SIMULATION OF AN EFFICIENT RANDOM INTERLEAVER FOR TURBO CODES

A Thesis

Submitted to the College of Engineering of

Nahrain University in Partial Fulfillment of the

Requirements for the Degree of Master of Science in

Electronic and Communications Engineering /

Modern Communications Engineering

by

Ammar Falih A. Hasan

(B.Sc. in Electronic Engineering 2004)

Jamadi al-thania **1429**
June **2008**

Certification

I certify that this thesis entitled "**Simulation of an Efficient Random Interleaver for Turbo Codes**" was prepared by **Ammar Falih A. Hasan,** under my supervision at Nahrain University/ College of Engineering in partial fulfillment of the requirements for the degree of Master of Science in Electronic and Communications Engineering / Modern Communications Engineering.

Signature:
Name: Prof. Dr. Raad Sami Fyath
(Supervisor)
Date: / / 2008

Signature:
Name: Dr. Jabir S. Aziz
(Head of Department)
Date: / /2008

Certificate

We certify, as an examining committee, that we have read this thesis entitled "**Simulation of an Efficient Random Interleaver for Turbo Codes**", examined the student **Ammar Falih A. Hasan** in its content and found it meets the standard of a thesis for the degree of Master of Science in Electronic and Communications Engineering/ Modern Communications Engineering.

Signature:
Name: Prof. Dr. Raad Sami Fyath
(Supervisor)
Date: / / 2008

Signature:
Name: Assist. Prof. Dr. Munqith S. Dawood
(Member)
Date: / /2008

Signature:
Name: Dr. Emad Shehab Ahmed
(Member)
Date: / /2008

Signature:
Name: Assist. Prof. Dr. Kader H. Al-Shara
(Chairman)
Date: / /2008

Approval of the College of Engineering

Signature:
Name: Prof. Dr. Muhsin J. Jweeg
(Dean)
Date: / /2008

Abstract

In this thesis, a modified version of an interleaver for turbo codes is proposed which is based on the concept behind the success of the S-random interleaver. The idea is to separate the bits as far as possible from each other in an attempt to overcome the occurrences of low weight codewords. The method for doing this is to distribute the bits on a circle based on a specified weighting between each bit and the other. This weight is found by adding the distances between the two adjacent bits, those distances are found from the shortest one out of two, inner distance and outer distance. The former is the absolute difference between the bits and the latter is the other way round. The proposed interleaver is combined with the simile constraint that leads to terminate both constituent recursive encoders to the zero state.

In this work, a software simulation package is built using C++-based object oriented approach to simulate turbo codes environment. A library of classes called "**GenData.lib**" is programmed completely from ground up. The library contains a comprehensive set of classes that represent different components incorporated in turbo coding communication systems. A program called "**TurboSim.exe**" is implemented based on these classes using Microsoft® Visual C++ 6 under Microsoft® Windows operating system.

Simulation results are presented to address the effect of number of iterations, simile constraint, puncturing, and interleaver size on the performance of turbo codes implemented with the proposed interleaver. The results are compared with those related to a random interleaver. The results indicate clearly that the proposed interleaver offers a BER of 1.0×10^{-6} at SNR=3.25 dB when a 16-state encoder is employed. This is to be compared with a BER=1.5×10^{-5} obtained with a random interleaver operating under same conditions (number of iterations=3 and interleaver size=315 bits).

Contents

List of Abbreviations

Abbreviations	Notations
APP	A Priori Probability
AWGN	Additive White Gaussian Noise
BCJR	Bahl, Cocke, Jelinek and Raviv
BER	Bit Error Rate
CC	Convolutional Code
GUI	Graphical User Interface
LLR	Log-Likelihood Ratio
MAP	Maximum A Posteriori
NRC	Non-Recursive Convolutional
OOP	Object Oriented Programming
OS	Operating System
PCCC	Parallel Concatenated Convolutional Code
PDF	Probability Density Function
QoS	Quality of Service
RSC	Recursive Systematic Convolutional
SNR	Signal-to-Noise Ratio

List of Symbols

Symbols	**Notations**
$A_k(s)$	Log of $a_k(s)$
$a_k(s)$	Joint probability from time=1 to k-1
$B_k(s)$	Log of $\beta_k(s)$
$\beta_k(s)$	Joint probability form time=k+1 to N
C_k	k^{th} code generated by the encoder
C_k^s	Signal part of C_k
C_k^p	Parity part of C_k
$d_{i,\min}$	Minimum distance for weight-i codewords applied to a convolutional encoder
$d_{i,\min}^{TC}$	Minimum distance for weight-i codewords applied to a turbo encoder
d_k	Weight of the k^{th} encoded codeword
D^j	i^{th} memory element in a convolutional encoder
E_b	Energy per bit
E_b/N_0	Signal-to-noise ratio
g_1	Upper definition function of a convolutional encoder
g_2	Lower definition function of a convolutional encoder
$G_k(s',s)$	Log of $\lambda_k(s',s)$
$G_k^e(s',s)$	Log of $\lambda_k^e(s',s)$
G_{NR}	Representation of a non-recursive convolutional encoder
G_R	Representation of a recursive systematic convolutional encoder
$\lambda_k(s',s)$	Instant joint probability at time k
$\lambda_k^e(s',s)$	Extrinsic instant joint probability at time k
L	Step size in a convolutional interleaver
$L(U_k)$	Log a posteriori probability or log-likelihood ration

$L^e(U_k)$	Extrinsic probability information
Lc	Channel reliability value
max	Maximum value function
n_i	Possible input events that produced $d_{i,\min}^{TC}$
N	Interleaver size
N_c	Number of constituent encoders in a turbo encoder
N_o	Noise spectral density
off	Offset
p	Delay unit of the constituent encoder
P_1	Parity generated from the first RSC encoder in turbo codes
P_2	Parity generated from the second RSC encoder in turbo codes
P_b	Bit error probability
$\Pi(i)$	Interleaving position of index i
Q	Q-function
r	Code rate
s	Current state of the encoder at time k
s'	Previous state of the encoder at time k-1
S	Specified distance used in S-random interleaver design
S_i	Signal bit generated by a source
u(D)	Code sequence
U_k	k^{th} bit generated by the source
ν	Number of memory units in a convolutional encoder
w_k	Number of errors for the k^{th} error codeword
x_k	k^{th} transmitted bit
x_k^s	signal part of x_k
x_k^p	parity part of x_k
X_s	Transmitted signal
X_p	Transmitted parity

y	Received data at the decoder
y_k	k^{th} received bit
y_k^s	Signal part of the k^{th} received bit
y_k^p	Parity part of the k^{th} received bit
y_p	Received parity
y_s	Received signal
y'_s	Interleaved received signal
$\overline{Y}_1^N$	Array of the received signals, $\{y_0, y_{1, \dots,} y_N\}$
??	Estimation predicted for missing data

List of Figures

Chapter One
Introduction

1.1 General Introduction

Turbo coding represents the most promising coding technique ever performed. Since its first appearance, plans have continuously been investigated for implementing it into many kinds of communication architectures. It is now found in fields starting from ground wireless communications to deep space satellite communications [1]. The powerful error correction capability of turbo codes is recognized and accepted for almost all types of channels leading to increased data rates, cheaper services, and ultimately improved Quality of Service (QoS). Turbo coding scheme combines some familiar ideas such as interleaving and parallel concatenation into a powerful coding structure that generates long frames of distinct and recoverable codewords.

The structure of a turbo encoder consists of a parallel concatenation of two or more constituent encoders of Recursive Systematic Convolutional (RSC) type [2]. The frame of information is applied to the first encoder and the resultant codewords enter an adder. The frame of information is then interleaved and fed to the next encoder and the resultant codewords are applied to the same adder. The resultant code is the combination of all the codes generated out of the adder.

The interleaver (also known as the permuter) is the component in which the frame of information is rearranged to produce a different codeword frame. The interleaver role gives two main benefits [3]

i- it spreads out burst errors because the data gets rearranged hence infected bits become distributed; and

ii- it avoids finite length codewords that might result out of constituent encoders in response to certain frames of data. The interleaver avoids finite length codewords by generating different codeword sets for all constituent encoders.

The design of the interleaver is not an important issue as long as it maintains large block size. To practically implement the scheme in communication systems, the interleaver size must be shortened to cope with the practical processing speeds and memory requirements. As the interleaver size is minimized, however, the performance degrades to an impractical point where the Signal-to-Noise Ratio (SNR) falls suddenly at certain level [4]. This level is dominated by the code's free distance which is defined as the minimum distance between the coded sequences. For a relatively small free distance codes, the bit error probability curve is almost flat in this region, this effect is often referred to as "error floor". Short interleaver lengths with random designs have been shown to give poor performance due to the high level error floor [5]. The interleaver design is the key for solving this obstacle. Careful design of an interleaver should maintain far coding distances between the generated codewords.

The most popular types of interleavers are the "random interleavers" where the interleaving sequence is generated in a random manner. The output results of such interleavers are not predictable, they might generate bad or good interleaver and they are often unrealizable for analysis. A new efficient method to generate random interleaver has been proposed by Divsalar and Pollara [6] and the resultant interleaver is called "S-random interleaver". Here a number for the position of the distribution is chosen randomly. The number is then compared with S previous numbers by measuring the distance between them. If the distance is less than S, the number is rejected and another number is generated. It has been proven that if S is chosen equal to $\sqrt{N/2}$, where N is the size of the interleaver, then the method converges [7].

This thesis addresses the design of an efficient interleaver to be incorporated into turbo codes.

1.2 Literature Survey

In 2001, Sadjadpour et al. [8] have described a new interleaver design for turbo codes with short block lengths based on an iterative decoding suitability criterion to design a two-step S-random interleaver. It has been found that this interleaver has much better Bit Error Rate (BER) performance than the S-random interleaver at low BER and results in a lower error floor for turbo codes.

In 2002, Zhang et al. [9] have proposed a new deterministic turbo code interleaver design based on a novel successive packing method. In this method, the interleaver is generated based on packing of the basis interleaver iteratively. It has been shown that successive packing-based interleavers possess the following desirable features: prunability, adaptability to various criteria, and pseudo-random distribution.

In 2002, Feng et al. [10] have proposed a code-matched interleaver design for turbo codes in which a particular interleaver is constructed to match the code weight distribution. The design method is based on the code distance spectrum. The low weight paths in the code trellis are eliminated so that they do not appear in the code trellis after interleaving. These paths give large contributions to the error probability in the SNR region of interest for practical communication systems. The resultant BER and frame error rate performance of turbo codes are improved, and the "error floor" is reduced considerably.

In 2004, Daneshgaran and Mulassano [11] have addressed the issue of pruning (i.e, shortening) a given interleaver in a Parallel Concatenated Convolutional Code (PCCC). They have presented a systematic technique for interleaver pruning and have demonstrated the average optimality of the strategy. The main benefits of the proposed strategy are: it applies to any interleaver, it leads

to a systematic hardware-efficient structure for the variable-length interleaver design, and it is optimal on the average.

In 2004, Daneshgaran and Laddomada [12] have presented two theorems that lead to a modification of the previously developed iterative interleaver growth algorithm that can be used to design optimized variable-length interleavers, whereby at every length the optimized permutation implemented by the interleaver is a single-cycle permutation. It has been found that the presented modifications result in a variable-length turbo codes with excellent performance.

In 2004, Hongyu et al. [13] have proposed a chaotic interleaver in turbo codes and a suggested novel method to design a chaotic interleaver called block-random chaotic interleaver. When compared with random interleavers and S-random interleavers, this design offers lower implementation complexity, lower system delay, and fewer parameters to be transferred. Simulation results show that the design is effective when employed in channel interleaving and turbo codes.

In 2005, Sun and Takeshita [14] have introduced a class of deterministic interleavers for turbo codes based on permutaion polynomials. The main characteristics of this class of interleavers is that they can be algebraically designed to fit a given component code. The interleaver has been generated by a few simple computations hence the storage of the interleaver tables is avoided. The results indicate that this class of interleavers outperforms S-random interleaver for short frame sizes while the performance is close to that of S-random interleavers for long frame sizes.

In 2005, Dinoi and Benedetto [15] have proposed a technique to obtain good prunable S-random interleavers to be used in parallel and serially concatenated codes. The design is based on an algorithm to extend the interleaver size in a wide interval of values, while keeping good spreading properties in the whole range. A

modified pruning technique has been used for this class of interleavers. It has been shown that the proposed technique produces good performance when it is also applied to non prunable S-random interleavers. The designed interleavers have been proven to behave in the whole size range as an optimal design.

In 2006, Laddomada and Scanavino [16] have proposed an iterative design for semi-random prunable interleavers for PCCC. The technique is based on the growth of a smaller interleaver up to the desired length N. The optimization has been achieved via a minimization using a cost function which is strictly related to both the correlation properties of the extrinsic information and the concept of separation of an interleaver. The designed interleavers are prunable and have a behavior very similar to the interleavers designed with techniques which maximize the spread of the permutation.

In 2007, Arif and Sheikh [17] have proposed a design of an interleaver that combines randomization and de-correlation property with regular permutation and yet guarantees an enlarged minimum distance. The design produces deterministic interleavers that outperform the random and semi-random interleavers particularly for turbo codes having short block length.

In 2008, Tsai et al. [18] have used a turbo encoder using various interleavers to generate a proposed selective-mapping scheme which doesn't require the transmission of side information and can reduce the peak to average power ratio in turbo coded orthogonal frequency-division multiplexing systems. It has been found that the waiver of side information can avoid the degradation of error rate performance which results from the incorrect recovery of the side information at the receiver in the conventional systems.

1.3 Aim of the Work

The aim of the work is to design an efficient interleaver to be incorporated into turbo codes. The aim is to be achieved through the following steps

i. To introduce a complete design of a turbo coding environment along with the analysis to implement various kinds of turbo coding layouts.
ii. To test various kinds of interleavers with various configurations.
iii. To design a new deterministic interleaver and compare its results with those related to previous counterparts.
iv. To build a convenient user interface using suitable programming language capable of implementing the design at high performance speeds and portability.

1.4 Thesis Layout

The thesis explains its subjects using the following layout

Chapter one gives a brief introduction to turbo codes and the role of interleavers in affecting their performance. Brief literature survey is also included here.

Chapter two explains in detail the concepts of the turbo coding/decoding systems along with mathematical framework and problems related to the systems.

Chapter three explains the main concepts and principles of operation behind the proposed interleaver. It also introduces briefly the programming styles and algorithm implementations experimented throughout the work.

Chapter four presents the main results and performance predictions related to turbo codes implemented with the proposed interleaver. The results are obtained using the developed software and compared with those related to a conventional random interleaver.

Chapter five gives the main conclusions drawn out of the work and suggests few recommendations for future work.

Chapter Two
Turbo Codes and Interleavers

2.1 Introduction

In this chapter, the structure of the turbo codes system is presented. The influence of the turbo codes components on the performance of the system is outlined. The Maximum A Posteriori (MAP) decoding algorithm derivation is introduced as well. The key role played by the interleaver and its effect on the turbo codes system is discussed. The chapter also introduces some of the common types of the interleavers.

2.2 Turbo Codes

Turbo codes are the coding algorithms that are the result of concatenating several components. Unlike traditional previous serial concatenations, turbo codes are formed by the parallel concatenation of constituent encoders as shown in Fig 2.1. The constituent encoders are the Recursive Systematic Convolutional (RSC) encoders. The two RSC encoders are by convention chosen of the same type. Before the information is applied to the second RSC encoder, it is permuted by a specially designed interleaver. The original information is sent out along with the result of the RSC encoders. The coding rate can be increased by the use of a puncturing mechanism out of both encoders.

The decoder of the turbo coding system is iterative in nature. It features two MAP decoding components that feedback the results iteratively as extrinsic information to each other as shown in Fig. 2.2 [19]. What makes MAP decoding algorithms so important is their ability to accept extra extrinsic information obtained out from other decoding components to assist in finding much more accurate results. In this mechanism, the two MAP decoders reconstruct the signal with the help of the two sent parities for the normal and the interleaved cases. This process

iterates for a number of times before concluding the result. The number of iterations can be determined by special performance measuring criteria [20].

An overview of each of the turbo coding components is discussed in the following subsections.

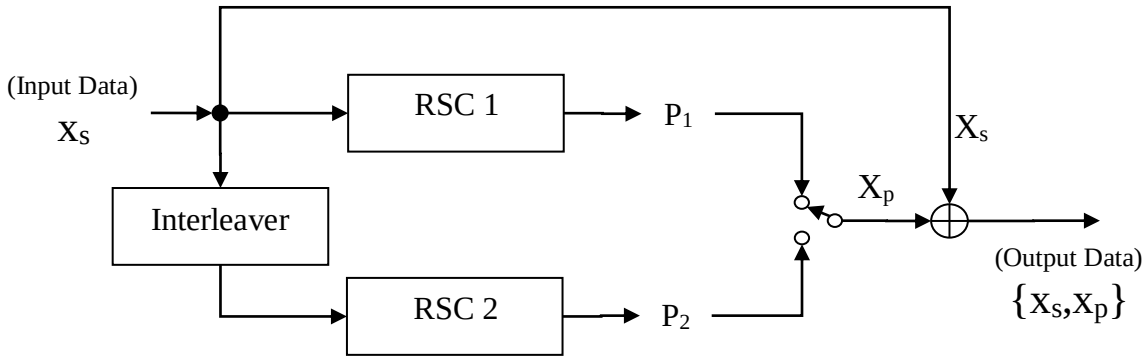


Fig. 2.1 Encoder of the turbo codes system.

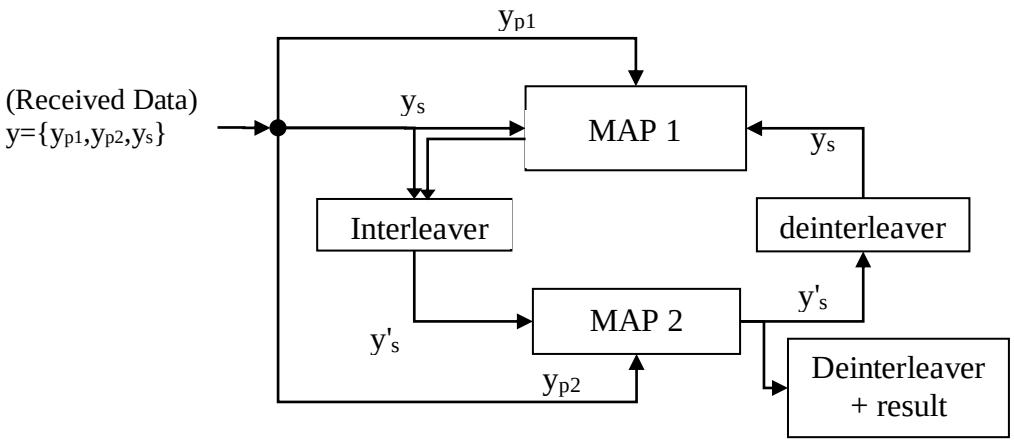


Fig. 2.2 Decoder of the turbo codes system [19].

2.2.1 Convolutional Codes

Convolutional Codes (CCs) are the type of codes that process the information continuously adding the extra parity bits interactively along the stream of data. While the implementation of convolutional encoding systems is relatively simple,

the process of decoding the resultant data stream at the receiving node can be quite complex. The “maximum likelihood” technique, first described by Viterbi, greatly simplifies the process of decoding convolutional codes. Convolutional encoding of data combined with Viterbi decoding at the receiving node is an accepted industry standard for digital channels [21].

A Non-Recursive Convolutional (NRC) encoder typically used in conjunction with an r=1/2 and ν=7 Viterbi decoder is shown in Fig. 2.3 [22]. The term "ν" defines the length or number of stages (memory units) in the shift register, the term "r" is the output rate (frequency) relative to the input rate (i.e. code rate). In Fig. 2.3, there are seven stages to the shift register; therefore, ν=7. In an r=1/2 system, the output rate is twice the input rate. In an r=2/3 system, three bits are output for every two bits that are input. The terms g_1 and g_2 represent the functions that define the structure of the convolutional encoder. In Fig. 2.3, $g_1=133_8$ represents the octal code for the upper connections to the shift register while $g_2=171_8$ describes the lower connections.

