the chosen random interleaver for the

given block size. This contrasts with the opinion given elsewhere in the literature[6] that regular interleavers should be avoided because of their poor distance properties.

The design of an interleaver for use in a punctured Turbo code was first considered by Barbulescu and Pietrobon [1994]. The odd-even interleaver proposed has the desirable property that if the parity bit directly associated with any information bit is punctured in the non-interleaved sequence, then the corresponding parity bit in the interleaved sequence is not punctured. It is stated that this gives a uniform distribution of the correcting capability of the codes, and results in better Turbo code performance. This design was further improved when the block-helical 'simile' interleaver was proposed [Barbulescu & Pietrobon, 1995], which combines the odd-even property with the possibility of using the same tail for interleaved and non-interleaved trellises.

Recently, a mathematical model of interleavers has been proposed by Andrews *et al.* [1997], who also used this to guide the generation of new interleavers for Turbo codes by modifying the S-random algorithm [1998]. It is interesting to note that practically all papers on interleaver design are based on this algorithm, which was designed to counter the effects of weight-2 input sequences only.

A more recent interleaver design technique has been presented by Hokfelt *et al.* [1999]. Their design technique is based on the statistical properties of the extrinsic information passed between decoders rather than on the Turbo code's distance properties. Another recent proposal by Yuan *et al.* [1998; 1999] is a modification of the S-random interleaver design which also takes into account the first few terms of the distance spectrum of the component codes.

[6] See, for example, [Divsalar & Pollara, 1995a].

Chapter 3

System Design and Functionality

3.1 Introduction

In this chapter, we will describe the major techniques and algorithms used in the Turbo codec and simulation environment. Everything will be considered from the mathematical/system viewpoint – practical implementation details are given in the following chapter. The modules considered in this chapter are repectively the Iterative (Turbo) algorithm and its components, the MAP decoder, and the Monte Carlo simulator.

3.2 Component Codes

The convolutional encoders are considered to be Finite State Machines (FSM) using the Mealy model. This allows us to use both recursive and non-recursive codes with the same implementation. Note that this assumption is effectively the same one taken in the design of the modified BCJR decoder. Also, our implementation only caters for symmetric Turbo codes (i.e. Turbo codes with identical component codes). This simplifies the handling of trellis termination for Recursive Systematic Convolutional (RSC) codes.

3.2.1 Non-Recursive Convolutional Codes

Non-Recursive Convolutional (NRC) codes are described by a matrix of generator polynomials. For an (n, k) convolutional code (i.e. there are k input bits and n output bits at every time step), the generator matrix $\mathbf{G}(D)$ has k rows and n columns:

$$\mathbf{G}(D) = \begin{pmatrix} g_{1,1}(D) & g_{1,2}(D) & \cdots & g_{1,n}(D) \\ g_{2,1}(D) & g_{2,2}(D) & \cdots & g_{2,n}(D) \\ \vdots & \vdots & \ddots & \vdots \\ g_{k,1}(D) & g_{k,2}(D) & \cdots & g_{k,n}(D) \end{pmatrix} \tag{3.1}$$

where D is the delay operator, the power of D denoting the number of time units a symbol is delayed with respect to the initial symbol in the sequence. The input vector $\mathbf{u}(D)$ consists of k columns:

$$\mathbf{u}(D) = \begin{pmatrix} u_1(D) & u_2(D) & \cdots & u_k(D) \end{pmatrix} \tag{3.2}$$

and the output vector $\mathbf{v}(D)$ consists of n columns:

$$\mathbf{v}(D) = \begin{pmatrix} v_1(D) & v_2(D) & \cdots & v_n(D) \end{pmatrix} \tag{3.3}$$

The output depends on the input according to the matrix multiplication:

$$\mathbf{v}(D) = \mathbf{u}(D).\mathbf{G}(D) \tag{3.4}$$

3.2.2 Recursive Systematic Convolutional Codes

In the case of a RSC code with k inputs and n outputs, we still have k input delay registers. In contrast with NRC codes, however, the input to each register is generated from the current input and from the state of the register according to a feedback polynomial. The generator polynomial in this case is given by:

$$\mathbf{G}(D) = \begin{pmatrix} 1 & 0 & \cdots & 0 & \frac{g_{1,k+1}(D)}{g_{1,1}(D)} & \cdots & \frac{g_{1,n}(D)}{g_{1,1}(D)} \\ 0 & 1 & \cdots & 0 & \frac{g_{2,k+1}(D)}{g_{2,2}(D)} & \cdots & \frac{g_{2,n}(D)}{g_{2,2}(D)} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & \frac{g_{k,k+1}(D)}{g_{k,k}(D)} & \cdots & \frac{g_{k,n}(D)}{g_{k,k}(D)} \end{pmatrix} \tag{3.5}$$

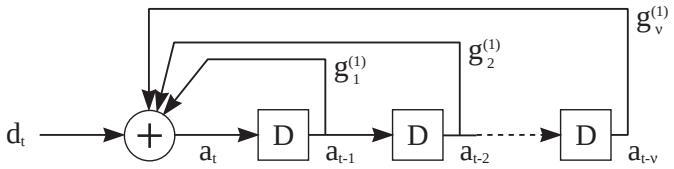


Figure 3.1: Feedback Connection of an RSC Encoder

where $g_{x,x}(D)$ is the feedback polynomial for input $x \in [1, k]$. Consider a feedback polynomial of order ν:

$$g_{x,x}(D) = \sum_{i=0}^{\nu} g_i^{(x)} D^i \tag{3.6}$$

Now consider an encoder with a single input ($k = 1$); the feedback connection, illustrated in Fig. 3.1, is given by:

$$g_{1,1}(D) = \sum_{i=0}^{\nu} g_i^{(1)} D^i \tag{3.7}$$

The bit stored in the first delay element, a_t, is given by:

$$a_t = d_t + \sum_{i=1}^{\nu} g_i^{(1)} a_{t-i} \pmod 2 \tag{3.8}$$

where d_t is the data bit passed to the encoder at time t. Rearranging the subject of the equation:

$$d_t = \sum_{i=0}^{\nu} g_i^{(1)} a_{t-i} \pmod 2 \tag{3.9}$$

where $g_0 = 1$, since the external input is always connected. Note that to terminate the encoder, a_t must be kept equal to zero for ν time steps; thus the input must be:

$$d_t = \sum_{i=1}^{\nu} g_i^{(1)} a_{t-i} \pmod 2 \tag{3.10}$$

for the values of t specifying the tail region, since $a_t = 0$.

3.3 Interleaver

The interleaver is a one-to-one mapping function that maps a sequence of τ bits into another sequence of τ bits. As used in Turbo codes, the interleaver performs this mapping by permuting the input bits:

$$\tilde{x}_t = x_{\lambda(t)}, \qquad t \in [0, \tau - 1] \tag{3.11}$$

where $\mathbf{x}_0^{\tau-1} = x_0, x_1, \ldots, x_{\tau-1}$ is the input sequence, $\tilde{\mathbf{x}}_0^{\tau-1} = \tilde{x}_0, \tilde{x}_1, \ldots, \tilde{x}_{\tau-1}$ is the interleaved sequence, and $\lambda(t)$ is the permuting function. Note that the permutation interleaver is actually a subset of the more general case when the interleaver is considered as a one-to-one mapping function. For application in Turbo coding, the interleaver / de-interleaver must be able to perform the following operations:

- Within the Turbo encoder, the interleaver should return the interleaved sequence corresponding to the given input sequence, as defined in Eqn. (3.11).
- In the decoder, the interleaver should map the probabilities associated with the input sequence to their corresponding position in the interleaved sequence. The reverse operation is also necessary, as shown in Fig. 2.1.

Our implementation also allows interleaver structures other than the permuting type usually used with Turbo codes. We also allow the use of dynamic interleavers, where the interleaver mapping function changes for successive frames. For example, a dynamic equiprobable random interleaver is a realisation of the uniform interleaver discussed in the literature[1]. This format allows the construction of any type of block interleaver; another interleaver structure, the convolutional interleaver[2], is not catered for in our implementation.

3.3.1 Trellis Termination

Terminating the trellis of a convolutional code with memory order ν requires ν tail bits. In the case of non-recursive codes, termination is trivial, since all ν bits should be zero. With recursive codes, however, the tail bits depend on the final state of the encoder prior to termination. This implies that if two data sequences cause the respective encoders to finish in different states, then the tail bits required will be different.

Our implementation handles two different possibilities for terminating the trellis of a Turbo code. The first is when the input sequence of τ bits (including the final ν tail bits) can be shown to be correctly terminated after passing through the interleaver(s). In this case, we only need to generate the tail bits once (for the input sequence). Also,

[1] See Section 2.3.4.

[2] This type of interleaver structure is usually used with stream-oriented Turbo codes.

at the decoding end, all BCJR decoders can assume that the trellis starts and ends at state zero. Guaranteeing that the interleaved sequences will also be correctly terminated depends on both the interleaver mapping and on the component codes used. It can be shown that for a RSC encoder of memory ν, the impulse response of the encoder is periodic with period $p \leq 2^{\nu} - 1$ [Barbulescu & Pietrobon, 1995]. Equality holds if the feedback polynomial is primitive, in which case the impulse response is a maximal length sequence. Now, for a permutation interleaver defined by Eqn. (3.11), the condition for the interleaver that allows both RSC encoders to end in the same state is [Blackert *et al.*, 1995]:

$$t = \lambda(t) \pmod{p} \quad \forall\, t \tag{3.12}$$

The second case handled by our implementation is the termination scheme proposed by Divsalar and Pollara [1995c], which we refer to as the JPL scheme. The first sequence is terminated, but the interleaver operates only on the initial $\tau - \nu$ data bits. Since the interleavers used by the JPL team (S-random) do not guarantee that the interleaved trellis will end in the same state, the tail bits required for the interleaved sequence may be different. The interleaved sequence is still terminated, but the tail bits used are not transmitted, and the decoder must consider all tail sequences to be equiprobable.

The final case handled by our implementation is for no termination. In this case, the data sequence does not have any tail bits appended to it, and the decoder cannot determine the final state of the trellis. Thus, the BCJR decoder is used with initial state zero and all possible states are considered equiprobable for the final state.

Another possibility, which cannot be handled directly by our implementation, is to terminate the first sequence and allow the interleaver to operate on all τ bits (including the final ν tail bits). In this case, unless the interleaver can guarantee that the interleaved trellis ends in the same state as the non-interleaved one, the final state of the interleaved trellis is indeterminate, and the BCJR decoder must be configured for an equiprobable final state.

Finally, as a special condition for terminated sequences, the interleaver can be forced to operate only on the initial $\tau - \nu$ data bits, and guarantee that the final state of the interleaved and non-interleaved trellis are the same (not necesarily zero). Thus, the

same tail can be used to terminate both sequences. An interleaver which satisfies this condition is termed *simile* [Barbulescu, 1996]. Simile interleavers are advantageous when the parity bits are punctured – for odd-even puncturing, since the tail sequence is the same, its odd parity bits are transmitted in one sequence, and the even parity bits in the other. Before decoding, these parity bits can be combined to increase the decoder's knowledge in the tail region. Effectively, a simile interleaver nullifies the puncturing effect in the tail region.

3.3.2 Odd-Even Interleavers

For an interleaver to be odd-even[3], the mapping function must satisfy

$$t = \lambda(t) \pmod{s} \quad \forall\, t \tag{3.13}$$

where s is the number of sets in the Turbo code (i.e. the number of parallel component codes). Ho *et al.* [1998] claim that such interleavers improve the performance of punctured Turbo codes; their claim is based on simulations of punctured Turbo codes with S-random interleavers designed with and without the restriction defined by Eqn. (3.13).

3.3.3 The Spreading Factor of an Interleaver

An interleaver has spreading factor (s_1, s_2) if any input bits that are less than s_1 places apart will be at least s_2 places apart at the output of the interleaver. That is, whenever $|i-j| < s_1$ then $|\lambda(i) - \lambda(j)| \geq s_2$. The interleaver is also said to have a spread S, where $S = \min(s_1, s_2)$.

3.4 Modulation and Channel Modelling

The modulator is modelled as a mapping function which takes one or more input bits[4] and associates them with a point in signal space (the modulation symbol). The average

[3] This property has also been called Mod-s, for Turbo codes with s parallel codes [Ho *et al.*, 1998].

[4] Actually, we do not restrict ourselves to the binary case, so *input symbols* would be more exact.

energy per modulation symbol (assuming all modulation symbols are equally utilised) together with the code rate R determines the average energy per bit E_b:

$$E_b = \frac{1}{R}\frac{1}{M}\sum_{i=1}^{M}|s_i|^2 \tag{3.14}$$

where s_i is the signal space representation of any one of M modulation symbols making up the modulation scheme.

The channel is modelled as an operator on points in signal space. At the demodulator, the probabilities for each information and parity bit are obtained from the channel by multiplying the probability densities for the modulation symbols associated with any particular bit. Thus, the Turbo decoder requires that each encoder parity output (having $n-k$ bits for an (n,k,ν) encoder) and each encoder input (k bits) should be represented by an integral number of modulation symbols. This is always the case with BPSK (where one modulation symbol encodes one bit), but is not necessarily so with other modulation schemes.

3.4.1 Additive White Gaussian Noise

The Additive White Gaussian Noise (AWGN) channel is represented as two independent and identically distributed random sequences, one for the in-phase component and one for the quadrature component of the noise signal. This noise signal is added geometrically to the information signal in signal space. The random sequences have a Gaussian profile with zero mean and a variance which depends on the unilateral noise power spectral density N_0.

Consider a random variable (RV) n_i representing the AWGN component in a communication system. We can say that:

$$n_i = n_{i_i} + j.n_{q_i} \tag{3.15}$$

where $j = \sqrt{-1}$, n_{i_i} and n_{q_i} are the in-phase and quadrature components of the noise. Note that both n_{i_i} and n_{q_i} are independent random variables with zero mean and vari-

ance σ^2. Now, the average noise energy per modulation symbol is given by:

$$\begin{aligned} N_0 &= \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} |n_i|^2 && (3.16)\\ &= \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} |n_{i_i} + jn_{q_i}|^2 && (3.17)\\ &= \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} (n_{i_i}^2 + n_{q_i}^2) && (3.18)\\ &= \lim_{N\to\infty} 2\sigma^2 && (3.19)\\ &= 2\sigma^2 && (3.20) \end{aligned}$$

Thus, the required standard deviation of the gaussian random sequences representing the in-phase and quadrature noise components is given by:

$$\sigma = \sqrt{\frac{N_0}{2}} \tag{3.21}$$

3.5 Puncturing

For maximum flexibility, puncturing is applied at signal space level. The modulator's output is a vector of points in signal space representing the information and parity bits for a frame, in the correct temporal order for transmission through the channel. The puncturing module then includes some of these points for transmission and skips those that should be punctured. At the receiving end, the probabilities for bits associated with punctured modulation symbols are set to be equiprobable.

3.6 MAP Decoding Algorithm

3.6.1 Introduction

The object of a single decoding stage is to determine the *a posteriori* probabilities of the information bits, given their *a priori* probabilities and the transition probabilities from each possible transmitted symbol to the received channel symbols. To maximise performance, the decision on any bit is based on the whole received sequence. The transmitted sequence is generated by passing the source bits through a convolutional

encoder. Usually, the encoder is started in the all-zero state, and the sequence is terminated, ensuring that the encoder also finishes in the all-zero state.

The algorithm presented by Bahl, Cocke, Jelinek and Raviv [1974] considers the general case of a Markov source transmitted through a discrete memoryless channel (DMC), and then applies this to the decoding of linear codes. Because of the generality of the algorithm, it is possible to decode sequences with *a priori* probabilities on each information bit. The algorithm also computes the *a posteriori* probabilities of the states and transitions; in our case, we only require the transition probabilities for decoding convolutional codes. This algorithm, called BCJR after the original authors, forms the basis for MAP decoding in Turbo codes.

We will start by defining the algorithm's inputs and outputs, and then give the algorithm equations. Note that to simplify the implementation, we have redefined the index values for a number of vectors and matrices to start from zero.

3.6.2 Definitions

Definition 3.6.1 (Source Message) *The information that is sent by the source is represented by the vector* $\mathbf{x}_0^{\tau-\nu-1}$. *A further* ν *tail symbols* $\mathbf{x}_{\tau-\nu}^{\tau-1}$ *augment the information symbols, ensuring that the final state of the encoder is zero. Each vector element* x_t *is one of* K *possible input symbols. Setting* $\nu = 0$ *we obtain an unterminated sequence, and the final state of the encoder is unknown. The source message is defined to be* $\mathbf{x}_0^{\tau-1}$.

Definition 3.6.2 (Encoder State) S_t *represents the state of the encoder at time* t*, that is, prior to the encoding of information symbol* x_t. *Thus, for an encoder which can be modelled as a finite state machine, the state* S_t *and the input* x_t *are the necessary and sufficient conditions to determine the next state* S_{t+1}. *Note that for a terminated sequence, the initial state* S_0 *and the final state* S_τ *are both zero. There are in all* M *valid states.*

Definition 3.6.3 (Encoded Message) *The encoder maps the source message* $\mathbf{x}_0^{\tau-1}$ *into the encoded message* $\mathbf{X}_0^{\tau-1}$. *Note that each vector element* X_t *of* $\mathbf{X}_0^{\tau-1}$ *can take any one of* N *values, where* $N \geq K$.

Definition 3.6.4 (Received Message) *The encoded message is modulated and transmitted through a noisy channel. At the receiving end, a message* $\mathbf{Y}_0^{\tau-1}$ *is captured, where each vector element* Y_t *is the corrupted modulation symbol (a point in signal space) corresponding to the encoded symbol* X_t*.*

Definition 3.6.5 (*A Priori* Probabilities) *Previous iterations will indicate, through the* a priori *probabilites, which input sequence is more likely.* $\pi_t(x)$ *is the* a priori *probability that* $x_t = x$*, where* x *is one of the* K *input symbols.*

Definition 3.6.6 (*A Posteriori* Probabilities) *The output of a single stage of the decoder is the set of* a posteriori *probabilities of the input sequence,* $\Pi_0^{\tau-1}(x)$*, where* $\Pi_t(x)$ *is the probability that* $x_t = x$*, given that we received the message* $\mathbf{Y}_0^{\tau-1}$*.*

3.6.3 The BCJR Algorithm

The general algorithm takes as input the *a priori* transition probabilities, output probabilities, and the channel crossover probabilities, respectively defined by:

$$p_t(m|m') = Pr[S_{t+1} = m|S_t = m'] \tag{3.22}$$

$$q_t(X|m', m) = Pr[X_t = X|S_{t-1} = m'; S_t = m] \tag{3.23}$$

$$R(Y_t|X) = Pr[\text{having transmitted } X \text{ if } Y_t \text{ was received}] \tag{3.24}$$

In the case of decoding a convolutional code, the probability of having transmitted any particular input at time t is obtained by summing:

$$Pr[S_t = m', S_{t+1} = m] = \frac{\sigma_t(m', m)}{P_y} \tag{3.25}$$

over all trellis paths between time t and $t+1$ which are associated with this input. The

terms on the right-hand side of the equation are defined by:

$$\begin{aligned} P_y &= Pr[Y_0^{\tau-1}] \\ &= \sum_m \lambda_\tau(m) && (3.26) \\ \sigma_t(m',m) &= Pr[S_{t-1}=m'; S_t=m; Y_0^{\tau-1}] \\ &= \alpha_{t-1}(m')\gamma_{t-1}(m',m)\beta_t(m) && (3.27) \\ \lambda_t(m) &= Pr[S_t=m; Y_0^{\tau-1}] \\ &= \alpha_t(m)\beta_t(m) && (3.28) \\ \alpha_t(m) &= Pr[S_t=m; Y_0^{t-1}] \\ &= \sum_{m'} \left[\alpha_{t-1}(m')\gamma_{t-1}(m',m)\right] && (3.29) \\ \beta_t(m) &= Pr[Y_t^{\tau-1}|S_t=m] \\ &= \sum_{m'} \left[\beta_{t+1}(m')\gamma_t(m,m')\right] && (3.30) \\ \gamma_t(m',m) &= Pr[S_{t+1}=m; Y_t|S_t=m'] \\ &= \sum_X \left[p_t(m|m')q_t(X|m',m)R(Y_t|X)\right] && (3.31) \end{aligned}$$

We restrict ourselves to the case where there is only a single output associated with any state transition. Thus, the summation for $\gamma_t(m',m)$ collapses, because for a given state transition (m',m) there is only one possible X, in which case $q_t(X|m',m)=1$. Thus, the equation for $\gamma_t(m',m)$ may be rewritten as:

$$\gamma_t(m',m) = p_t(m|m')R(Y_t|X) \tag{3.32}$$

where X is a function of (m',m). Boundary conditions for $\alpha_t(m)$ and $\beta_t(m)$ are given by:

$$\alpha_0(m) = Pr[S_0=m] \tag{3.33}$$

$$\beta_\tau(m) = Pr[S_\tau=m] \tag{3.34}$$

Usually, the convolutional encoder is started in state zero, thus:

$$\alpha_0(m) = \begin{cases} 1 & m=0, \\ 0 & \text{otherwise.} \end{cases} \tag{3.35}$$

Similarly, when the input sequence is correctly terminated, the encoder ends in state zero, such that:

$$\beta_\tau(m) = \begin{cases} 1 & m = 0, \\ 0 & \text{otherwise.} \end{cases} \tag{3.36}$$

Note that the functions for $\alpha_t(m)$ and $\beta_t(m)$ are recursive. Thus, at least one of these must be evaluated and stored before any result can be computed. However, in the interest of performance, it is better if this is done to both function and also to the $\gamma_t(m', m)$ function.

Considering the $\gamma_t(m', m)$ matrix, we observe that $\gamma_t(m', m)$ only has a non-zero value for the connected set of (m', m). This is given by the set of possible inputs x for every state m'. We redefine the $\gamma_t(m', m)$ matrix to take advantage of this as follows:

$$\gamma_t(m', x) = p_t(m|m')R(Y_t|X) \tag{3.37}$$

where m and X are functions of (m', x). The algorithm may be simplified by redefining its inputs as follows:

$$p_t(m|m') = \pi_t(x) \tag{3.38}$$

$$R_t(X) = R(Y_t|X) \tag{3.39}$$

where x is the input associated with the transition from state m' to state m. Redefining the $\gamma_t(m', m)$ function in terms of (m', x):

$$\gamma_t(m', x) = \pi_t(x)R_t(X) \tag{3.40}$$

Arranging the remaining equations,

$$\alpha_t(m) = \sum_{m'} \left[\alpha_{t-1}(m')\gamma_{t-1}(m', x)\right] \tag{3.41}$$

where x is a function of (m', m) and the summation is over the connected set of (m', m). Similarly,

$$\beta_t(m) = \sum_{m'} \left[\beta_{t+1}(m')\gamma_t(m, x)\right] \tag{3.42}$$

where x is a function of (m', m) and the summation is over the connected set of (m', m). The $\sigma_t(m', m)$ metric is also redefined to contain only the connected set of (m', m),

$$\sigma_t(m', x) = \alpha_{t-1}(m')\gamma_{t-1}(m', x)\beta_t(m) \tag{3.43}$$

where m is a function of (m', x). Finally,

$$\lambda_t(m) = \alpha_t(m)\beta_t(m) \tag{3.44}$$

$$P_y = \sum_m \lambda_\tau(m) \tag{3.45}$$

The end result of the MAP decoder is given by:

$$\Pi_t(x) = \sum_m \frac{\sigma_t(m, x)}{P_y} \tag{3.46}$$

3.7 Monte Carlo Simulator

Simulations are performed using the Monte Carlo technique with a uniform sampling of the search space [Press *et al.*, 1992; Jeruchim *et al.*, 1992]. Thus, assuming that the results of the experiments have a Gaussian distribution[5], the estimate[6] is given by:

$$\varepsilon = \hat{\mu} \pm c\hat{\sigma} \tag{3.47}$$

where $\hat{\mu}$ and $\hat{\sigma}$ are respectively the mean and variance of the population of results, and are given by:

$$\hat{\mu} = \mu \tag{3.48}$$

$$\hat{\sigma} = \frac{\sigma}{\sqrt{N}} \tag{3.49}$$

μ and σ are respectively the mean and variance of the samples of results, and η is the number of samples, and are respectively given by:

$$\mu = \frac{1}{\eta}\sum_{i=1}^{\eta} x_i \tag{3.50}$$

$$\sigma^2 = \frac{1}{\eta - 1}\sum_{i=1}^{\eta}(x_i - \mu)^2 \tag{3.51}$$

$$= \frac{1}{\eta - 1}\left\{\sum_{i=1}^{\eta} x_i^2 - \eta\mu^2\right\} \tag{3.52}$$

In Eqn. (3.47), c is a factor that depends on the required confidence level of the estimate. For a confidence level of χ (expressed as a fraction), c is governed by the equation:

$$Q(c) = \frac{1}{2}(1 - \chi) \tag{3.53}$$

[5]This is usually justified, particularly when a large number of experiments is performed.

[6]The sought result, which is the output of the Monte Carlo simulation

and $Q(x)$ is the well-known complementary error function:

$$Q(x) = \frac{1}{\sqrt{2\pi}} \int_x^\infty e^{-\frac{t^2}{2}} \mathrm{d}t \tag{3.54}$$

$$= \frac{1}{2} \mathrm{erfc}\left(\frac{x}{\sqrt{2}}\right) \tag{3.55}$$

Thus, for a tolerance $\pm\rho$, expressed as a fraction of the mean at a fractional confidence χ, the Monte Carlo simulation will stop when:

$$\rho \geq c\frac{\hat{\sigma}}{\hat{\mu}} \tag{3.56}$$

$$\geq Q^{-1}\left(\frac{1}{2}(1-\chi)\right)\frac{\hat{\sigma}}{\hat{\mu}} \tag{3.57}$$

The inverse of $Q(x)$ cannot be determined algebraically, so a numerical solution has to be used. The secant method [Press *et al.*, 1992] is a suitable technique, since the required result is already bounded (x must be in $[0, 1]$), and the function itself is continuous.

Chapter 4

Implementation Details

4.1 Introduction

The practical issues concerning the implementation of the Turbo codec and simulation environment are discussed in this chapter. The major topics include the Monte Carlo simulator, designed to allow multi-result experiments and to make full use of multi-processor facilities, and the BCJR algorithm, with the associated numerical accuracy problems. Other topics included in this chapter involve the provision of low-level facilities. Although the discussions in this chapter are tied closer to the hardware and software used, descriptions are kept at algorithm level.

