INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-40947-3
Multibit Decoding of Turbo Codes, by Pierre-Paul Sauvé

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science in the Department of Electrical and Computer Engineering, University of Toronto, October, 1998.

Abstract

Multibit turbo decoding is the name given to a modification of the turbo decoding algorithm (iterative bit-by-bit sum-product decoding), where the sum-product decoding is done on groups of $g$ bits instead of single bits. By decoding over groups, the decoder may use any dependence between bits of a group to aid in error correction. This thesis presents multibit decoding, and explores the issues affecting the performance of this decoding method when using binary antipodal modulation over an additive white Gaussian noise channel. The main results are that properly designed non-punctured multibit systems for $g = 2$ equal or better the performance conventional turbo codes, for both relatively short and long block lengths. The number of operations required by multibit decoding for performance equivalent to conventional decoding is lower, sometimes reaching $1/2$. Other advantages of multibit decoding include reduced memory requirements, and the possibility of doing more operations in parallel.
Acknowledgments

First, I would like to thank my thesis supervisor, Prof. Frank Kschischang. He has been a constant source of enthusiasm and enlightenment.

I would also like to thank my friends of the communication and electronics group for many interesting discussions that have helped me in my work.

Thank you, Bernadette, for keeping me sane and fed during the last few weeks; maraming salamat.

Finally, I gratefully acknowledge the financial support provided by the University of Toronto (U of T Open fellowship), ITRC, NSERC, and Research in Motion.
Contents

List of Tables ........................................................................................................ vi
List of Figures ......................................................................................................... xi

1 Introduction ........................................................................................................ 1
  1.1 Error Control Coding ................................................................................... 1
  1.2 Turbo Codes ................................................................................................. 3
  1.3 Multibit decoding ......................................................................................... 6
  1.4 Thesis Organization ..................................................................................... 8

2 Background .......................................................................................................... 9
  2.1 Factor Graphs ............................................................................................... 9
    2.1.1 Basic Concepts ..................................................................................... 9
    2.1.2 Representing Codes With Indicator Functions ..................................... 10
    2.1.3 Factor Graphs of Probability Density Functions ................................. 12
    2.1.4 The Sum-Product Algorithm ............................................................... 14
    2.1.5 Sum-Product Algorithm on graphs with cycles ............................... 16
    2.1.6 Application To Coding: the MAP Algorithm .................................... 17
    2.1.7 Example: Forward Backward Algorithm on a Convolutional Code .. 17
  2.2 Compound Codes And Iterative Decoding .................................................... 18
  2.3 Turbo Codes ............................................................................................... 20

3 Multibit Turbo Decoding .................................................................................... 23
  3.1 General Concept ......................................................................................... 23
  3.2 Applying Multibit to Turbo Decoding ......................................................... 24
3.3 Decoding Algorithm .................................. 26
3.4 Decoding Complexity .................................. 26
3.5 Memory Requirements ................................. 28
3.6 Choosing Grouped Interleavers ....................... 29
   3.6.1 RSC Code response to weight two inputs ....... 29
   3.6.2 Avoiding low-weight code words in turbo-codes . . 30
   3.6.3 Simple “Weight-Two” criterion .................. 31
   3.6.4 Modifying Interleaver Selection Criteria for use with Group Interleavers 32
   3.6.5 The Spread Rule ................................ 33
3.7 Puncturing Methods .................................. 34
3.8 Sub-Block Operations ................................ 35

4 Experimental Results ................................ 38
   4.1 Simulation Software ................................ 38
   4.2 System Model ...................................... 39
   4.3 Experiments ....................................... 41
      4.3.1 Impact of grouped interleaver on code performance . 41
      4.3.2 Effect of grouping on decoding performance .... 41
      4.3.3 Real Systems with Random Interleavers .......... 42
   4.4 Puncturing Methods ................................ 44
   4.5 Interleaver Optimization ............................ 45
   4.6 Multibit vs. Conventional for short blocks ......... 50
   4.7 Performance and Complexity ........................ 51
   4.8 Lessons Learned ................................... 55