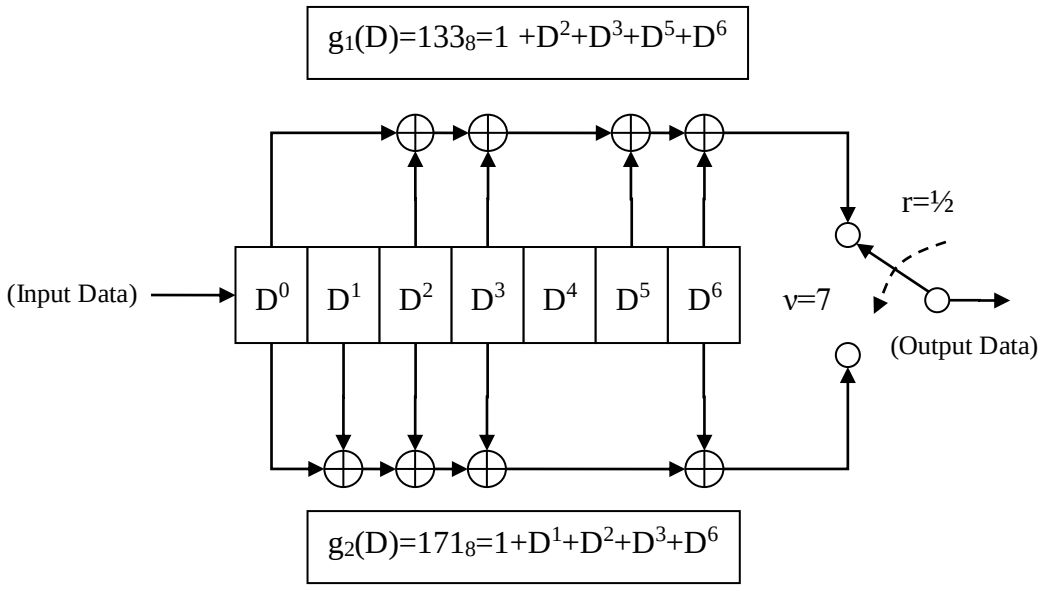


Fig. 2.3 A typical NRC encoder

Another arrangement of the convolutional encoder is the Recursive Systematic Convolutional (RSC) form [23]. In this form, the encoder is considered as a two set equations for feed-forward and feed-backward description, as shown in Fig. 2.4. The reason for using the RSC type rather than the NRC encoder in turbo codes is because it has the advantage of generating infinite weight code sequences out of input code sequences non divisible by the feedback equation.

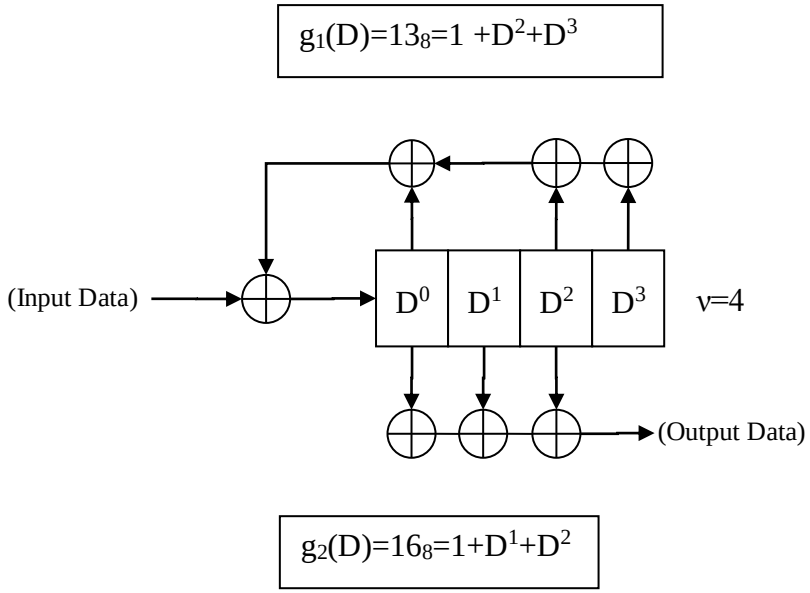


Fig. 2.4 RSC constituent encoder used in turbo codes.

The RSC encoder is merely a simple manipulation form of the normal NRC encoder and hence both possess same set of code sequences. This can be proved by knowing that the generation matrix for a rate 1/2 NRC encoder has the form

$$G_{NR}(D) = [g_1(D), g_2(D)] \quad \ldots(2.1)$$

and the equivalent RSC encoder's generator matrix is given by [24]

$$G_R(D) = \left[1, \frac{g_2(D)}{g_1(D)}\right] \quad \ldots(2.2)$$

It can be seen that any code sequence $u(D)$ input to the former equation can have the same result out of the latter one simply by applying the function $g_1(D)$ over the input resulting in the code sequence $u(D)g_1(D)$. The preference use of the RSC

encoder over the NRC appears when combined with an interleaver that manipulates the input frame to form another code sequence. The use of an interleaver overcomes finite weight code sequences problem.

2.2.2 Puncturing

Puncturing is the process of canceling out one part of the encoders generated parity bits at each run; this will increase the code rate (r) for the turbo coding system from ⅓ to ½ as shown in Fig. 2.5. In this figure, the following letters are assumed

- S_i Signal bit generated by the data source (having ± 1 states).
- P_1 Parity bit generated out of the first RSC encoder.
- P_2 Parity bit generated out of the second RSC encoder.
- ?? Predicted estimation of the missing data.

At the receiving end, the received data elements are furnished to cover the required data to be processed by the decoder. With puncturing, the decoder is informed to fill the unsent or (blocked) data elements with equal probability values (e.g., zeros if the sent elements having ± 1 values) indicating a priori knowledge that these places are not sent and their replacements are estimated. From the channel side, code rate (r) should by changed from ⅓ to ½ indicating increase in the transmitted signal power by a ratio of $\frac{1/2}{1/3} = 1.5$ or 1.76 dB [25].

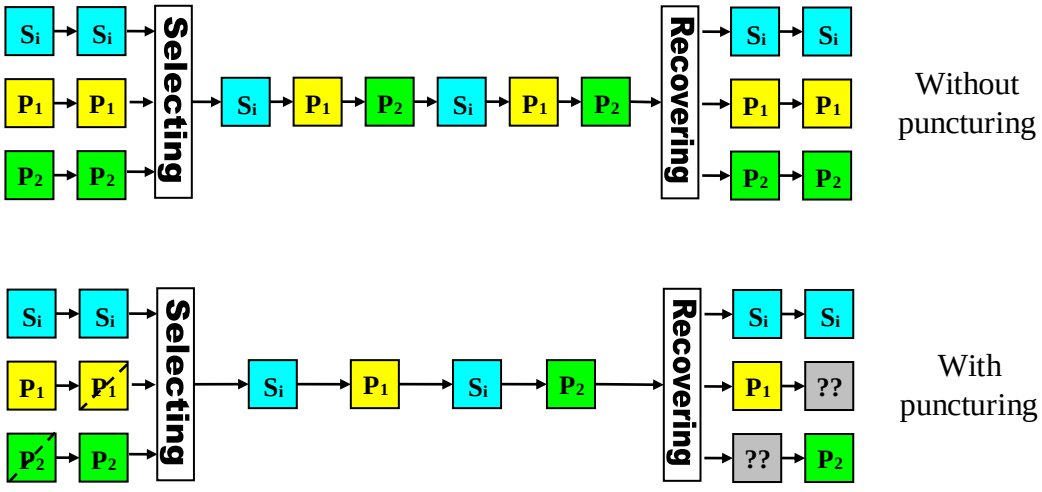


Fig. 2.5 Implementation of the puncturing mechanism.

2.3 Maximum A Posteriori Decoding Algorithm

MAP decoders make optimal symbol-by-symbol decisions, as well as providing soft reliability information which is necessary in the turbo coding scheme. The Bahl, Cocke, Jelinek and Raviv (BCJR) decoding algorithm [26] is the most commonly used MAP algorithm in turbo decoding. This section introduces the mathematical framework used to describe the MAP decoder. The results will be used in the next chapter to simulate the performance of turbo codes incorporating different types of interleaving.

2.3.1 Basic Concepts

The information life path for the system under consideration is shown in Fig. 2.6. The source produces the data U_k (a single bit at time k that carries on one of the two values +1 or -1). The RSC encoder adds extra parity bit to the source bit yielding $C_k = \{C_k^s, C_k^p\}$, where $C_k^s = U_k$ is the signal part and C_k^p is the parity part. To compare with the non coded case, the encoded transmitted signal is assumed to be packed by increasing the bit rate by a factor of code rate (r). This process reduces the signal power per bit for the assumed bit rate by the same factor and the resulting signal is $x_k = \{x_k^s, x_k^p\}$, where $x_k = rC_k$ [27].

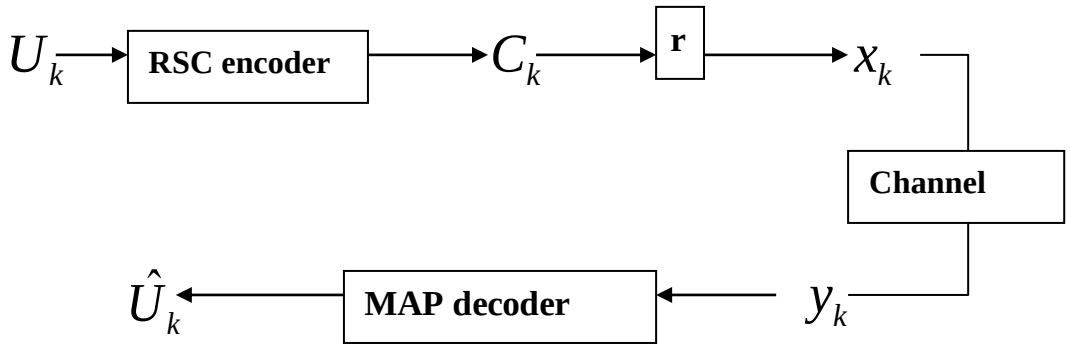


Fig. 2.6 Life cycle path of information from source to destination

The received noisy signal $y_k = \{y_k^s, y_k^p\}$ enters the decoder to be processed for estimating the transmitted signal $\hat{U}_k$. The decoder estimates the original information using the following two equations [28]

$$\hat{U}_k = sign[L(U_k)] \quad \ldots(2.3)$$

$$L(U_k) = \ln\left[\frac{P(U_k = +1 \mid \vec{Y}_1^N)}{P(U_k = -1 \mid \vec{Y}_1^N)}\right] \quad \ldots(2.4)$$

where *sign()* is the signum function, $L(U_k)$ is the log A Posteriori Probability (APP) or Log-Likelihood Ratio (LLR) [29], P is the probability and $\vec{Y}_1^N = \{y_1, y_2, ..., y_N\}$, where $y_k = \{y_k^s, y_k^p\}$.

According to Baye's theorem, the relationship between the conditional and joint probability in eq. 2.4 is given by [30]

$$P(U_k \mid \vec{Y}_1^N) = p(U_k, \vec{Y}_1^N) / p(\vec{Y}_1^N) \quad \ldots(2.5)$$

Note that uppercase P is used to designate binary probability and lowercase p is used to designate the PDF (Probability Density Function) of a continuously valued signal. By combining eqs. 2.4 and 2.5

$$L(U_k) = \ln\left[\frac{P(U_k = +1, \vec{Y}_1^N) / p(\vec{Y}_1^N)}{P(U_k = -1, \vec{Y}_1^N) / p(\vec{Y}_1^N)}\right]$$

$$= \ln\left[\frac{p(U_k = +1, \vec{Y}_1^N)}{p(U_k = -1, \vec{Y}_1^N)}\right] \quad \ldots(2.6)$$

To solve for $p(U_k, \vec{Y}_1^N)$, the formula is divided into parts

$$p(U_k, \vec{Y}_1^N) = p(U_k, \vec{Y}_1^{k-1}, y_k, \vec{Y}_{k+1}^N)$$

$$= \sum_{s'} p(s', U_k, \vec{Y}_1^{k-1}, y_k, \vec{Y}_{k+1}^N)$$

$$= \sum_{s'} p(s', \vec{Y}_1^{k-1}) p(U_k, y_k, \vec{Y}_{k+1}^N \mid s', \vec{Y}_1^{k-1})$$

$$= \sum_{s'} p(s', \vec{Y}_1^{k-1}) p(U_k, y_k, \vec{Y}_{k+1}^N \mid s') \quad \ldots(2.7)$$

where s represents the current state of the encoder at time k, s' is the previous state at time k-1, and $\vec{Y}_1^{k-1}$ was omitted from $p(U_k, y_k, \vec{Y}_{k+1}^N \mid s', \vec{Y}_1^{K-1})$ because the current

probability at time k and future ones doesn't depend on past events. Equation 2.7 is further simplified as

$$p\left(U_k,\vec{Y}_1^N\right)=\sum_{S'}p\left(s',\vec{Y}_1^{k-1}\right)p\left(U_k,y_k,\vec{Y}_{k+1}^N\mid s'\right)$$
$$=\sum_{S'}p\left(s',\vec{Y}_1^{k-1}\right)p\left(U_k,y_k\mid s'\right)p\left(\vec{Y}_{k+1}^N\mid s',U_k,y_k\right)$$
$$=\sum_{S'}p\left(s',\vec{Y}_1^{k-1}\right)p\left(U_k,y_k\mid s'\right)p\left(\vec{Y}_{k+1}^N\mid s\right) \quad \ldots(2.8)$$

The three terms given in eq. 2.8 form a chronological view of the probabilities [26]. They are symbolized as

$$p\left(U_k,\vec{Y}_1^N\right)=\sum_{U}\alpha_{k-1}(s')\lambda_k(s',s)\beta_k(s) \quad \ldots(2.9)$$

where $\alpha_{k-1}\left(s'\right)$ represents the joint probabilities from time=1 to k-1, $\beta_k(s)$ represents the joint probabilities of future state transitions from time=k+1 until N and $\lambda_k(s',s)$ is the instant joint probability at time k. Substituting eq. 2.9 into eq. 2.6 yields

$$L\left(U_k\right)=\ln\left[\frac{\sum_{U^+}\alpha_{k-1}(s')\lambda_k(s',s)\beta_k(s)}{\sum_{U^-}\alpha_{k-1}(s')\lambda_k(s',s)\beta_k(s)}\right] \quad \ldots(2.10)$$

where $\alpha_k(s)=p\left(s,\vec{Y}_1^k\right)$, $\lambda_k(s',s)=p\left(U_k,y_k\mid s'\right)$ and $\beta_k(s)=p\left(\vec{Y}_{k+1}^N\mid s\right)$.

2.3.2 Calculation of Different Probabilities

Each term in eq. 2.9 is calculated separately using different approaches as follows [31].

The past probabilities of eq. 2.9 are calculated recursively as follows

$$\alpha_k(s)=p\left(s,\vec{Y}_1^k\right)=\sum_{S'}p\left(s',s,\vec{Y}_1^{k-1},y_k\right)$$
$$=\sum_{S'}p\left(s',\vec{Y}_1^{k-1}\right)p\left(s,y_k\mid s'\right)$$
$$=\sum_{S'}p\left(s',\vec{Y}_1^{k-1}\right)p\left(U_k,y_k\mid s'\right)$$
$$\alpha_k(s)=\sum_{S'}\alpha_{k-1}(s')\lambda_k(s',s) \quad \ldots(2.11)$$

Since the encoder is assumed to start at the zero state, the initial values given are $\alpha_0(0)=1$ and $\alpha_0(s\neq 0)=0$.

The future probabilities of eq. 2.9 are calculated recursively as follows

$$\beta_{k-1}(s') = p(\vec{Y}_k^N \mid s') = \sum_S p(s, \vec{Y}_{k+1}^N, y_k \mid s')$$

$$= \sum_S p(s, y_k \mid s') p(\vec{Y}_{k+1}^N \mid s', s, y_k)$$

$$\beta_{k-1}(s') = \sum_S p(U_k, y_k \mid s') p(\vec{Y}_{k+1}^N \mid s)$$

$$\beta_{k-1}(s') = \sum_S \lambda_k(s', s) \beta_k(s) \qquad \ldots(2.12)$$

Since the encoder is assumed to terminate at the zero state, the initial values given are $\beta_N(0) = 1$ and $\beta_N(s \neq 0) = 0$.

The transition probability $\lambda_k(s', s)$ is the probability that the encoder state transits from s_{k-1} to s_k at time k caused by input U_k and is calculated as follows

$$\lambda_k(s', s) = p(U_k, y_k \mid s')$$

$$= P(U_k \mid s') p(y_k \mid s', U_k)$$

$$= P(U_k) p(y_k \mid C_k) \qquad \ldots(2.13)$$

where $U_k = C_k^s$, $U_k = \pm 1$ and $C_k = \{C_k^s, C_k^p\}$. $P(U_k)$ is the APP(A Priori Probability) obtained by previous estimations and is calculated in the following section.

2.3.3 Turbo Decoding Process

The turbo decoding process relies on extrinsic information gained accumulatively by individual decoders; if the information processed by the decoders are mutually independent the result will improve significantly [32].

Let $L^e(U_k)$ be defined as the extrinsic probability information gained by other sources of calculations. $L^e(U_k)$ is given as

$$L^e(U_k) = \ln\left[\frac{P(U_k = +1)}{P(U_k = -1)}\right] \qquad \ldots(2.14)$$

Let

$$P(U_k = +1) = P^+, \; P(U_k = -1) = P^-$$

Then

$$L^e(U_k) = \ln\left[\frac{P^+}{P^-}\right] \text{ and } \frac{P^+}{P^-} = \exp\left[L^e(U_k)\right] \qquad \ldots(2.15)$$

$$\frac{P^+}{P^-} = \frac{P(U_k = +1)}{P(U_k = -1)} \qquad \text{where} \quad P(U_k = +1) + P(U_k = -1) = 1$$

$$\frac{P^+}{P^-} = \frac{P(U_k = +1)}{1 - P(U_k = +1)}$$

$$P(U_k = +1) = \frac{P^+/P^-}{1 + P^+/P^-} = \frac{1}{1 + P^-/P^+} = \frac{\sqrt{P^-/P^+ \; P^+/P^-}}{1 + P^-/P^+}$$

$$= \frac{\sqrt{P^-/P^+}}{1 + P^-/P^+}\left(\sqrt{P^+/P^-}\right)^{+1} \qquad \ldots(2.16)$$

$$P(U_k = -1) = \frac{P^-/P^+}{1 + P^-/P^+} = \frac{\sqrt{P^-/P^+ \; P^-/P^+}}{1 + P^-/P^+}$$

$$= \frac{\sqrt{P^-/P^+}}{1 + P^-/P^+}\left(\sqrt{P^+/P^-}\right)^{-1} \qquad \ldots(2.17)$$

Combining eqs. 2.16 and 2.17 gives

$$P(U_k) = \frac{\sqrt{P^-/P^+}}{1 + P^-/P^+}\left(\sqrt{P^+/P^-}\right)^{U_k} \qquad \ldots(2.18)$$

Let

$$A_k = \frac{\sqrt{P^-/P^+}}{1 + P^-/P^+} \qquad \ldots(2.19)$$

Combining eqs. 2.15, 2.18 and 2.19 gives

$$P(U_k) = A_k\left(\sqrt{\exp\left[L^e(U_k)\right]}\right)^{U_k}$$

$$= A_k\left(\exp\left[\tfrac{1}{2} U_k L^e(U_k)\right]\right) \qquad \ldots(2.20)$$

To solve the second part of eq. 2.13 ($p(y_k \mid C_k)$), the signal and parity are assumed to travel independently through the channel and hence are calculated as

$$p(y_k \mid C_k) = p(y_k^s, y_y^p \mid C_k^s, C_k^p)$$
$$= p(y_k^s \mid C_k^s) p(y_k^p \mid C_k^p) \qquad \ldots(2.21)$$

The probabilities given in eq. 2.21 represent the PDF of the noise function that acts on the original transmitted information until it reaches the destination as $\{y_k^s, y_k^p\}$. The assumed PDF function is a pure Additive White Gaussian Noise (AWGN) function given in Fig. 2.7 [30].

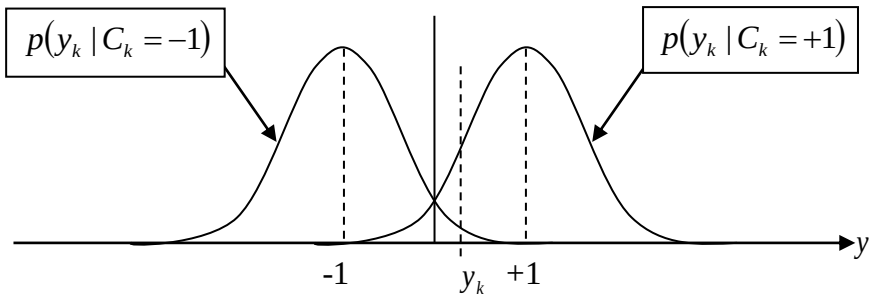


Fig. 2.7 AWGN PDF noise function assumed to act on the transmitted signal [30].