4.1.1 Choice of Language

For flexibility, the same simulator is required to work with different communication system designs. In particular, it should allow the use of different channel (noise) models, modulation schemes, information sources, source codecs, and channel coding schemes. At its simplest (since we are primarily concerned with the analysis of the channel code), the system will use an equiprobable binary source and a direct-mapping source coder, BPSK modulation, and transmission over an AWGN channel. However, every part needs to be replaceable. All this lends itself naturally to an Object-Oriented Programming (OOP) model.

The second major requirement, speed, suggests the use of an efficient compiled language. Also, for a further improvement in performance, parallel execution can be used at various levels. Together with the OOP suggestion, these led to the choice of the C++ programming language [Stroustrup, 1991]. For parallel programming, the Message-Passing Interface (MPI) model [MPI, 1995] was chosen for its simplicity and also for the possibility of executing in parallel over a cluster of workstations. This is particularly attractive for large-grain parallelism, where overheads are smaller.

4.2 Monte Carlo Simulator

Our simulator implementation differs from the basic system described in Section 3.7 in the following respects:

- Experiments are allowed to return multiple results; for example, simulating a communication system based on Turbo codes will return the BER and Frame Error Rate (FER) at every iteration. The Monte Carlo simulator stops when *all* results are within the required tolerance limits.
- Parallelism at the simulator level is large-grain, and thus represents an opportunity for high efficiency utilisation of resources. Also, such parallelism is not communication intensive, facilitating the use of parallel systems with low inter-processor bandwidth, such as a multi-computer system.

4.2.1 Multi-Processing

For uniform sampling, all samples are independent. Thus, it is simple to compute different samples on separate processing nodes. Independence of the results from different nodes is ensured by seeding the random generators differently for each node. In our implementation, we have used the Local Area Multicomputer (LAM) library from Ohio Supercomputer Centre [LAM, 1996].

```
begin
    Initialise results
    Send a work command to all slaves
    repeat
        Wait for any slave to return a result
        Update estimate with slave's result
        Compute accuracy reached
        if required accuracy not reached yet
            Send a work command to slave that returned result
        end if
    until required accuracy reached
    Wait for results from remaining slaves
    Update estimate with slaves' results
end
```

Figure 4.1: Root Process Algorithm

4.2.2 Dynamic Load Balancing

The parallel system is implemented using a Master-Slave model. The master (or root) process distributes work among the slave processes, gathers the results, and computes the estimate. The slaves are used to compute a sample and return its result; since all samples are independent, the order of arrival of results is not important. This makes it very easy to implement dynamic load balancing for maximum utilisation of resources. The basic algorithm (at the root process) is given in Fig. 4.1.

4.2.3 Initialising the Master-Slave Parallel System

To simplify the design process, both master and slave component programs in the parallel system are implemented in a single executable. Also, since the parallel programs are started on different computer systems (there is no shared memory), each slave must go through the initialisation procedure separately. In this scenario, we must make sure that before parallel execution starts, all programs have the required data initialised with the same values. Similarly, a mechanism must be provided for the root process to change simulation parameters in the slaves (for example, to simulate the same system at a different SNR). The initialisation procedure is depicted by the flowchart in Fig. 4.2. The program can check whether it is a master or slave by reading the size of the par-

ent communicator; if it is of size zero, then the running program must be the master, otherwise, it is a slave.

The root program must initialise the whole system by checking the topology of the available system, and spawning a slave per processor. The slave routine merely has to respond to the commands given by the master, exiting when this is requested. To avoid hogging resources, all slaves automatically decrease their process priority on initialisation. This does not imply a performance penalty unless other users are also using the system.

The cluster of workstations used included a number of single- and dual-processor machines as well as a quad-processor machine (all processors were Intel Pentium Pro 200MHz). Due to limitations in the MPI 1.1 standard regarding the initialisation of processes, the standard LAM model only caters for machines as processing nodes, and processes are distributed across machines in round-robin fashion. Now, spawning has to be performed in a single MPI call, because all slave processes need to be in the same MPI communicator. This makes it impossible to start exactly one process per processor.

In order to work around this problem, the class which handles MPI (a higher level interface to the LAM routines, written specifically for a client-server setup) initially spawns a number of processes equal to the number of machines, and asks each process to return the number of processors on that particular machine. In this way, the root process builds a table containing the number of processors in each machine. Next, the root process spawns a number of processes equal to νN, where N is the number of machines and ν is the largest number of processors in any single machine. At this point, there are exactly ν processes per machine. Now while this is what we want for any machine with ν processors, there are extra processes on other machines. Thus, finally, the root process will kill $\nu - n_i$ processes on each machine, where n_i is the number of processors in machine i. This algorithm is described in Fig. 4.3.

4.2.4 Handling Dead-Lock

Under certain conditions, a large proportion of the samples have a zero value. In such cases, the gaussian profile assumption is violated, and the estimate will be seen as

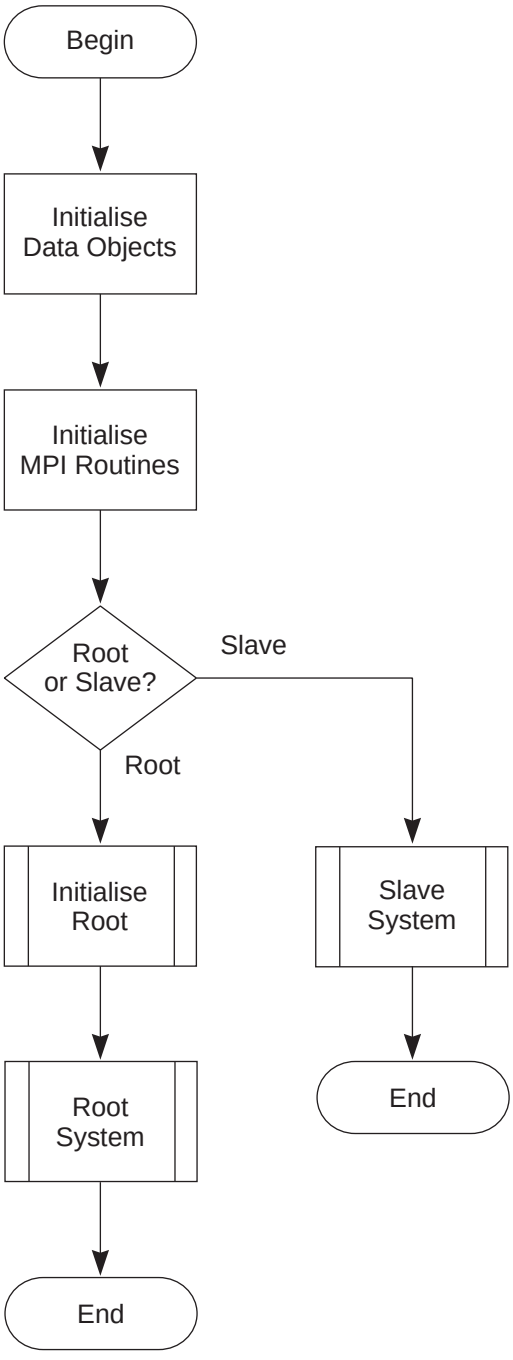


Figure 4.2: Initialisation of MPI System

```
begin
    Read the number of nodes (N) in the LAM system
    Spawn N processes (one per machine)
    for each of the N processes
        Request process to return the number of CPUs in that machine
        Read the returned result, and store it in a table
        Request process to die
    end for
    Find the largest number of processors per node, ν
    Spawn νN processes (ν per machine)
    Kill extraneous processes on machines with less than ν processors
end
```

Figure 4.3: Root Process Initialisation

converging very slowly. This happens, for example, when computing the error-count profile (i.e. the histogram of the number of bits in error per frame). The solution is not to consider the computed accuracy of such estimates when checking if the required accuracy has been reached. This is done by considering only results for which the fraction of non-zero estimates is greater than $\frac{1}{\lambda}$, where λ is chosen by the user.

4.3 BCJR Algorithm

A major implementation hurdle of the probability-based BCJR algorithm is the large numerical range of the α and β metrics, which increases exponentially with block size. This problem has led researchers around the world to implement the BCJR algorithm in the Log domain (using LLR). Our solution is to implement the BCJR algorithm using the template feature of C++ to allow the use of any mathematical precision required. A number of higher-precision arithmetic classes are used in our implementation, providing a range of compromises between accuracy and speed / memory requirements. These are described below; an evaluation of their accuracy and speed, and how well they perform within a Turbo code, is given in the following chapter.

4.3.1 Infinite Precision

Initially, the accuracy problem was avoided by providing a class which can compute on real numbers with infinite accuracy. The class is built as a C++ interface to the GNU Multi-Precision library [Granlund, 1996], using overloaded operators for addition, subtraction, multiplication and division. Conversion to and from double-precision values is also provided. While its usefulness is undeniable, such an implementation naturally suffers a significant performance disadvantage. It is particularly useful as a reference for assessing the suitability of other strategies.

4.3.2 Extended Precision Floating-Point

Since the major requirement for the BCJR algorithm is extended range rather than increased precision, a more optimised solution to the problem is provided by a class with a 32-bit exponent. To simplify implementation, the mantissa is held as a standard IEEE 64-bit floating-point number. After each arithmetic operation, the result is normalised. This class results in a significant increase in performance and reduction in memory usage. However, both speed and memory are still compromised for the benefit of accuracy.

4.3.3 Log-Scale Floating-Point

The usual solution to this dilemma in practical Turbo codes is to store the natural logarithms of the numbers, rather than the value itself. While this clearly solves the range problem, its effect on the resolution of the numbers is not clear. Working with logarithms also introduces a new difficulty – computing the addition of two numbers. A practical solution is to compute the addition of A and B using the corresponding logarithm values a and b as follows:

$$-\log(A+B) = -\log\left(e^{-a}+e^{-b}\right) \quad (4.1)$$

$$= \min(a,b) - \log\left(1+e^{-|a-b|}\right) \quad (4.2)$$

where $a = -\log(A)$ and $b = -\log(B)$. Thus, the addition becomes a comparison operation (to select the smaller of a or b) and a function that depends on the difference

between a and b. It can be shown that the largest value of $f(c) = \log(1 + e^{-c})$ for $c \geq 0$ is equal to $\log(2)$ and that as c increases, $f(c)$ quickly tends to zero. This means that $f(c)$ can be easily implemented using a small look-up table by quantising c [Pietrobon, 1995].

4.4 Other Support Facilities

4.4.1 Random Number Generator

Pseudo-random sequences are used in various parts of the system, including the channel model and a number of interleaver creation techniques. The random generator must provide the following interface:

- Seeding function.
- A function which returns a random integer modulo m, where m is passed as a user parameter, and a function returning a random floating point number in $[0, 1]$. The numbers returned by both of these functions have a flat probability distribution in the allowed range.
- Another function returning a random floating-point number with a gaussian probability distribution (zero mean and unit variance, or user-supplied variance).

Random sequences are generated using the subtractive algorithm, originally due to Knuth [1981]; our implementation, however, is based on the description given by Press *et al.* [1992]. This algorithm was chosen because it has a very long period (necessary for simulating low bit-error rates) and also because it does not suffer from low-order correlations, simplifying the generation of random bit sequences.

Gaussian profile reshaping is performed using the Box-Muller Transform [Press *et al.*, 1992]. This transform has the advantage that two Gaussian random numbers are generated from two random number with a flat profile. Thus, the period of the underlying generator is not reduced. Another good property is that the profile of the random values generated is exactly Gaussian, even in the tail regions.

Chapter 5

Testing

5.1 Introduction

In this chapter we document the testing performed on the Turbo codec and simulation environment. The testing strategy involved validating the simpler modules (those that can be tested separately) first, and then proceeding to the more complex modules (which make use of other modules internally). In practice, we first tested the Monte Carlo simulator with an uncoded transmission. This is because the expected BER can be easily computed mathematically for comparison. Next, we tested the MAP decoder by comparing simulated results with a Union bound for BER. We also confirm the accuracy of the various numerical representations used with the BCJR algorithm by comparing simulation results for the different techniques. Finally, we test the Turbo decoder by comparing with known bounds and with published results.

5.2 Uncoded Transmission

Initial testing involved the uncoded transmission of information using BPSK modulation over the AWGN channel. The expected bit error rate can be easily calculated for comparison with the simulation results [Proakis, 1995]:

$$P_b = Q\left(\sqrt{2.\frac{E_b}{N_0}}\right) \tag{5.1}$$

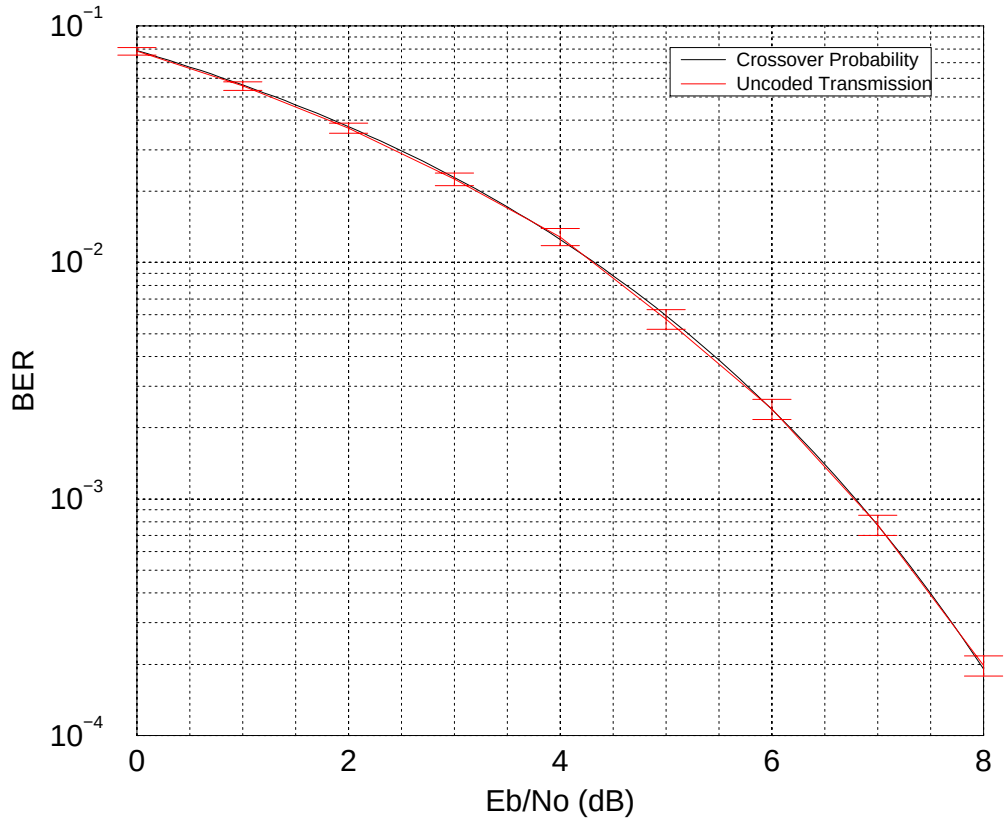


Figure 5.1: Uncoded Transmission

Blocks of 1000 bits were transmitted without channel coding, and an estimate of the BER computed using the Monte Carlo algorithm (with a tolerance of $\pm 10\%$ at a confidence level of 95%). This was compared with the expected BER calculated using Eqn. (5.1), as shown in Fig. 5.1. It is clear that the expected value is well within the simulation tolerance limits, also shown in the figure.

5.3 Upper Bound for MAP Decoder

The bit error performance of a linear block code can be upper bounded by the union bound [Cover & Thomas, 1991]:

$$P_b \leq \sum_{w=1}^{\tau} \sum_{d=d_{min}}^{\infty} A(w,d).\frac{w}{\tau}.Q\left(\sqrt{2.d.R.\frac{E_b}{N_0}}\right) \tag{5.2}$$

where $A(w,d)$ is the number of codewords of input weight w and total weight d. The code's block size is given by the number of information bits τ and the code rate R.

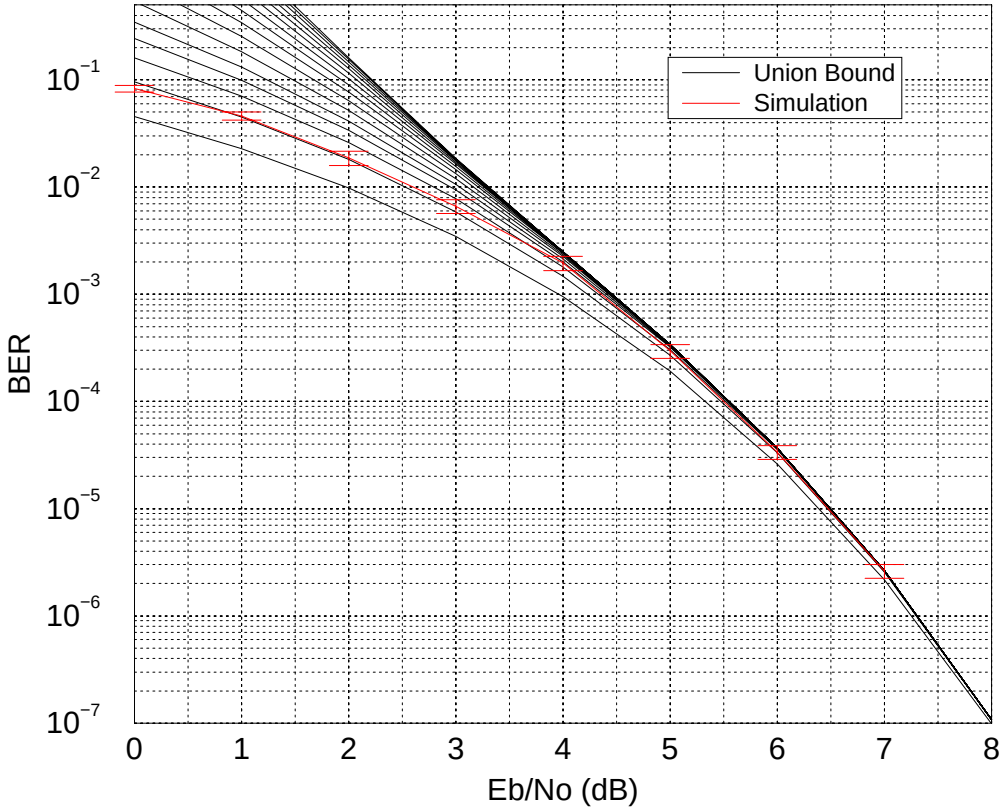


Figure 5.2: MAP decoder compared with Union Bound

Consider a rate $\frac{1}{2}$ RSC code with generator $(1, 5/7)$[1]. Now, encoding an information block of 24 bits (and a further 2 bits for tailing) will result in a $(52, 24)$ equivalent block code. After obtaining the $A(w, d)$ matrix up to $d = 20$ for this code, the union bound can be computed. The code's performance can be simulated using a MAP decoder based on the BCJR algorithm. Fig. 5.2 compares the simulation result with the union bound (shown for increasing codeword weight considered in the summation). As expected, the bound is very tight for high SNR, where the bound converges rapidly. It can also be noted that the first few terms of the summation can be used instead of the complete bound as a more accurate predictor of performance.

5.4 BCJR Arithmetic Accuracy

In order to confirm the accuracy of the faster arithmetic routines used to speed up the BCJR algorithm (as discussed in Section 4.3), we simulate the same terminated RSC

[1]Polynomials are denoted as g_a or g_a/g_b, where g_a is the feedforward polynomial and g_b is the feedback polynomial, expressed in octal.

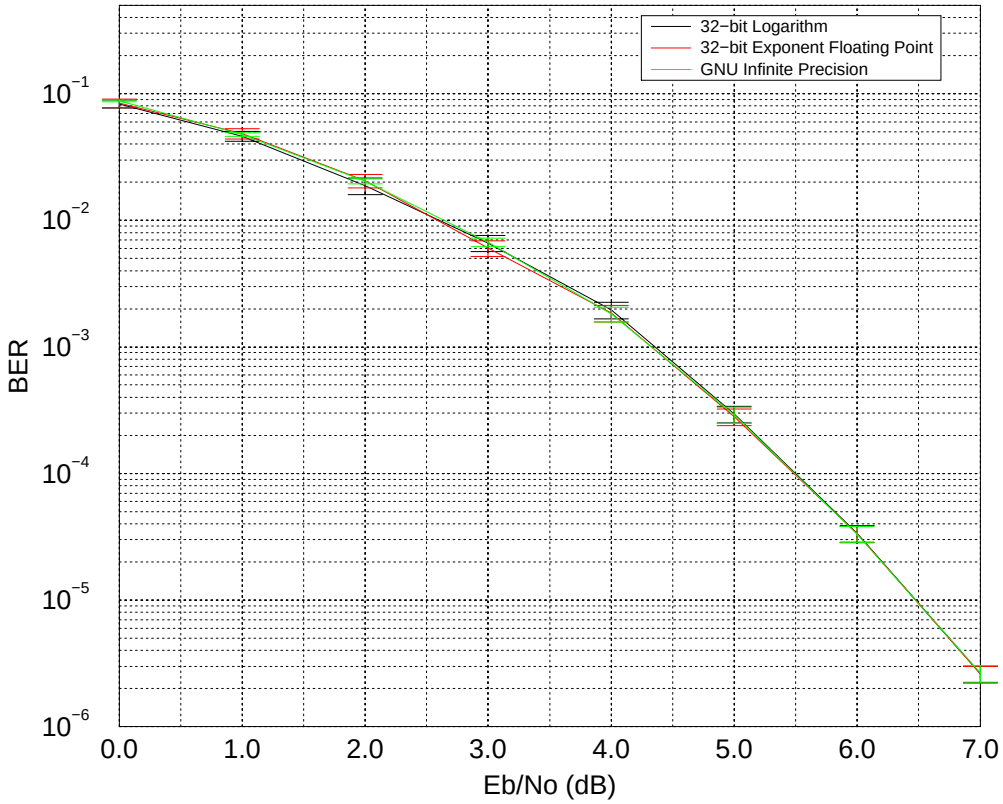


Figure 5.3: Comparison of Arithmetic Routines for MAP decoder

code used in Section 5.3 using all three arithmetic models. As can be seen from Fig. 5.3, using the faster routines does not result in any noticeable difference in performance.

5.5 Upper Bound for Turbo Decoder

The same union bound given in Eqn. (5.2) can be used with Turbo codes, assuming that we can obtain the $A(w, d)$ matrix. For Turbo codes, this is not a trivial task because the block sizes are usually large and the presence of the interleaver complicates the problem. There are two solutions to this dilemma – the first is to work with very small block sizes. An alternative solution, which can be used with the larger block sizes normally associated with Turbo codes, is to assume the presence of a *uniform interleaver*.

A rate $\frac{1}{3}$ Turbo code is created using RSC component codes with generator $(1, 7/5)$, and a uniform interleaver. The information block length is 1000 bits, and both the original and the interleaved sequences are tailed using the method described by Divsalar and

d	A_d
8	0.0039881
9	0.0079605
10	0.011918
11	0.015861
12	0.019887
13	0.024188
14	0.029048
15	0.034846
16	0.065768
17	0.1457
18	0.2984
19	0.5472
20	0.9171
21	1.437
22	2.144
23	3.09
24	4.465
25	6.716
26	10.67
27	17.65
28	29.61
29	49.31
30	80.57
31	128.6
32	201.3
33	311.5
34	481.2
35	748.8

Table 5.1: Coefficients used to compute BER bound for $(3006, 1000)$ Turbo code

Pollara [1995c]. This results in a $(3006, 1000)$ Turbo code. The union bound for this code can be computed using the following equation:

$$P_b \approx \sum_d A_d.Q\left(\sqrt{2.d.R.\frac{E_b}{N_0}}\right) \tag{5.3}$$

where A_d is the total information weight of all codewords of weight d divided by the number of information bits per codeword, as defined by Eqn. (6.3) on page 71. The summation usually operates on a truncated set of codeword weights. For the Turbo code with $\tau = 1000$, the set of coefficients used to compute the bound is given in Table 5.1, as quoted by Benedetto and Montorsi [1996b]. The bound is compared with a simulation of the code's performance in Fig. 5.4.

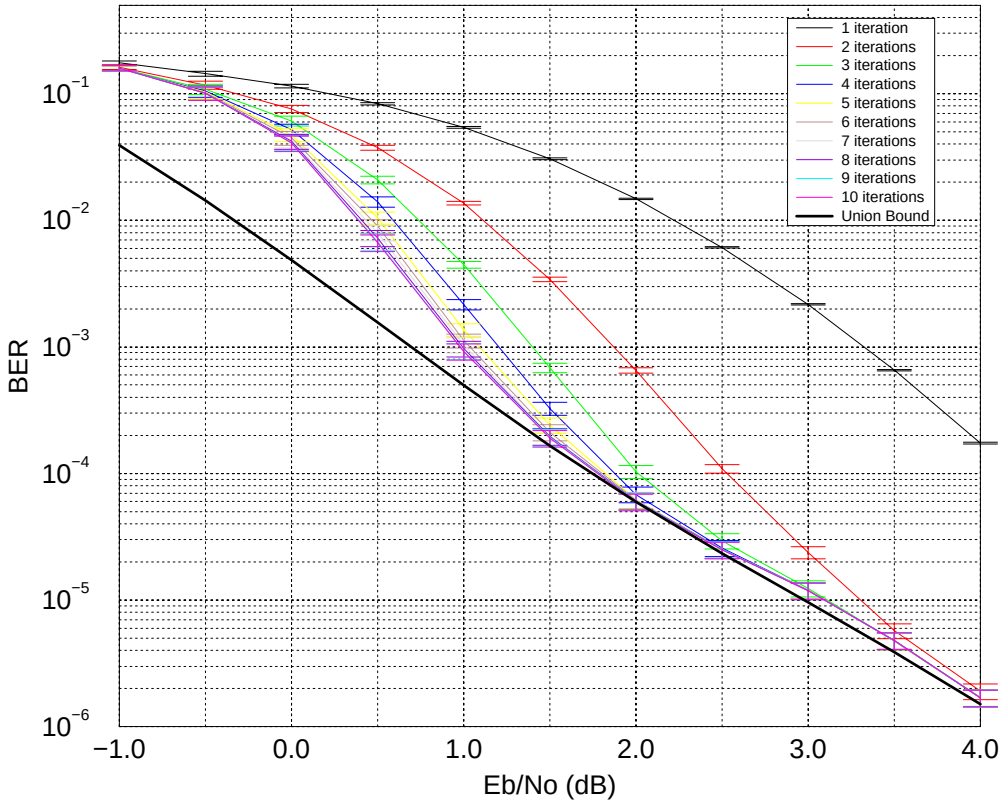


Figure 5.4: Turbo decoder compared with Union Bound

5.6 Comparison with Published Results

Finally, we compare our simulator with results published by other independent researchers. The first setup that we replicated consists of a rate $\frac{1}{2}$ Turbo code[2], constructed from two RSC component codes with generator $(1, 5/7)$, a helical interleaver with 17 rows and 6 columns, and an odd-even puncturing pattern. The 102-bit information sequence is passed through the interleaver, and a 2-bit tail appended, resulting in a $(208, 102)$ Turbo code. Since the interleaver is simile, the same tail can be used for both interleaved and non-interleaved sequences. Our simulation results, shown in Fig. 5.5, agree with those published by Barbulescu [1996, p. 41] for 8 iterations (our code is the same as the TNIE code given in Fig. 3.11).