5 Conclusions .............................................. 56
   5.1 Summary ........................................... 56
   5.2 Contributions ...................................... 56
   5.3 Suggestions for Further Work ....................... 57
List of Tables

3.1 Memory requirements for multibit turbo decoding as a function of the number of information bits $K$, the grouping factor $g$, and $|S|$, the number of states. No likelihoods ratios are used in the representation. ........................................... 28

3.2 Minimum interleaver size to find a spread interleaver for various combinations of $g$, $\tau$ and $\sigma$ ................................................................. 35

4.1 Complexity increase factor for different values of $g$, assuming equal cost for addition and multiplication. The values are normalized with respect to $\kappa$ for $g = 1$. ................................................................. 51
List of Figures

1.1 Graph showing the Bit Error Rate vs. Signal to Noise Ratio (dB) performance from [5]. The information block length was $K=65535$, $R=0.5$ and RSC codes were $21/37$. The number of decoding iterations performed shown on the curves. ........................................ 3

1.2 Encoding of a turbo-code ........................................ 114 4

1.3 $21/37$ RSC code ........................................ 114 5

1.4 BER vs. SNR performance of a conventional turbo code after 12 iterations (dotted) and a quaternary multibit code after 8 iterations (solid). Components were $23/35$ RSC codes, rate is $1/3$ and information block length is $K=16384$ bits. ........................................ 114 7

2.1 Example of a simple factor graph ........................................ 114 10

2.2 Factor graph of a truncated convolutional code, showing the use of auxiliary variables (double circles). ........................................ 114 12

2.3 Factor graph of a simple block code, including the variables and local functions for a memoryless channel ........................................ 114 13

2.4 Various levels of simplification of a coding system shown as factor graph:
   a) a general coding system
   b) simplifications after $y$ has been observed
   c) simplification for a memoryless channel ........................................ 114 14

2.5 Factor graph fragment showing the messages on one edge, as updated by the sum-product algorithm ........................................ 114 15

2.6 Factor graph representing a hidden Markov model, or terminated convolutional code: the variable nodes with the double circles are unobserved state variables ........................................ 114 17
2.7 Fragment of a the factor graph of one component of a turbo code, including observations from the channel. This fragment represents one state transition of the component and its associated messages.

2.8 Example of iterative schedule for a two component compound code: one iteration is shown.

2.9 Factor graph of a rate 1/3 turbo code.

2.10 Schematic showing the steep descent (A) and “error floor” (B) regions of a typical turbo code’s BER vs. SNR curve.

3.1 Schematic representing multibit decoding for different group sizes $g$, from left to right, $g = 1, 2, 4$. The number of dots on the variable nodes indicate the number of bits treated as single variables.

3.2 Schematic diagram of two interleavers: at left, a conventional turbo code interleaver and at right, an interleaver suitable for $g = 2$ decoding.

3.3 Factor graphs for conventional turbo code (left) and the equivalent $g = 2$ realization. Thicker edges show where messages have higher dimensionality.

3.4 Trellis diagram showing the allowed state transition of a 5/7 code for two local function nodes of the conventional (left) realization and for one transition of the $g = 2$ multibit realization (right).

3.5 Fragment of a convolutional code’s factor graph corresponding to one state transition and the associated messages.

3.6 21/37 RSC code.

3.7 Block diagram of taboos for increasing minimum distance and reducing multiplicity of second minimum distance in an interleaver.

3.8 Block diagram showing derivation of taboos for “weight-2” criterion in an $g = 2$ blocked interleaver for $\tau = 5$.

3.9 Block diagram of taboos for spread interleaver.

3.10 Factor graph of the input symbols and one component code from a turbo code, with added sub-block substitutions.
4.1 Effect of blocked interleaver on conventional turbo code for blocks of \( b = 1 \) (dotted), \( b = 2 \) (solid) and \( b = 4 \) (dashed) bits. Code is 23/35, \( K = 1024 \), 8 iterations.