If $\{x_k^s, x_k^p\}$ is transmitted, eq. 2.21 is simplified as follows [32]

$$p(y_k \mid C_k) = p(y_k^s \mid C_k^s) p(y_k^p \mid C_k^p)$$
$$= \exp\left(\frac{-(y_k^s - x_k^s)^2}{2\sigma^2}\right) \exp\left(\frac{-(y_k^p - x_k^p)^2}{2\sigma^2}\right)$$
$$= \exp\left(\frac{-y_k^{s^2} + 2x_k^s y_k^s - x_k^{s^2}}{2\sigma^2} + \frac{-y_k^{p^2} + 2x_k^p y_k^p - x_k^{p^2}}{2\sigma^2}\right)$$
$$= \exp\left(\frac{-(y_k^{s^2} + y_k^{p^2} + x_k^{s^2} + x_k^{p^2}) + 2(x_k^s y_k^s + x_k^p y_k^p)}{2\sigma^2}\right) \qquad \ldots(2.22)$$

Let

$$B_k = \exp\left(\frac{-(y_k^{s^2} + y_k^{p^2} + x_k^{s^2} + x_k^{p^2})}{2\sigma^2}\right) \quad \ldots(2.23)$$

Substituting eq. 2.23 into eq. 2.22 yields

$$p(y_k \mid C_k) = B_k \exp\left[\frac{x_k^s y_k^s + x_k^p y_k^p}{\sigma^2}\right]$$

$$= B_k \exp\left[\frac{1}{2}\frac{2r}{\sigma^2}\left(C_k^s y_k^s + C_k^p y_k^p\right)\right]$$

$$= B_k \exp\left[\frac{1}{2} L_c\left(C_k^s y_k^s + C_k^p y_k^p\right)\right] \quad \ldots(2.24)$$

where $x_k = rC_k$, $L_c = \frac{2r}{\sigma^2}$ is the channel reliability value [30] and r = code rate.

Substituting eqs. 2.20 and 2.24 into eq. 2.13 yields

$$\lambda_k(s', s) = A_k B_k \exp\left[\frac{1}{2} U_k L^e(U_k)\right] \exp\left[\frac{1}{2} L_c\left(C_k^s y_k^s + C_k^p y_k^p\right)\right]$$

$$= A_k B_k \exp\left[\frac{1}{2} U_k \left(L^e(U_k) + L_c y_k^s\right)\right] \lambda_k^e(s', s) \quad \ldots(2.25)$$

where $\lambda_k^e(s', s)$ is the extrinsic instant joint probability and is given by

$$\lambda_k^e(s', s) = \exp\left[\frac{1}{2} L_c C_k^p y_k^p\right] \quad \ldots(2.26)$$

Combining eq. 2.10 and eq. 2.25 gives [32]

$$L(U_k) = \ln\left[\frac{\sum_{U^+} \alpha_{k-1}(s')\beta_k(s) A_k B_k \exp\left[\frac{1}{2} U_k \left(L^e(U_k) + L_c y_k^s\right)\right] \lambda_k^e(s', s)}{\sum_{U^-} \alpha_{k-1}(s')\beta_k(s) A_k B_k \exp\left[\frac{1}{2} U_k \left(L^e(U_k) + L_c y_k^s\right)\right] \lambda_k^e(s', s)}\right]$$

$$= \ln\left[\frac{A_k B_k \exp\left[\frac{1}{2}(+1)\left(L^e(U_k) + L_c y_k^s\right)\right] \sum_{U^+} \alpha_{k-1}(s')\beta_k(s)\lambda_k^e(s', s)}{A_k B_k \exp\left[\frac{1}{2}(-1)\left(L^e(U_k) + L_c y_k^s\right)\right] \sum_{U^-} \alpha_{k-1}(s')\beta_k(s)\lambda_k^e(s', s)}\right]$$

$$= L^e(U_k) + L_c y_k^s + \ln\left[\frac{\sum_{U^+} \alpha_{k-1}(s')\beta_k(s)\lambda_k^e(s', s)}{\sum_{U^-} \alpha_{k-1}(s')\beta_k(s)\lambda_k^e(s', s)}\right] \quad \ldots(2.27)$$

In eq. 2.27, A_k and B_k are extracted out of the summation because they don't depend on the state transition of the encoder as can be seen by reconsidering eqs. 2.19 and eq. 2.23 and are omitted from the numerator and denominator because they

are constants. The final result presented by eq. 2.27 shows clearly that estimating the signal at time k is the result of adding the a priori information gained by other decoders given by $L^e(U_k)$ plus the reading of the detector of the receiver sometimes called *channel value* given by $L_c y_k^s$ plus the result of the decoding process of the decoder which itself can be used as extrinsic a priori information for other decoders [19].

2.3.4 Simplified Calculations

The MAP algorithm provides a flexible theoretical solution to the needs of turbo decoding process; it solves complicated interleaved data and provides at the same time soft output information to be used by other decoders. The drawback of the algorithm, however, is its complexity. It makes extensive use of logarithmic mathematical functions which take long processing time and consumes practical data precisions [33]. The data precision problem can be solved by converting the calculations from the time domain to the logarithmic domain. By implementing the Log-domain calculation, the calculations would by much simplified as shown by the following derivations [34]

Let

$$A_k(s) = \ln(\alpha_k(s)) \qquad B_k(s) = \ln(\beta_k(s))$$
$$G_k(s',s) = \ln(\lambda_k(s',s)) \qquad G_k^e(s',s) = \ln(\lambda_k^e(s',s)) \qquad \ldots(2.28)$$

Then

$$\alpha_k(s) = \exp[A_k(s)] \qquad \beta_k(s) = \exp[B_k(s)]$$
$$\lambda_k(s',s) = \exp[G_k(s',s)] \qquad \lambda_k^e(s',s) = \exp[G_k^e(s',s)] \qquad \ldots(2.29)$$

By using eqs. 2.28 and 2.29, the former MAP derivations are simplified as follows. Substituting eqs. 2.28 and 2.29 into eq. 2.11 yields

$$A_k(s) = \ln\left(\sum_{s'} \alpha_{k-1}(s')\lambda_k(s',s)\right)$$
$$= \ln\left[\sum_{s'} \exp\left(A_{k-1}(s') + G_k(s',s)\right)\right] \qquad \ldots(2.30)$$

where $A_0(0) = 0$ and $A_0(s \neq 0) = -\infty$.

Substituting eqs. 2.28 and 2.29 into eq. 2.12 yields

$$B_{k-1}(s') = \ln\left[\sum_s \lambda_k(s',s)\beta_k(s)\right]$$
$$= \ln\left[\sum_s \exp\left(B_k(s)+G_k(s',s)\right)\right] \quad \ldots(2.31)$$

where $B_N(0)=0$ and $B_N(s\neq 0)=-\infty$.

Substituting eqs. 2.28 and 2.29 into eqs. 2.25 and 2.26 gives

$$G_k(s',s) = \tfrac{1}{2}U_k\left(L^e(U_k)+y_k^s L_c\right)+G_k^e(s',s) \quad \ldots(2.32)$$

$$G_k^e(s',s) = \tfrac{1}{2}L_c y_k^p C_k^p \quad \ldots(2.33)$$

Substituting eqs. 2.28 and 2.29 into eq. 2.27 gives

$$L(U_k) = L^e(U_k)+L_c y_k^s + \ln\left[\frac{\sum_{U^+}\exp\left(A_{k-1}(s')+B_k(s)+G_k^e(s',s)\right)}{\sum_{U^-}\exp\left(A_{k-1}(s')+B_k(s)+G_k^e(s',s)\right)}\right] \quad \ldots(2.34)$$

The calculations now use fewer logarithmic equations and more importantly keep the precisions at stable practical implementation values because the numbers are transformed to the logarithmic domain.

The complexity of the previous derivations due to the extensive use of the logarithms equations can be further simplified by the use of suboptimal logarithmic oriented derivations which make use of the max() function. The max() function is stated as follows

Let

$$\mathrm{E}(x^i) = \ln\left[\sum_i e^{x_i}\right]$$

Then

$$\mathrm{E}(x^i) = \max_i[x_i] + \ln\left[\sum_i e^{x_i}\right] - \max_i[x_i]$$
$$= \max_i[x_i] + \ln\left[\sum_i e^{x_i}\right] - \ln\left[\exp\left(\max_i[x_i]\right)\right]$$

$$= \max_i [x_i] + \ln\left[\frac{\sum_i e^{x_i}}{\exp\left(\max_i [x_i]\right)}\right]$$

$$= \max_i [x_i] + \ln\left[\sum_i \exp\left(x_i - \max_i [x_i]\right)\right] \quad \ldots(2.35)$$

In most cases the second part of eq. 2.35, usually called the correction term, approaches a negligible value that can safely be dropped, hence simplified as

$$\mathrm{E}(x^i) = \ln\left[\sum_i e^{x_i}\right] \cong \max_i [x_i] \quad \ldots(2.36)$$

The statement of eq. 2.36 is used to greatly simplify the Log-MAP derivations to be used for more practical implementations as shown in the following derivations.

Substituting eq. 2.36 into eqs. 2.30 and 2.31 gives

$$A_k(s) = \max_i (A_{k-1}(s') + G_k(s', s)) \quad \ldots(2.37)$$

$$B_{k-1}(s') = \max_i (B_k(s) + G_k(s', s)) \quad \ldots(2.38)$$

Substituting 2.35 into eq. 2.34 yields

$$L(U_k) = L^e(U_k) + L_c y_k^s + \max_{U^+}\left[A_{k-1}(s') + B_k(s) + G_k^e(s', s)\right] - \max_{U^-}\left[A_{k-1}(s') + B_k(s) + G_k^e(s', s)\right] \quad \ldots(2.39)$$

2.4 Interleavers

This section introduces the main concepts behind interleavers employed in turbo codes and presents the main features of different types of interleavers.

2.4.1 Concepts of Interleavers

An interleaver is the part of the turbo coding system which rearranges the ordering of a data sequence in a one-to-one deterministic format in a way that helps forming another codeword to be coded using the RSC component. The interleaver is the most important part of the turbo coding algorithm and many researches have

been investigated to provide the best designs. The design of short frame length interleavers is a sensitive process that if neglected by assuming random permutation designs will result in bad performance systems. An interleaver serves two important tasks in turbo codes [35]

i- It constructs long block codes from small memory convolutional codes by uncorrelating the source of information applied to the two RSC encoders. Long codes can approach the Shannon capacity.

ii- It spreads out burst errors.

The need for two uncorrelated codes is to assist in delivering two different codes of the same source of information to exchange their confidence of the results at the decoder side in case one of them has errors. The design of the interleaver plays the essential role in a well behaved turbo coding communication system. Interleavers affect directly the performance of the system especially when the required size of the frame is to be short.

The decoders used in the receiving side assume both encoders start and terminate at the zero states. Trellis termination of the first encoder can be established by inserting few additional tail bits to the input information driving the encoder to terminate at the zero state, but the interleaver permutes on the second data on the second encoder which terminates it into a different state. To solve this, two solutions have been introduced [36]

i- Designing simile interleavers. Simile interleavers are designed in a way that terminates both encoders into the same state at the end of the calculation process; hence termination is only required to be done on the first encoder.

ii- The use of additional redundant terminating bits. These additional bits are appended to the end of both outgoing blocks of parity bits.

The difference between the two termination methods is that the first one adds a constraint to the design of the interleaver to make use of the double termination advantage, while the second approach adds additional complexity to the design of

the communication system but removes all constraints on the design of the interleaver. The termination method used in this thesis, as will be explained later, is the design of an interleaver incorporating simile method.

2.4.2 Types of Interleavers

There are many types of interleavers employed with turbo codes. In general the interleavers can be classified into block, convolutional, random and linear interleavers [37]. Some of the interleaver types are

a- Block Interleaver

It is also known as row-column interleaver, since it sets the data into rows (left to right and top to bottom) and retrieves them by columns (top to bottom and right to left). The data are distributed systematically separating each bit from the other in a constant length as shown in Fig. 2.8 [38].

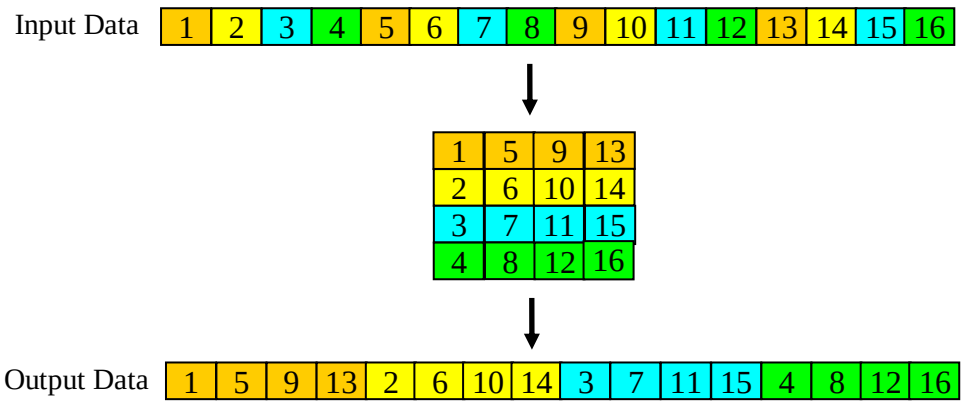


Fig. 2.8 Row-Column Interleaver.

b- Diagonal Interleaver

The method is the same as that in the row-column method but the data are retrieved in a diagonal manner as shown in Fig. 2.9.

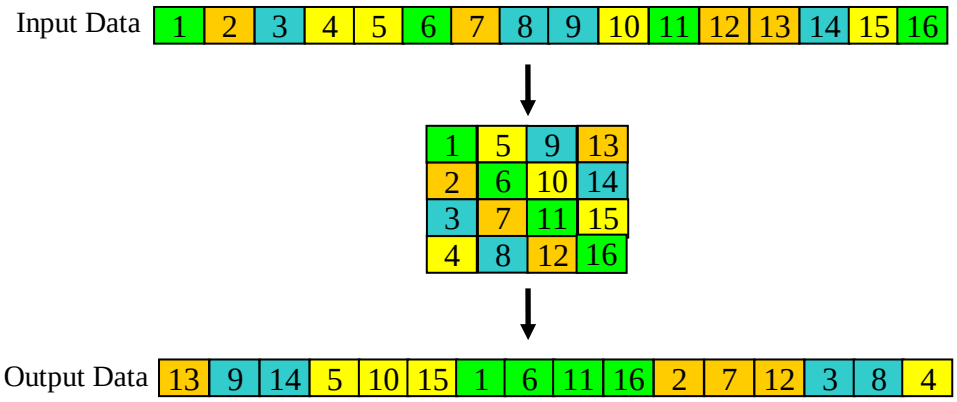


Fig. 2.9 Diagonal Interleaver

c- Convolutional Interleavers

A convolutional interleaver reads the source data from a set of N delay registers as shown in Fig. 2.10 where N is the interleaver size. Each delay register delays data by L bits more than its previous register. The generated numbers are modulated with N to obtain a set of numbers from 0 to N-1. Convolutional interleavers are also known as circular shifting interleavers because they circulate in a definite step size through all N bits [39].

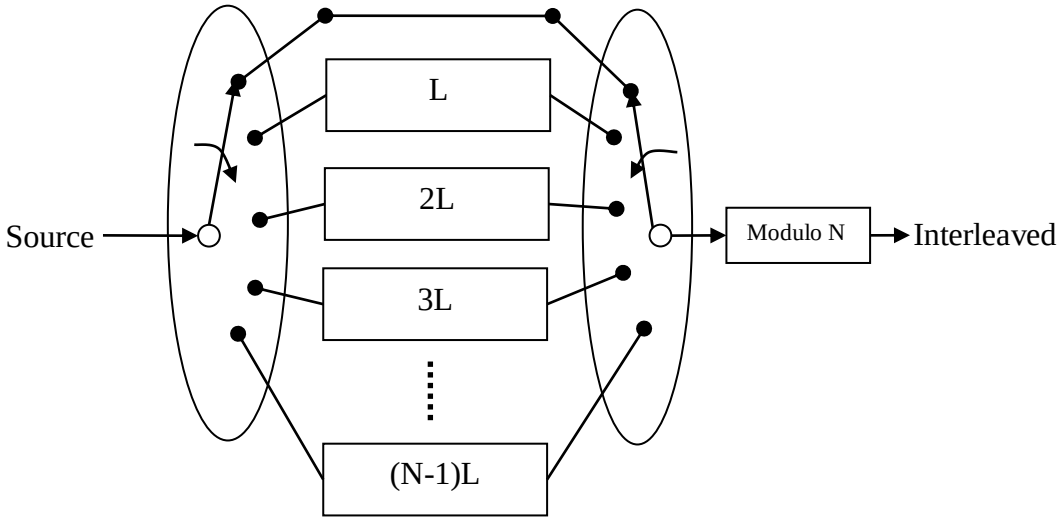


Fig. 2.10 Convolutional interleaver module

Convolution interleavers read the data according to the equation

$$\Pi(i) = (iL + off) \bmod N$$

where L=step size, i=index, *off* =offset and N=Interleaver size.

For example let N=10, *off* = 0 and L=3

Data (i): 0 1 2 3 4 5 6 7 8 9

Intereleaved data ($\Pi(i)$): 0 3 6 9 2 5 8 1 4 7

A Convolutional interleaver is considered as a type of linear interleavers. Linear interleavers are the ones that generate the interleaving matrix from a mathematical expression saving memory needs for large interleaver sizes.

d- Pseudorandom Interleaver

In this case the data are distributed randomly. This process results in un-predicted types of interleavers that might perform well but most of the cases would result in bad interleavers.

e- S-Random Interleaver

The S-random interleaver design is based on the selection of random numbers of distribution according to the condition that separates the numbers by a specified distance. The idea is to choose an arbitrary length S which will be compared with the distance between a chosen number and S of previously selected ones. If the distance is less than S, the number is rejected and another one is selected. This design ensures that no two successive numbers would be possible to join together in the interleaved version of the data. The method continues until all numbers are selected, as shown in Fig. 2.11. It is found that if S is selected less than $\sqrt{N/2}$ where N is the interleaver size, the method converges [40].

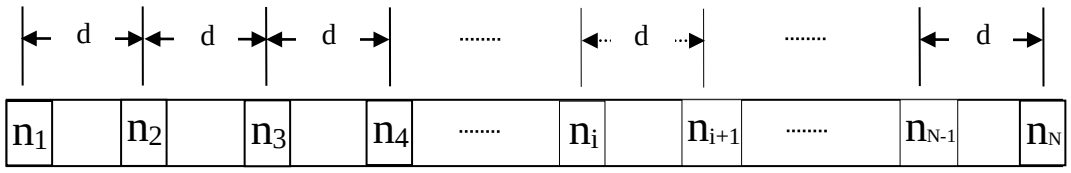


Fig. 2.11 Construction of the S-Random interleaver. The absolute distance d between the sequentially selected numbers should be larger than the specified length S.

2.4.3 Analysis of Interleaving-Based Turbo Codes

For the recursive encoder, the code sequence will be of the finite weight if and only if the input sequence is divisible by $g_1(D)$ hence the following assumptions are assumed [41]

i- A weight-one input will produce an infinite weight output (for such an input is never divisible by polynomial $g_1(D)$).

ii- For any non-trivial $g_1(D)$, there exists a family of weight-two inputs of the form $D^j\left(1+D^{q-1}\right)$, $j \geq 0$, which produce finite weight outputs, i.e., which are divisible by $g_1(D)$. When $g_1(D)$ is a primitive polynomial of the degree *m*, then $q = 2^m$; more generally, *q*-1 is the length of the pseudorandom sequence generated by $g_1(D)$.

The turbo encoder is linear (maps the input to a fixed output) because all of its components are linear. The RSC encoders are linear, the permuter can be represented as a linear permutation matrix and the puncturor is linear because all codewords share the same puncturing locations. Because the turbo encoder is linear its performance can be measured by considering the case of all zeros bits sequence and noticing the error probabilities.

If the 0^{th} codeword was transmitted (all zeros codeword), the decoder would choose the k^{th} codeword (assuming error codeword) with the probability [29]

$$P_s(k) = Q\left(\sqrt{\frac{2d_k r E_b}{N_o}}\right) \qquad \ldots(2.40)$$

where Q is the Q-function, d_k is the weight of the k^{th} encoded codeword, r is the code rate, E_b is the energy per bit and N_o is the noise spectral density.