The second setup is for a rate $\frac{1}{3}$ Turbo code constructed from two RSC component codes with generator $(1, 5/7)$ and a 1024-bit random interleaver[3]; the sequences are

[2]Note that the code rate is not exactly $\frac{1}{2}$, but this is the closest convenient fraction.

[3]The mapping used is included in the CD (`montorsi-1024.dat`).

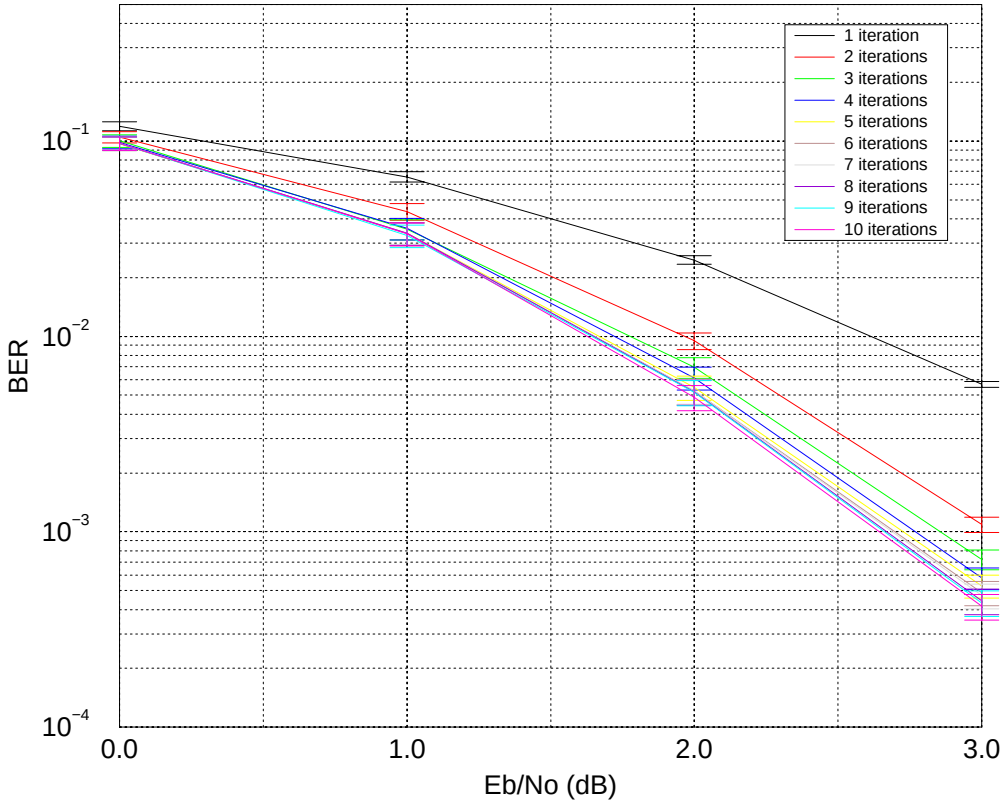


Figure 5.5: Simulation of $(208, 102)$ Turbo code

terminated using the JPL scheme[4]. Our simulation results after ten iterations are shown in Fig. 5.6, together with the results quoted by Montorsi[5]. Note that for all points, we simulated more than twice the number of blocks used by Montorsi. Furthermore, our choice for the number of blocks to be simulated is not *ad hoc*, but is based on the required tolerance and confidence level. We also observed that for a small number of samples (less than 100), the distribution of sample results is skewed.

[4]See Section 2.3.6.

[5]Published on the web at `http://hp0tlc.polito.it/~montorsi/simulation.html`.

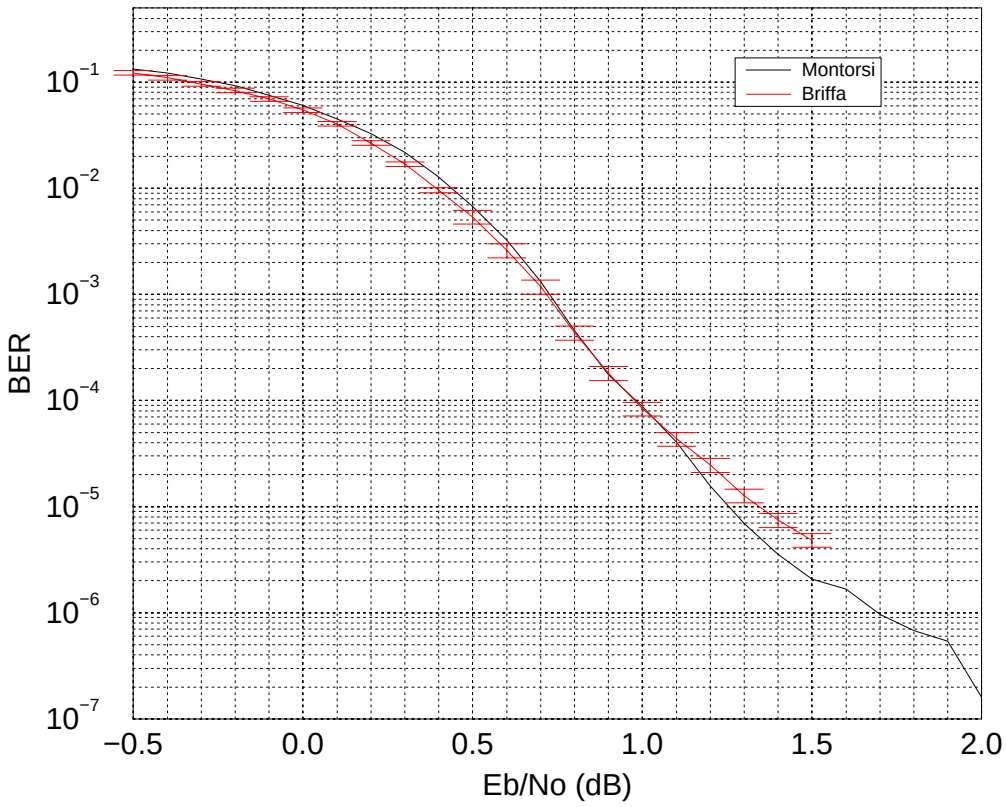


Figure 5.6: Turbo decoder comparison with Montorsi

Part II

Interleaver Design for Turbo Codes

Chapter 6

Overview

6.1 Introduction

It is widely accepted by the research community that the interleaver plays a very important part in the construction of good Turbo codes. There are valid indications as to what makes a good interleaver; unfortunately, however, there is no comprehensive explanation or computable quantification of the effects of interleaver choice on the resultant Turbo code. In this chapter, we draw together known results and attempt to use these to determine which parameters make a good interleaver.

6.2 Interleaving in Coding Systems

6.2.1 Classical Use of Interleavers

Interleavers have been used in communication systems for a long time; the classical use is to randomise the location of errors, enabling the use of random-error-correcting codes on channels with burst error patterns. Typically, bursty channels would include fading channels often found in wireless transmission. Another use of interleaving is in concatenated coding, where the output of the outer-stage decoder exhibits burst error patterns (as happens with a Viterbi decoder). Hence, the most important parameter of the interleaver in this case is its ability to spread error bursts such that they appear as isolated errors to the inner-stage decoder. Naturally, the optimum interleaver would

achieve this with the minimum memory [Ramsey, 1970]. It should be clear that the required parameters of the interleaver in this case depend uniquely on the inner and outer codes and decoders used.

6.2.2 Interleavers in Turbo Codes

Turbo coding, however, also introduced a further dimension to what is required from the interleaver – this involves the effects of the iterative algorithm and the passing of intrinsic information between successive decoder stages. In this context, the interleaver has often been explained as reducing the correlation between the parity bits corresponding to the original and interleaved data frames. However, for reasons which will be explained later, this terminology isn't quite accurate.

In the original paper introducing Turbo codes, Berrou and Glavieux [1993] already showed an exceptional understanding of some of the most important parameters making a good interleaver. In particular:

- Increasing the block size (and hence the size of the interleaver) results in improved performance.
- The interleaver should randomise the input sequence in order to avoid particular low-weight patterns mapping onto themselves, reducing the effective free distance of the resulting Turbo Code.

Initially, the interleavers used with Turbo Codes were block interleavers. Such interleavers can be described by a $T \times T$ matrix where every row and every column contains a single 1 and $T - 1$ zeros. If a 1 occurs in the i^{th} row and the j^{th} column, then the interleaver moves the i^{th} input symbol to the j^{th} output position. Recently, however, another kind of interleaver has re-emerged – the convolutional interleaver would classically consist of a set of T shift registers of increasing length, with the input sequence being multiplexed into T subsequences, thus introducing a different delay for each subsequence. The output sequence is obtained by de-multiplexing the outputs of the shift registers. In particular, such interleavers have been applied to stream-oriented Turbo codes [Hall & Wilson, 1998].

6.3 Performance of Turbo Codes

Before discussing what makes a good interleaver, we will briefly analyse the performance of Turbo codes, based on their distance spectrum. This analysis assumes a ML decoder, and was published by Perez *et al.* [1996] and also by Benedetto and Montorsi [1995b; 1996a; 1996b]. While the iterative decoder used in Turbo codes is not a ML decoder, its performance is very close, so that the analysis is a valid explanation of Turbo code performance.

6.3.1 Recursive and Non-Recursive Constituent Codes

The part played by the use of RSC codes as components for Turbo codes is critical. Although NRC codes perform better than RSC codes when used on their own, the particular weight distribution of RSC codes results in a significant improvement within the Turbo coding context. Consider, as an example, the 2-state systematic NRC code with generator $(1, 3)$ and RSC code with generator $(1, 2/3)$. When used with a MAP decoder and a block size $\tau = 100$ (including a 1-bit tail), the BER performance of the NRC code is superior to that of the RSC code, as shown in Fig. 6.1.

The difference between the two codes becomes immediately apparent when we consider the weight distribution of the terminated convolutional codes. The weight distributions for NRC and RSC codes up to a codeword weight of 20 are shown in Figs. 6.2 and 6.3 respectively. On the graphs darker regions denote a higher multiplicity, where multiplicity is the number of codewords with the specified input and output weight. For each weight of the input sequence, the weight of codewords generated by the RSC code span a broad range with a rather uniform multiplicity. On the other hand, for input sequences with low weight, the NRC code generates a small set of codeword weights of low value.

In a Turbo code, assuming a permuting interleaver, the interleaved sequence has the same weight at the input sequence. However, since the order of the bits is changed, the weight of the parity bits for the interleaved sequence may be different from that of the input sequence. When NRC codes are used, low-weight inputs are associated with low-weight parity sequences. This means that the parity sequences for both the

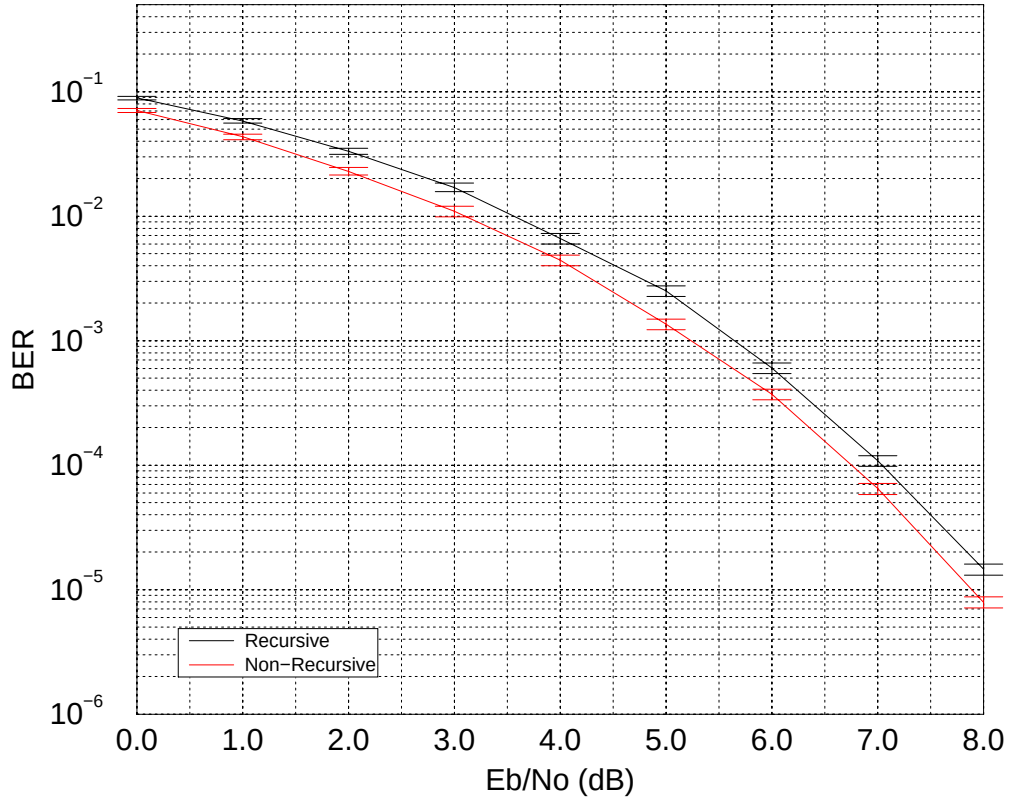


Figure 6.1: BER performance of 2-state RSC and NRC codes

input and interleaved sequences will have a low weight. On the other hand, if RSC codes are used, there is a good chance (depending on the interleaver mapping) that inputs associated with a low-weight parity will be mapped to an interleaved sequence associated with a high-weight parity.

6.3.2 Performance Bound

The BER of a linear block code can be upper bounded by the union bound[1]:

$$P_b \leq \sum_{w=1}^{\tau} \sum_{d=d_{free}}^{\infty} A(w,d).\frac{w}{\tau}.Q\left(\sqrt{2.d.R.\frac{E_b}{N_0}}\right) \tag{5.2}$$

where $A(w,d)$ is the number of codewords of input weight w and total weight d. The code's block size is given by the number of information bits τ and the code rate R. Ignoring the effect of the tail (assuming that the tail length $\nu \ll \tau$), we can use our usual definition of τ as being the length of the whole source sequence, including the

[1]See also Sections 5.3 and 5.5.

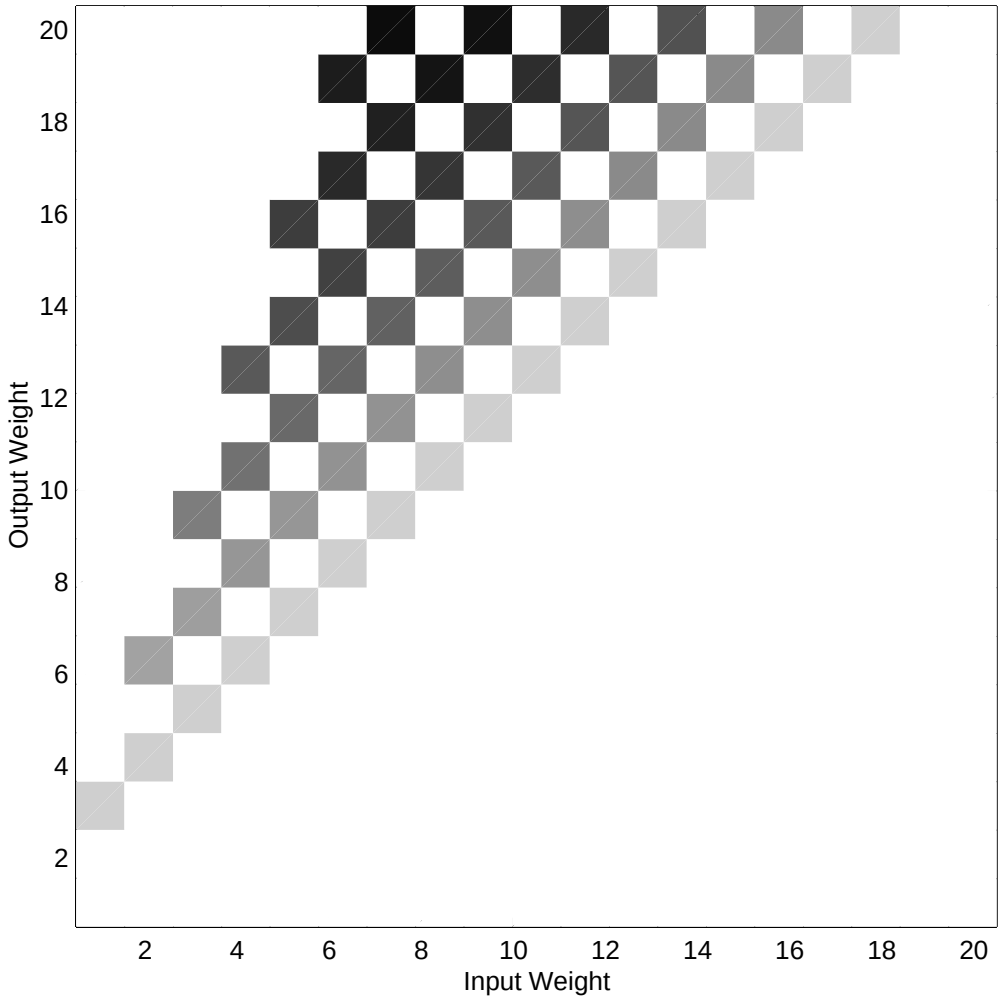


Figure 6.2: Weight distribution of a terminated NRC code ($\tau = 100$)

tail. Thus, changing the order of summation:

$$P_b \leq \sum_{d=d_{free}}^{\infty} \left[\sum_{w=1}^{\tau} A(w,d).\frac{w}{\tau} \right] .Q\left(\sqrt{2.d.R.\frac{E_b}{N_0}} \right) \tag{6.1}$$

$$\leq \sum_{d=d_{free}}^{\infty} A_d.Q\left(\sqrt{2.d.R.\frac{E_b}{N_0}} \right) \tag{6.2}$$

where A_d is the total information weight of all codewords of weight d divided by the number of information bits per codeword, as defined by:

$$A_d = \sum_{w=1}^{\tau} A(w,d).\frac{w}{\tau} \tag{6.3}$$

Now, define N_d to be the number of codewords of total weight d and w_d to be their

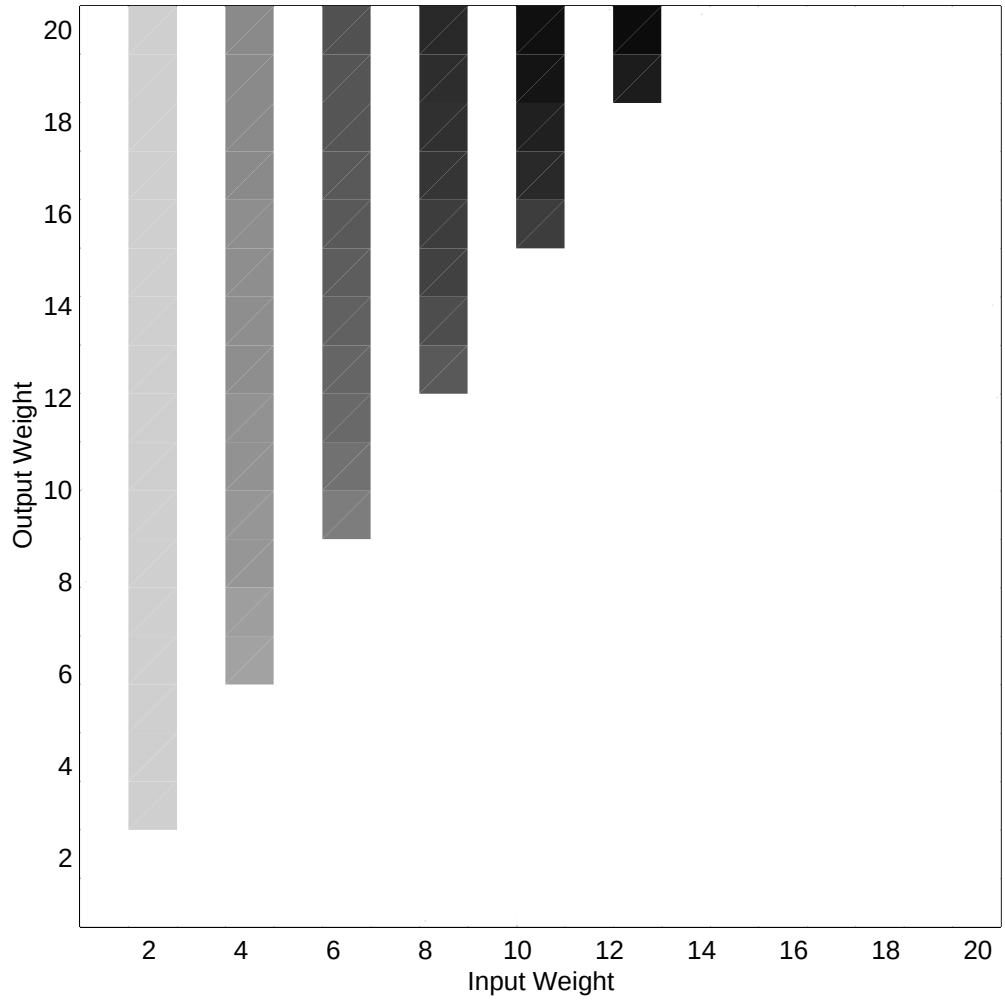


Figure 6.3: Weight distribution of a terminated RSC code ($\tau = 100$)

average information weight. Thus,

$$N_d.w_d = \sum_{w=1}^{\tau} A(w,d).w \tag{6.4}$$

$$\therefore A_d = w_d.\frac{N_d}{\tau} \tag{6.5}$$

where $\frac{N_d}{\tau}$ is called the *effective multiplicity* of codewords of weight d. Finally, we can insert this into the bound:

$$P_b \leq \sum_{d=d_{free}}^{\infty} w_d.\frac{N_d}{\tau}.Q\left(\sqrt{2.d.R.\frac{E_b}{N_0}}\right) \tag{6.6}$$

6.3.3 Free Distance and Performance at High SNR

It can be seen from Eqn. (6.6) that the BER contribution of codewords of weight d depends on three terms:

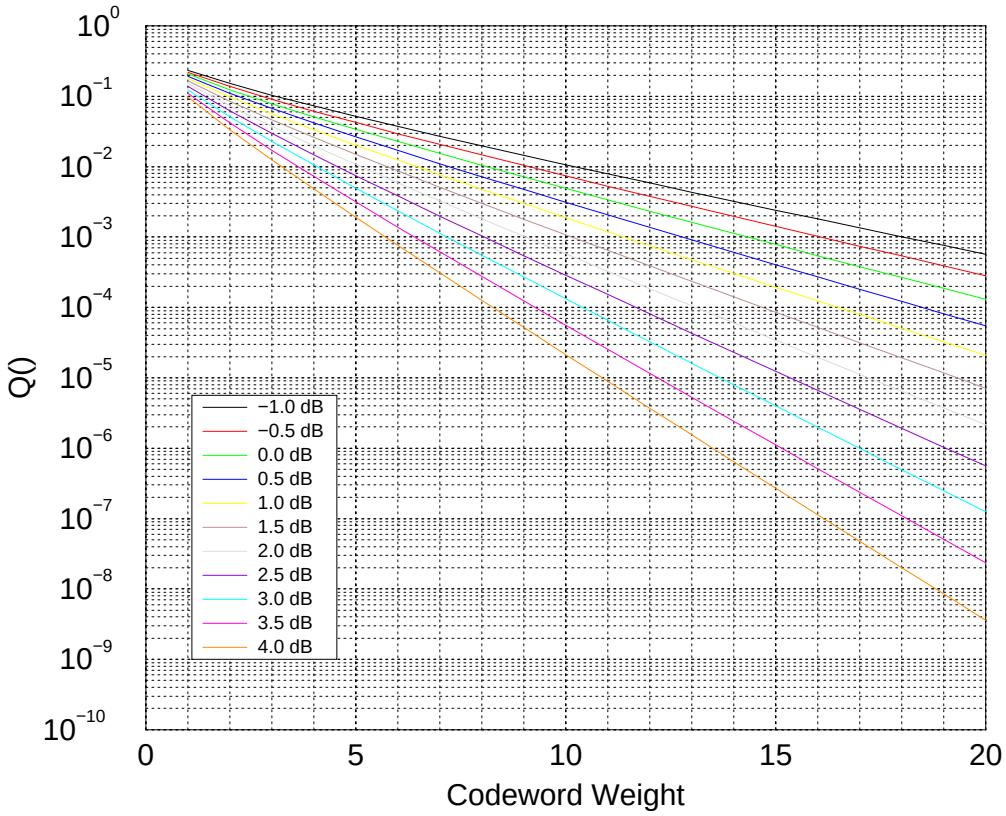


Figure 6.4: BER contribution of the complementary error function ($R = \frac{1}{3}$)

- Their average information weight w_d.
- Their effective multiplicity $\frac{N_d}{\tau}$.
- The complementary error function $Q\left(\sqrt{2.d.R.\frac{E_b}{N_0}}\right)$.

While the first two terms depend uniquely on the code structure, the final term depends also on the channel SNR. In order to visualise its effect, we plot the value of the $Q()$ function for a typical range of channel SNR and for codeword weights up to $d = 20$. The plot, shown in Fig. 6.4, also assumes a code rate of $\frac{1}{3}$.

It is clear that as the SNR increases, the contribution due to larger codeword weights is reduced. Asymptotically, the BER performance of the code is dominated by the free distance term (for $d = d_{free}$) at high SNR. Thus for Turbo codes the asymptotic performance approaches

$$P_b \approx w_{free}.\frac{N_{free}}{\tau}.Q\left(\sqrt{2.d_{free}.R.\frac{E_b}{N_0}}\right) \tag{6.7}$$

where N_{free} is the multiplicity of free distance codewords and w_{free} is the average weight of their corresponding information sequences. In a Turbo code with a random interleaver a few low-weight input sequences associated with low-weight parity may be mapped to interleaved sequences which are also associated with low-weight parity. Such codewords lower the free distance of the Turbo code, causing the error floor.