4.2 Identical \( K = 1024 \) bit 23/35 turbo codes. Decoded as \( g = 1 \) with interleaver in blocks of \( b = 2 \) (dotted), and \( g = 2 \) (solid) at 8 iterations.

4.3 Identical \( K = 1024 \) bit 23/35 turbo codes. Decoded as \( g = 1 \) with interleaver in blocks of \( b = 4 \) (dotted), and \( g = 4 \) (solid) at 4,8 iterations.

4.4 23/35 turbo code, \( K = 1024 \) information bits, randomly chosen interleavers, \( g = 1 \) (dotted), \( g = 2 \) (solid) and \( g = 4 \) (dashed) at 8 iterations.

4.5 23/35 turbo code, \( K = 16384 \) information bits, randomly chosen interleavers showing iterations 8, 12 and 16 for \( g = 1 \) (dotted) and \( g = 2 \) (solid).

4.6 Comparison of 23/35 codes rate 1/2 punctured turbo codes with information block length of 256 bits: \( g = 1 \) (dotted), \( g = 2 \) sub-block puncturing (solid), and \( g = 2 \) block puncturing (dashed).

4.7 Comparison of various attempts to improve interleavers for \( g = 1 \), \( K = 256 \), decoded with 2 and 8 iterations: \( a) \) is random choice, \( b) \) is simple optimization avoiding \( d = \tau \) only, and \( c) \) is a spread interleaver with \( \sigma = 10 \).

4.8 Comparison of various attempts to improve interleavers for \( g = 2 \), \( K = 256 \) bits, 8 iterations: \( a) \) avoids \( d = \tau \), \( b) \) avoids \( d = \tau, 2\tau \), \( c) \) is a random interleaver for comparison, \( d) \) has \( \sigma = 7 \) and \( e) \) has \( \sigma = 4 \) while avoiding \( d = \tau \).

4.9 Comparison of various attempts to improve interleavers for \( g = 4 \), \( K = 256 \) bits, 8 iterations: \( a) \) is random interleaver for comparison, \( b) \) avoids \( d = \tau, 2\tau \), \( c) \) has \( \approx \sigma = 4 \) while avoiding \( d = \tau, 2\tau \) and \( d) \) has \( \sigma = 5 \).

4.10 Comparison of two codes having \( K = 256 \) information bits for iterations 2,4 and 8. Both use spread interleavers. The dotted line is a \( g = 1 \) code with \( \sigma = 10 \) and the solid line is for a \( g = 2 \) code with \( \sigma = 7 \).

4.11 Relative Complexity vs. SNR required to achieve a BER of \( 10^{-3} \) for 23/35 codes, \( K = 16384 \) bits, \( g = 1, 2 \).

4.12 Relative Complexity vs. SNR required to achieve a BER of \( 10^{-4} \) for 23/35 codes, \( K = 16384 \) bits, \( g = 1, 2 \).
4.13 Relative Complexity vs. SNR required to achieve a BER of $10^{-5}$ for 23/35 codes, $K=16384$ bits, $g = 1, 2$. ................................................................. 53

4.14 Relative Complexity vs. SNR required to achieve a BER of $10^{-3}$ for 23/35 codes, $K=256$ bits, $g = 1, 2, 4$. ................................................................. 53

4.15 Relative Complexity vs. SNR required to achieve a BER of $10^{-4}$ for 23/35 codes, $K=256$ bits, $g = 1, 2, 4$. ................................................................. 54

4.16 Relative Complexity vs. SNR required to achieve a BER of $10^{-5}$ for 23/35 codes, $K=256$ bits, $g = 1, 2, 4$. ................................................................. 54

A.1 Booch diagram of the single-node construction generic factor graph classes. 60

A.2 Booch diagram of the vector related generic factor graph classes. .......................... 60

A.3 Booch diagram of the high-level coding classes implementing a turbo code. .......................... 61