It is known that the larger the argument of the Q-function, the smaller the probability of error. Hence the design of a good encoder ensures that d_k is as large as possible. This can be seen for the RSC encoder from the second assumption. For the given k^{th} codeword of eq. 2.40 the bit error probability is given by

$$P_e(k \mid 0) = \frac{w_k}{N} Q\left(\sqrt{\frac{2d_k r E_b}{N_o}}\right) \qquad \ldots(2.41)$$

where w_k is the number of errors for the k^{th} error codeword and N is the interleaver size. To measure the performance of the turbo codes, the bit error probabilities, P_b, for all possible set of k^{th} codewords must be considered. The P_b for a given E_b/N_o can be given as

$$P_e \le \sum_{k=1}^{2^N-1} P_b(k \mid 0) = \sum_{k=1}^{2^N-1} \frac{w_k}{N} Q\left(\sqrt{\frac{2rd_k E_b}{N_o}}\right) \qquad \ldots(2.42)$$

The summation of all possible codewords from $k=1$ to $k=2^N-1$ can be further split into groups of the same weight codewords (i.e. having same no. of bits) with different set of permutation [29]

$$P_e \le \sum_{k=1}^{2^N-1} \frac{w_k}{N} Q\left(\sqrt{\frac{2rd_k E_b}{N_o}}\right) \le \sum_{w=1}^{N} \sum_{v=1}^{\binom{N}{w}} \frac{w}{N} Q\left(\sqrt{\frac{2rd_{wv} E_b}{N_o}}\right) \qquad \ldots(2.43)$$

where $\binom{N}{w} = \frac{N!}{w!(N-w)!}$ is the set of all possible permutations for the codewords of weight w.

For $w=1$(weight one input), RSC encoders will generate infinite (very high) weight codewords (d_{1v}), thus giving a very small (negligible) probabilities of error in eq. 2.43. For $w=2$ (weight two input), from the $\binom{N}{2}$ possibilities only a fraction will be divisible by $g_1(D)$ (hence produce finite weight output) and from these possible events only certain ones will give the minimum possible output codeword,

$d_{2,\min}$. Further, due to the presence of the interleaver it is unlikely that the second RSC encoder will give the same result. However, for random interleavers, it is possible to generate two sequences with same $d_{2,\min}$ as shown in Fig. 2.12.

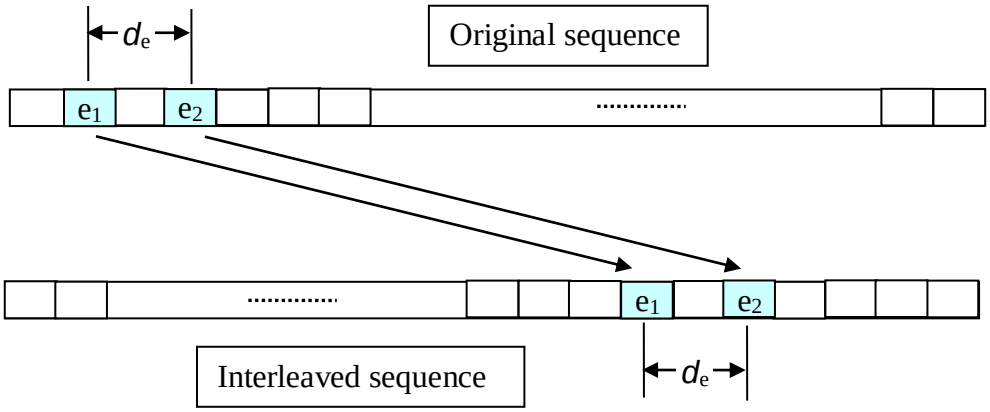


Fig. 2.12 A sequence with two error bits at a specified distance. If the two bits generate a finite codeword and interleaved at the same distance, the performance of the turbo code will largely be degraded.

Let one neglect all the possibilities, except for some minimum-weight turbo codes which is given by [42]

$$d_{2,\min}^{TC} = \sum_{j=1}^{N_C} d_{2,\min}^{(j)} \qquad \ldots(2.44)$$

where $d_{2,\min}^{TC}$ = minimum distance for the weight-2 codewords applied to a turbo encoder, $d_{2,\min}^{(j)}$ = the weight-2 input minimum distance for the jth constituent code, and N_c is the number of constituent RSC encoders. The value of $d_{2,\min}^{TC}$ depends on the interleaver design used in the turbo coding system.

Referring to eq. 2.43, and considering the case $w=2$, it can be approximated as [29]

$$\sum_{v=1}^{\binom{N}{2}} \frac{2}{N} Q\left(\sqrt{\frac{2rd_{2,v}E_b}{N_o}}\right) \cong \frac{2n_2}{N} Q\left(\sqrt{\frac{2rd_{2,\min}^{TC}E_b}{N_o}}\right) \qquad \ldots(2.45)$$

where n_2 = the possible input events that produce $d_{2,\min}^{TC}$.

If $w=3$ is considered, a similar argument can give $d_{3,\min}^{TC}$. Most of the $\binom{N}{3}$ terms for w_3 inputs in eq. 2.43 can be neglected because the number of weight-3 inputs (n_3) divisible by $g_1(D)$ is of the order of n_2. This can be expected by noticing that $w=3$ can be considered as a combination of $w=1$ and $w=2$ thus will result in $n_3 << n_2$. Similar analysis can be done for $n_4, n_5 ...etc$.

From the terms that are neglected for eq. 2.45 the approximate P_b for a given E_b/N_o is given by [29]

$$P_e \cong \max_{w\geq 2}\left\{\frac{wn_w}{N}Q\left(\sqrt{\frac{2rd_{w,\min}^{TC}E_b}{N_o}}\right)\right\} \qquad \ldots(2.46)$$

which suggests that P_b can be approximated by the maximum effective values for a given event that will generate $d_{w,\min}^{TC}$. The terms n_w and $d_{w,\min}^{TC}$ are functions of the particular interleaver.

Note from eq. 2.46 that P_b depends on $(1/N)$, which means that the probability of error can simply be decreased by increasing the interleaver length; this effect is called "interleaver gain".

Chapter Three
Proposed Interleaving Scheme

3.1 Introduction

This chapter proposes the design of a new interleaver. The new interleaver is based on the conceptual idea behind the success of the S-Random interleaver. The chapter introduces the procedures and methods used to design the framework of simulation. Different layouts and parameters of turbo coding system are incorporated for testing purposes.

3.2 Simile Constraint

The simile constraint is a condition applied to the design of the interleaver to make both Recursive Systematic Convolutional (RSC) encoders terminate at the same state. The method used to implement the constraint is given as follows

i- Checking that the interleaver size is divisible by the delay unit p where $p = 2^{\nu} - 1$ with ν is the number of memory elements in the encoder. To do this, check that $N \bmod p = 0$. For an N=18 and an encoder with ν=2, p is calculated as p=3 and N mod p=0.

ii- Creating p arrays, each one is of size N / p. For the example shown in Fig. 3.1a , 3 arrays of size 6 are created as shown in Fig. 3.1b.

iii- Distributing the numbers 1 to N into the p arrays in a way that for any input i from (1 to N): Sequence number = i mod p, position in sequence=i/p, as shown in Fig. 3.1b.

iv- Interleaving the sequences of p arrays independently as shown in Fig. 3.1c.

v- Reconstructing the original sequence by rearranging the values of the arrays as $\{S_0^0, S_0^1, S_0^2, \cdots, S_0^{P-1}, S_1^0, \cdots, S_{I-1}^{P-1}\}$ where S_i^k is the i^{th} value of the sequence k, as shown in Fig. 3.1d.

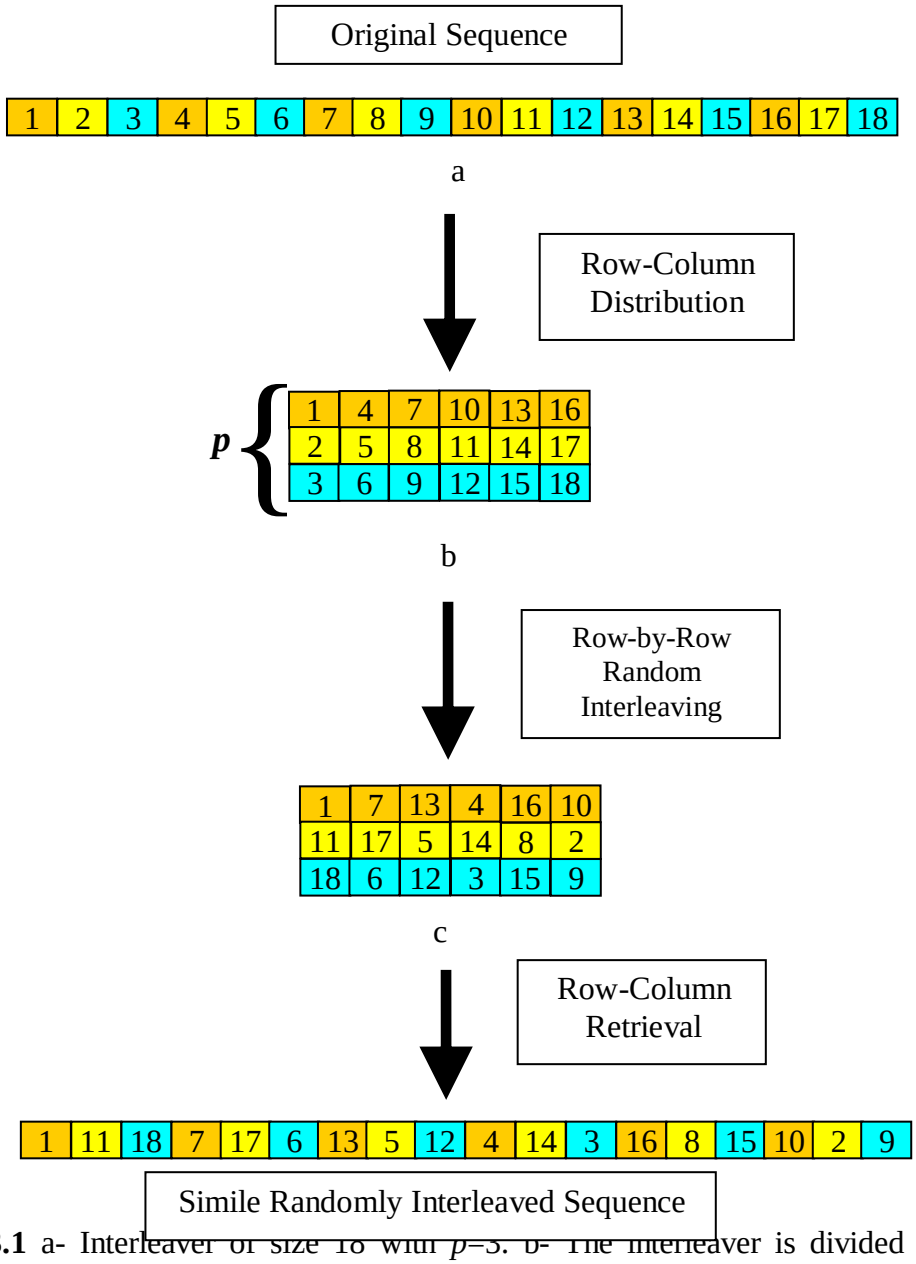


Fig. 3.1 a- Interleaver of size 18 with p=3. b- The interleaver is divided into p sequences. c- The sequences interleaved individually. d- The interleaver reconstructed.

From point (iv) it is clear that a simile interleaver can be merged with other types of interleavers [3]. This constraint leads to a better result on the behavior of

the performance graphs. It is hard to implement the simile constraint on the design of S-Random interleaver due to the reordering of the bits.

3.3 The New Interleaver Design: Circle Distribution Interleaver

It has been shown that the interleaver design is focused mainly on two targets

i- The trellis termination for forcing both encoders to end at the zero state, where this problem is solved by applying the simile constraint.

ii- Avoid generating finite length code sequences out of both encoders.

The second target is handled in this work by presenting a new method for distributing the bits. The method is to separate all the data items as far as possible from each other in a way that ensures minimizing the possibilities of producing finite codewords.

The algorithm is explained as follows

i- Shortest distance: A distance between two numbers is measured as either the inner distance or the outer distance, where the inner distance is calculated as the absolute difference between the numbers while the outer distance is the sum of the distances with the edges of the interleaver bounds as can be seen in Fig. 3.2. The shortest distance is taken as the shortest one out of both. For example for an interleaver of size 100, the distances between the numbers 10 and 90 are calculated as 80 and 20 for the inner and outer distances respectively. The shortest distance in this case is the outer distance which is 20.

ii- The distribution of numbers is done on a circle which is filled sequentially with one number after the other.

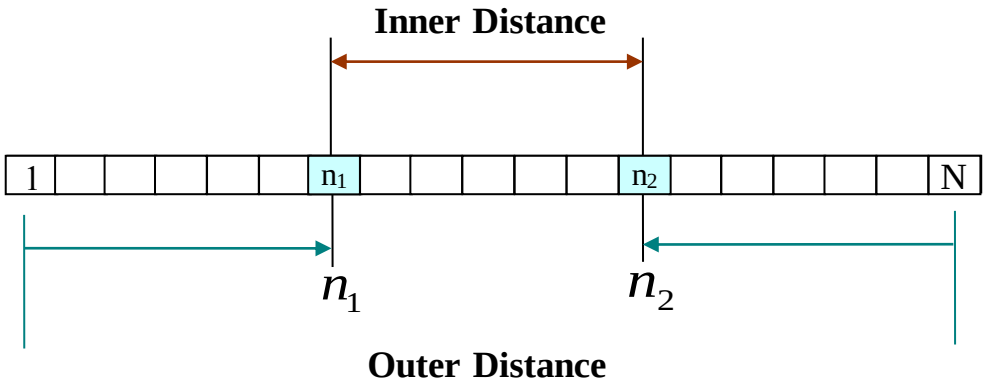


Fig. 3.2 Inner and outer distances between any two numbers.

iii- The algorithm used to distribute the numbers on the circle is given as follows

a- For a sequential circle of numbers, the distances between the numbers are defined as given in the example shown in Fig. 3.3.

b- Two sequential numbers are chosen and distributed on the circle, see Fig. 3.4.

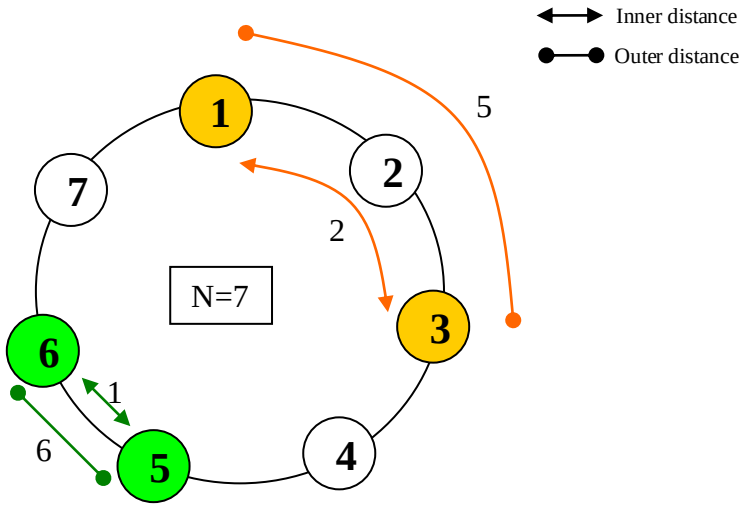


Fig. 3.3 Example of an interleaver of size 7. The figure shows the inner and outer distance for two numbers.

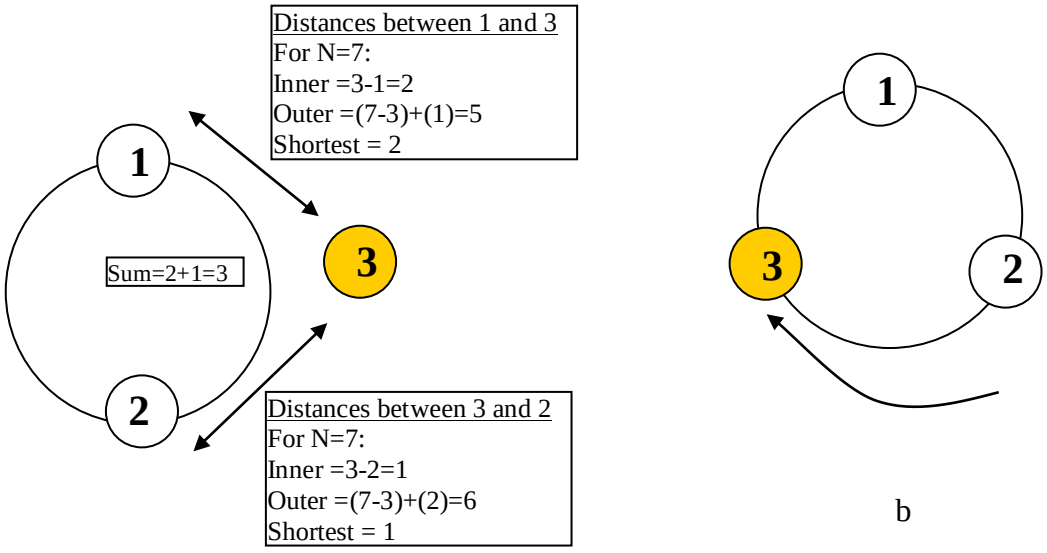


Fig. 3.4 a- The first two numbers are distributed on the circle. A third number searches the distances between the two numbers. b- The number is inserted into any of the two sides which is selected to be the last.

c- A new number is chosen sequentially after the previously selected ones.

d- The new number cycles through all the set of numbers within the circle and keeps a memory of the sum of the shortest distances between it and every two adjacent numbers, see Fig. 3.5a.

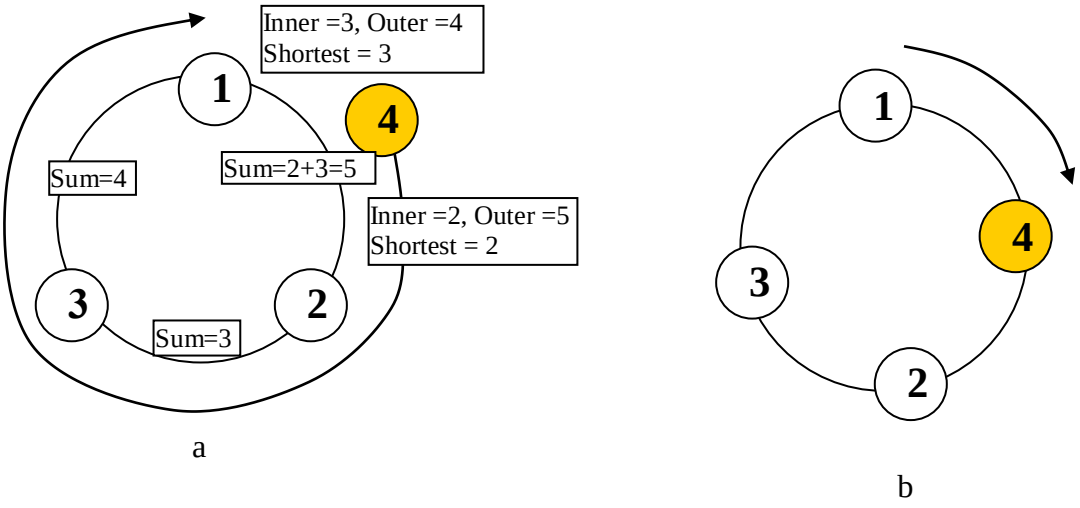


Fig. 3.5 a-The fourth number keeps searching all the distances between any two sets of numbers in the circle. b- It chooses to fit itself in the position where the sum of the two distances is the maximum.

e- Finally it chooses to insert itself between the set of two numbers that have the maximum sum of saved short distances If there exist more than one position that have the same maximum distance, the last one is selected for programming efficiency purposes, see Fig. 3.5b.

f- The process continues until all the sets of numbers are inserted, see Fig. 3.6.