6.3.4 Spectral Thinning and Performance at Low-Medium SNR

Consider a Turbo code with a uniform interleaver. Several low-weight input sequences associated with low-weight parity will be mapped to interleaved sequences which are associated with high-weight parity (since the average parity weight is high even for low-weight input). Compared with the weight distribution of its component RSC code, the number of low-weight input sequences associated with high-weight parity is increased while those associated with a low-weight parity remain few. This phenomenon is termed *spectral thinning* because the low-weight portion of the distance spectrum retains a low multiplicity.

Thus, although the free distance of a Turbo code may be small compared with a Maximum Free Distance (MFD) convolutional code of larger memory, the multiplicity of low-weight codewords is lower (not just for the free distance codewords). This means that at low-medium SNR, when the effect of higher-weight codewords cannot be neglected, the performance of a Turbo code is better. This phenomenon was first explained in detail by Perez *et al.* [1996].

6.3.5 Interleaver Input-Output Distance Spectrum

In order to visualise the spread of an interleaver and how it is achieved, we plot the Input-Output Distance Spectrum (IODS). This graph is a two-dimensional histogram, with the axes being the distance between all possible bit-pairs at the input and output of the interleaver; the darker the region the more instances with the given parameters. For comparison, we plot the IODS (limiting ourselves to a distance of 100) for a Square and Berrou-Glavieux interleaver of equal size in Figs. 6.5 and 6.6 respectively. These interleavers are used in Chapter 8, and are listed in Table 8.1 on page 104.

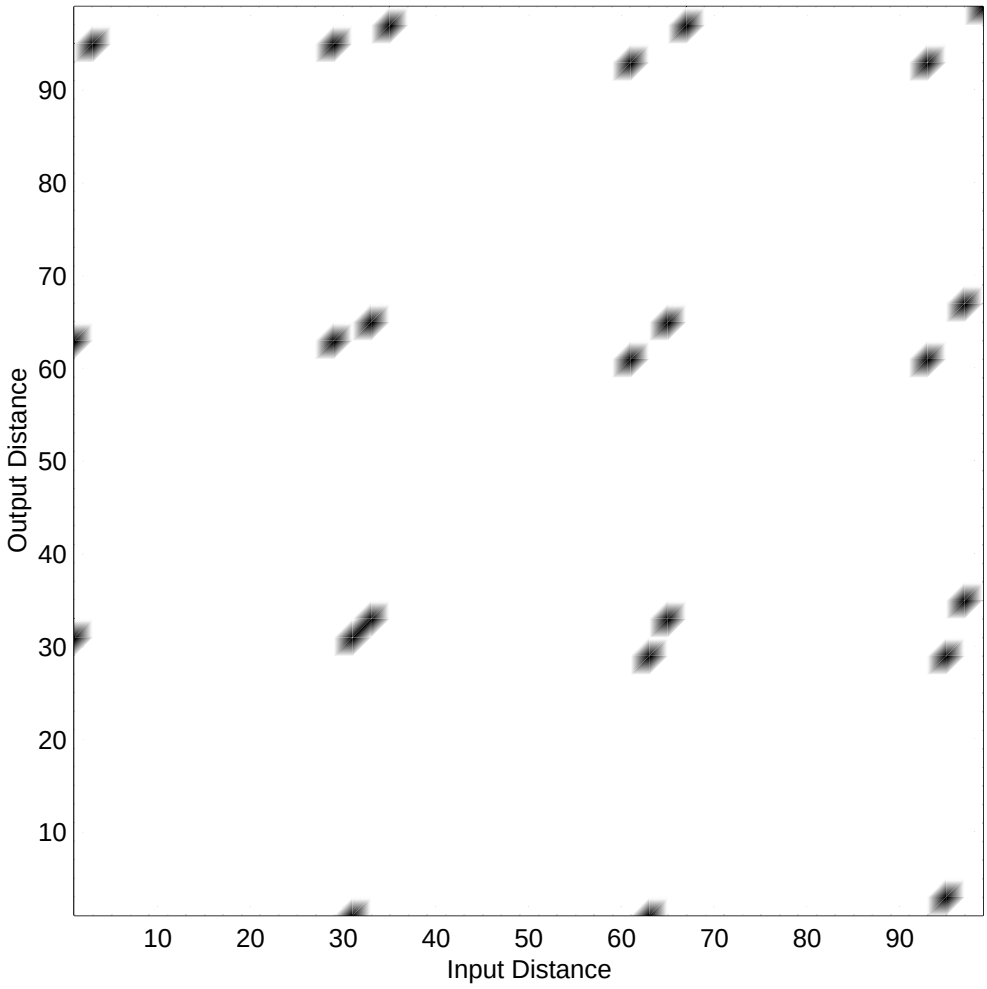


Figure 6.5: IODS for Square Interleaver

Note how the Square interleaver concentrates bit-pairs in very small regions, while the Berrou-Glavieux interleaver (through the pseudo-random perturbations) spreads out the regions. For this reason, the peak frequency for the Square interleaver is almost four times that of the Berrou-Glavieux interleaver.

6.4 Interleaver Requirements

While restricting ourselves to block interleaver structures, which can describe any interleaver used in a block-oriented Turbo code, we discuss below the main requirements for interleaver design. The simultaneous optimisation of all requirements is a problem of significant complexity; unfortunately, even their separate optimisation is non-trivial because of the inter-dependant restrictions.

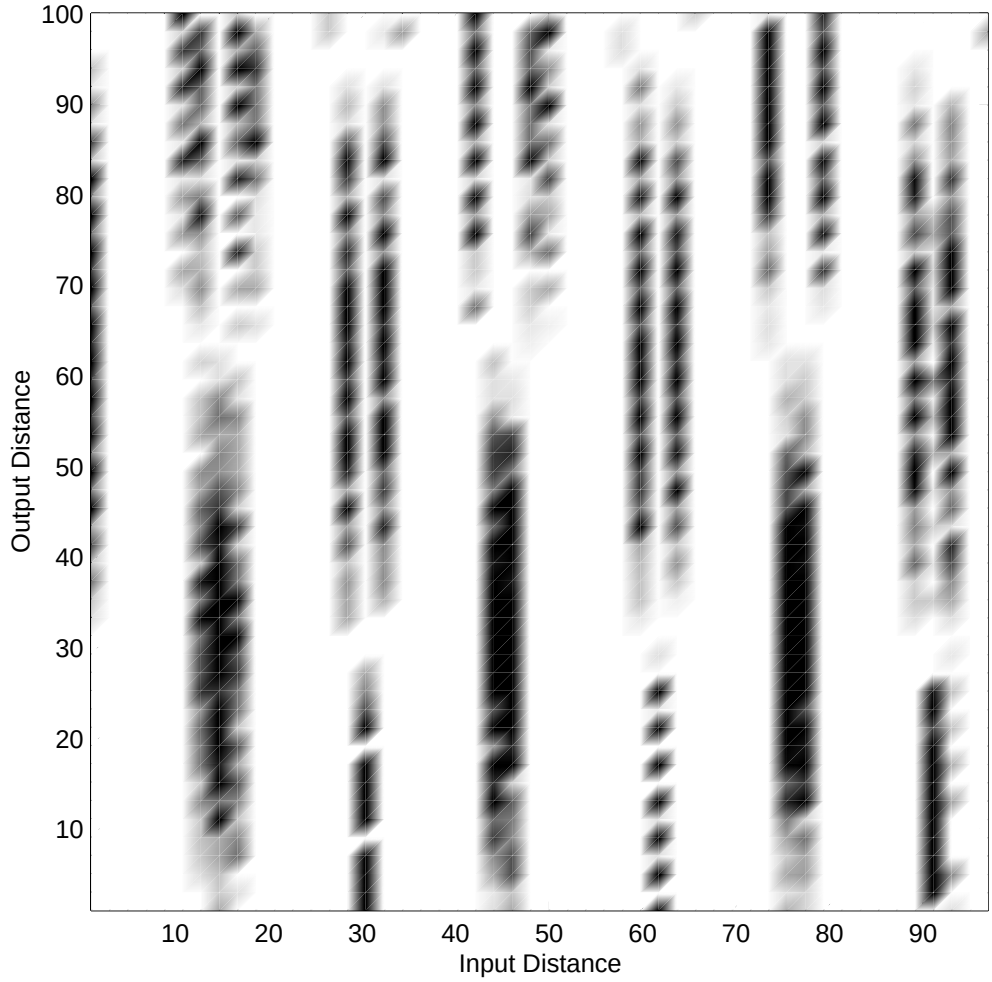


Figure 6.6: IODS for Berrou-Glavieux Interleaver

6.4.1 Block Size

The length of the interleaver should be as large as possible for improved performance. It can be seen from Eqn. (6.7) that as block size increases (assuming that the free distance and multiplicity are unchanged) the effective multiplicity of the free distance term is decreased. This lowers the error floor asymptote without changing its slope, in proportion with the increase in block size. This effect has been suitably documented and quantified by the JPL team [Dolinar *et al.*, 1998]. Note, however, that increasing the interleaver size also increases the decoder delay by a corresponding amount.

6.4.2 Interleaver Mapping Randomisation

The importance of randomisation in the interleaver is evident and accepted. One of the reasons given for randomisation is the breaking of low-weight sequences to increase the free distance and hence improve the code's asymptotic performance [Berrou & Glavieux, 1996; Perez *et al.*, 1996].

The effect of randomisation as a method of reducing correlation between the parity sequences, and its effects on the iterative nature of the Turbo decoder is still unclear. This reason is close to the classical use of interleavers, allowing the use of random-error-correcting codes in situations where there is a marked temporal correlation in noise patterns. The requirement for randomisation as opposed to spreading is still unclear in this sense.

6.4.3 Interleaver Spread

In order to spread error bursts between successive decoders, the interleaver must ensure that bit pairs which are close (within 5ν, where ν is the memory of the constituent codes) in the original sequence will be further apart in the interleaved sequence. The limit of 5ν is chosen because in a convolutional code, an error in the received sequence will affect the decoder's performance for bits within this distance of the error. This can be visualised by observing the confidence at the output of a MAP decoder when the input has a single hard error.

Fig. 6.7 shows the confidence at the output of a MAP decoder for every bit in a frame consisting of 102 data bits and 2 tail bits. The code used is a $(n = 2, k = 1, \nu = 2)$ RSC code with polynomials $(1, 5/7)$. The confidence metric is computed as the ratio p_0/p_1, where p_0 and p_1 are the *a posteriori* probabilities of the correct and incorrect bit, respectively, at a given position within the frame. Notice how a single hard error at position 52 affects approximately ten (5ν) bits on either side. As one would expect, closer bits are affected most; the effect practically vanishes beyond the distance of 5ν bits. Other points worth noting are that tail and lead bits have a higher confidence (because in those cases the decoder knows the start / final state with certainty), and that the confidence graph is symmetric about the point containing the error.

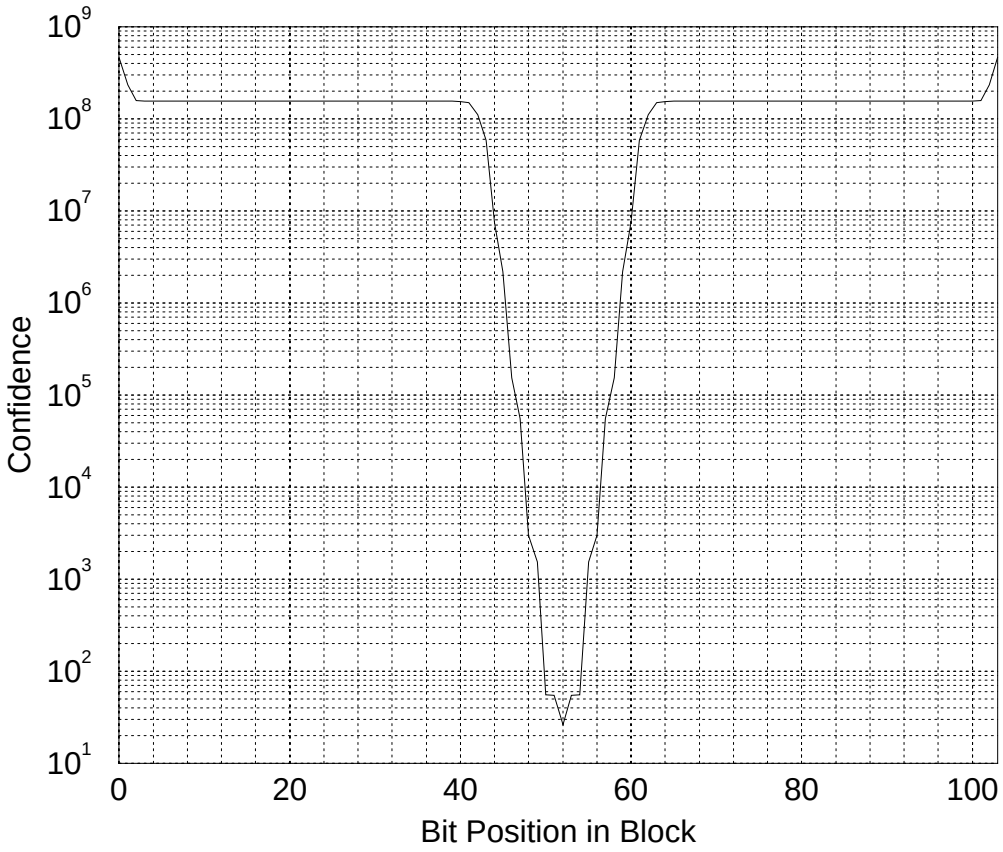


Figure 6.7: Error Propagation in a MAP Decoder

A quantification of the effect of interleaver spread on the Turbo code's performance is still missing. Also, the effect of multiplicity of points in the IODS close to the spread boundary is not known.

6.4.4 Interleavers for Punctured Codes

Some work has been done on the effect of puncturing. In particular, the JPL group has computed a list of the best rate-$\frac{1}{2}$ constituent codes with odd-even puncuring [Divsalar & Pollara, 1995b], based on the minimum weights of parity sequences for low-weight information sequences. Another contribution in this area, this time due to Barbulescu [1996], is the restriction imposed on the interleaver to make it odd-even.

6.4.5 Trellis Termination

The original Turbo code propositions ignored the effect of terminating the information sequences. While the effect of this is considered insignificant for large block sizes, it

may have a detrimental effect on small codes. A sufficient condition on the interleaver to ensure that both the original and the interleaved sequence are terminated is given by Blackert *et al.* [1995]. A further improvement is given by Barbulescu [1996]; by further restricting the interleaver, both the original and interleaved sequence have the same tail (and thus the same parity sequence in the tail section). This effectively removes the effect of puncturing in the tail section.

Chapter 7

Interleaver Types

The interleaver types considered in our analysis can be subdivided into three categories. Standard interleavers include those used with Turbo codes in the literature, while the Reference interleavers have been designed with the primary intention of gaining some insight into the problem. Finally, the collection of Optimised interleavers includes our efforts at interleaver design and others already mentioned in the literature.

7.1 Standard Interleavers

7.1.1 Rectangular Interleaver

The classical block interleaver has well-known spread properties [Ramsey, 1970]. In particular, Ramsey proves that we can construct an optimal[1] interleaver with a spread of (R, C) by creating a rectangular interleaver with R rows and C columns, where information is written row-wise and read column-wise and the following conditions are satisfied:

- $R + 1$ and C are relatively prime.
- $R + 1 < C$. Note that a Square interleaver cannot obey this condition.

[1] Optimality is in the sense of minimum encoder delay and minimum combined storage capacity for the interleaver and de-interleaver.

We can formalise the mapping function for such a rectangular interleaver by the equation:

$$\lambda(t) = [t \bmod R].C + (t \operatorname{div} R) \tag{7.1}$$

Theorem 7.1.1 *A rectangular interleaver is odd-even if R and C are both odd.*

Proof. Consider a rectangular interleaver as defined by Eqn. (7.1). Now, let

$$i = t \bmod R \tag{7.2}$$

$$j = t \operatorname{div} R \tag{7.3}$$

Thus, we can rewrite the interleaver mapping as

$$\lambda(t) = i.C + j \tag{7.4}$$

$$t = j.R + i \tag{7.5}$$

Now, for an odd-even interleaver, $\lambda(t)$ must be even when t is even, and odd when t is odd. Thus, $|\lambda(t) - t|$ must be even for all values of t. Now, for the rectangular interleaver considered,

$$|\lambda(t) - t| = |(i.C + j) - (j.R + i)| \tag{7.6}$$

$$= |i.(C-1) - j.(R-1)| \tag{7.7}$$

To ensure that $|\lambda(t) - t|$ is even, $(C-1)$ and $(R-1)$ must be both even. Hence, R and C must be odd.

QED.

The regularity of a rectangular interleaver, while making it very simple to implement, causes problems when used in a Turbo code. For certain input patterns of weight $w = 4$, when writing in the rectangular interleaver, the four 1's are at the corners of a square or rectangle, the sides of which have lengths equal to a multiple of p (the length of the impulse response of the RSC component code). An example of this is shown in Fig. 7.1. For such cases, the parity sequences for both the interleaved and non-interleaved stream will have a low weight, impairing the performance of the resulting Turbo code. This phenomenon is well documented in the literature [Berrou & Glavieux, 1996].

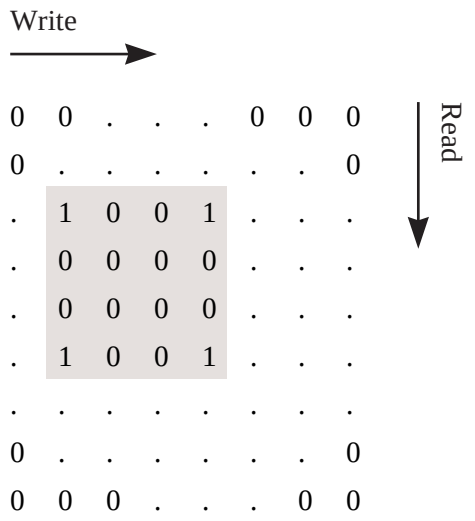


Figure 7.1: Example weight-4 sequence not broken by a rectangular interleaver

7.1.2 Berrou-Glavieux Interleaver

The interleaver used by Berrou and Glavieux in the first Turbo code to appear in the literature is a rectangular interleaver modified to avoid the known flaws. For an $\varpi \times \varpi$ interleaver memory (ϖ being a power of two), the information is written row-wise and read pseudo-randomly as described by

$$i_r = \left(\frac{\varpi}{2} + 1\right)(i + j) \pmod{\varpi} \tag{7.8}$$

$$\xi = (i + j) \pmod{8} \tag{7.9}$$

$$j_r = [P(\xi).(j + 1)] - 1 \pmod{\varpi} \tag{7.10}$$

where i, j are respectively the row and column for writing and i_r, j_r are the row and column for reading. Thus the interleaver mapping is given by

$$\lambda(i.\varpi + j) = i_r.\varpi + j_r \tag{7.11}$$

$P(\xi)$ is a number, relatively prime with ϖ, which is a function of the line address ξ. For all cases where ϖ is a power of two, the function $P(\xi)$ as given in Table 7.1 may be used. This is the function used in the original Turbo code by Berrou *et al.* [1993], but was first documented in a subsequent paper [1996].

Note that reading is performed diagonally in order to avoid possible effects of a relation between ϖ and the period of puncturing. Also, a multiplying factor $\left(\frac{\varpi}{2} + 1\right)$ is used to prevent two neighbouring data written on two consecutive lines from remain-

ξ	$P(\xi)$
0	17
1	37
2	19
3	29
4	41
5	23
6	13
7	7

Table 7.1: Pseudo-Random Function for Berrou-Glavieux Interleaver

ing neighbours in reading. Together with the pseudo-randomness introduced by $P(\xi)$, this avoids the input weight $w = 4$ problem associated with the rectangular interleaver.

7.1.3 Helical Interleaver

This type of interleaver, which is another modification of the rectangular interleaver, was originally proposed by Barbulescu [1995; 1996]. For a helical interleaver with R rows and C columns[2], data is written row-wise and read diagonally, starting from the bottom-left entry, as specified by

$$\lambda(t) = i_r.C + j_r \tag{7.12}$$

$$i_r = C.R - 1 - t \pmod{R} \tag{7.13}$$

$$j_r = t \pmod{C} \tag{7.14}$$

It can be shown that the interleaver is odd-even if C is even. Also, the interleaver is correctly-terminating if C is a multiple of $\nu + 1$ for RSC constituent codes with a 'full' feedback polynomial (i.e. one which adds all the delayed bits at the encoder input). A further restriction on the interleaver construction is that R and C have to be relatively prime.

[2]Barbulescu refers to the number of columns as the depth of the interleaver.

7.2 Reference Interleavers

7.2.1 Uniform Interleaver

This is by definition the average of all possible interleavers, assuming a random choice with a uniform distribution. It is useful because a number of bounds actually assume this kind of interleaver – thus, simulating a Turbo code with a uniform interleaver enables a fair comparison with such bounds. It is also indicative of a performance that should be achievable without too much effort.

The formal definition of such an interleaver implies that the performance of the resulting Turbo code can be computed by averaging the results of Turbo codes with all possible interleavers. While this solution is impractical (there are $\tau!$ different interleavers for a block of size τ), we can achieve a statistically valid result by simulating a large number of blocks with several different interleavers. In our implementation, we simulate a Turbo code with a different random interleaver for each frame. This means that:

- We assume that the sought result actually exists – that is, we assume that the mean of the samples converges as the number of samples taken is increased.
- The distribution of the random interleavers is flat – that is, all interleavers are equally probable for any frame being simulated.

While the simulation time is increased for a uniform interleaver (compared with other Turbo codes with the same parameters but a fixed interleaver), we have not encountered cases where the assumption taken is not valid. Also, the simulations agree with bounds which assume a uniform interleaver (see Section 5.5).

7.2.2 Flat Interleaver

The flat interleaver simply replicates the incoming sequence. It is arguably the worst choice for an interleaver, but may be used to gain insight into the interleaver design

problem. The mapping function for such an interleaver is trivial:

$$\lambda(t) = t \tag{7.15}$$

Note that the flat interleaver is always odd-even and correctly-terminating.

7.2.3 Barrel-Shifting Interleaver

The intention of the Barrel-Shifting interleaver is to separate the interleaved bits from the original bits without disturbing their order. The permutation function for this interleaver is simply to transpose all input bits by a fixed number of positions in the output sequence, wrapping at the frame boundary. This is defined by

$$\lambda(t) = t + \zeta \pmod{\tau} \tag{7.16}$$

where τ is the block size and ζ is the shift distance. This interleaver can be made odd-even by choosing ζ to be even. Also, for a correctly-terminating interleaver, ζ needs to be a multiple of p, the period of the RSC component code.

7.2.4 One-Time Pad Interleaver

Rather than shuffling the information sequence in time, the OTP interleaver adds a known random sequence (the one-time pad) to the information sequence. In this way, any correlation between the parity sequences associated with the straight and interleaved frames is broken, while the interleaver delay is zero. Possible variations include making the interleaver correctly-terminated (by constraining the random sequence to be itself a correctly terminated information sequence), renewable (i.e. having a different random sequence for each data frame), or by combining the OTP interleaver with a shuffling interleaver. Note that because the OTP interleaver does not shuffle the bits along the time axis, this interleaver must necessarily be odd-even.

```
procedure s-random
begin
    Generate an empty interleaver array of size τ
    for i = 0 to τ − 1
        repeat
            Generate a random integer in [0, τ − 1]
        until distance between generated integer and S previously generated
            integers is greater than S
    end for
end
```

Figure 7.2: S-Random Interleaver – Design algorithm

7.3 Optimised Interleavers

7.3.1 S-random Interleaver

Practically all Turbo code interleaver design techniques in the literature are based on the S-random interleaver generation algorithm proposed by the JPL team [Divsalar & Pollara, 1995a]. This interleaver design is a randomly chosen interleaver with a restriction on its spread. The algorithm for choosing an interleaver with spread S is given in Fig. 7.2. The searching time for this algorithm increases with S and is not guaranteed to finish successfully. Divsalar and Pollara have observed, however, that choosing $S < \sqrt{\frac{\tau}{2}}$ usually produces a result in a reasonable time.

The main problems with this technique are that it is not guaranteed to produce the required interleaver and that it only aims at achieving a spread S. Seen in IODS space, this means that a square of side S starting at the origin must be kept empty. The distribution of points outside this square is not considered. Usually, with a large interleaver that is randomly chosen, the distribution outside the spread boundary is approximately flat. This can be seen in Fig. 7.3, which shows the IODS for an S-random interleaver of size $\tau = 1024$. Compared with interleavers having higher regularity, such as the Berrou-Glavieux interleaver of Fig. 6.6 (page 76), the flattening of the IODS reduces the peak frequency. For example, the peak frequency of the Berrou-Glavieux interleaver is about eight times that of the S-random interleaver.

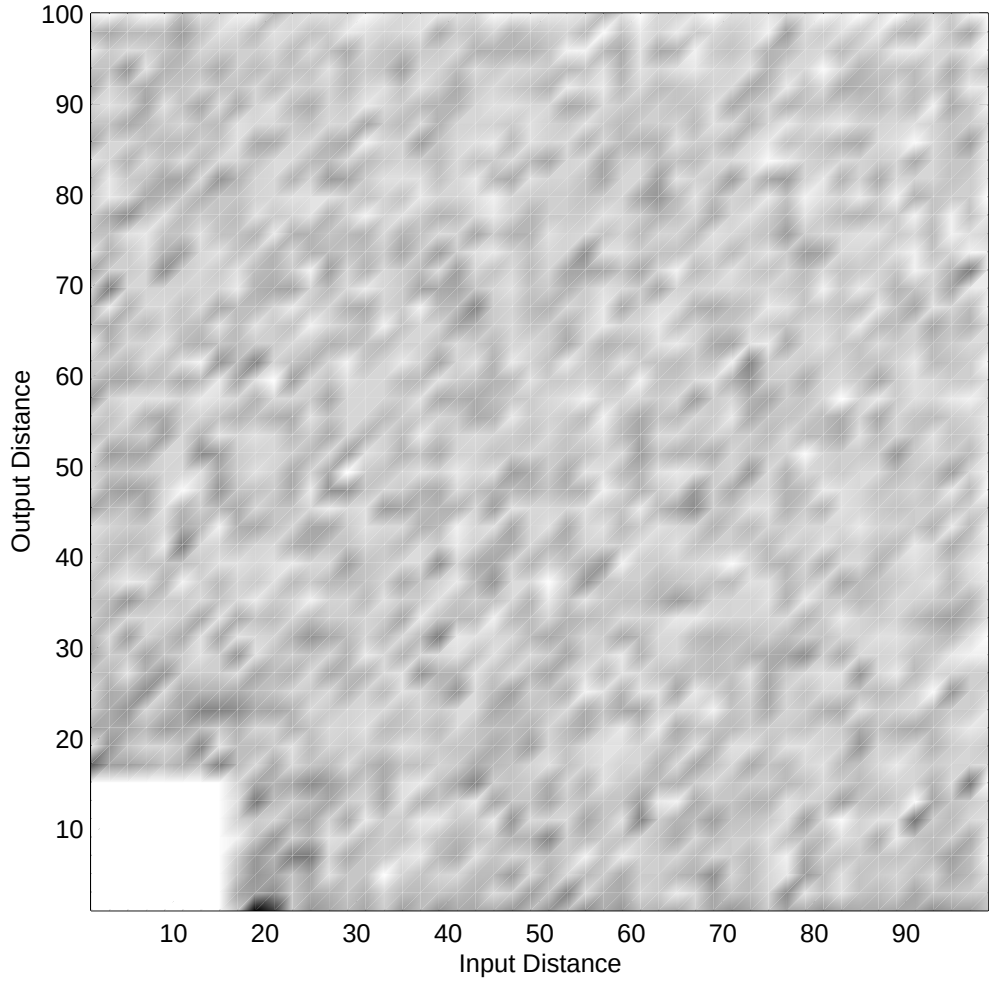


Figure 7.3: IODS for S-random Interleaver

7.3.2 Simulated Annealing Interleaver

In order to simultaneously optimise the various requirements, one possibility is to define an energy function based on those requirements and use the simulated annealing algorithm. Although this is not guaranteed to converge to an optimal solution, in our experience it can be used with success in various optimisation problems with large dimensionality (as in this case). Also, such an interleaver would have the advantage of appearing similar to a random one.

In our investigations, a number of error functions have been tested. The main design criterion in our case was a modification of the 'high spreading factor' criterion used by the JPL team in their S-random interleaver. While we aimed for a high spreading factor (at least 5ν, where ν is the component code's memory order), we also wanted the worst-case spreads to have low multiplicity. The limit of 5ν was reached after analysing the

procedure *simulated annealing*
begin
Initialise state $X = X_0$
Initialise temperature $T = T_0$
repeat
repeat
Choose X', a perturbation of X
Let $\Delta E = Energy(X') - Energy(X)$
if $\Delta E < 0$ **or** $random[0,1) < e^{-\frac{\Delta E}{T}}$
Change state $X = X'$
end if
until several state changes or too many iterations
Update temperature according to annealing schedule
until final temperature reached or configuration is stable
end

Figure 7.4: Simulated Annealing – Basic algorithm

confidence at the output of a MAP decoder when the input had a single hard error. It is not difficult to constrain the interleaver in order to make it correctly-terminating. Since this design technique works directly on the interleaver's Input-Output Distance Spectrum (IODS) rather than restricting only particular input-output distance conditions, it should result in improved performance, because we are not exclusively considering weight-2 input sequences.

7.3.2.1 Interleaver Design with Simulated Annealing

The Simulated Annealing (SA) algorithm for global minimisation is now a well-proven and accepted technique, and has been used successfully in various applications for minimisation of a combinatorial (rather than a continuous) function [Press *et al.*, 1992]. The method is a close analogy with thermodynamics, particularly with the way that slowly cooled crystals achieve a state of minimum energy.

The minimisation algorithm implemented in our case is the Metropolis Algorithm, where a downhill move is always performed, while an uphill move is accepted with a probability $e^{\frac{E_{new}-E_{old}}{T}}$, where T is the temperature, E_{old} is the energy before the move and E_{new} is the energy after the move being considered. In our case, we design interleavers according to a predefined set of requirements by definining an energy function

based on those criteria. Temperature is a virtual value which is decreased exponentially. The basic algorithm is outlined in Fig. 7.4. The parameters that can be varied include:

- the energy function
- the perturbation scheme
- the initial code assignment X_0
- the initial and final temperatures
- the annealing schedule
- the number of iterations allowed at every temperature
- the number of state changes allowed per temperature (if less than the maximum number of iterations)

It is generally preferred to have a perturbation scheme and energy function chosen such that the variation of energy at every perturbation be as smooth as possible. Also, the perturbation scheme must be chosen so as not to exclude any part of the search space. We have attempted to meet these conditions in our case by restricting perturbations to a swap of two random interleaver entries.

The initial assignment X_0 should not make any difference in the result if the other parameters are chosen correctly. This is because at the initial (sufficiently high) temperature, the system's energy will rise considerably as most perturbations will be accepted. Thus, a random interleaver should be as good as any other. This has been verified experimentally.

To allow the system to migrate towards a high-energy state in the initial phase, the starting temperature should be chosen such that $T_0 \gg \Delta E_{avg}$. This ensures that uphill changes have a high probability of occurrence at initial temperatures. Note that the choice of a good perturbation scheme is necessary such that ΔE has a sufficiently restricted variance.

The algorithm stops when the state configuration becomes stable. This can be implemented by stopping the algorithm after, say, five reductions in temperature occur with-

out any new state being accepted. In the general case, this is enough, but a minimum temperature condition is usually also applied. This minimum temperature should be chosen such that it is sufficiently lower than the temperatures at which the configuration becomes stable in the normal case.

Since the probability of acceptance of an uphill move decreases exponentially with temperature, the annealing schedule is generally a geometric decrease of temperature $T_{i+1} = \alpha T_i$, where α lies in the range $0.90 \rightarrow 0.99$. Values of α closer to 1.0 make the temperature decrease more slowly, and generally results in better annealing, with a penalty to be paid as an increased simulation time.

The maximum number of iterations to be performed at every temperature should be large enough to allow the configuration to reach an energy state typical of that temperature. In practice, the number of iterations should be such that a significant proportion of the possible perturbations are tried. The only adverse effect of choosing too high a value is a proportional increase in computation time. It may be noted that this value is particularly important at low temperatures, where very few state changes are in fact accepted. At high temperatures, most proposed changes will be accepted, so the neccessary number of iterations is lower at high temperatures. This is achieved by adding another limit to the number of iterations – the maximum number of accepted state changes, which should be lower then the maximum number of iterations, typically by an order of magnitude. Thus, at high temperatures, the number of iterations performed is limited by the number of accepted state changes, while at lower temperatures, it is limited by the maximum number of iterations.

To improve the performance of the resulting Turbo code, we need to minimise the number of points close to the origin in the interleaver's IODS. With this in mind, we choose an energy function which sums over all points in the IODS, with increased weight on points close to the origin. This attempts to 'push' bit-pair points away from the origin, also increasing the spread of the interleaver. One particular energy function that can be used is:

$$E = \sum_{i,j} \frac{5.\nu}{\sqrt{(i-j)^2 + [\lambda(i) - \lambda(j)]^2}} \tag{7.17}$$

where $i, j \in [0, \tau - 1]$, $i > j$, τ is the interleaver's size, ν is the encoder memory, and

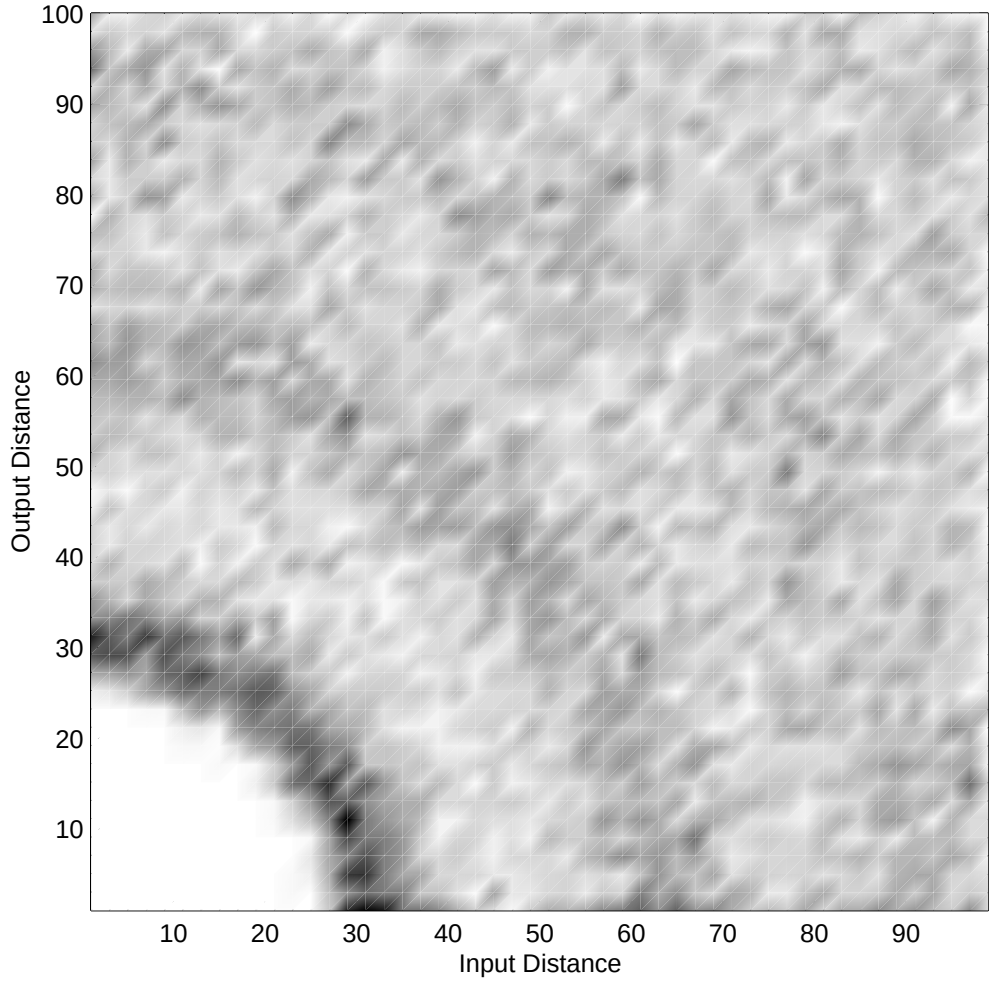


Figure 7.5: IODS for Simulated Annealing Interleaver

$\lambda()$ is the interleaving function. Note that the denominator $\sqrt{(i-j)^2 + [\lambda(i) - \lambda(j)]^2}$ is the radial distance from the origin of the point described by the bit pair i, j.

7.3.2.2 Comparison of IODS with S-Random Interleavers

In contrast with the JPL technique, our algorithm does not guarantee a particular spread. On the other hand, it pushes points away from the origin even beyond the spread boundary. The IODS for an interleaver designed using Simulated Annealing is shown in Fig. 7.5. For comparison, see also the IODS for an S-random interleaver, designed using the JPL algorithm, in Fig. 7.3. Note that the IODS peak frequency of the Simulated Annealing interleaver is comparable with that of the S-random interleaver.

7.3.2.3 Correctly-Terminating Interleavers

For an interleaver to be correctly-terminating, the mapping function must satisfy

$$t = \lambda(t) \pmod{p} \quad \forall\, t \tag{3.12}$$

where p is the encoder's periodicity[3]. To design correctly-terminating interleavers using simulated annealing, it is sufficient to:

- Create an initial interleaver which satisfies Eqn. (3.12).
- Modify the perturbation function to choose two random positions t_1 and t_2 which satisfy:

$$t_1 = t_2 \pmod{p} \tag{7.18}$$

Note that these conditions are tied to the component codes' parameters, and restrict the search space to the set of correctly-terminating interleavers. This makes it harder to increase the interleaver spread.

The IODS for two small interleavers designed using simulated annealing with and without the correct-termination restriction are given in Figs. 7.6 and 7.7 respectively. The interleavers are those used for codes H and F respectively in Chapter 9, and are listed in Table 9.1 on page 111. It can be observed from the graphs how the restriction for termination degrades the interleaver spread.

7.3.2.4 Odd-Even (Mod-s) Interleavers

For an interleaver to be odd-even, the mapping function must satisfy

$$t = \lambda(t) \pmod{s} \quad \forall\, t \tag{3.13}$$

where s is the number of sets in the Turbo code (i.e. the number of parallel component codes). Such an interleaver is supposed to perform better in the presence of puncturing. For an interleaver to be both odd-even and correctly-terminating, it must satisfy

[3] See Section 3.3.1.

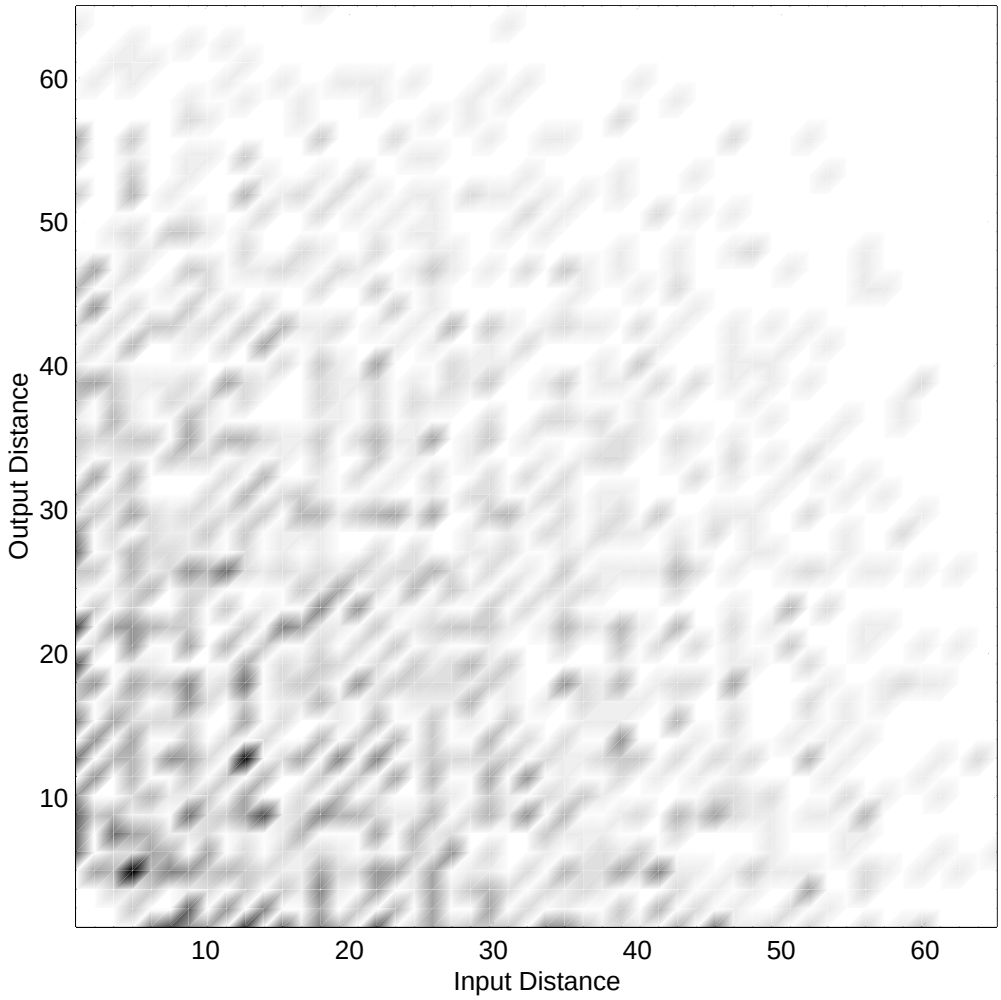


Figure 7.6: IODS for a Small Simulated Annealing Interleaver (Self-Terminating)

both conditions specified above; thus:

$$t = \lambda(t) \pmod{\mathrm{lcm}(s, p)} \quad \forall\, t \tag{7.19}$$

where $\mathrm{lcm}(a, b)$ gives the lowest common multiple of a and b. Since our analysis is restricted to Turbo codes without puncturing, this modification to the simulated annealing algorithm has not been implemented.

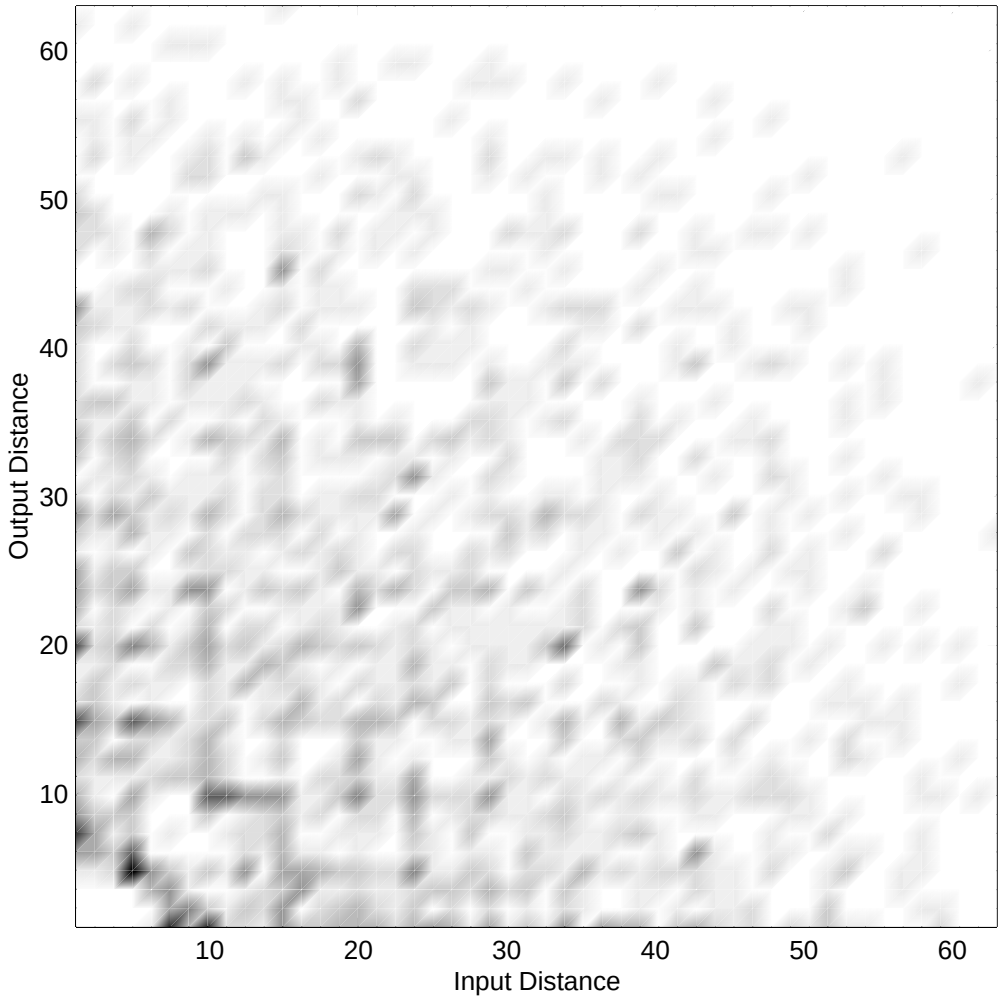


Figure 7.7: IODS for a Small Simulated Annealing Interleaver (Non-Terminating)

Chapter 8

Interleaver Design for Large Frames

8.1 Introduction

In this chapter we consider the interleaver design problem for large block sizes, where the effect of trellis termination is less marked. This is done by comparing the performance of various interleavers with a similar block size; novel interleaver schemes are also used, from which we gather some further insight into the problem. Finally, the performance of an optimised interleaver design technique based on simulated annealing is considered.

8.2 Performance Reference

We restrict ourselves to unpunctured rate-$\frac{1}{3}$ symmetric Turbo codes with encoder memory $\nu = 2$ and generator $(1, 5/7)$. In order to avoid the effects of trellis termination, we also choose a relatively large block size $\tau = 1024$. The effect of interleaver choice for Turbo codes with this component code and block size are indicative of what can be expected with other good component codes and larger block sizes. The design problem of choosing the best component codes has been tackled by the JPL team [Divsalar & Pollara, 1995b].

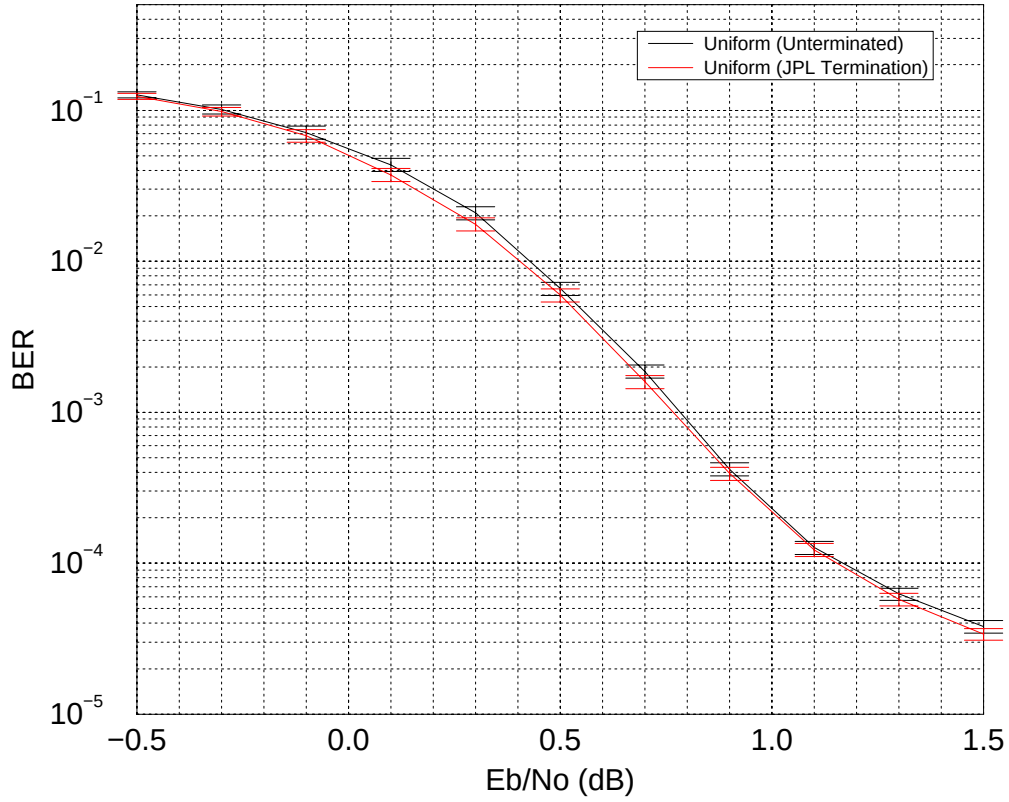


Figure 8.1: Turbo code BER simulation (large block size) – Uniform interleaver

As a reference for performance, we implement a uniform interleaver by using a different random interleaver for every block simulated[1]. To confirm the validity of the statement that trellis termination does not have a significant effect at the chosen block size, we simulate the uniform interleaver with and without termination. To terminate the uniform interleaver we use the scheme proposed by the JPL team[2]. The Bit Error Rate (BER) and Frame Error Rate (FER) results for these two Turbo codes (E and F in Table 8.1 on page 104) are shown respectively in Figs. 8.1 and 8.2; all simulations were performed using 10 iterations, with a target tolerance of $\pm 10\%$ at a confidence of 95% – these tolerance limits are also shown in the graphs. As expected, trellis termination does not significantly affect the performance of Turbo codes with encoder memory $\nu = 2$ at this block size.

[1] See Section 7.2.1.

[2] See Section 2.3.6.

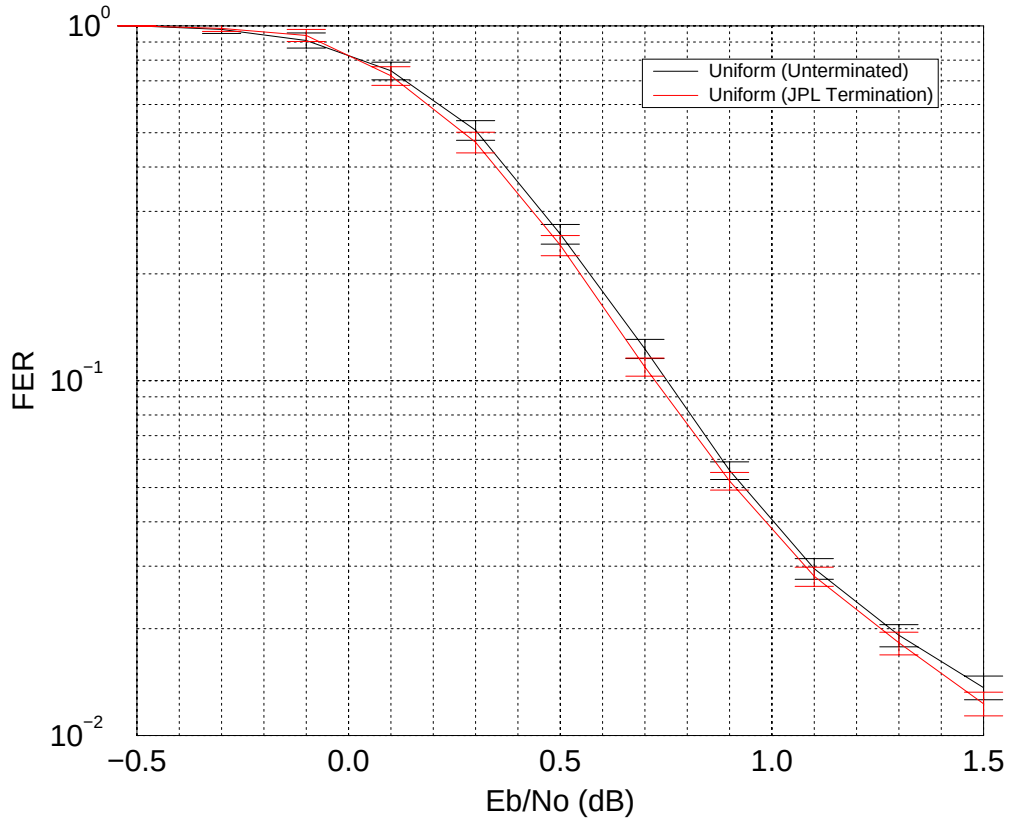


Figure 8.2: Turbo code FER simulation (large block size) – Uniform interleaver

8.3 Regular Interleavers

It has mostly been argued that interleavers with a high regularity perform poorly in Turbo codes. This is confirmed in Figs. 8.3 and 8.4 where we simulate Turbo codes with Square, Rectangular, and Helical interleavers (codes A, B, and C in Table 8.1). For comparison, the performance of a uniform interleaver (code E) is also shown in the graphs. Note that the Rectangular and Helical interleavers satisfy all the restrictions detailed in [Ramsey, 1970] and [Barbulescu & Pietrobon, 1995] respectively:

Rectangular: for the interleaver with $R = 21$ rows and $C = 49$ columns,

- $R + 1$ and C are relatively prime and $R + 1 < C$.
- The interleaver spread is greater than 5ν.
- Additionally, R and C are both odd, so that the interleaver is odd-even. This allows the same interleaver to be fairly compared with other such interleavers in a study of punctured Turbo codes.