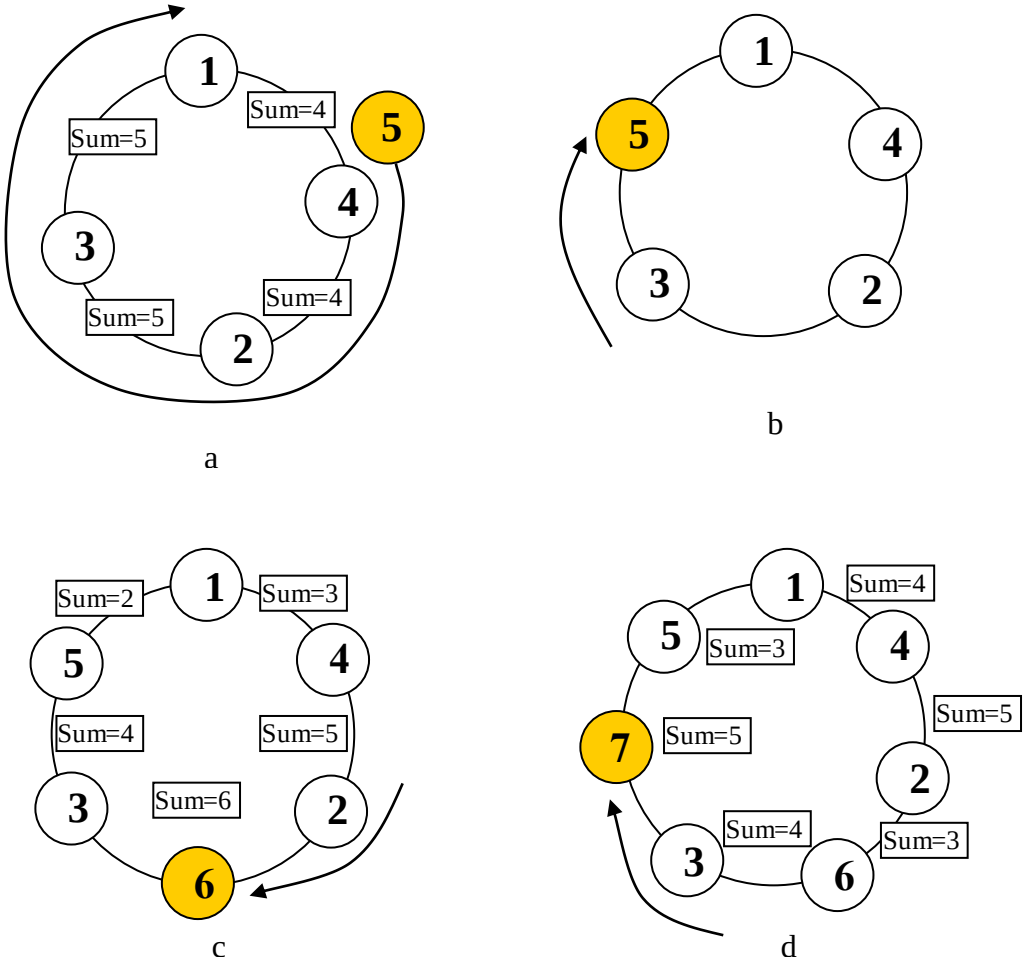


Fig. 3.6 The rest of the numbers are distributed according to the algorithm.

3.4 Software Implementation of the Proposed Interleaving Scheme

In this work, the framework environment for the analysis of turbo codes is programmed using the Object Oriented Programming (OOP) approach based on C++ language. This section provides a description of the implementation program and algorithms used in the work of simulation.

3.4.1 Procedural Programming

For most programmers, procedural programming languages are always the preferred choice over the much complicated OOP such as C++. It is well known that procedural programs depend on functions that are grouped into libraries. These functions are used to process input data according to the desired tasks which are usually noticed by their names. At the first sight the method seems reasonable and no demanding improvements are required. This is correct for programs that are directed for write once benefit once problems because any simple change required to be done on some function would affect the flow of the algorithm for the complete program.

Object oriented programming uses different approaches in solving problems; it uses programming blocks called objects to describe the components of the problem. An object contains properties that hold its status and contains methods that change these properties. A program might contain a single or more objects that could do their required tasks by calling their methods. The object itself might contain as many objects as it would need. If a problem is detected, each object is tested individually to make sure it does its work correctly. Objects are not the programs themselves; they are instances of special descriptive programs called classes. Classes describe the data that the objects represent and hold all the functions that are to be acting on these data. To modify or develop an OOP program, simply add new functionality and improvements to the classes that represent them without affecting the implementing program.

3.4.2 The Developed Software Package

In this work, a library of classes called "**GenData.lib**" is developed and programmed to access the turbo codes environment. The classes represent each component of the turbo coding system individually. The library includes classes that initiate turbo coding simulations with different layouts and parameters. Each class might have constructors, access functions and helper functions. Some classes have public member variables and some have file handling utilities as shown in Fig. 3.7. The name of the class reveals its use and type.

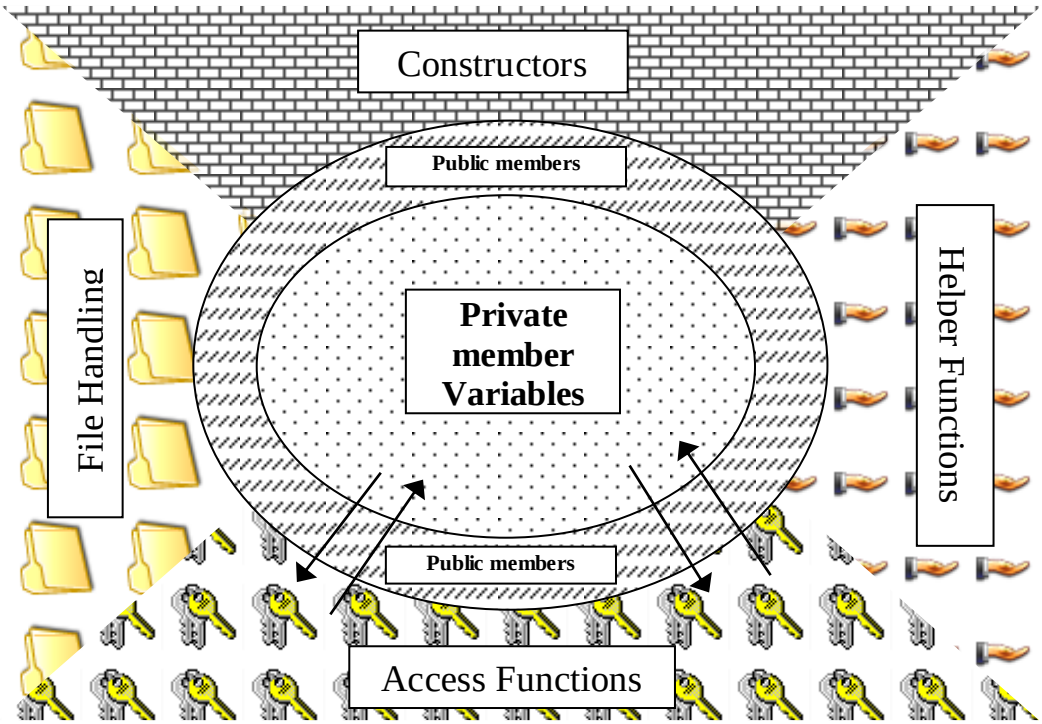


Fig. 3.7 A typical class structure.

The main set of these classes are explained as follows

a- Data Stream Class (DataStr)

This class is used to instantiate a stream of data of a specific size, the data are represented as an array of type "double" values. Associated with the stream is a group of functions that are collectively used to manipulate the data with the desired tasks.

Constructors

i- "**DataStr(int s)**" : creates the stream with the specified size. The data within the stream are not initialized.

ii- "**DataStr(const DataStr& dataStream)**" : default assignment constructor.

iii- "**DataStr(DataStr* dataStream)**" : creates a data stream copying the contents of the given dataStream parameter.

Access functions

i- "**DataStr& operator = (const DataStr& dataStream)**" : default assignment function.

ii- "**void setBits()**" : assigns random bits values to the stream.

iii- "**void setBits(double choice)**" : assigns the given choice to all the bits.

iv- "**void setBits(double* copy)**" : assigns the bits the contents of the given copy array.

v- "**double* getBits()**" : returns a pointer to the bits.

vi- "**double* getCBits() const**" : a constant version of the above function.

vii- "**int getSize() const**" : returns the size of the stream.

Helper functions

i- "**DataStr permute(const InterMat *mat, bool option)**" : returns a permuted version of the given data stream according to the given

interleaver matrix. The option parameter indicates whether to interleave or deinterleave the stream; true=interleave, false=deinterleave.

ii- "**DataStr encode(StateTran *encoder)**" : returns parity data stream from the given data according to the given encoder.

iii- "**DataStr hard()**" : returns hard limited version of the data. (+1.0,-1.0).

iv- "**void tail(StateTran *encoder)**" : tails the last bits of this data to lead to the zero state according to the given encoder in use and zero state initialization is assumed.

v- "**int compare(DataStr *original)**" : compares the data with the given data stream and returns number of errors.

vi- "**void halfPuncture(bool position)**" : punctures the data for a rate 1/2 simulation, this is used to omit the information received at the decoder as a result of puncturing occurring at the encoder. The value of position determines the starting point of puncturing; false=start puncturing from first number, true=start from second number.

File Handling

i- "**bool saveData(const char* fileName)**" : saves the data stream to a file with the name specified and returns true on success.

ii- "**bool loadData(const char* fileName)**" : loads the data from a file with the name specified, and returns true on success.

b- Interleaver Matrix Class (InterMat)

This class constructs the interleaver matrix array according to the given interleaver type. The class supports the addition of the simile constraint on the design and it is used to interleave or deinterleaver the data according to the desired tasks.

Constructors

i- "**InterMat(Interleaver l, int blockSize, int index=1)**" : constructs the array of interleaving matrix given the option of the desired interleaver and desired block size. The index argument is used by the interleavers for starting point distribution options.

ii- "**InterMat(Interleaver l, int N, int p, bool simile)**" : constructs a simile interleaver concatenating the given interleaver type. N (the interleaver size) must be divisible by the delay unit of the encoder (p). The delay unit p is given by $p = 2^{v} - 1$, where v is the number of delay elements of the encoder. The simile option is just to indicate the simile operation, recommended to always be true.

Access function

"**int* getMat() const**" : returns a pointer to the matrix of interleaving.

Helper Function

"**void isConsistent()**" : a diagnostic function that returns true if the built interleaver has the distribution numbers distributed from 1 to matrix size.

File Handling

i- "**bool saveMat(const char* fileName)**" : saves the matrix to a file with the specified name and returns true on success.

ii- "**bool loadMat(const char* filename)**" : loads the matrix from a file with the specified name and returns true on success.

c- State Transition Class (StateTran)

An instance of this class represents an RSC encoder with a latent number that holds the encoder current state. The class supports RSC encoders of types 4, 8 and 16 states with different feed-back and feed-forward functions.

Constructors

i- "**StateTran(int s=0)**" : constructs the default transition table of states and initializes the state to the given input or to the zero state by default.

ii- "**StateTran(bool g1[], bool g2[])**" : constructs a zero initialized transition table according to the given g1 and g2 arrays. The input arrays must be of size 5, also the element g1[0] must always be true to complete the feedback equation.

Access functions

i- "**void reset(int newState=0)**" : assigns the given state to the object or reset it to the zero state by default.

ii- "**int getState()**" : returns the current state of the object.

Helper functions

i- "**bool getParity(int input)**" : returns the bit result of adding the forward transfer function on the given state.

ii- "**int getNextState(int input)**" : returns the next state according to the given input.

iii- "**int getPreviousState(int input)**" : returns the previous state if the given input is assumed.

iv- "**double tailer(int state)**" : returns a value to be used by a data stream to lead to the zero state if the given state is assumed.

v- "**int getMemories()**" : returns the number of active memory elements of the encoder.

vi- "**bool isValid()**" : returns the validity state of the encoder, an encoder is valid if all its states have previous states for inputs 0 and 1.

vii- "**int filter(int state)**" : omits the unused bits for 8 and 4 states cases according to the type of the encoder.

d- Decoding Class (BCJR)

This class is used to build a decoding component object. The decoder is built using the optimized BCJR algorithm. The class supports both Log-MAP and Max-Log-Map algorithms.

Constructor

"**BCJR(DataStr *message, DataStr *parity, double Lc, StateTran *encoder)**" : constructs and initializes a BCJR component with the given message, parity and encoder in use. The reliability of the channel value Lc given to the constructor is calculated from the equation $Lc = 4SNR\_channel$ where *SNR_channel* is the signal-to-noise ratio of the channel.

Access functions

i- "**void setLeIn(DataStr *extrinsic)**" : sets the input extrinsic information to the decoder. This function must be called before processing the decoder.

ii- "**DataStr getLeOut()**" : processes the overall data and calculates the output extrinsic information from the component and returns the result.

iii- "**DataStr getFinal()**" : adds (input message + input extrinsic + output extrinsic) as the final decoding result. Both setLeIn and getLeOut functions must be called before calling this function.

e- Channel Simulation Class (Channel)

This class represents the process of simulating channel effects on the streaming data and implements the decoding process for recovering the original signal. The Channel class supports functions for adding noise, decoding and comparing the streams with original information signal.

Constructors

i- "**Channel(DataStr *msg, DataStr *par1, DataStr *par2, double SNR_channel, bool punctured=false)**" : constructs three DataStr

objects and initializes them using the given inputs. The given SNR_channel represents the SNR of the output of the encoder and the punctured option is used for describing the punctured status of the sent parities; false=not punctured, true = punctured.

ii- "**Channel(DataStr *msg, double SNR_channel)**" : constructs single DataStr object and initializes it using the given input. Using this constructor disables the coding specialized functions (i.e. decode function).

Access functions

i- "**DataStr getDat(int choice)**" : returns a data stream object that represents a copy of one of the three internal DataStr objects. The given choice is the DataStr object to be returned; 0=message,1=par1 and 2=par2.

ii- "**bool isPunctured()**": returns the state of puncturing.

Helper functions

i- "**void applyNoise()**" : applies the noise of the channel to the data.

ii- "**DataStr decode(int iterations, InterMat *interleaver, StateTran *encoder)**" : decodes the three DataStr objects with the turbo decoding algorithm discussed in chapter two according to the given interleaver and decoder types in use. The iterations argument specifies the number of iterations that the decoder will simulate.

3.4.3 The Implemented Software

A block diagram of the implemented software is shown in Fig. 3.8. A block of data of size N is applied to Encoder1 and at the same time is interleaved by an interleaver and applied to Encoder2. The original source of information and the two encoded data blocks are fed to the channel. The channel block applies the noise according to the required SNR assuming ±1 signal is transmitted. The puncturing is handled also by the channel by neglecting the punctured positions of the sent data. The decoding is done inside the channel class. The resultant estimation block stream

is compared with the original source to calculate the number of errors. The resultant probability of error is plotted against the applied SNR to find the Bit Error Rate (BER) characteristics.

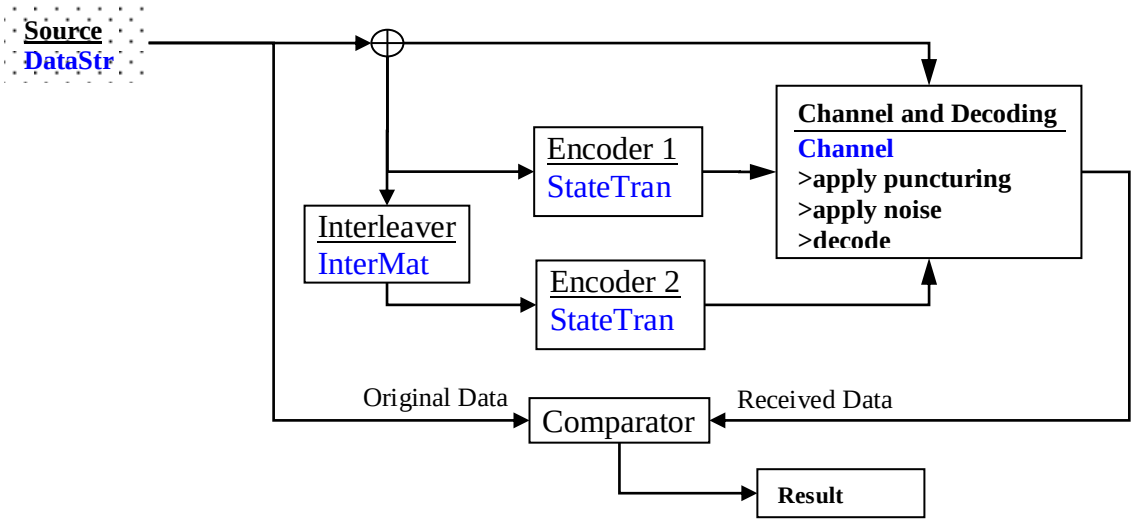


Fig. 3.8 Block diagram of the implemented software.

The implementation of the library is done using Microsoft® Visual C++6 under Microsoft® Windows Operating System (OS). A program named "**TurboSim.exe**" is programmed for simulation. The Graphical User Interface (GUI) of the program is shown in Figs. 3.9 and 3.10. The GUI allows simulating different tests. The standard characteristics form of parameters shown on the right provides a tool for changing size of the interleaver, number of packets involved, number of iterations, puncturing states and others. If the number of packets is set to zero, the program will simulate an uncoded system and the interleaver size will be the number of uncoded bits sent through the channel. The program provides the use of different RSC encoder shapes with memory number equal to 2, 3, or 4 memory elements.

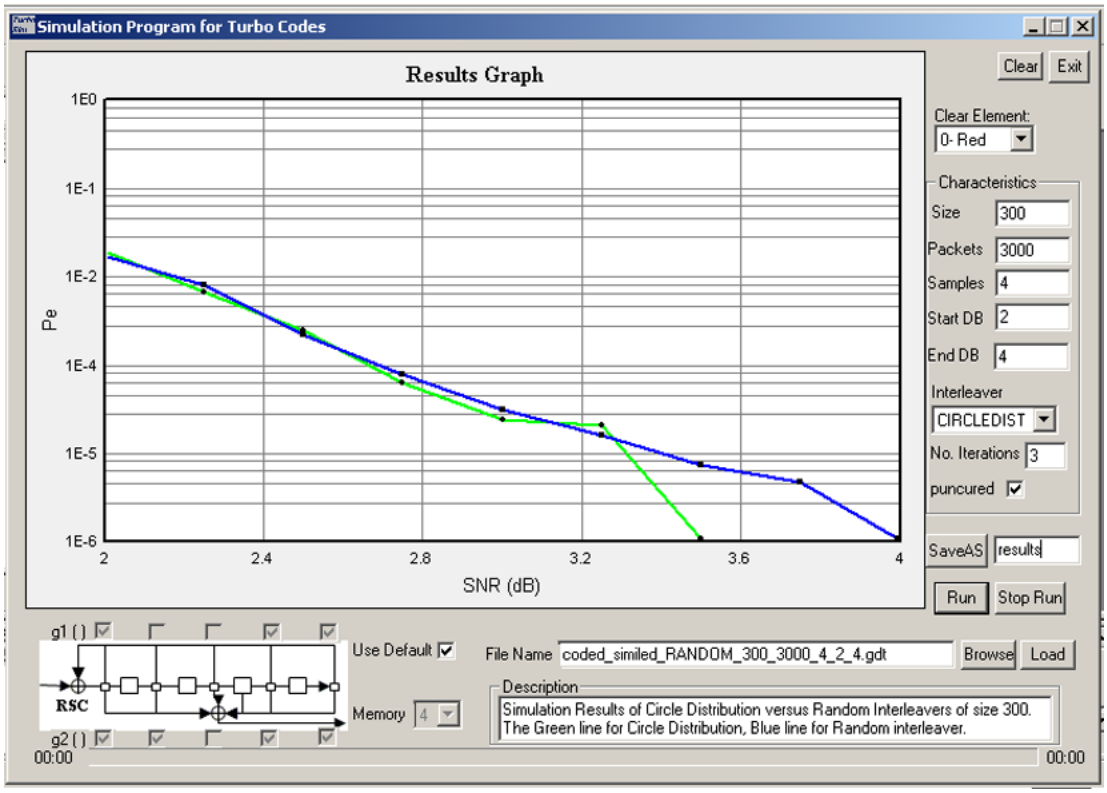


Fig. 3.9 GUI view of the TurboSim program.

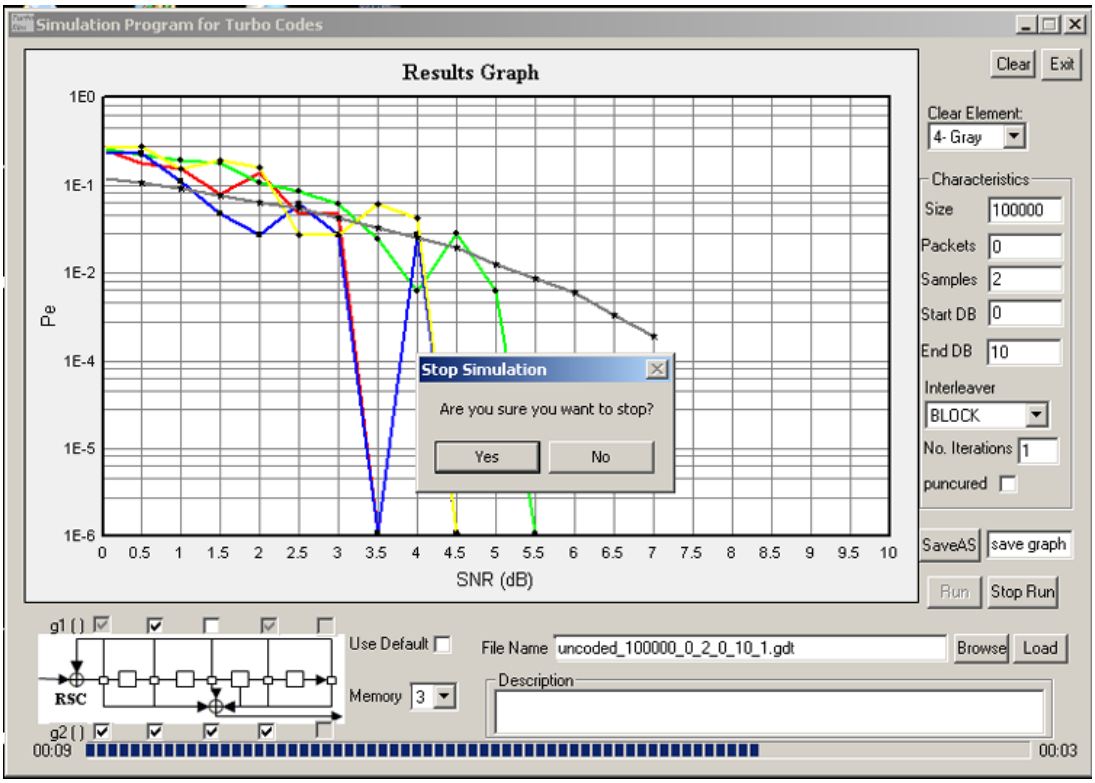


Fig. 3.10 GUI view of TurboSim Program showing the capabilities of stopping simulation, multi-drawing and time handling.