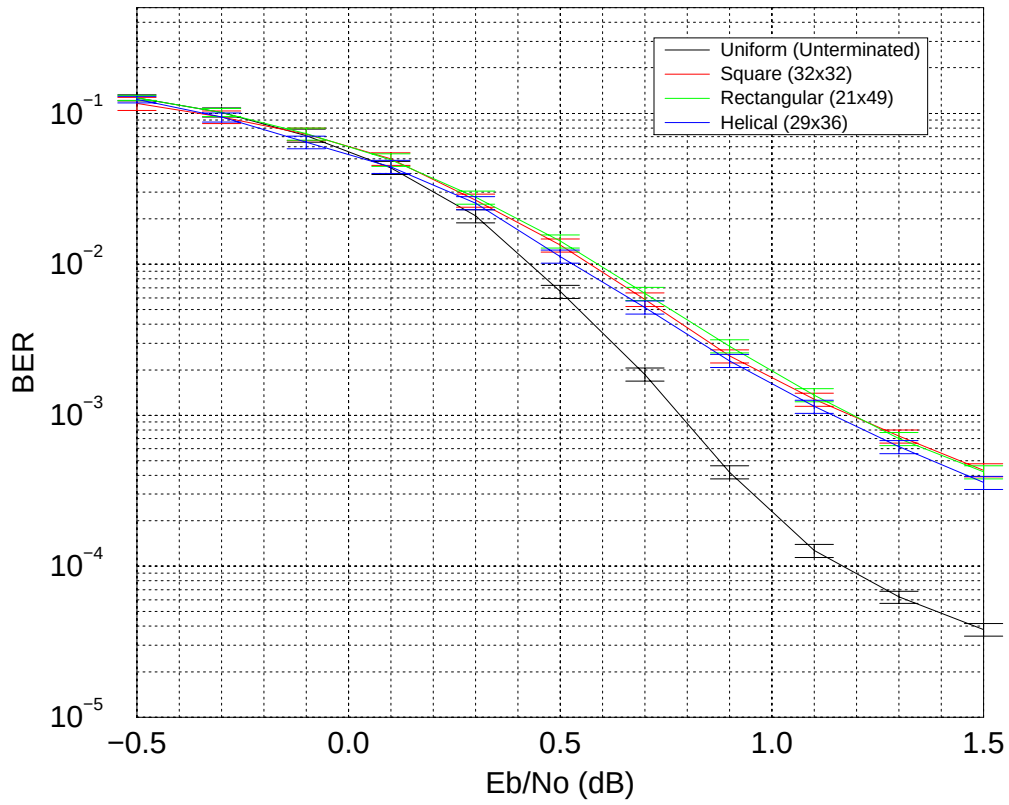


Figure 8.3: Turbo code BER simulation (large block size) – Regular interleavers

Helical: for the interleaver with $R = 29$ rows and $C = 36$ columns,

- R and C are relatively prime.
- C is a multiple of $\nu + 1$ and the RSC code's feedback polynomial is full, making the interleaver simile. This allows the same tail to be used with both the interleaved and non-interleaved sequences. In our implementation, we use the Tail Not Interleaved scheme proposed in [Barbulescu, 1996], which was shown to give better results.
- Additionally, C is even, so that the interleaver is odd-even. This allows the same interleaver to be fairly compared with other such interleavers in a a study of punctured Turbo codes.

It is interesting to note that the BER performance improvement of the Helical interleaver over the Square and Rectangular interleavers is minimal, and can probably be attributed mostly to the termination. Its FER performance, however, is significantly better, though still far from the uniform interleaver.

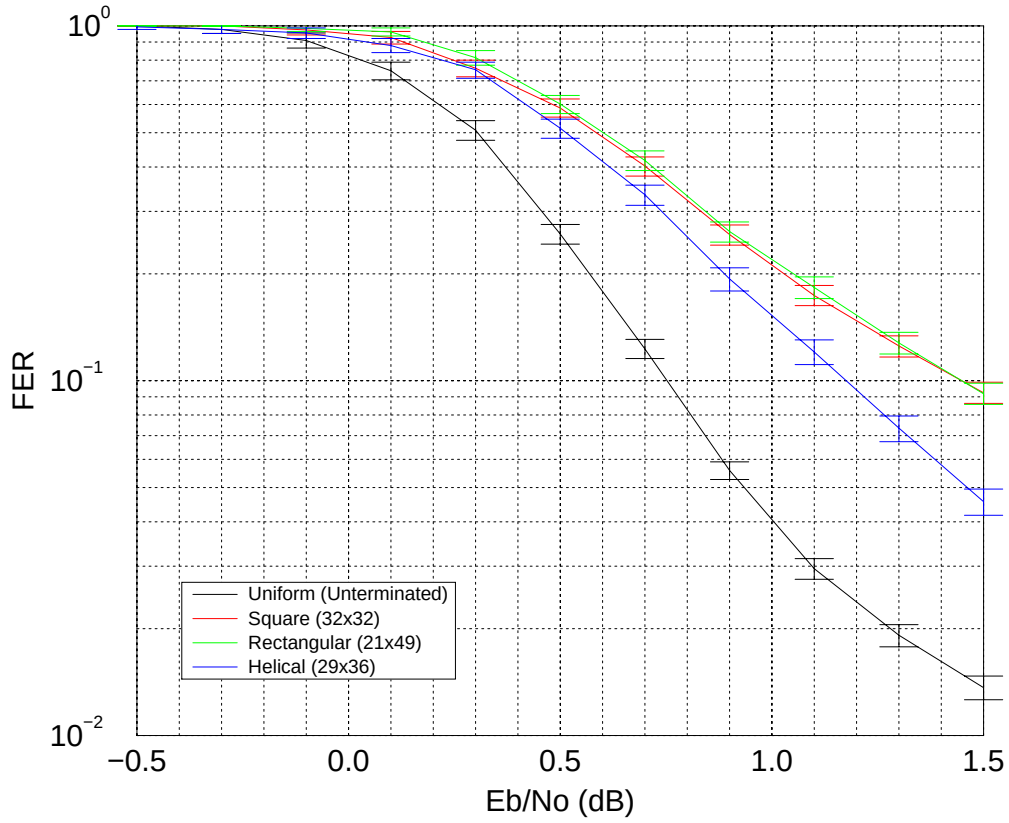


Figure 8.4: Turbo code FER simulation (large block size) – Regular interleavers

8.4 Randomised Interleavers

The interleaver originally used by Berrou *et al.* , described in [Berrou & Glavieux, 1996], is essentially a Square interleaver with some pseudo-random perturbations. However, its performance is significantly better than a regular Square interleaver with the same dimensions. A comparison between the Berrou-Glavieux interleaver (code D) and the Square interleaver (code A) is shown in Figs. 8.5 and 8.6. For comparison, the performance of a uniform interleaver (code E) is also shown in the graphs.

8.5 Analysis of Bad Interleavers

8.5.1 Barrel-Shifting Interleaver

Before we attempt to create an optimised interleaver, we identify two parameters that do not improve performance. The first parameter is increasing the distance between a bit's position in the input and its position in the interleaved stream (i.e. $|\lambda(t) - t|$).

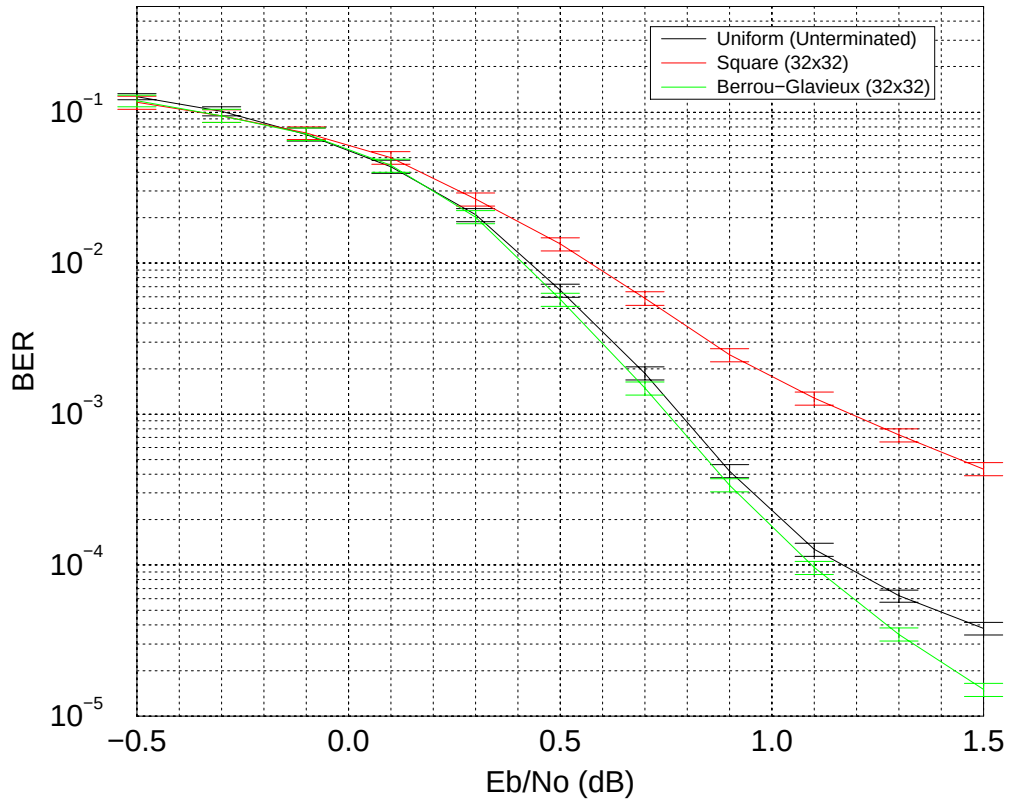


Figure 8.5: Turbo code BER simulation (large block size) – Randomised interleavers

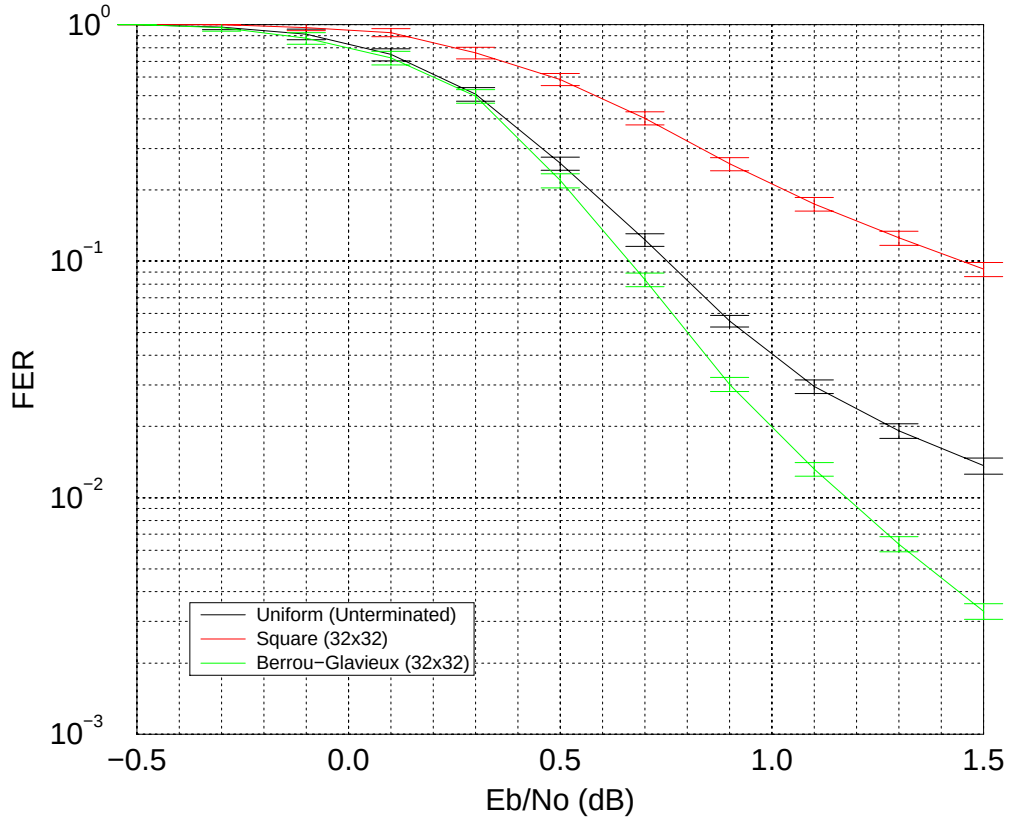


Figure 8.6: Turbo code FER simulation (large block size) – Randomised interleavers

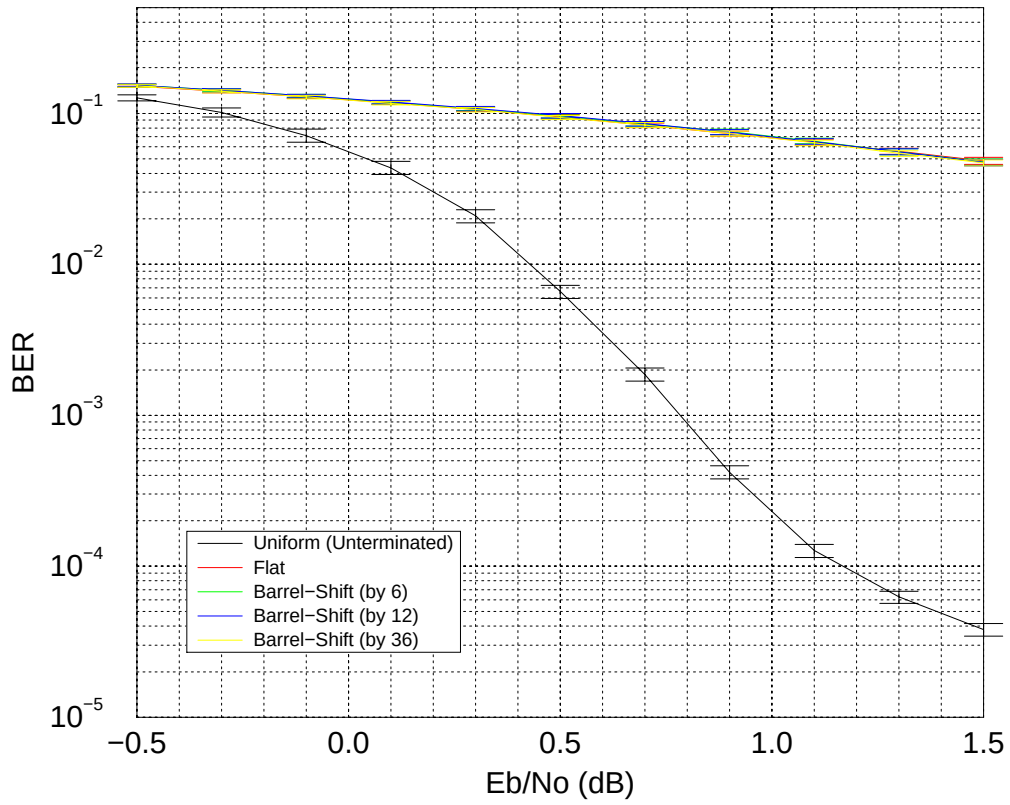


Figure 8.7: Turbo code BER simulation (large block size) – Barrel-shift interleaver

The barrel-shifting interleaver[3] is one which specifies this distance (ζ) while keeping everything else the same as in the input stream. We compare the BER performance of this interleaver for various values of ζ (codes H, I, J) with a flat interleaver (code G) and a uniform interleaver (code E) in Fig. 8.7.

Note how the performance is very poor for such an interleaver, and increasing ζ has negligible effect. Note also how the performance of this interleaver is practically identical to that of a flat interleaver (which can also be considered as a special case barrel-shift interleaver, with $\zeta = 0$). The FER performance is not shown because the flat and barrel-shifting interleavers give a FER of 100% within the SNR range considered.

8.5.2 One-Time Pad Interleaver

The second interleaver type considered is the OTP interleaver[4], where the input stream is not permuted in time, but rather has a random sequence added to it. The perfor-

[3] See Section 7.2.3.

[4] See Section 7.2.4.

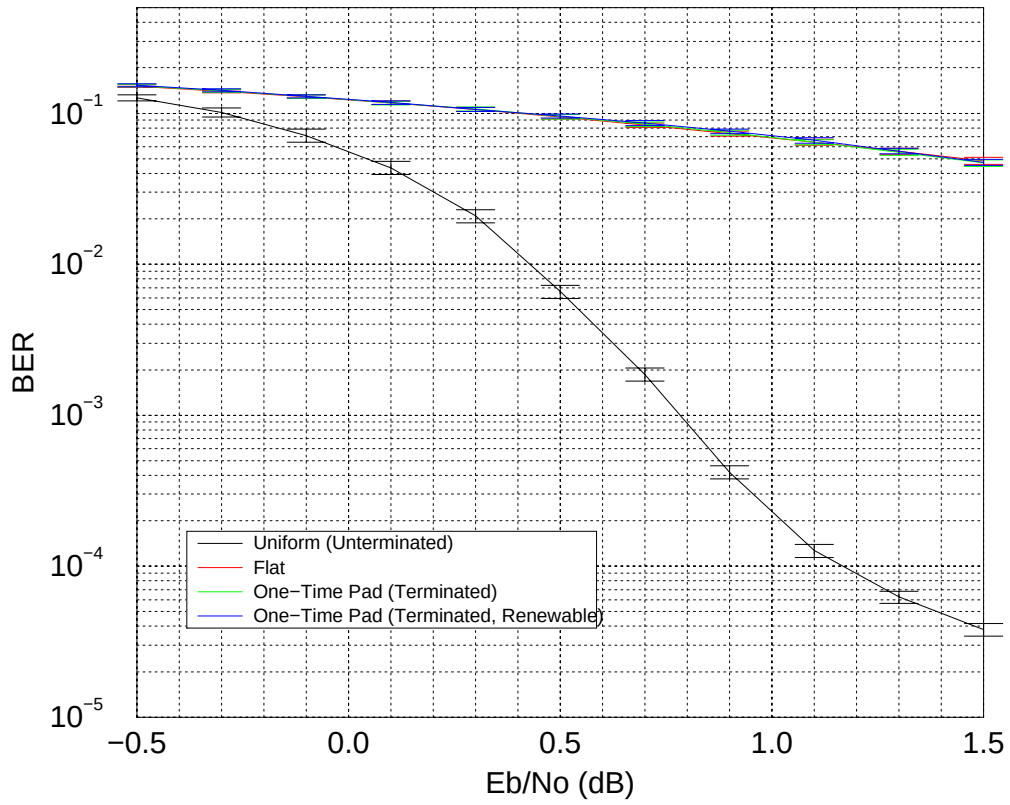


Figure 8.8: Turbo code BER simulation (large block size) – One-time pad interleaver

mance of a random OTP interleaver (code K) is compared to a flat (code G) and a uniform (code E) interleaver in Fig. 8.8. The average performance of all OTP interleavers, computed by using a renewable OTP interleaver (code L) is also included in the graph.

As for the barrel-shift interleaver, the performance of the OTP interleaver is very poor, being practically identical to the flat interleaver. This indicates that it is not the lack of correlation between input and interleaved sequences that gives a Turbo code its good performance, but rather the temporal permutation function of standard interleavers.

8.6 Optimised Interleaver Design

The performance of a Turbo code using the Simulated Annealing interleaver[5] (code M) is shown in Figs. 8.9 and 8.10. For comparison, codes using the uniform interleaver (code E) and the Berrou-Glavieux interleaver (code D) are also shown. As can be seen from the graphs, optimising the interleaver improves both the BER and FER perfor-

[5]The mapping used is included in the enclosed CD (`sai-1024_2-type9.dat`), see Appendix A.

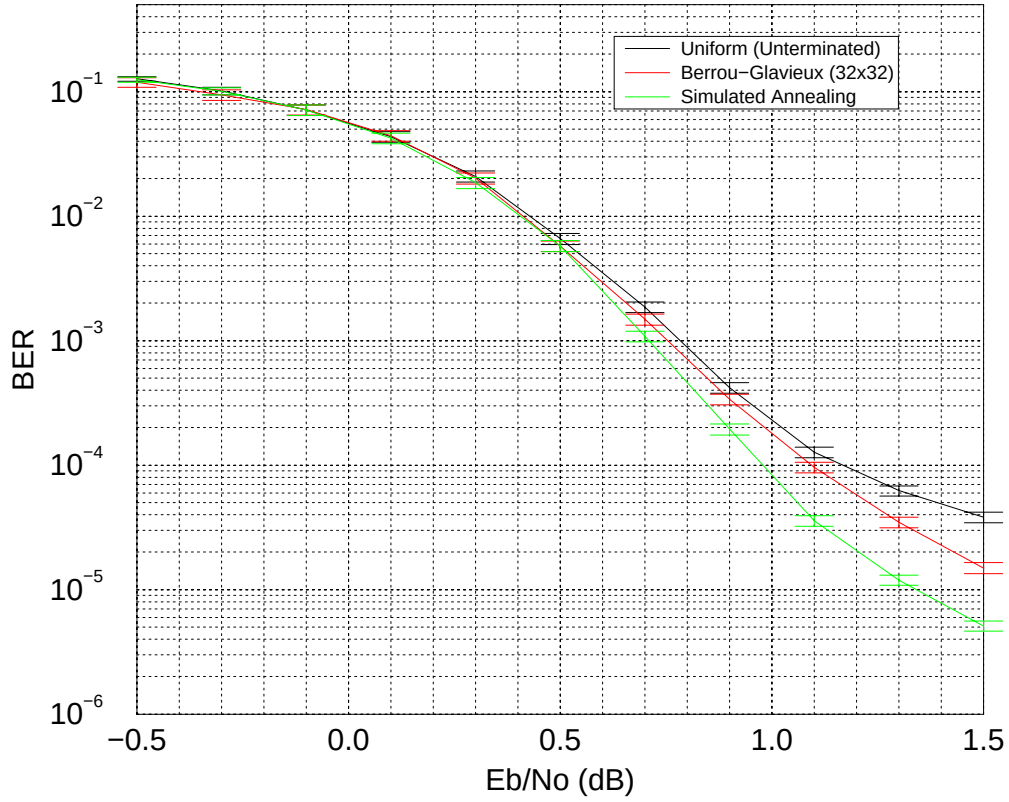


Figure 8.9: Turbo code BER simulation (large block size) – Optimised interleavers

mance of the Turbo code. Code M can achieve a BER of 10^{-5} at $\frac{E_b}{N_0} = 1.35$ dB.

Other energy functions based on the radial distance from the origin of IODS points have also been tried, with very similar results. It seems that the combinatorial restrictions imposed on the interleaver structure do not allow significantly better interleavers (in the sense of increased distance of IODS points from the origin) to be constructed. If there is anything further to be gained in interleaver design for Turbo codes, other factors need to be considered (such as puncturing, or higher-order distance statistics).

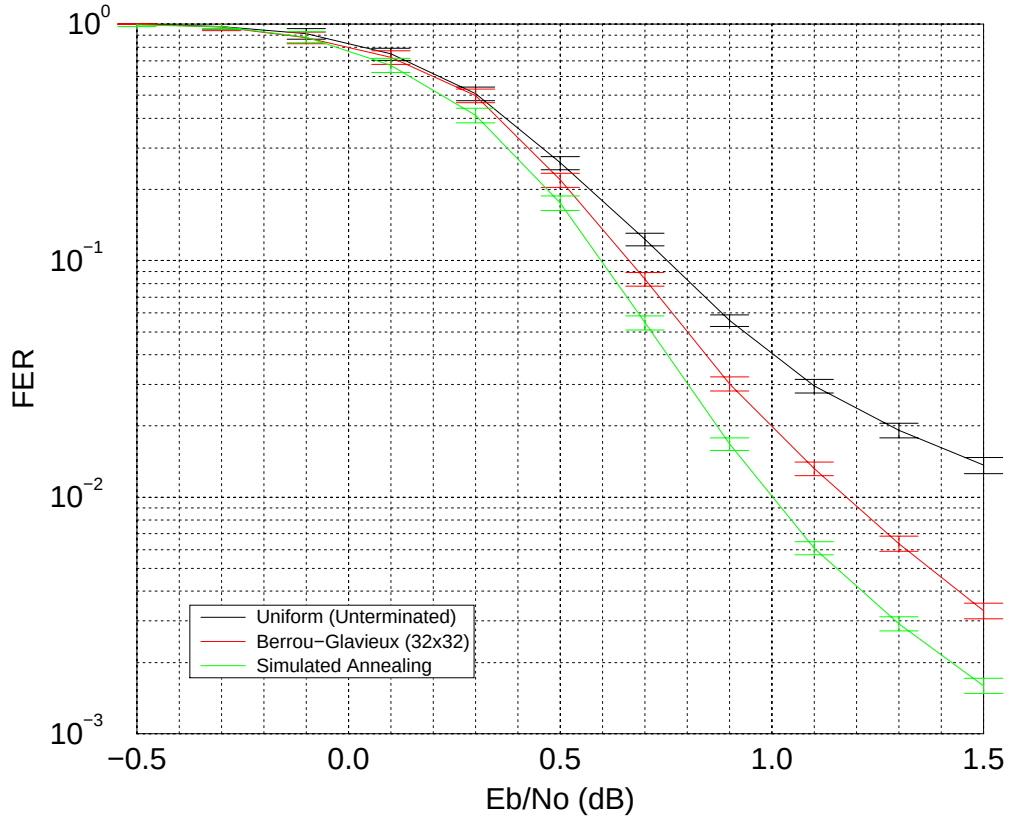


Figure 8.10: Turbo code FER simulation (large block size) – Optimised interleavers

Label	*Type*	*Parameters*	*Termination*	*Code Size*
A	Square	32×32	None	$(3072, 1024)$
B	Rectangular	21×49	None	$(3087, 1029)$
C	Helical	29×36	Simile	$(3138, 1044)$
D	Berrou-Glavieux	32×32	None	$(3072, 1024)$
E	Uniform	(n / a)	None	$(3072, 1024)$
F	Uniform	(n / a)	JPL	$(3078, 1024)$
G	Flat	(n / a)	Simile	$(3078, 1024)$
H	Barrel-Shift	$\zeta = 6$	Standard	$(3078, 1024)$
I	Barrel-Shift	$\zeta = 12$	Standard	$(3078, 1024)$
J	Barrel-Shift	$\zeta = 36$	Standard	$(3078, 1024)$
K	One-Time Pad	Terminated	Standard	$(3078, 1024)$
L	One-Time Pad	Terminated, Renewable	Standard	$(3078, 1024)$
M	Simulated Annealing	(see text)	None	$(3072, 1024)$

Table 8.1: Interleavers for a 1024-bit Frame

Chapter 9

Interleaver Design for Small Frames

9.1 Introduction

In this chapter we consider the interleaver design problem for small block sizes. The effect of various trellis termination schemes is analysed by comparing the performance of the uniform interleaver with and without termination. We also analyse the performance of different termination schemes when used with an optimised interleaver.