Chapter Four
Simulation Results

4.1 Introduction

The "TurboSim" program discussed in the previous chapter is used here to obtain simulation results for the proposed interleaver. Results related to S-random interleaver are also included for comparison purposes. The parameters of the interleaver are changed to reflect their effects on the system performance. The performance comparison is implemented by plotting the number of errors related to a given frame size signified as probability of errors (Pe) against the Signal-to-Noise Ratio (SNR). Unless otherwise specified, the parameters used in the simulations are given as follows

a. Interleaver size = 315.
b. Number of RSC memories = 4.
c. Number of iterations = 3.
d. Simile constraint is applied.
e. Puncturing is implemented.

4.2 Effect of Number of Iterations

The number of iterations has a very obvious effect on improving the decoding result. Increasing the number of iterations increases the commutation between received noisy signals and increases the confidence in the correct choice [19]. The size of the interleaver is chosen to be 10000 bits long with no simile constraint to overcome high error floor results. The simulation results presented in Figs. 4.1 and 4.2 clearly show the effect of increasing the number of iterations on the performance of turbo codes. The number of iterations used in the following simulation tests is chosen to be 3 as a compromise between processing time and performance enhancement. The results also show that the proposed interleaver has no significant performance improvement over the random interleaver at a fixed value of number of iterations and in the absence of the simile constraint.

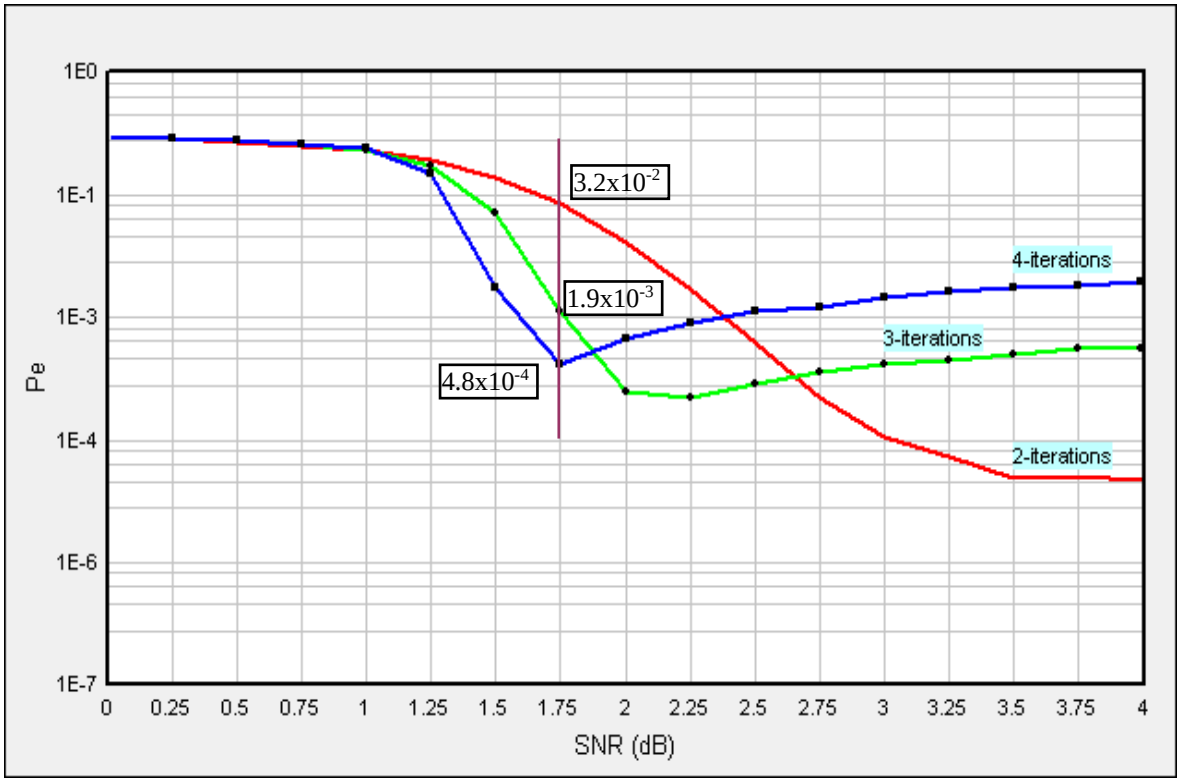


Fig. 4.1 Effect of the number of iterations on the performance of the proposed interleaver in the absence of simile constraint.

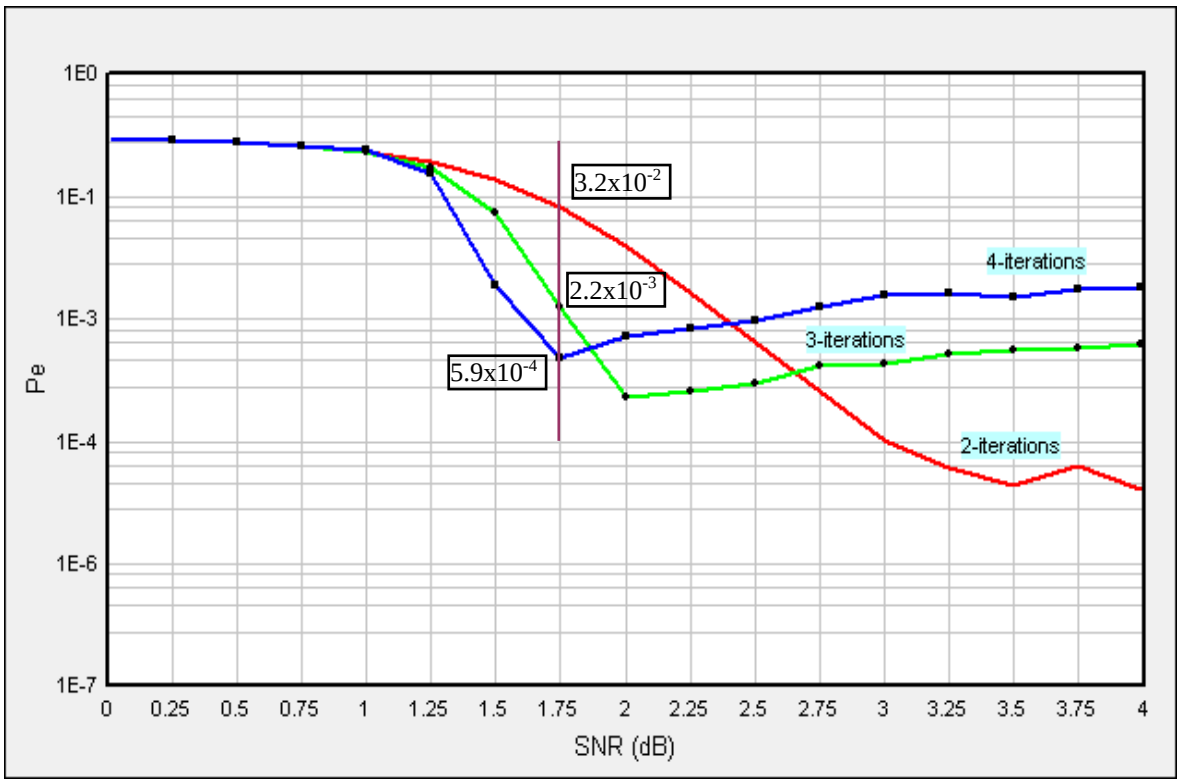


Fig. 4.2 Effect of the number of iterations on the performance of a random interleaver in the absence of simile constraint.

For example, a SNR of 1.75 dB offers a BER of $3.2\text{x}10^{-2}$, $1.9\text{x}10^{-3}$ and $4.8\text{x}10^{-4}$ when the proposed interleaver is implemented with number of iterations 2, 3 and 4, respectively. These values are to be compared with $3.2\text{x}10^{-2}$, $2.2\text{x}10^{-3}$ and $5.9\text{x}10^{-4}$, respectively, when a random interleaver is used. Note further, increasing the SNR to 3.5 dB yields a BER of $2.4\text{x}10^{-5}$, $6.3\text{x}10^{-4}$ and $3.6\text{x}10^{-3}$, when the proposed interleaver is implemented with number of iterations 2, 3 and 4, respectively. For a random interleaver, the BER becomes $1.9\text{x}10^{-5}$, $6.9\text{x}10^{-4}$ and $3.0\text{x}10^{-3}$, respectively at SNR 3.5 dB.

4.3 Effect of Simile Constraint

The decoder architecture discussed in section 2.3 is based on the trellis transition of states. It assumes that both encoders start and end at the same state. The termination of the first encoder is easily handled by the addition of tailing bits but the interleaver complicates the process for the second encoder. The simile constraint discussed in section 3.2 solves the problem. The simile constraint works with only interleaver sizes divisible by p parameter given by $p = 2^{\nu} - 1$ where ν=number of memory elements in the encoder.

The test run in the previous section is repeated here with simile constraint applied to an interleaver of size 10005 with ν=4 and p=15. The constraint is applicable here because the number N=10005 is divisible by the number p=15 (N mod p=0). The results are shown, respectively, in Figs. 4.3 and 4.4 for the proposed and a random interleaver. The results shown in Fig. 4.4 show the effect of ending both encoders at the same state for different numbers of iterations for the random interleaver. The conclusion can also be noted for the proposed scheme with no significant change in performance as shown in Fig. 4.3. For example, a SNR of 1.75 dB yields a BER of $3.0\text{x}10^{-2}$, $1.6\text{x}10^{-3}$ and $5.4\text{x}10^{-6}$ when the proposed interleaver is implemented with number of iterations 2, 3 and 5, respectively. The values are to be compared with $3.0\text{x}10^{-2}$, $1.2\text{x}10^{-3}$ and $6.3\text{x}10^{-6}$, respectively, when a random interleaver is used.

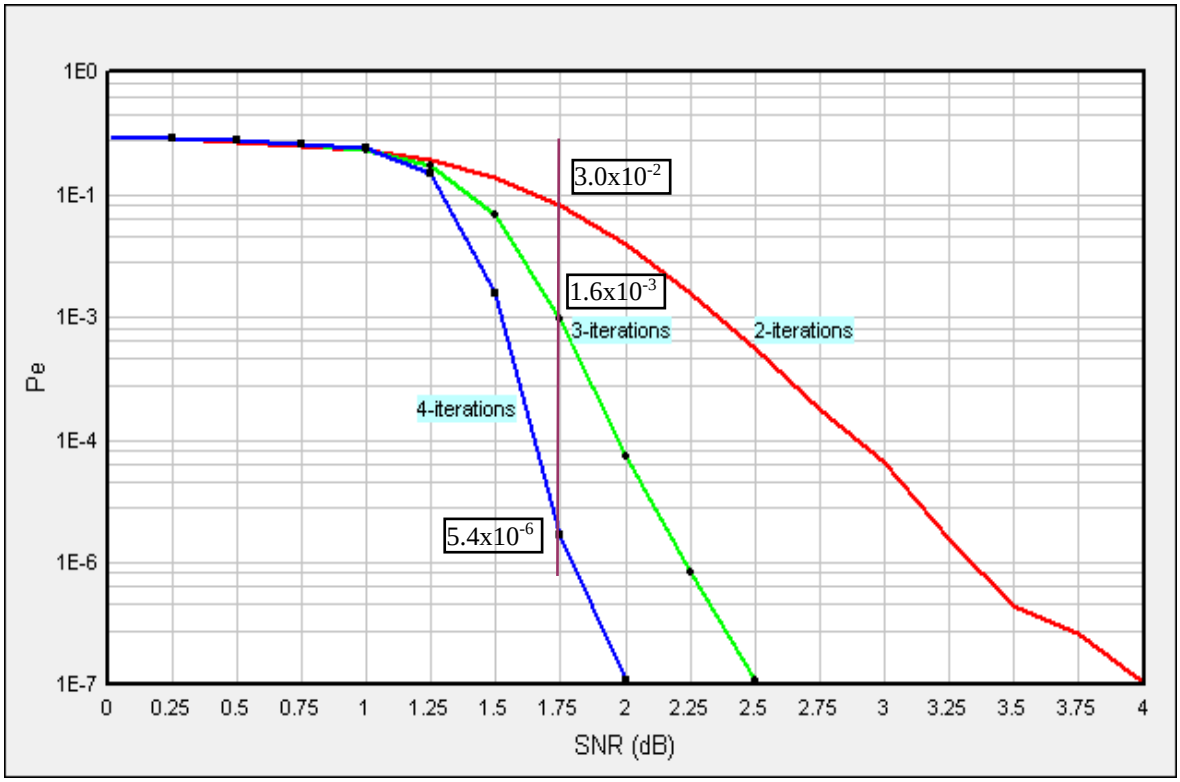


Fig. 4.3 Effect of simile constraint on the performance of the proposed interleaver for different iteration numbers.

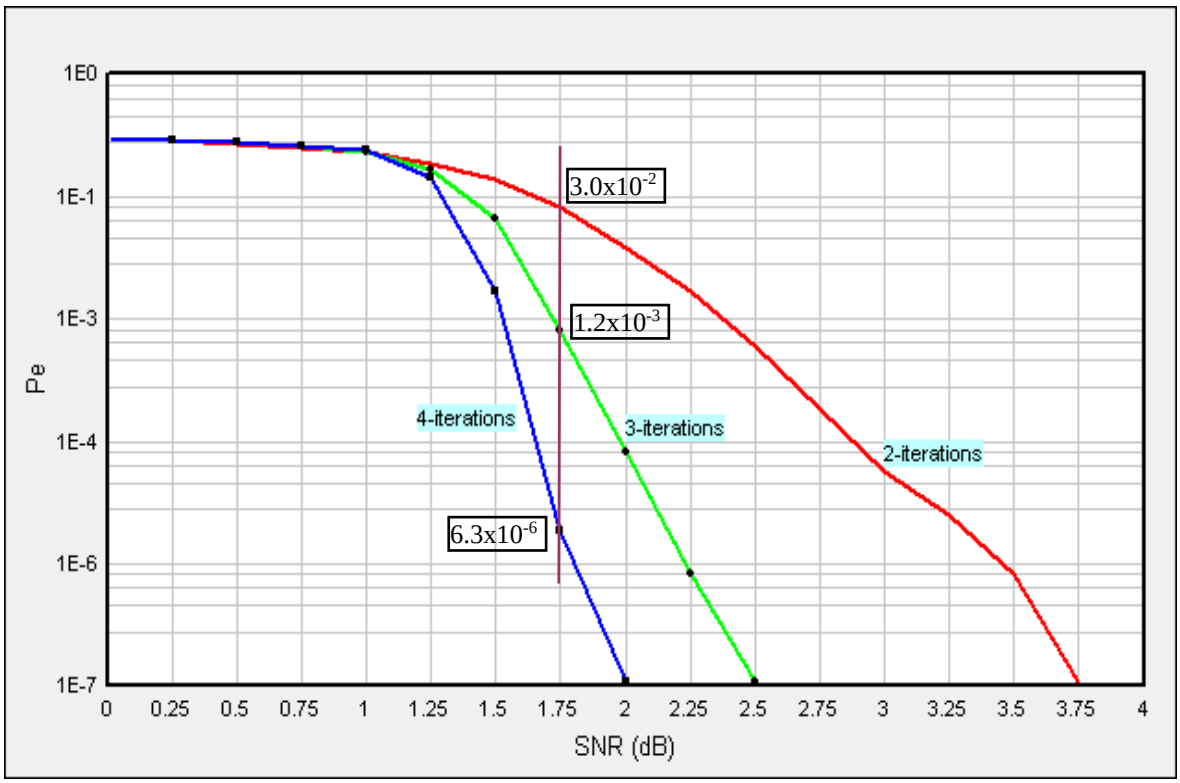


Fig. 4.4 Effect of simile constraint on the performance of a random interleaver for different iteration numbers.

4.4 Effect of Interleaver Size

The interleaver size has a significant effect on the design. Increasing the size of the interleaver decreases the probability of generating low weight codewords. Additionally the size of interleaver has an "Interleaving gain" effect on the performance as discussed in section 2.4. The effect for changing the size of the proposed interleaver is tested for 315, 1005 and 10005 bit lengths as shown in Fig. 4.5. The same tests are run for a random interleaver as shown in Fig. 4.6. The results show that increasing the size of the interleaver enhances its performance. For a SNR= 2.25 dB, the proposed interleaver gives a BER of $2.3x10^{-3}$, $1.4x10^{-4}$ and $2.0x10^{-6}$ when the interleaver size is set to 315, 1005 and 10005, respectively, these values are to be compared with BER of $2.6x10^{-3}$, $1.5x10^{-4}$ and $2.0x10^{-6}$, respectively, for a random interleaver. For practical implementations minimizing the size of the interleaver is a demanding constraint in the design of communication systems to overcome processing limits.

4.5 Effect of Puncturing

Puncturing in turbo coding is used to increase the code rate of the system. However, the performance of the system is decreased because it sends less information about the original signals.

The effect of puncturing on the performance of the proposed and random interleavers is shown, respectively, in Figs. 4.7 and 4.8. At SNR= 2.75 dB, the proposed interleaver offers a BER of $7.1x10^{-5}$ and $1.0x10^{-7}$ in the presence and absence of puncturing, respectively. These values are to be compared with a BER= $2.2x10^{-4}$ and $4.0x10^{-6}$, respectively, for the random interleaver at SNR=2.75 dB.

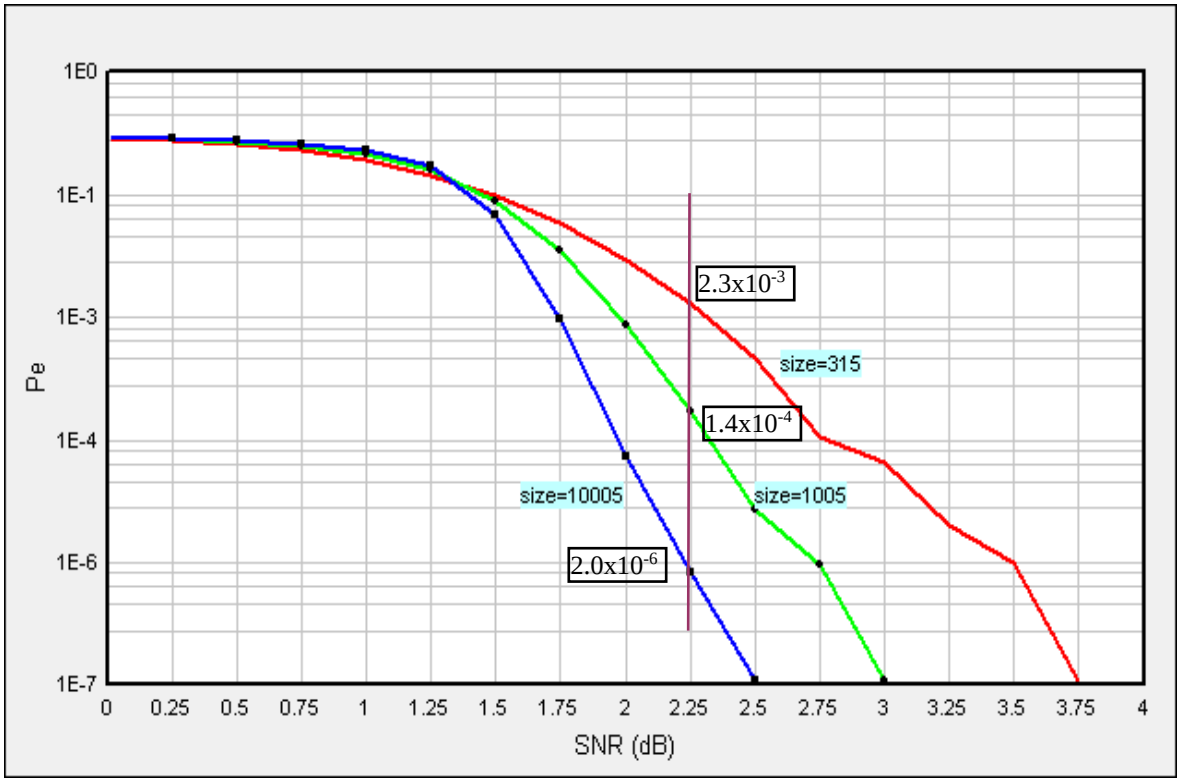


Fig. 4.5 Effect of interleaver size on the performance of the proposed interleaver designed with simile constraint.

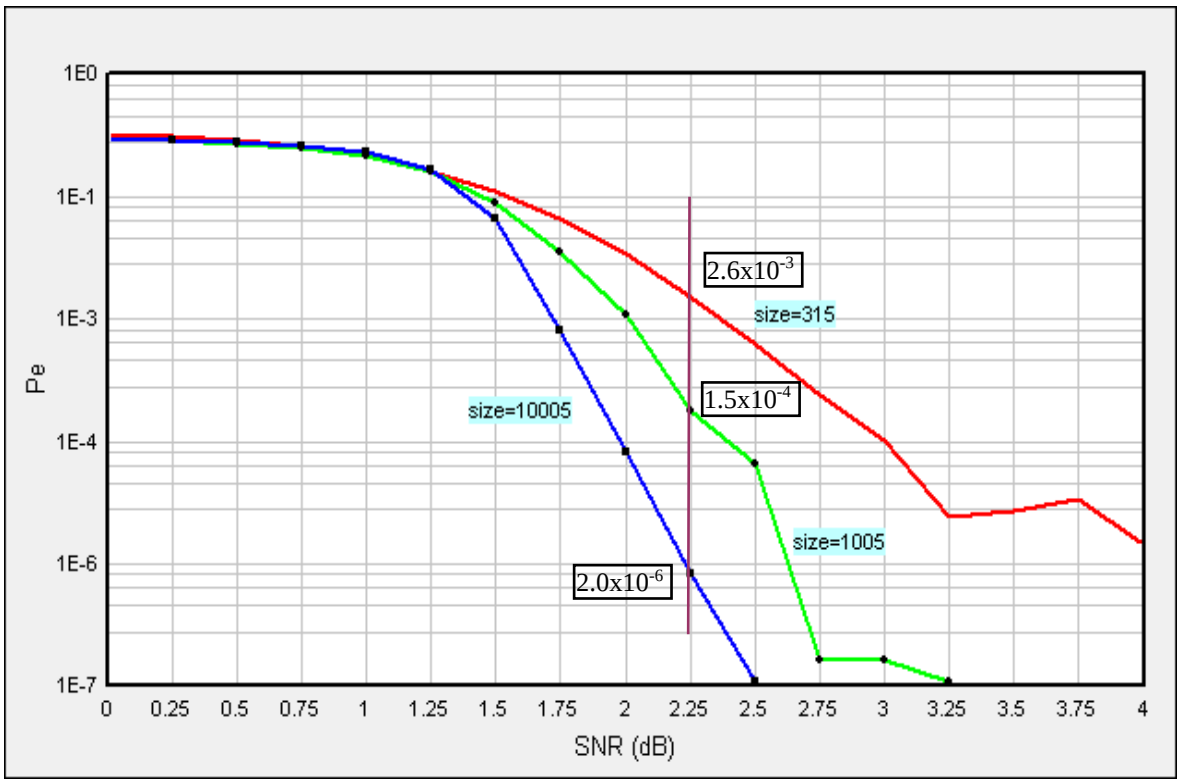


Fig. 4.6 Effect of the interleaver size on the performance of a random interleaver design with simile constraint.

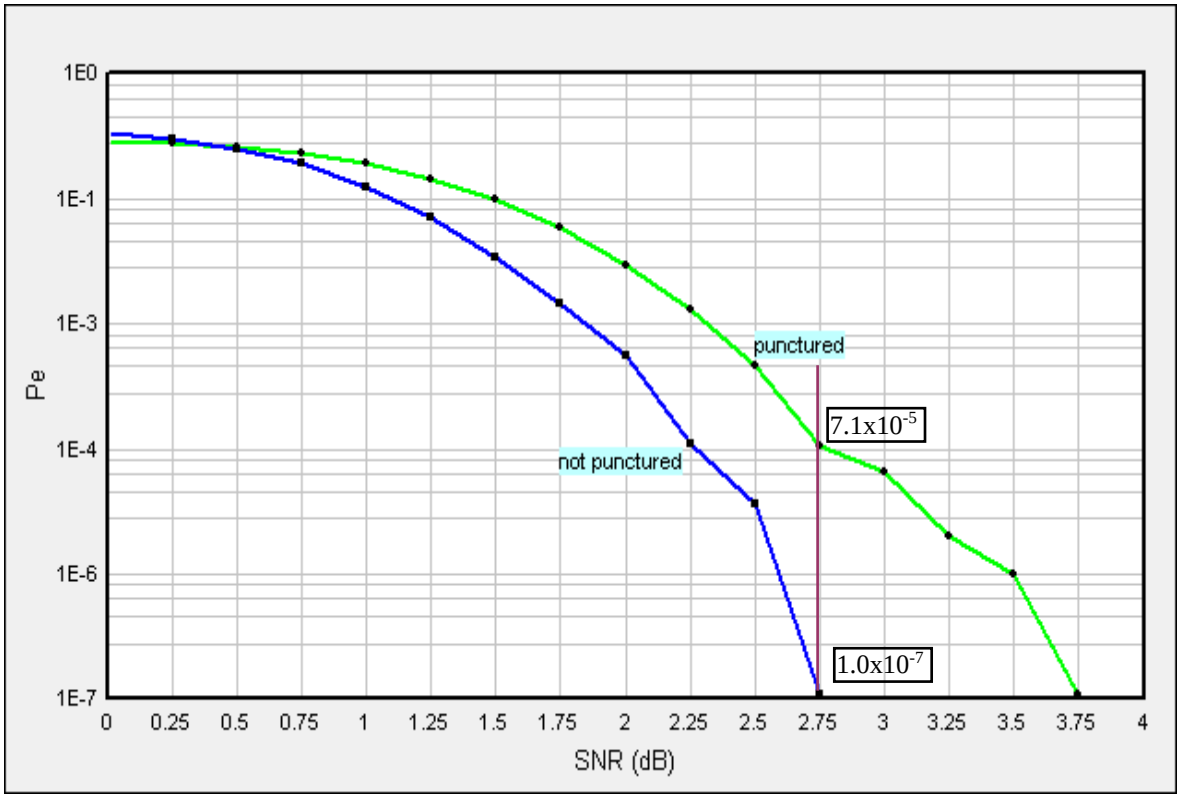


Fig. 4.7 Effect of puncturing on the performance of the proposed interleaver.

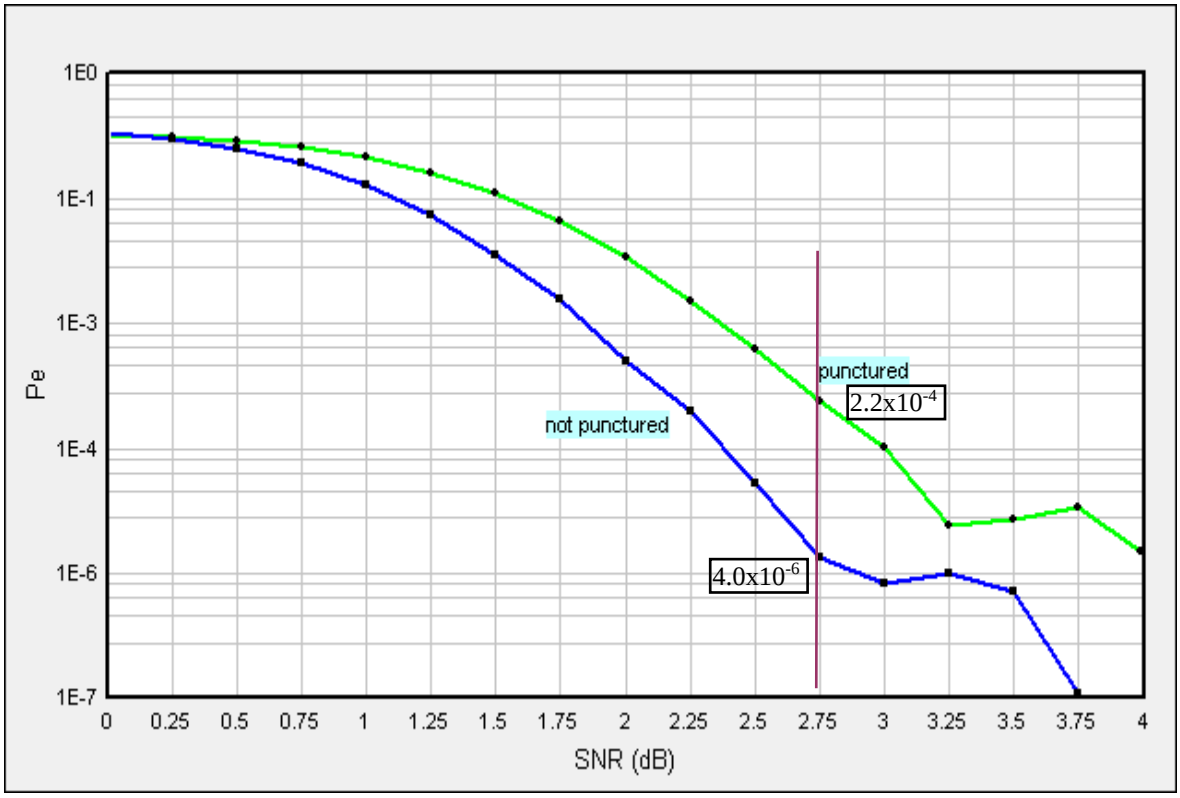


Fig. 4.8 Effect of puncturing on the performance of a random interleaver.

4.6 Performance Comparison

The concept of the S-Random interleaver is discussed in Chapter 2. The circle distribution interleaver design discussed in section 3.3 is based on the same concepts but a different approach is used to distribute the symbols. Therefore, it is worth getting clear comparison picture between their performances. In this subsection, the performance of both interleavers are compared for turbo coding implemented with ν=3 and ν=4 and the results are depicted in Figs. 4.9 and 4.10, respectively, with SNR> 2.75 dB, the proposed interleaver offers a better BER performance and this effect is more pronounced as ν increases from 3 (i.e, 8-state encoder) to 4 (i.e, 16-state encoder). For example, at a SNR= 3.25 dB, the proposed interleaver offers a BER of $1.8\text{x}10^{-5}$ and $1\text{x}10^{-6}$ when ν=3 and 4, respectively. These values are to be compared with $6.8\text{x}10^{-5}$ and $1.5\text{x}10^{-5}$, respectively, for the random interleaver.

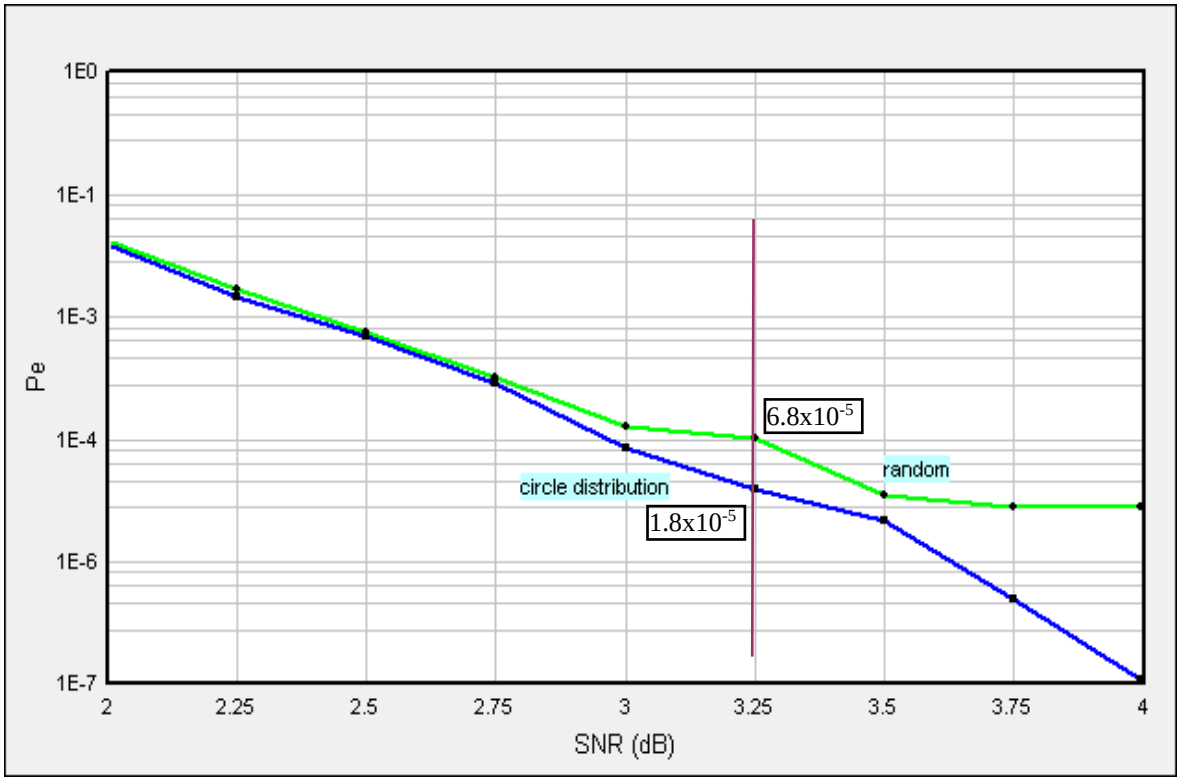


Fig. 4.9 Effect of implementing the proposed interleaver over a random interleaver for an 8-state encoder (memory= 3).

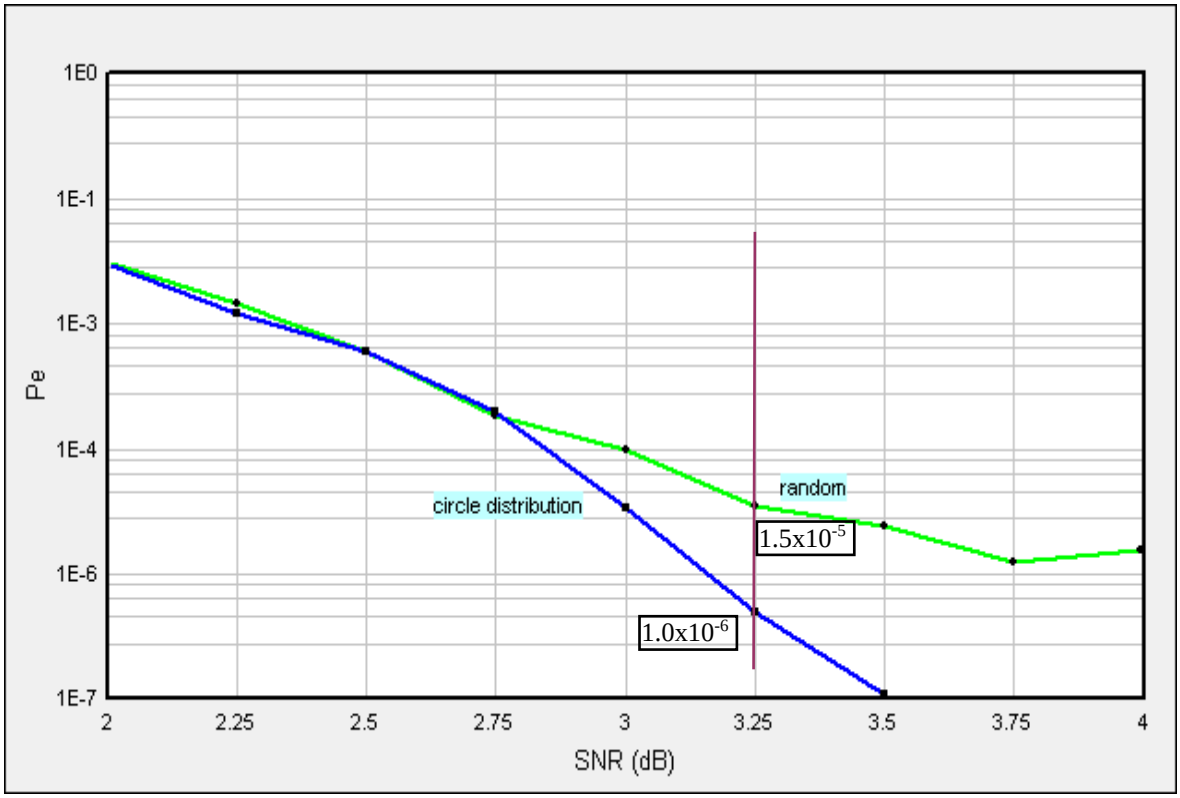


Fig. 4.10 Effect of implementing the proposed interleaver over a random interleaver for a 16-state encoder (memory= 4).

Chapter Five
Conclusions and Suggestions for Future Work

5.1 Conclusions

In this thesis, a new interleaver design is proposed which is based on a systematic non-random approach. Further, a software framework for turbo coding simulation is built based on object oriented programming approach. The main conclusions drawn from this study are

i- Increasing the number of iterations on the decoder side improves the system performance notably. At SNR= 1.75 dB, the system offers a BER of 3.2×10^{-2}, 1.9×10^{-3} and 4.8×10^{-4} when the proposed interleaver is implemented without simile constraint with number of iterations 2, 3 and 4, respectively.

ii- Increasing the size of the interleaver improves the performance of the system. At SNR= 2.25 dB, the proposed interleaver gives a BER of 2.3×10^{-3}, 1.4×10^{-4} and 2.0×10^{-6} when the interleaver size is set to 315, 1005 and 10005, respectively.

iii- Ending both encoders at the same state is a very important constraint on the proposed interleaver to reduce the level of error floor. At SNR= 1.75, the proposed interleaver offers a BER of 3.0×10^{-2}, 1.6×10^{-3} and 5.4×10^{-6} for a number of iterations of 2, 3 and 4, respectively, when simile constraint is imposed. These values are to be compared with 3.2×10^{-2}, 1.9×10^{-3} and 4.8×10^{-4}, respectively, when no simile constraint is implemented. After the given SNR value, the performance degrades for the latter case.

iv- Puncturing the sent data to increase the code rate reduces system performance. At SNR= 2.75 dB, the proposed interleaver offers a BER of 7.1×10^{-5} and 1.0×10^{-7} in the presence and absence of puncturing, respectively.

v- The proposed interleaver performs better than the random interleaver when the SNR> 2.75 dB and this effect is more pronounced when a 16-state encoder is employed as compared with an 8-state encoder. At a SNR= 3.25 dB, the proposed interleaver offers a BER of 1.8×10^{-5} and 1.0×10^{-6} when ν=3 and 4, respectively. These values are to be compared with 6.8×10^{-5} and 1.5×10^{-5}, respectively, for the random interleaver.

5.2 Suggestions for Future Work

The following issues are suggested for future investigation

i- The algorithm of the circle distribution interleaver can be improved by making the search region lie within four adjacent numbers instead of two. This will allow the distribution of bits with higher distances between adjacent numbers especially when it is used for lower memory RSC encoders.

ii- To implement the proposed scheme with 8-state and 4-state encoders but with improved starting points.

iii- To use different approaches to terminate both encoders at the same state.

iv- To develop software classes to support different kinds of interleavers and use the simulation results for comparison purposes with those related to the proposed interleaver.

References

[1] 23: K. Sripimanwat, "Turbo code applications: A journey from a paper to realization", Springer, Netherlands, 2005.

[2] 11: A. Glavieux, "Channel coding in communication networks from theory to turbo codes", Antony Rowe Limited, Chippenham, Wiltshire, 2007.

[3] 36: J. B. Anderson, "Trellis and turbo coding", IEEE Press and John Wiley and Sons Inc., New Jersey, USA, 2004.

[4] 48: M. Breiling, "A logarithmic upper bound on the minimum distance of turbo codes", IEEE Transactions on Information Theory, vol. 50, no. 8, pp. 1692-1710, August, 2004.

[5] 25: P. Frossard, "FEC performance in multimedia streaming", IEEE Communications Letters, vol. 5, no. 3, pp. 122-124, March 2001.

[6] 24: D. Divsalar and F. Pollara, "Turbo codes for PCS applications", IEEE International Conference on Communications in Seattle ICC'95, vol. 1, pp. 54-59, June 1995.

[7] 59: L. Trifina, V. Munteanu, and D. Traniceriu, "Increasing S Parameter of Interleavers", International Symposium on Signals, Circuits and Systems, vol. 2, pp. 1-4, July, 2007.