9.2 Performance Reference

As in Chapter 8 we restrict ourselves to unpunctured rate-$\frac{1}{3}$ symmetric Turbo codes with encoder memory $\nu = 2$ and generator $(1, 5/7)$. In contrast, though, we choose a very small block size $\tau = 64$ to analyse the effect of termination where it should be most pronounced. The effect of interleaver choice for Turbo codes with this component code and block size are indicative of what can be expected with other good component codes and similarly small block sizes.

We simulate the uniform interleaver with and without termination. To terminate the uniform interleaver we use the scheme proposed by the JPL team[1]. The Bit Error Rate (BER) and Frame Error Rate (FER) results for these two Turbo codes (A and B in Table 9.1 on page 111) are shown respectively in Figs. 9.1 and 9.2; all simulations were

[1] See Section 2.3.6.

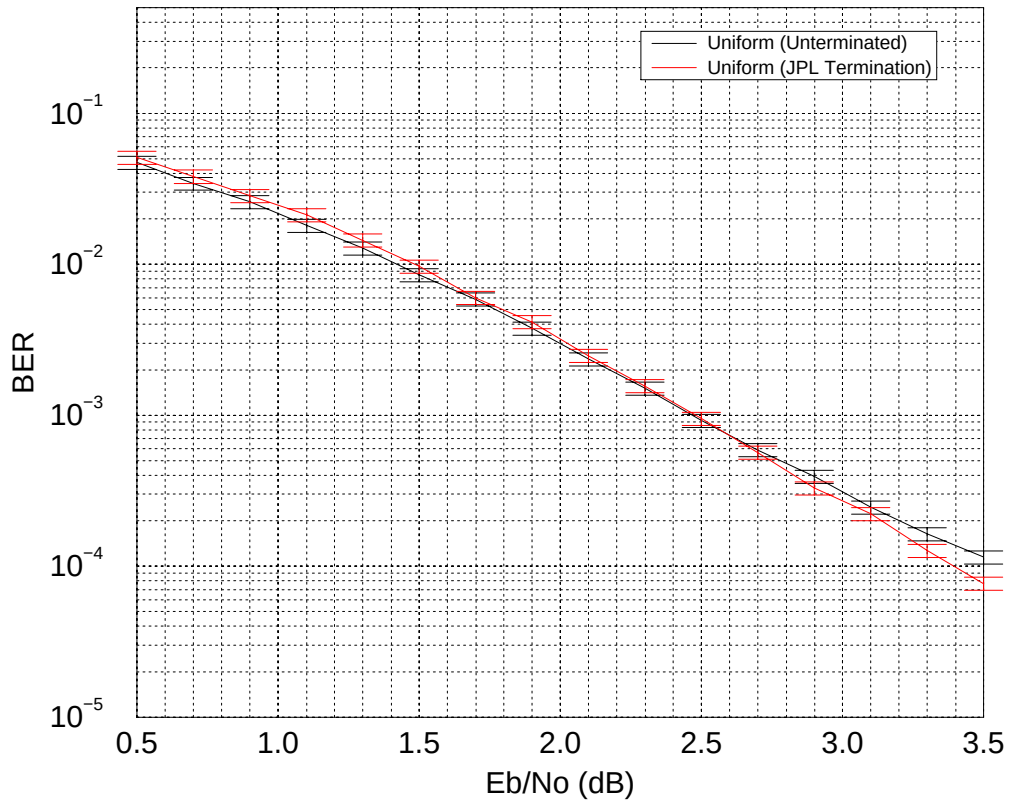


Figure 9.1: Turbo code BER simulation (small block size) – Uniform interleaver

performed using 10 iterations, with a target tolerance of $\pm 10\%$ at a confidence of 95%. While at low SNR the performance of the two codes is almost identical, at high SNR trellis termination improves the performance of Turbo codes with encoder memory $\nu = 2$ at this block size. In particular, the FER performance is markedly better and the BER performance is marginally improved.

9.3 Deterministic Interleavers

The performance of Turbo codes with Square, Berrou-Glavieux, and Helical interleaver types (codes C, D, and E in Table 9.1) are shown in Figs. 9.3 and 9.4. For comparison, the performance of the terminated uniform interleaver (code B) is also shown in the graphs. The Helical interleaver satisfies all necessary restrictions [Barbulescu & Pietrobon, 1995]:

- R and C are relatively prime.

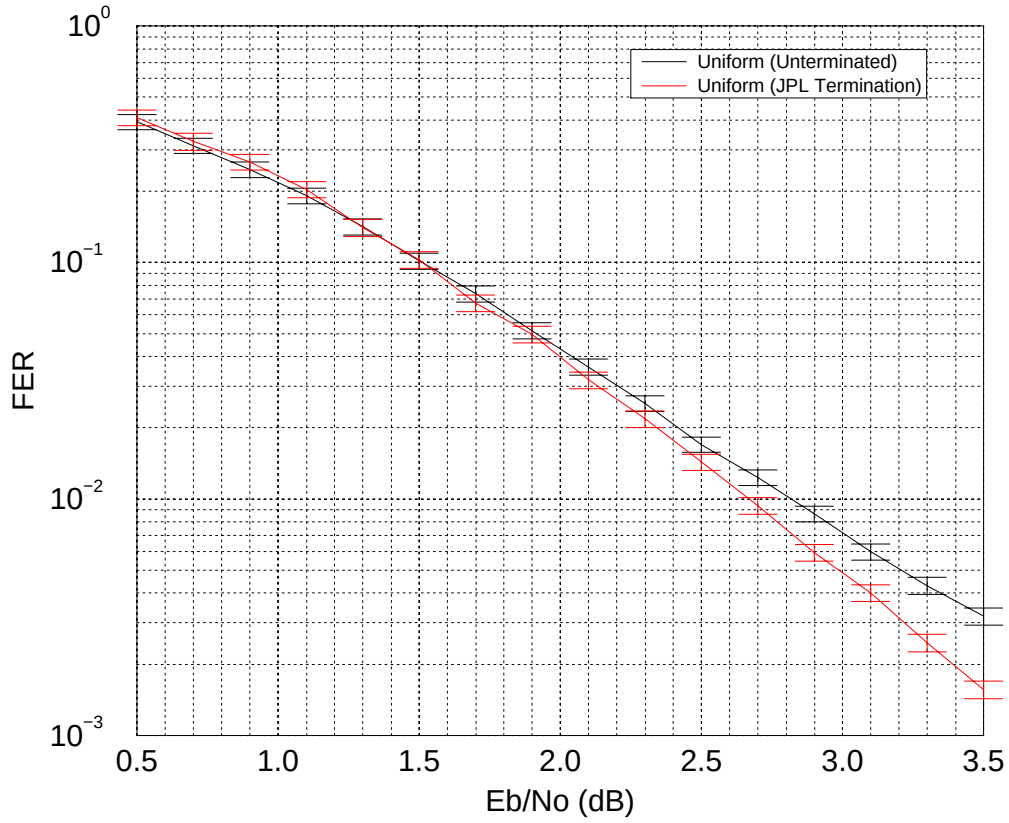


Figure 9.2: Turbo code FER simulation (small block size) – Uniform interleaver

- C is a multiple of $\nu + 1$ and the RSC code's feedback polynomial is full, making the interleaver simile.
- Additionally, C is even, so that the interleaver is odd-even. This allows the same interleaver to be fairly compared with other such interleavers in a a study of punctured Turbo codes.

As for Turbo codes with large interleavers, the Square interleaver performs worst and the Berrou-Glavieux interleaver performs better than the uniform interleaver. In this case, however, the Helical interleaver performs better than the uniform interleaver both in terms of BER and FER.

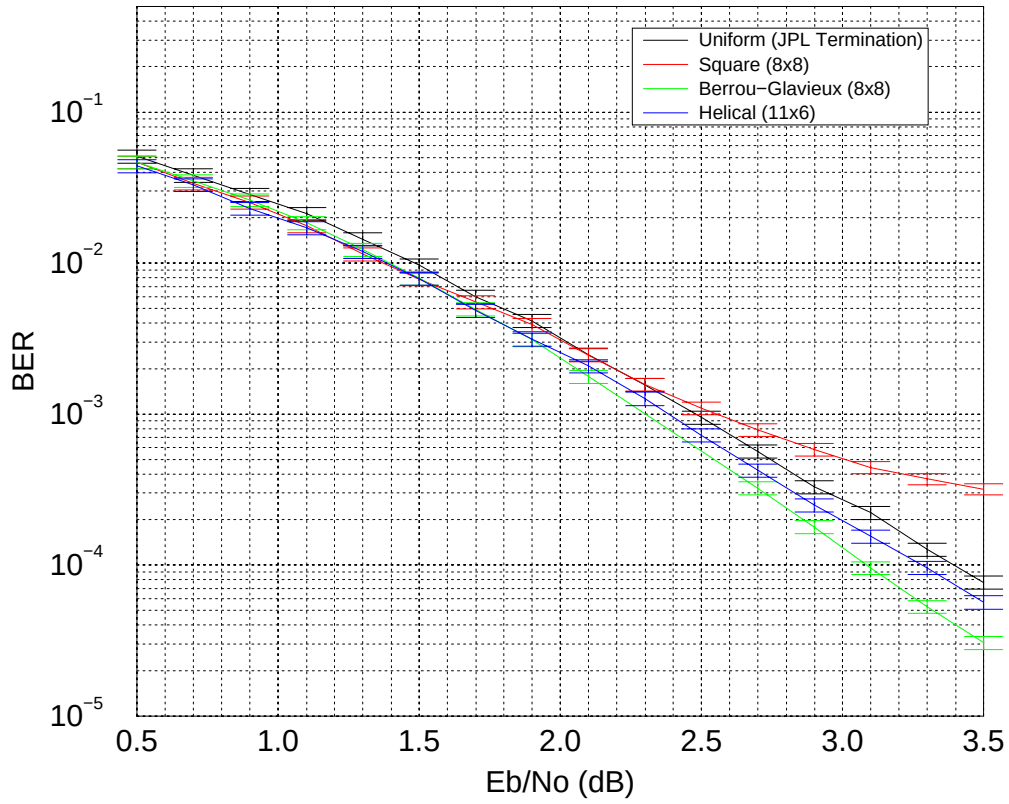


Figure 9.3: Turbo code BER simulation (small block size) – Deterministic interleavers

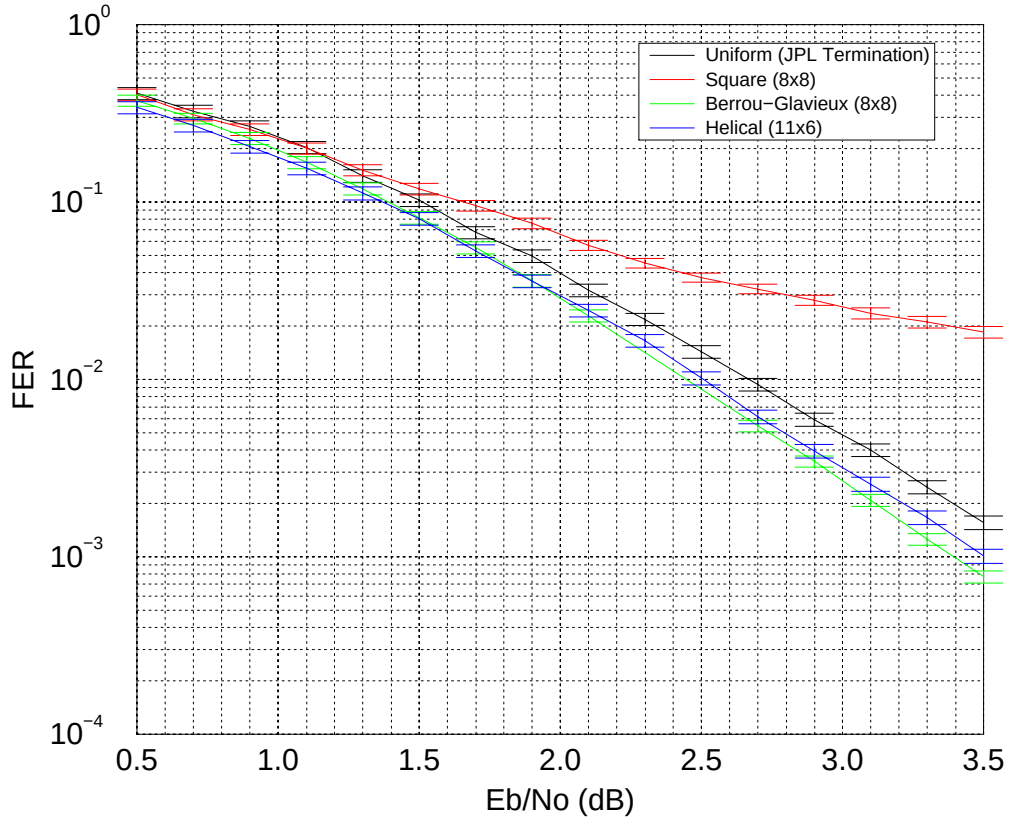


Figure 9.4: Turbo code FER simulation (small block size) – Deterministic interleavers

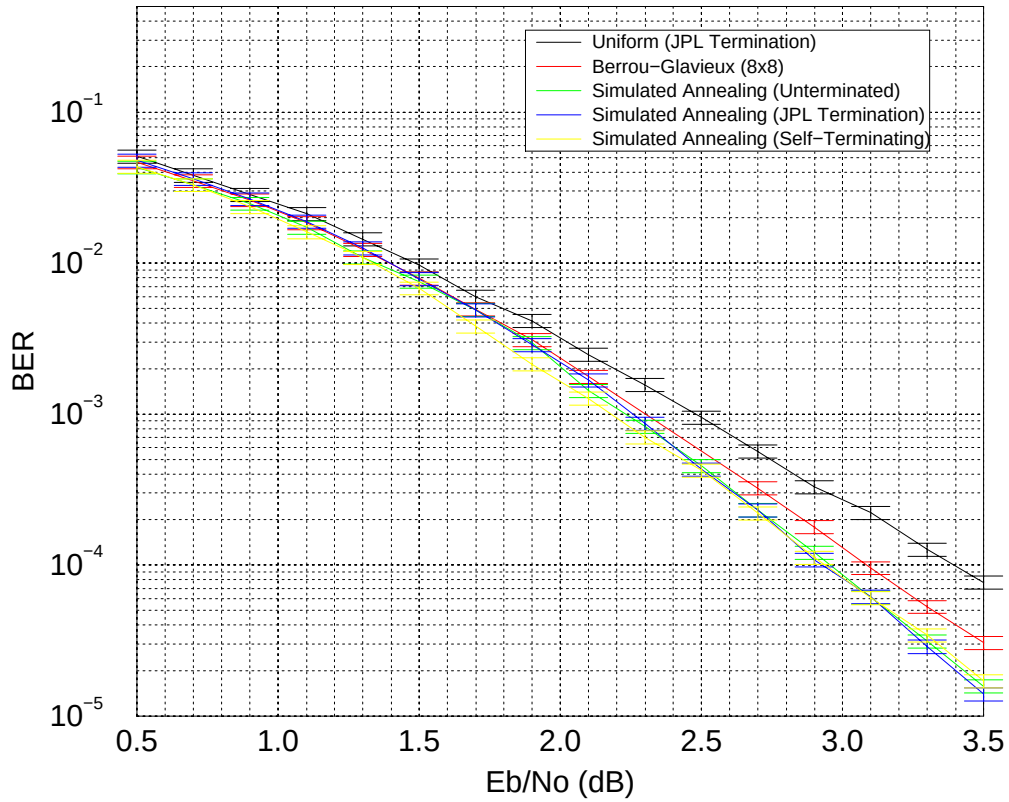


Figure 9.5: Turbo code BER simulation (small block size) – Optimised interleavers

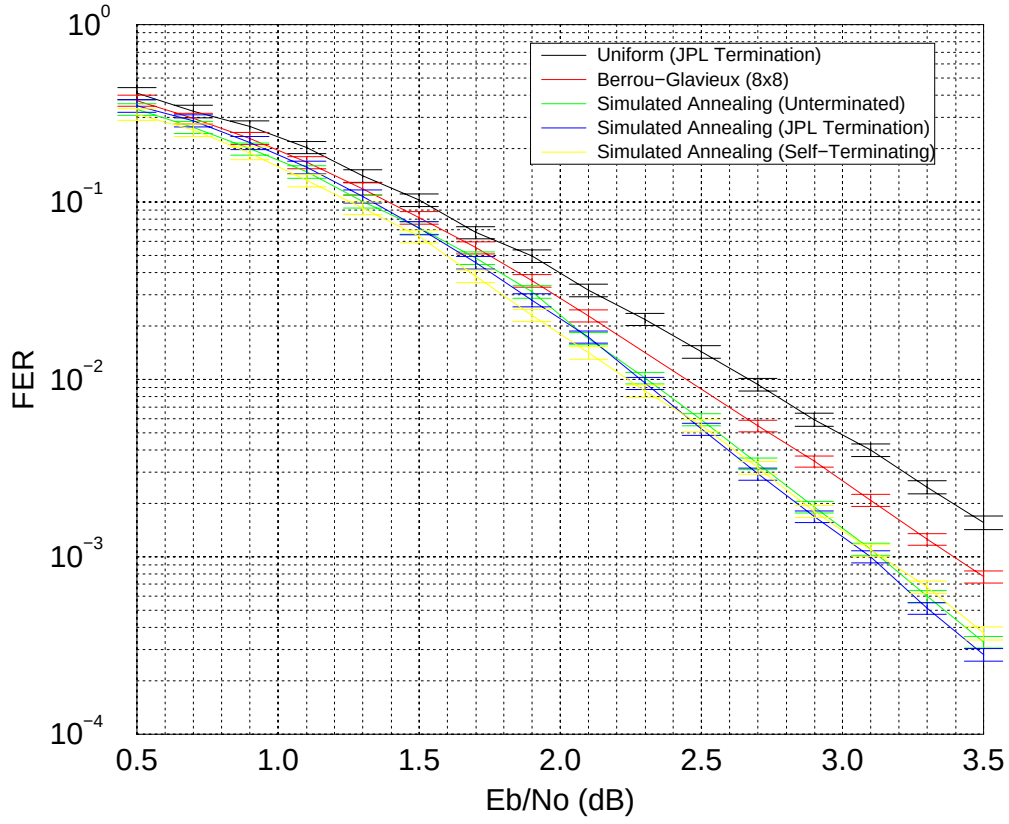


Figure 9.6: Turbo code FER simulation (small block size) – Optimised interleavers

9.4 Optimised Interleaver Design

The performance of Turbo codes with interleavers designed using Simulated Annealing[2] (codes F, G, and H) are shown in Figs. 9.5 and 9.6. For comparison, codes using the terminated uniform interleaver (code B) and the Berrou-Glavieux interleaver (code D) are also shown. As can be seen from the graphs, optimising the interleaver improves both the BER and FER performance. However, with an optimised interleaver design, the effect of termination becomes negligible, both when the JPL termination scheme is used with a non-terminating interleaver (code G) and also when a self-terminating interleaver is designed (code H). Consider both cases separately:

- When the JPL termination scheme is used in code G, the same interleaver is used as the one for code F. The only difference is that a further two tail bits are appended to the input and interleaved sequences. Since the BJCR algorithm is used, an unterminated sequence (as in code F) is equivalent to a terminated sequence where no tail information is available. Thus, the information advantage of code G over code F is the (complete) tail information for the input sequence and the tail parity information for the interleaved sequence. However, this extra information comes at a cost of code rate reduction by a factor of $\frac{66}{64}$. It would seem from the simulation results that the information advantage is offset by the code rate reduction for this particular interleaver. It can be seen from the uniform interleaver simulations, however, that on the average the JPL termination scheme does provide a performance improvement. This advantage seems to depend on the particular interleaver used.

- In desigining an interleaver which self-terminates, we restrict the search space[3], effectively reducing the algorithm's ability at increasing the interleaver spread. For this reason, while code H has the advantage of being correctly-terminating (with complete tail information for both decoders), it not only suffers the penalty of code rate reduction (as for code G), but also a small reduction in spread.

[2]The mappings used are included in the enclosed CD (`sai-64_2-type9.dat` for codes F and G, `sai-66_2-type9-term.dat` for code H), see Appendix A.

[3]See Section 7.3.2.3.

Label	*Type*	*Parameters*	*Termination*	*Code Size*
A	Uniform	(n / a)	None	$(192, 64)$
B	Uniform	(n / a)	JPL	$(198, 64)$
C	Square	8×8	None	$(192, 64)$
D	Berrou-Glavieux	8×8	None	$(192, 64)$
E	Helical	11×6	Simile	$(204, 66)$
F	Simulated Annealing	(see text)	None	$(192, 64)$
G	Simulated Annealing	(see text)	JPL	$(198, 64)$
H	Simulated Annealing	(see text)	Self	$(198, 64)$

Table 9.1: Interleavers for a 64-bit Frame

In conclusion, while termination is beneficial for the average interleaver, its effect on optimised interleavers is negligible and not always beneficial. Considering that designing an interleaver without termination is simpler and that there is also a minor gain in code rate, the design of non-terminating interleavers should be preferred for all block sizes.

Chapter 10

Conclusions

10.1 Introduction

In this chapter we give our comments on the Turbo codec and simulation software, detailing the restrictions and advantages of the design followed. We also comment on the interleaver design problem, and give some suggestions for future work.

10.2 Turbo Codec

The major restrictions imposed by our implementation of the Turbo codec system are that the Turbo code must be symmetric and that it cannot be stream-oriented. The BCJR algorithm itself supports stream-oriented decoding because it allows the user to specify the start and end state probabilities, and can be used for any block size (memory permitting). Also, the extension to non-symmetric Turbo codes is supported by the object-oriented architecture; however, its implementation raises non-trivial issues regarding termination (if this is required).

Otherwise, the codec is extremely flexible, allowing the user to specify any interleaver mapping via plug-in modules, including the possibility to load the mapping from a file. Similarly, any puncturing pattern may be used with the modular system implemented. This allows future research on puncture Turbo codes, including the possibility of using non-deterministic puncturing patterns.

10.3 Monte Carlo Simulator

To minimise simulation times for the required accuracy, the simulator automatically stops when the variance of the result drops below a threshold depending on the tolerance and confidence requested. This cut-off assumes that the Monte Carlo samples come from a Gaussian distribution. While this is usually true when a large number of samples is taken, our simulator does not verify that the distribution is indeed Gaussian. The only safety measure is a lower threshold on the number of samples used, a measure that is necessary when simulating at low SNR where the simulation converges very rapidly. Ideally, a Chi-square test should be included to verify the Normality assumption.

10.4 Interleaver Design

We have proposed a new interleaver design technique, and shown that it performs better than other regular and partly-randomised designs. For the case of small interleavers, it is shown that while correct termination improves the performance for an average interleaver, its effect on Turbo codes with optimised interleavers is negligible. Considering that designing an interleaver without termination is simpler and that there is also a minor gain in code rate, the design of non-terminating interleavers should be preferred for all block sizes.

Using our simulated annealing design technique it is easier to include restrictions which make the interleaver correctly-terminating or odd-even. While the S-random algorithm serves well for specifying interleaver spread, we believe that our algorithm is better suited for more sophisticated design criteria.

Utilising some performance enhancement techniques, the complexity of the energy function grows only as $O(\tau)$, making it suitable for use with large block sizes. However, increasing the block size also requires the simulated annealing algorithm to perform more perturbations at every temperature, so that the overall complexity of the design technique actually grows larger than $O(\tau)$.

10.5 Future Work

10.5.1 Improving the Simulated Annealing Design

While we have shown that simulated annealing can be used to design optimised interleavers, the energy functions used are based on the 'high spreading factor' criterion used by the JPL team in their S-random interleaver. Thus, the performance of codes with our interleavers are not expected to be significantly better than those using S-random interleavers.

More sophisticated energy functions will need to be considered if any further improvement is sought. Possible directions of research may include:

- Energy functions that take into consideration the fact that not all weight-two input sequences produce a finite-length parity sequence. Those sequences resulting in a parity of infinite weight should be given less importance since they will not adversely affect the Turbo code's weight spectrum.
- Consideration of sequences of weight greater than two which produce finite length parity sequences should also improve performance, particularly at the lower SNR values. This is, however, a problem of considerable complexity, and requires the energy function to be matched to the particular component codes used.

10.5.2 Interleaver Design for Punctured Codes

Our analysis has taken into consideration only unpunctured rate-$\frac{1}{3}$ symmetric Turbo codes. The problem of interleaver design for punctured codes also necessitates an appropriate interleaver optimisation. Some work has already been done on punctured codes, particularly indicating the usefulness of an odd-even interleaver structure. Modification of the simulated annealing algorithm to create odd-even interleavers is trivial[1], and should provide a preliminary design strategy. The effect of trellis termination on punctured codes needs to be investigated, particularly for small block sizes.

[1] See Section 7.3.2.4.

10.5.3 Advanced Puncturing

Given the flexibility of our Turbo codec and simulation system, the possibility of using more complex puncturing schemes is also open to investigation. In our investigations we have observed that the corrective capability of Turbo codes is not always uniform over all data bits (depending on the interleaver) and also that the importance of the codeword bits are not all identical. Thus, it may be possible to define non-deterministic puncturing patterns of the same rate which improve performance, by puncturing those bits which are less important.

10.5.4 Interleaver Design for Unequal Error Protection

Basing ourselves on the observation that the interleaver structure affects the distribution of corrective power over the set of input bits, we conjecture that it should be possible to design an interleaver in such a way as to create a Turbo code with a required distribution of protection. For example, the spread in a region of the input sequence may be slackened in order to improve the spread in another region.