[8] 15: H. R. Sadjadpour, M. Salehi, N. J. A. Solane, and G. Nebe, " Interleaver design for short block length turbo codes", IEEE Journal on Selected Areas in Communications, vol. 19, no. 5, pp. 1-4, May 2001.

[9] 16: X. Zhang, Y. Q. Shi, H. Chen, A. M. Haimovich, A. Vetro, and H. Sun., "Successive packing based interleaver design for turbo codes", IEEE International Symposium on Circuits and Systems ISCAS'02, vol. 1, pp. 17-20, 2002.

[10] 14: W. Feng, J. Yuan, and B. S. Vucetic, "A code-matched interleaver design for turbo codes", IEEE Transactions on Communications, vol. 50, no. 6, pp. 926-937, June, 2002.

[11] 18: F. Daneshgaran and P. Mulassano, "Interleaver pruning for construction of variable length turbo codes", IEEE Transactions on Information Theory, vol. 50, no. 3, pp. 455-466, March, 2004.

[12] 19: F. Daneshgaram and M. Laddomada, "Optimized prunable single-cycle interleavers for turbo codes", IEEE Transactions on Communications, vol. 52, no. 6, pp. 899-909, June, 2004.

[13] 17: Z. Hongyu, L. Wang, Q. Yuan, H. Wang, and J. Yu, "A chaotic interleaver used in turbo codes", International Conference on Communications Circuits and Systems ICCCAS '04, vol. 1, pp. 38-42, June, 2004.

[14] 13: J. Sun and O. Y. Takeshita, "Interleavers for turbo codes using permutation polynomials over integer rings", IEEE Transactions on Information Theory, vol. 51, no. 1, pp. 101-119, January, 2005.

[15] 20: L. Dinoi and S. Benedetto, "Design of fast-prunable S-random interleavers", IEEE Transactions on Wireless Communications, vol. 4, no. 5, pp. 2540-2548, September, 2005.

[16] 21: M. Laddomada and B. Scanavino, "A cost-function based technique for design of good prunable interleavers for turbo codes", IEEE Transactions on Wireless Communications, vol. 5, no. 8, pp. 1953-1958, August, 2006.

[17] 22: M. Arif and N. M. Sheikh, "A noval design of deterministic interleaver for turbo codes", International Conference on Electrical Engineering ICEE'07, Lahore, Pakistan, pp. 1-5, April, 2007.

[18] 57: Y. Tsai, S. Deng, K.Chen, and M. Lin, "Turbo Coded OFDM for Reducing PAPR and Error Rates", IEEE Transactions on Wireless Communications, vol. 7, no. 1, pp. 84-89, January, 2008.

[19] 41: S. Brink, "Convergence behavior of iteratively decoded parallel concatenated codes", IEEE Transactions on Communications, vol. 49, no. 10, pp. 1727-1737, October, 2001.

[20] 60: N. Y. Yu, M. G. Kim, Y. S. Kim, S. U. Chung, "Efficient stopping criterion for iterative decoding of turbo codes", IEEE Electronics Letters, vol. 39, no. 1, pp. 73-75, January, 2003.

[21] 33: M. R. Soleymani, Y. Gao and U. Vilaipornsawai, "Turbo coding for satellite and wireless communications", Kluwer Academic Publishers, Norwell, Massachusetts, USA, 2002.

[22] 39: A. J. Han Vinck, P. Dolezal, and K. Young-Gil, "Convolutional encoder state estimation", IEEE Transactions on Information Theory, vol. 44, no. 4, pp. 1604-1608, July, 1998.

[23] 52: A. Msir, F. Monteiro, A. Dandache, and B. Lepley, "A high speed encoder for recursive systematic convolutive codes", Proceedings of the Eighth IEEE International On-Line Testing Workshop, pp. 51-55, November, 2002.

[24] 12: Y. Liao, "A new short memory turbo code with good BER performance and low decoding complexity", IEEE 58th Vehicular Technology Conference VTC'03, vol. 5, pp. 3179-3182, October, 2003.

[25] 29: M. A. Kousa and A. H. Mugaibel, "Puncturing effects on turbo codes", IEE Proceedings in Communications, vol. 149, no. 3, pp. 132-138, June, 2002.

[26] 30: J. H. Gunther, D. Keller, and T. K. Moon, "A generalized BCJR algorithm and its use in turbo synchronization", IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '05), vol. 3, pp. 837-840, March, 2005.

[27] 45: A. J. Viterbi, "An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes", IEEE Journal on Selected Areas in Communications, vol. 16, no. 2, pp. 260-264, February, 1998.

[28] 1: W. J. Gross and P. G. Gulak, "Simplified MAP algorithm suitable for implementation of turbo codes", IEE Electronics Letters, vol. 34, no. 16, pp. 1577-1578, August, 1998.

[29] 31: W. E. Ryan, "Concatenated codes and iterative decoding", Wiley Encyclopedia of Telecommunications (J. G. Proakis, ed.) New York, Wiley and Sons, 2003.

[30] 32: B. Sklar, "Digital Communications: Fundamentals and Applications, 2nd Edition", Prentice Hall PTR, 2001.

[31] 44: C. Schurgers, F. Catthoor, and M. Engels, "Memory optimization of MAP turbo decoder algorithms", IEEE Transactions on Very Large Scale Integration (VLSI), vol. 9, no. 2, pp. 305-312, April, 2001.

[32] 46: S. ten Brink, "Iterative decoding trajectories of parallel concatenated codes", Proceedings on 3rd IEEE/ITG Conference Source Channel Coding, pp. 75-80, Munich, Germany, January, 2000.

[33] 40: J. Vogt and A. Finger, "Improving the max-log-MAP turbo decoder", IEEE Electronics Letters, vol. 36, no. 23, pp. 1937-1939, November, 2000.

[34] 34: H. Claussen, H. R. Karimi, and B. Mulgrew, "Improved Max-Log-MAP turbo decoding by maximization of mutual information transfer", EURASIP Journal on Applied Signal Processing, vol. 6, pp. 820-827, January, 2005.

[35] 47: S. Benedetto, G. Montorsi, and D. Divsalar, "Concatenated convolutional codes with interleavers", IEEE Communications Magazine, vol. 41, no. 8, pp. 102-109, August, 2003.

[36] 5: J. Hockfelt, O. Edfors, and T. Maseng, "On the theory and performance of trellis termination methods for turbo codes", IEEE Journal On Selected Areas in Communications, vol. 19, no. 5, pp. 838-847, May, 2001.

[37] 35: J. C. Moreira and P. G. Farrell, "Essentials of error control coding", John Wiley and Sons Inc., Weinheim, Germany, 2006.

[38] 51: V. Tarokh and B. M. Hochwald, "Existence and construction of block interleavers", IEEE International Conference on Communications ICC'02, vol. 3, pp. 1855-1857, New York, USA, April, 2002.

[39] 37: H. C. Lentmaier, M. Costello, and K. Zigangirov, "Designing linear interleavers for multiple turbo codes," IEEE Proceedings on 8th International Symposium on Communications Theory and Applications, pp. 252–257, July, 2005.

[40] 54: K. V. Koutsouvelis and C. E. Dimakis, "A low complexity algorithm for generating turbo code S-random interleavers", On-Line Springer Engineering Journal on Wireless Personal Communications, December, 2007.
http://www.springerlink.com/index/6h77745w25466258.pdf

[41] 43: M. S. C. Ho, S. S. Pietrobon, and T. Giles, "Improving the constituent codes of turbo encoders", IEEE Global Telecommunications Conference GLOBECOM'98, vol. 6, pp. 3525-2529, August, 1998.
[42] 53: I. Chatzigeorgiou and I. J. Wassell, "Revisiting the calculation of effective free distance of turbo codes", IEEE Electronics Letters, vol. 44, no. 1, pp. 43-44, January, 2008.

Appendix A
GenData.lib classes

In this Appendix, the list of all classes included in "GenData version 1.09" library is listed. A detailed description of the behavior and use of each class is given. Theses classes are used collectively in programming "TurboSim.exe" program. Each class might have constructors, access functions and helper functions. Some classes have public member variables and some have file handling utilities. The name of the class reveals its use and type.

1- Global parameters and functions

the global parameters and functions are used by the classes to access shared properties and general functions. The items are listed as follows

- "bool ADD_CORRECTION" : A constant Boolean indicating weather to add correction to the decoding algorithm, This directly influences the use of either Log-MAP or the Max-Log-MAP algorithms. Default = false indicating the use of Max-Log-MAP.
- "void add(bool a,bool b)" : Adds two Booleans logically.
- "enum Interleaver" : An enumeration that defines all the basic interleavers supported by this version of the library.

  The Interleavers supported are

  - SAME : means the data left uninterleaved.
  - REVERSE : the data are permuted reversely(last bit first, first bit last and so on).
  - RANDOM : the data are permuted randomly.
  - BLOCK : data are set in rows and are retrieved by columns.
  - CIRCLEDIST: distributes the data into a circle where the data compete to be as far as possible from each other.

2- Class Convert

This class is used to translate an integer number to its binary form expressed as an array of Boolean bits.

Public member variables

- "bool* array" : array is a pointer to the binary digits expressed in Boolean started from right as the first bit and ending on the left as the last bit.
- "int state" : state is the integer form of the data.

Constructors

- "Convert(int memorySize)" : constructs the object with the given array size, the array size determines the number of Boolean bits to be used.

Access functions

- "Convert& operator=(const Convert &copy)" : default assignment function.
- "int getSize() const" : returns the number of bits(i.e. array size).

Helper functions

- "bool* inttobool()" : returns array after calculating its translation from state.
- "bool* inttobool(int newState)" : assigns state and then returns its array bit translation.
- "int booltoint()" : returns state after calculating its translation from array.
- "int booltoint(bool *copy)" : assigns the values of copy to array and then returns its state translation.
- "void shiftRight()" : rotates the bits one step to the right and translates state.
- "void shiftLeft()" : rotates the bits one step to the left and translates state.

3- Class SqMatrix

A helper class that is used to build matrix related interleavers.

Public member variables

- "int *inter" : a pointer to the matrix of the interleaver.

Constructors

- "SqMatrix(int size)" : constructs a square matrix where row by column multiplication would result into the given size.
- "SqMatrix(int rows, int columns)" : constructs a square matrix with manual entry of rows and columns sizes.

Access Functions

- "int getRows()" : returns number of rows.
- "int getColumns()": returns number of columns.
- "int getSize()" : returns the given size.
- "void print()" : a diagnostic function used to print the matrix in the console terminal.

Helper functions

- "void setBlock()" : builds a row column interleaver matrix from the internal square matrix.

4- Class InterMat

Class InterMat is used to build the array of the interleaver, which is constructed by applying the desired interleaving technique option.

Constructors

- "InterMat(Interleaver l, int blockSize, int index=1)" : Constructs the array of interleaving matrix given the option of the desired interleaver and desired block size. The index argument is used by the interleavers for special distributions options.
- "InterMat(Interleaver l, int N, int p, bool similed)" : Constructs a simile interleaver concatenating the given interleaver type. Make sure that N(the interleaver size) is divisible by p(the delay unit of the encoder). The delay unit p is given by $p=2^{v}-1$, where v is the No. of delay elements of the encoder. The similed option is just to indicate the similed operation, recommended to always be true.

Access functions

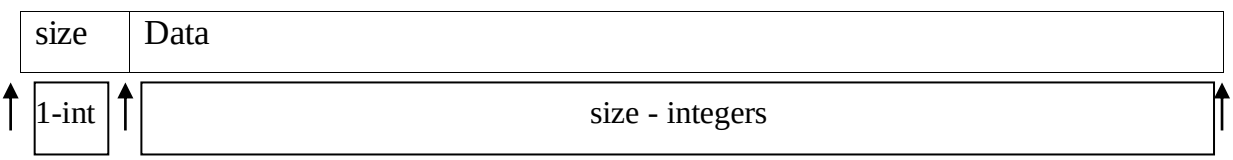
- "int* getMat() const" : returns the pointer to the matrix of interleaving.

Helper Functions

- "void isConsistent()" : a diagnostic function that returns true if the built interleaver having the distribution No.s distributed from 1 to matrix size.

File Handling

The form for handling the files is



- “bool saveMat(const char* fileName)” : saves the matrix to a file with the specified name and returns true on success.
- “bool loadMat(const char* filename)” :loads the matrix from a file with the name specified and returns true on success.

5- Class StateTran

Holds the transition table of states according to a given encoder. The encoder used is a 4-bit memory RSC with transfer function(by default) in octal (g1,g2)=(23,33) or in binary (g1,g2)=(10011,11011) this is the optimum encoder which minimizes the bit error probability by maximizing the effective free distance. It is possible as well to supply user defined encoder.

Constructors

- "StateTran(int s=0)" : constructs the default transition table of states and initializes the state to the given input or to the zero state by default.
- "StateTran(bool g1[], bool g2[])" : constructs a zero initialized transition table according to the given g1 and g2 arrays. Be careful to supply input arrays of size 5, also be aware that element g1[0] must always be true to complete the feedback eq.

Access functions

- "void reset(int newState=0)" : assigns the given state to the object or reset it to the zero state by default.
- "int getState()" : returns the current state of the object.

Helper functions

- "bool getParity(int input)" : returns the bit result of adding the forward transfer function on the given state.
- "int getNextState(int input)" : returns the next state according to the given input.
- "int getPreviousState(int input)" : returns the previous state if the given input is assumed.

- “double tailer(int state)” : returns a value to be used by a data stream to lead to the zero state if the given state is assumed.
- “int getMemories()” : returns the the number of active memory elements of the encoder.
- “bool isValid()” : returns the validity state of the encoder, an encoder is valid if all its states have previous states for inputs 0 and 1.
- “int filter(int state)” : omits the unused bits for 8 and 4 states cases according to the type of the encoder.

6- Class DataStr

Used to instantiate a stream of data of a specific size, the data are represented as an array of type double values. Associated with the stream are group of functions that are collectively used to manipulate the data with the desired tasks.

Constructors

- "DataStr(int s)" : Creates the stream with the specified size. The data within the stream are not initialized.
- "DataStr(const DataStr& dataStream)" : Default assignment constructor.
- "DataStr(DataStr* dataStream)" : Creates a data stream copying the contents of the given dataStream.

Access functions

- "DataStr& operator = (const DataStr& dataStream)" : Default assignment function.
- "void setBits()" : Assigns random bits values to the stream.
- "void setBits(double choice)" : Assigns the given double to all the bits.
- "void setBits(double* copy)" : Assigns the bits the contents of the given copy array.
- "double* getBits()" : returns a pointer to the bits.
- "double* getCBits() const" : a constant version of the above function.
- "int getSize() const" : Returns the size of the stream.

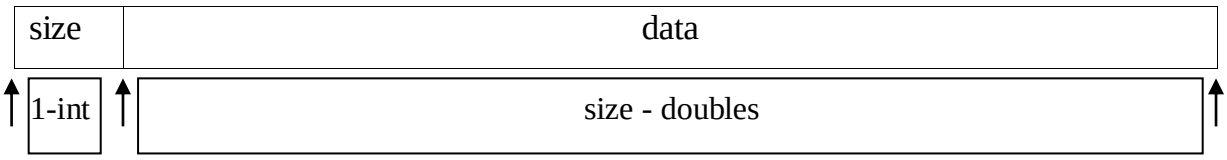
Helper functions

- "DataStr permute(const InterMat *mat, bool option)" : returns a permuted version of the given data stream according to the given interleaver matrix. option indicates interleave or deinterleave : true=interleave, false=deinterleave.

- "DataStr encode(StateTran *encoder)" : returns parity data stream from the given data according to the given encoder.
- "DataStr hard()" : returns hard limited version of the data. (+1.0,-1.0).
- "void tail(StateTran *encoder)" : tails the last bits of this data to lead to the zero state according to the given encoder in use and zero state initialization is assumed.
- "int compare(DataStr *original)" : compares the data with the given data stream and returns number of errors.
- "void halfPuncture(bool position)" : punctures the data for 1/2 rate simulation, this is used to omit the information received at the decoder as a result of puncturing that occurred at the encoder. The value of position determines the starting point of puncturing where: position=false-> start puncturing from first No., position=true-> start from second No.

File Handling

The form for handling the files is



- "bool saveData(const char* fileName)" : saves the data stream to a file with the name specified, and returns true on success.
- "bool loadData(const char* fileName)" : loads the data from a file with the name specified, and returns true on success.

7- Class BCJR

A class used to build a decoding component object. The decoder is built using the optimized BCJR algorithm. The ADD_CORRECTION global variable directly affects this component. If ADD_CORRECTION is false the component will use MAX_LOG_MAP algorithm, otherwise a correction term will be added to the data making the component act as LOG_MAP algorithm.

Constructors

- "BCJR(DataStr *message, DataStr *parity, double Lchannel, StateTran *encoder)" : Constructs and initializes a BCJR component with the given message, parity and encoder in use. The reliability of the channel value Lc given to the constructor is calculated from the equation : $Lc = 4SNR\_channel$ where SNR_channel is the signal-to-noise ratio of the channel.

Access functions

- "void setLeIn(DataStr *extrinsic)" : Sets the input extrinsic information to the decoder. Make sure to call this function before processing the decoder.
- "DataStr getLeOut()" : processes the overall data and calculates the output extrinsic information from the component and returns the result.
- "DataStr getFinal()" : adds(input message+input extrinsic+output extrinsic) as the final decoding result. Make sure to call both setLeIn and getLeOut before calling this function.

8- Class Channel

Channel class is an implementation of the GenData library classes that simulates a channel having a collection of data streams and features noise addition along with a decoding process function that implements turbo decoding algorithm of individual components.

Constructors

- "Channel(DataStr *msg, DataStr *par1, DataStr *par2, double SNR_channel, bool punctured=false)" : constructs three DataStr objects and initializes them using the given inputs. The given SNR_channel represents the SNR of the output of the encoder until it reaches the decoder side and the punctured bool is for describing the punctured status of the sent parities: false=not punctured, true = punctured.
- "Channel(DataStr *msg, double SNR_channel)" : constructs single DataStr object and initialize it using the given input. Using this constructor disables the coding specialized functions (i.e. decode).

Access functions

- "DataStr getDat(int choice)" : returns a data stream object that represents a copy of one of the three internal DataStr objects. The given choice is the DataStr object to be returned : 0=message,1=par1 and 2=par2. for the single input case, the single data is always returned.
- "bool isPunctured()": returns the state of puncturing.

Helper functions:

- "void applyNoise()" : applies the noise of the channel to the data.
- "DataStr decode(int iterations, InterMat *interleaver, StateTran *encoder)" : decodes the three DataStr objects with a turbo decoding algorithm according to the given interleaver and decoder in use. The iterations argument Specify the number of iterations that the decoder will iterate.

9- Class Use

Use class is an implementation of the GenData library, that is capable of simulating coded and uncoded experiments. The class supports saving the data by opening a file stream before processing. The file stream is closed by the destructor or manually by calling closeStream function.

Constructors

- "Use(int packetSize)": initializes the uncoded experiment. Processing function will operate on the packet data without coding considerations.
- "Use(int interleaverSize, bool puncturedState, Interleaver intType= RANDOM, bool SIMILED= false, int p=15, bool defaultEncoder= true, int iIterations=3)": initializes the coded experiment. Processing function will simulate a turbo coding/decoding system initialized to a random interleaver with the given size. Puncturing mechanism is supported by the specified puncturedState: false=no puncturing is assumed, true=puncuring is done on the data. intType argument specifies the interleaver to be used, RANDOM is assumed by default, the new similed interleaver is supported along with the ρ parameter, defaultEncoder indicates weather the encoder should be left for later initialization, true means that the encoder is initialized with default encoder structure, iIterations represents the number of iterations for the decoder.

Helper functions

- "double processAndSave(double SNR_dB, int noOfPackets)": process and returns the Pe value at the given SNR in dB for the given No. of packets.
- "bool initializeEncoder(bool g1[],bool g2[])": if the encoder is left for later initialization, this function is used to build an encoder scheme according to the given values, make sure that g1 and g2 are of size 5, also make sure that g1[0]=true.

File Handling

The form for handling the files is

Pe	SNR_dB	Pe	SNR_dB	Pe	SNR_dB	Pe	SNR_dB	Pe	SNR_dB

↑ double ↑ double

The size of the file is not saved, the implementing programs should either read all the file or use EOF capabilities as supported by their environments.

- "bool openStream(const char* filename)": creates a new file with the specified name. call this function before processing in order to save processed data. A warning will be printed if not called.
- "bool saveItem(const void* item, int itemSize)": allows to add extra data to the opened stream such as additional information about the experiment. item is the structure of the data, itemSize is the size of the data.
- "bool closeStream(const char* filename)": closes the file stream. This function is called by the destructor if the file is opened.