References

ANDREWS, KENNETH S., HEEGARD, CHRIS, & KOZEN, DEXTER. 1997 (June). *A Theory of Interleavers*. Technical Report TR97-1634. Department of Computer Science, Cornell University.

ANDREWS, KENNETH S., HEEGARD, CHRIS, & KOZEN, DEXTER. 1998 (Aug. 16–21st,). Interleaver Design Methods for Turbo Codes. *Page 420 of: Proceedings of the IEEE International Symposium on Information Theory*.

BAHL, L. R., COCKE, J., JELINEK, F., & RAVIV, J. 1974. Optimal Decoding of Linear Codes for Minimizing Symbol Error Rate. *IEEE Transactions on Information Theory*, Mar., 284–287.

BARBULESCU, SORIN ADRIAN. 1996 (Feb.). *Iterative Decoding of Turbo Codes and Other Concatenated Codes*. Ph.D. thesis, University of South Australia, School of Electronic Engineering, Faculty of Engineering.

BARBULESCU, SORIN ADRIAN, & PIETROBON, STEVEN SILVIO. 1994. Interleaver Design for Turbo Codes. *Electronics Letters*, **30**(25), 2107–2108.

BARBULESCU, SORIN ADRIAN, & PIETROBON, STEVEN SILVIO. 1995. Terminating the Trellis of Turbo-Codes in the Same State. *Electronics Letters*, **31**(1), 22–23.

BENEDETTO, SERGIO, & MONTORSI, GUIDO. 1995a. Average Performance of Parallel Concatenated Block Codes. *Electronics Letters*, **31**(3), 156–158.

BENEDETTO, SERGIO, & MONTORSI, GUIDO. 1995b. Performance Evaluation of Turbo Codes. *Electronics Letters*, **31**(3), 163–165.

BENEDETTO, SERGIO, & MONTORSI, GUIDO. 1996a. Design of Parallel Concatenated Convolutional Codes. *IEEE Transactions on Communications*, **44**(5), 591–600.

BENEDETTO, SERGIO, & MONTORSI, GUIDO. 1996b. Unveiling Turbo Codes – Some Results on Parallel Concatenated Coding Schemes. *IEEE Transactions on Information Theory*, **42**(2), 409–428.

BERROU, CLAUDE, & GLAVIEUX, ALAIN. 1996. Near Optimum Error Correcting Coding and Decoding: Turbo-Codes. *IEEE Transactions on Communications*, **44**(10), 1261–1271.

BERROU, CLAUDE, GLAVIEUX, ALAIN, & THITIMAJSHIMA, PUNJA. 1993 (May). Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes. *Pages 1064–1070 of: Proceedings of the IEEE International Conference on Communications*.

BLACKERT, W. J., HALL, ERIC K., & WILSON, STEPHEN G. 1995. Turbo Code Termination and Interleaver Conditions. *Electronics Letters*, **31**(Jan.), 2082–2083.

COVER, T., & THOMAS, J. 1991. *Elements of Information Theory*. Wiley Interscience, New York.

DIVSALAR, DARIUSH, & POLLARA, FABRIZIO. 1995a (May 15th,). *Multiple Turbo Codes for Deep-Space Communications*. TDA Progress Report 42-121. Jet Propulsion Laboratory, California Institute of Technology.

DIVSALAR, DARIUSH, & POLLARA, FABRIZIO. 1995b (Nov. 15th,). *On the Design of Turbo Codes*. TDA Progress Report 42-123. Jet Propulsion Laboratory, California Institute of Technology.

DIVSALAR, DARIUSH, & POLLARA, FABRIZIO. 1995c (Feb. 15th,). *Turbo Codes for Deep-Space Communications*. TDA Progress Report 42-120. Jet Propulsion Laboratory, California Institute of Technology.

DIVSALAR, DARIUSH, DOLINAR, SAM, & POLLARA, FABRIZIO. 1995 (Aug. 15th,). *Transfer Function Bounds on the Performance of Turbo Codes*. TDA Progress Report 42-122. Jet Propulsion Laboratory, California Institute of Technology.

DOLINAR, SAM, & DIVSALAR, DARIUSH. 1995 (Aug. 15th,). *Weight Distributions for Turbo Codes Using Random and Nonrandom Permutations*. TDA Progress Report 42-122. Jet Propulsion Laboratory, California Institute of Technology.

DOLINAR, SAM, DIVSALAR, DARIUSH, & POLLARA, FABRIZIO. 1998 (May 15th,). *Code Performance as a Function of Block Size*. TDA Progress Report 42-133. Jet Propulsion Laboratory, California Institute of Technology.

FORNEY, JR., G. DAVID. 1966. *Concatenated Codes*. Cambridge, MA: M.I.T. Press.

GRANLUND, TORBJÖRN. 1996 (June). *GNU MP: The GNU Multiple Precision Arithmetic Library*. Free Software Foundation. Version 2.0.2.

HAGENAUER, JOACHIM. 1994. Soft is Better than Hard. *Pages 155–171 of:* BLAHUT, D. (ed), *Communications, Coding and Cryptology*. Kluwer.

HAGENAUER, JOACHIM, & HOEHER, PETER. 1989 (Nov.). A Viterbi Algorithm with Soft-Decision Outputs and its Applications. *Pages 1680–1686 of: Proceedings of GLOBECOM '89*.

HAGENAUER, JOACHIM, ROBERTSON, PATRICK, & PAPKE, LUTZ. 1994 (Oct. 26–28th,). Iterative (TURBO) Decoding of Systematic Convolutional Codes with the MAP and SOVA Algorithms. *In: Proceedings of the ITG Conference on Source and Channel Coding*.

HALL, ERIC K., & WILSON, STEPHEN G. 1998 (Aug. 16–21st,). Convolutional Interleavers for Stream-Oriented Parallel Concatenated Convolutional Codes. *In: Proceedings of the IEEE International Symposium on Information Theory*.

HO, MARK S. C., PIETROBON, STEVEN S., & GILES, TIM. 1998 (Nov.). Interleavers for Punctured Turbo Codes. *Pages 520–524 of: Proceedings of the IEEE Asia-Pacific Conference on Communications and Singapore International Conference on Communication Systems*, vol. 2.

HOKFELT, JOHAN, EDFORS, OVE, & MASENG, TORLEIV. 1999 (June 6–10th,). Interleaver Design for Turbo Codes Based on the Performance of Iterative Decoding. *In: Proceedings of the IEEE International Conference on Communications.*

JERUCHIM, MICHEL C., BALABAN, PHILIP, & SHANMUGAN, K. SAM. 1992. *Simulation of Communication Systems.* Plenum Press, New York.

JUNG, P., & NASSHAN, M. 1994a. Dependence of the Error Performance of Turbo-Codes on the Interleaver Structure in Short Frame Transmission Systems. *Electronics Letters*, **30**(4), 287–288.

JUNG, P., & NASSHAN, M. 1994b. Performance Evaluation of Turbo Codes for Short Frame Transmission Systems. *Electronics Letters*, **30**(2), 111–113.

KNUTH, DONALD E. 1981. *Seminumerical Algorithms.* Second edn. The Art of Computer Programming, vol. 2. Addison Wesley.

LAM. 1996 (Nov. 11th,). *MPI Primer / Developing with LAM.* Ohio Supercomputer Centre, The Ohio State University.

MASSEY, JAMES L. 1984. The How and Why of Channel Coding. *Pages 67–73 of: Proceedings of the 1984 Zurich Seminar on Digital Communications.*

MPI. 1995 (June 12th,). *MPI: A Message-Passing Interface Standard.* Message Passing Interface Forum. Version 1.1.

PEREZ, LANCE C., SEGHERS, JAN, & COSTELLO, JR., DANIEL J. 1996. A Distance Spectrum Interpretation of Turbo Codes. *IEEE Transactions on Information Theory*, **42**(6), 1698–1709.

PIETROBON, STEVEN SILVIO. 1995 (Sept.). Implementation and Performance of a Serial MAP Decoder for use in an Iterative Turbo Decoder. *Page 471 of: Proceedings of the IEEE International Symposium on Information Theory*.

PRESS, WILLIAM H., TEUKOLSKY, SAUL A., VETTERLING, WILLIAM T., & FLANNERY, BRIAN P. 1992. *Numerical Recipes in C: The Art of Scientific Computing.* Second edn. Cambridge Universty Press.

PROAKIS, JOHN G. 1995. *Digital Communications.* Third edn. McGraw Hill International.

RAMSEY, JOHN L. 1970. Realization of Optimum Interleavers. *IEEE Transactions on Information Theory*, **16**(3), 338–345.

REED, MARK C., & PIETROBON, STEVEN SILVIO. 1996. Turbo-Code Termination Schemes and a Novel Alternative for Short Frames. *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications,* Oct. 15–18th,, 354–358.

ROBERTSON, PATRICK. 1994a (Dec.). Illuminating the Structure of Code and Decoder of Parallel Concatenated Recursive Systematic (Turbo) Codes. *Pages 1298–1303 of: Proceedings of GLOBECOM '94.*

ROBERTSON, PATRICK. 1994b (Sept.). Improving Decoder and Code Structure of Parallel Concatenated Recursive Systematic (Turbo) Codes. *Pages 183–187 of: Third Annual International Conference on Universal Personal Communication*.

SHANNON, CLAUDE E. 1948. A Mathematical Theory of Communication. *Bell System Technical Journal*, **23**, 379–423, 623–656.

STROUSTRUP, BJARNE. 1991. *The C++ Programming Language*. Second edn. Addison Wesley.

YUAN, JINHONG, VUCETIC, BRANKA, & FENG, WEN. 1998 (Aug. 16–21st,). Combined Turbo Codes and Interleaver Design. *In: Proceedings of the IEEE International Symposium on Information Theory*.

YUAN, JINHONG, VUCETIC, BRANKA, & FENG, WEN. 1999. Combined Turbo Codes and Interleaver Design. *IEEE Transactions on Communications*, **47**(4), 484–487.

Appendix A

CD Contents

A.1 Overview

The enclosed CD contains:

- An electronic version of this manuscript and of all the included graphs, with an associated viewer. The manuscript, in Adobe Postscript and PDF formats, can be found in the directory `Dissertation`. The graphs (in EPS format) are in `Dissertation/Graphs` and a copy of GhostView and GhostScript can be found in `Dissertation/Viewer`. The graph filenames are listed in Table A.1 along with the figure numbers they refer to and the page where they can be found.

- The source code for the Turbo codec and simulation environment. This can be found in `Source`, together with the Makefiles used to control compilation. A brief description of how to use the Turbo codec and Monte Carlo simulator can be found in Appendix B.

- Detailed simulation results used to plot the graphs in this thesis. These are further described in Section A.2.

- Mapping definitions for all the non-deterministic interleavers used - these were not included in the printed manuscript for reasons of space and legibility. They can be found in directory `Interleavers`.

Figure	*Page*	*Filename*
5.1	59	ber-uncoded.eps
5.2	60	rsc-2_1_3-111_101-52_24.eps
5.3	61	rsc-2_1_3-111_101-52_24-arith.eps
5.4	63	tc-2_1_3-101_111-3006_1000-uniform-null.eps
5.5	64	tc-2_1_3-111_101-208_102-helical17x6-stipple.eps
5.6	65	tc-2_1_3-111_101-3078_1024-montorsi-null.eps
2.2	29	ber-typical.eps
7.3	87	idp-montorsi-1024.eps
7.5	91	idp-sai-1024_2-type9.eps
7.6	93	idp-sai-66_2-type9-term.eps
7.7	94	idp-sai-64_2-type9.eps
6.1	70	cc-2_1_2-11_10-200_99.eps
6.2	71	irwef-nrc-2_1_2-11_10-20-100.eps
6.3	72	irwef-rsc-2_1_2-11_10-20-100.eps
6.4	73	q-0.333333-1_20.eps
6.5	75	idp-rectangular32x32.eps
6.6	76	idp-berrou-32.eps
6.7	78	confidence.eps
8.1	96	tc-2_1_3-111_101-3xxx_1xxx-uniform-ber-null.eps
8.2	97	tc-2_1_3-111_101-3xxx_1xxx-uniform-fer-null.eps
8.3	98	tc-2_1_3-111_101-3xxx_1xxx-regular-ber-null.eps
8.4	99	tc-2_1_3-111_101-3xxx_1xxx-regular-fer-null.eps
8.5	100	tc-2_1_3-111_101-3xxx_1xxx-berrou-ber-null.eps
8.6	100	tc-2_1_3-111_101-3xxx_1xxx-berrou-fer-null.eps
8.7	101	tc-2_1_3-111_101-3xxx_1xxx-shift-ber-null.eps
8.8	102	tc-2_1_3-111_101-3xxx_1xxx-otp-ber-null.eps
8.9	103	tc-2_1_3-111_101-3xxx_1xxx-sai-ber-null.eps
8.10	104	tc-2_1_3-111_101-3xxx_1xxx-sai-fer-null.eps
9.1	106	tc-2_1_3-111_101-192_64-uniform-ber-null.eps
9.2	107	tc-2_1_3-111_101-192_64-uniform-fer-null.eps
9.3	108	tc-2_1_3-111_101-192_64-deterministic-ber-null.eps
9.4	108	tc-2_1_3-111_101-192_64-deterministic-fer-null.eps
9.5	109	tc-2_1_3-111_101-192_64-sai-ber-null.eps
9.6	109	tc-2_1_3-111_101-192_64-sai-fer-null.eps

Table A.1: Filename Reference – Graphs

A.2 Results

The directory `Results` contains the simulation results used to plot the graphs in this thesis. A list of the result files is given in Tables A.2 and A.3 along with a brief description. The simulation results are in UNIX text format and suited for machine-reading. They consist of:

- A header detailing the software modules used, together with their version numbers and compilation (build) date. These lines are prefixed with '`# VCS:`'.
- A section giving the simulation details. For machine readability, these lines are prefixed by: '`#% <title>:`', where `<title>` is one of:

  `Codec` – gives the code type and size.

  `Codec2` – gives further details about the code's construction, including constituent codes, interleaver, type of termination used, and puncturing pattern (if any).

  `Tolerance` – the simulation tolerance.

  `Confidence` – the statistical confidence of the simulation tolerance.

  `Date` – the date and time when the simulation was started.

- A number of lines giving the BER and FER values after every iteration simulated, together with the tolerance for each value. Optionally, the number of frames simulated may also be given. Each line gives the results for a particular value of $\frac{E_b}{N_0}$, and is formatted in tab-separated columns as follows:
  - The first column will always contain the value of $\frac{E_b}{N_0}$ in dB.
  - Next, the results for the first iteration are given, in the following format: '`<BER> <BERtol> <FER> <FERtol>`', where `<BER>` and `<FER>` are the Bit and Frame error rates respectively, and `<BERtol>` and `<FERtol>` are their respective tolerances as absolute values.
  - The results for successive iterations follow in the same format.
  - Finally, the number of frames simulated at the given SNR may be listed in an additional column.

Filename	*Reference*
nrc-2_1_2-11_10-200_99.dat	2-state NRC code (Section 6.3.1)
rsc-2_1_2-11_10-200_99.dat	2-state RSC code
ee-nrc-2_1_2-11_10-20.dat	Error events for 2-state NRC code, used to compute the weight spectrum
ee-rsc-2_1_2-11_10-20.dat	Error events for 2-state RSC code
irwef-nrc-2_1_2-11_10-20-100.dat	Weight spectrum of 2-state NRC code
irwef-rsc-2_1_2-11_10-20-100.dat	Weight spectrum of 2-state RSC code
q-0.333333-1_20.dat	BER contribution of the complementary error function (Section 6.3.3)
idp-berrou-32.dat	IODS – Berrou (Fig. 6.6)
idp-montorsi-1024.dat	IODS – S-random (Fig. 7.3)
idp-rectangular32x32.dat	IODS – Square (Fig. 6.5)
idp-sai-1024_2-type9.dat	IODS – SA (Fig. 7.5)
idp-sai-64_2-type9.dat	IODS – SA non-terminating (Fig. 7.7)
idp-sai-66_2-type9-term.dat	IODS – SA self-terminating (Fig. 7.6)
confidence.dat	MAP confidence (Section 6.4.3)
uncoded-theoretical.dat	Expected BER – uncoded (Eqn. (5.1))
nrc-1_1_1-1-1000_1000-logreal.dat	Simulation of uncoded transmission
bound-rsc-2_1_3-111_101-52_24-20.dat	Union bound of $(52, 24)$ convolutional code (Section 5.3)
rsc-2_1_3-111_101-52_24-logreal.dat	Simulation using 32-bit Logarithm arithmetic (Fig. 5.3)
rsc-2_1_3-111_101-52_24-mpgnu.dat	Simulation using GNU Infinite Precision arithmetic
rsc-2_1_3-111_101-52_24-mpreal.dat	Simulation using 32-bit Exponent Floating-Point arithmetic
bound-tc-2_1_3-101_111-3006_1000-uniform-35.dat	Union bound of $(3006, 1000)$ Turbo code (Section 5.5)
tc-2_1_3-101_111-3006_1000-uniform-jplterm-null.dat	Simulation of $(3006, 1000)$ Turbo code
tc-2_1_3-111_101-208_102-helical17x6-stipple.dat	Simulation of $(208, 102)$ Turbo code (Fig. 5.5)
tc-2_1_3-111_101-3078_1024-montorsi-null.dat	Simulation of Turbo code with Montorsi's S-random interleaver (Fig. 5.6)
tc-2_1_3-111_101-3078_1024-montorsi-publishedber.dat	Montorsi's published BER results
tc-2_1_3-111_101-3078_1024-montorsi-publishedfer.dat	Montorsi's published FER results

Table A.2: Filename Reference – Results (part 1)

Filename	*Reference*
tc-2_1_3-111_101-3072_1024-rectangular32x32-null.dat	Code A in Table 8.1, page 104
tc-2_1_3-111_101-3087_1029-rectangular21x49-null.dat	Code B in Table 8.1
tc-2_1_3-111_101-3138_1044-helical29x36-null.dat	Code C in Table 8.1
tc-2_1_3-111_101-3072_1024-berrou32-null.dat	Code D in Table 8.1
tc-2_1_3-111_101-3072_1024-uniform-null.dat	Code E in Table 8.1
tc-2_1_3-111_101-3078_1024-uniform-jplterm-null.dat	Code F in Table 8.1
tc-2_1_3-111_101-3078_1024-flat-null.dat	Code G in Table 8.1
tc-2_1_3-111_101-3078_1024-shift6-null.dat	Code H in Table 8.1
tc-2_1_3-111_101-3078_1024-shift12-null.dat	Code I in Table 8.1
tc-2_1_3-111_101-3078_1024-shift36-null.dat	Code J in Table 8.1
tc-2_1_3-111_101-3078_1024-otp-null.dat	Code K in Table 8.1
tc-2_1_3-111_101-3078_1024-otp_renewable-null.dat	Code L in Table 8.1
tc-2_1_3-111_101-3072_1024-sai-1024_2-type9-null.dat	Code M in Table 8.1
tc-2_1_3-111_101-192_64-uniform-null.dat	Code A in Table 9.1, page 111
tc-2_1_3-111_101-198_64-uniform-jplterm-null.dat	Code B in Table 9.1
tc-2_1_3-111_101-192_64-rectangular8x8-null.dat	Code C in Table 9.1
tc-2_1_3-111_101-192_64-berrou8-null.dat	Code D in Table 9.1
tc-2_1_3-111_101-204_66-helical11x6-null.dat	Code E in Table 9.1
tc-2_1_3-111_101-192_64-sai-64_2-type9-null.dat	Code F in Table 9.1
tc-2_1_3-111_101-198_64-sai-64_2-type9-jplterm-null.dat	Code G in Table 9.1
tc-2_1_3-111_101-198_64-sai-66_2-type9-term-null.dat	Code H in Table 9.1

Table A.3: Filename Reference – Results (part 2)

Appendix B

Using the Software

B.1 Introduction

This appendix gives a brief description of how the Turbo codec and Monte Carlo simulator can be used within a C++ program, illustrating with examples. The use of the multi-processing with the Monte Carlo system is also described.

B.2 Creating a Turbo Codec

A Turbo codec object is created by the code fragment:

```
turbo<logreal> codec(encoder, modem, punc, chan, tau,
   sets, *inter, iter, simile, endatzero);
```

where `logreal` is the class defining 32-bit logarithm arithmetic, to be used in the codec's implementation of the BCJR algorithm. In order, the parameters for creating a Turbo codec are:

`encoder` is the convolutional encoder used as component code in the Turbo code. Note that only symmetric Turbo codes are currently supported, thus all encoders are considered identical. The encoder is defined as a Finite State Machine (FSM) – objects for NRC and RSC codes are already defined, and their creation is described in Section B.2.1.

`modem` defines the modulation / demodulation function which maps the encoder's output to channel symbols. A BPSK modulation scheme is defined simply by the code fragment: `bpsk modem;`

`punc` defines the puncturing pattern used. Implemented definitions for puncturing patterns include:

- Null, for no puncturing: `puncture_null punc(tau,s)`
- Stipple, for odd-even puncturing: `puncture_stipple punc(tau,s)`
- File, for reading the puncturing pattern from a text file: `puncture_file punc(path,tau,s)`

where `tau` is the block length and `s` is the number of parallel outputs (one more than the number of component codes).

`chan` defines the channel model. An AWGN channel is defined by the following code fragment: `awgn chan;`
The SNR can be defined after binding the channel object to the Turbo codec during creation.

`tau` is the length of the input sequence (including tail bits).

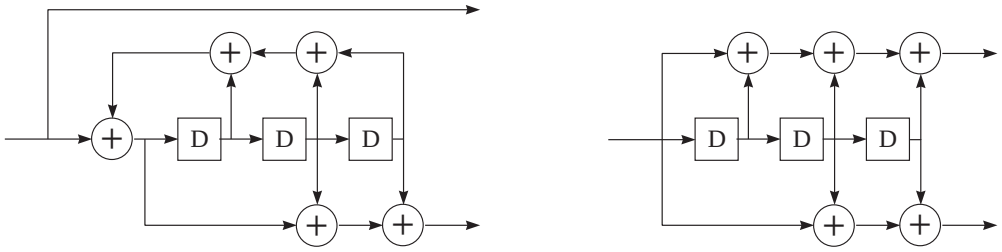
`sets` is the number of parallel codes being concatenated.

`inter` is a pointer to an array of `sets` $-$ 1 interleaver structures. The creation of interleavers is described in Section B.2.2.

`iter` is the number of iterations to be simulated.

`simile` is a boolean, and when true denotes that the interleaver used is simile.

`endatzero` is a boolean, and when true denotes that the interleaver used is correctly-terminating.



(a) (b)

Figure B.1: Example RSC and NRC encoders

B.2.1 Creating Component Codes

The creators for NRC and RSC encoders respectively are given below:

```
nrcc(k, n, generator);
rscc(k, n, generator);
```

where integers `k` and `n` are the number of input and output bits respectively. The code's generator matrix is given by `generator`, which is a `k` by `n` matrix of bitfields. As an example, consider creating the encoders described in Fig. B.1. The RSC encoder (a) is created by the following C++ code fragment:

```
const int k=1, n=2;
matrix<bitfield> gen(k, n);
gen(0,0) = "1111";
gen(0,1) = "1011";
rscc(k, n, gen);
```

Note how the first generator matrix entry described the feedback polynomial. Similarly, the NRC encoder (b) is created by:

```
const int k=1, n=2;
matrix<bitfield> gen(k, n);
gen(0,0) = "1111";
gen(0,1) = "1011";
nrcc(k, n, gen);
```

B.2.2 Creating Interleaver Structures

This aspect of the Turbo codec is one of the most developed, and consequently there are a large number of interleaver implementations. The creation of the interleaver types used in this thesis is described below:

- Non-deterministic interleavers may be created by loading the interleaver mapping from a text file, as in the code fragment below:

  ```
  file_lut inter(path, tau, forceterm);
  ```

  where `path` is the filename, `tau` is the interleaver size (including any tail used), and `forceterm` is a boolean specifying the use of JPL termination when true.

- Uniform interleavers: `uniform_lut inter(tau, forceterm);`

- Flat interleavers: `flat_lut inter(tau);`

- Barrel-Shift interleavers: `shift_lut inter(amount, tau);`
  where `amount` is the shift distance ζ.

- One-Time-Pad interleaver structures are defined by:

  ```
  onetimepad inter(*encoder, tau, terminated, renewable);
  ```

  where `renewable` is true for a renewable OTP interleaver, `terminated` is true for a self-terminating interleaver, and `encoder` is a pointer to the component code used. The encoder is necessary to create terminating interleavers.

- Helical interleavers: `helical inter(tau, rows, cols);`

- Rectangular interleavers: `rectangular inter(tau, rows, cols);`

- Berrou-Glavieux (`M` $\times$ `M`) interleavers: `berrou inter(M);`

B.3 Performing a Monte Carlo Simulation

In order to simulate the constructed code, we first need to create a Monte Carlo experiment which simulates a random frame and returns the BER and FER after each iteration. This is done by the code fragment:

```
const bool fast = true;
randgen src;
commsys system(&src, &chan, &codec, fast);
```

where `src` is defined as an equiprobable source, `chan` and `codec` are as defined for the Turbo codec, and `fast` is a boolean which enables a performance enhancement technique. When enabled, Turbo decoding stops when successive iterations return identical frames (indicating convergence).

Once the experiment object is created, it is simply bound to a new Monte Carlo object, and the required tolerance and confidence can be set, as in:

```
montecarlo estimator(&system);
estimator.set_confidence(confidence);
estimator.set_accuracy(accuracy);
```

where `confidence` and `accuracy` are fractional values. Next, we can enable multi-processing:

```
cmpi::enable(&argc, &argv, priority);
```

where `priority` is the UNIX job nice value that will be given to child processes. We can now start the simulation loop by considering a range of SNR values, and start the estimator at each of these values. When the estimator returns, we can output the returned estimate values and the tolerance at each estimate:

```
for(double SNR = SNRmin; SNR <= SNRmax; SNR += SNRstep)
   {
   cerr << "Simulating system at Eb/No = " << SNR << "\n";
   chan.set_snr(SNR);
   vector<double> estimate, tolerance;
   estimator.estimate(estimate, tolerance);

   cout << SNR;
   for(int i=0; i<system.count(); i++)
      cout << "\t" << estimate(i) << "\t" \
           << estimate(i)*tolerance(i);
   cout << "\t" << estimator.get_samplecount() << "\n";
   }
```

Finally, when the simulation is over, we can disable multi-processing:

```
cmpi::disable();
```