C.1 23/35, $K=16384$, $g=1$ ................................................................. 69
C.2 23/35, $K=16384$, $g=2$ ................................................................. 69
C.3 23/35, $K=16386$, $g=3$ ................................................................. 69
C.4 23/35, $K=16384$, $g=4$ ................................................................. 69
C.5 23/35, $K=16386$, $g=3$, random sub-block interleaving ........................................... 70
C.6 23/35, $K=16386$, $g=3$, random sub-block lin. substitutions ........................................... 70
C.7 23/35, $K=4096$, $g=1$ ................................................................. 71
C.8 23/35, $K=4096$, $g=1$ ................................................................. 71
C.9 23/35, $K=4096$, $g=2$ ................................................................. 71
C.10 23/35, $K=1024$, $g=1$ ................................................................. 72
C.11 23/35, $K=1024$, $g=1$, inter $\sigma = 4$, avoids $d = \tau, 2\tau$ ........................................... 72
C.12 23/35, $K=1024$, $g=1$, inter $\sigma = 10$ ........................................... 72
C.13 23/35, $K=1024$, $g=2$ ................................................................. 72
C.14 23/35, $K=1024$, $g=1$, inter. blocked by 2 ........................................... 73
C.15 23/35, $K=1024$, $g=1$, inter. blocked by 2 ........................................... 73
C.16 23/35, $K=1024$, $g=1$, inter. blocked by 4 ........................................... 73
C.17 23/35, $K=256$, $g=1$ ................................................................. 74
C.18 23/35, $K=256$, $g=1$, inter. avoids $d = \tau, 2\tau$ ........................................... 74
Chapter 1

Introduction

1.1 Error Control Coding

Error control coding aims to achieve reliable communication over noisy channels. To achieve reliable communication, messages must be mapped to codewords that are distinguishable at the channel output. The set of codewords is called a code. Fundamentally, code design involves finding clever ways of mapping information to codewords at the transmitter and mapping the noise-corrupted received words back to information at the receiver.

Coding adds redundancy to the information that is to be transmitted. The amount of redundancy is usually measured by the rate $R$ of a code. The rate is a an important design parameter, and it is the ratio of information symbols to the total number of symbols transmitted, including redundancy:

$$R = \frac{\text{number of information symbols}}{\text{total number of symbols transmitted}}.$$

The best code designs, given the type and severity of noise expected, minimize the information symbol error rates, the computational complexity of encoding and decoding, the latency (transmission and decoding delay), while maximizing the rate.

At first, people thought that to increase the reliability of a communication system, the rate would have to decrease and that error free communications in noise implied a rate of zero. Then in 1948, Claude Shannon [22] showed that codes can be constructed for any given channel which allow the communications through it to be as reliable as one wishes, with one
condition: the rate of the code cannot be higher than the channel’s capacity $C$. The capacity of a channel is defined as the maximum rate at which reliable communication is possible.

For example, an additive white Gaussian noise (AWGN) channel where the average transmitting energy is constrained to be $E_b$ per bit has a capacity $C$ given by:

$$C = \frac{1}{2} \log_2(1 + \frac{2E_b R}{N_o}) \text{ bits per transmission}, \quad (1.1)$$

where $N_o/2$ is the two-sided noise power spectral density. If one wants to communicate without error over an AWGN channel using a rate $R$, a simple manipulation of (1.1) allows us to see that the minimum signal to noise ratio $E_b/N_o$ required is:

$$\frac{E_b}{N_o} \geq \frac{2^{2R} - 1}{2R}, \quad (1.2)$$

which is often referred to as the Shannon Limit.

Although Shannon showed the existence of codes of non-zero rate to communicate reliably in noise, his arguments were non-constructive: the proof involves the creation of arbitrarily long, randomly chosen codes which are impossible to store or decode in practice. To give an example of the type of codes used in Shannon’s argument, let us say that we are planning to transmit with a code of rate 1/2. We will use a block code, meaning that the information to be sent is broken up into vectors of $K$ bits and each of the possible $2^K$ input vectors is mapped to a vector of $N$ bits. Shannon’s arguments involve a random choice of mappings (the $N$ vectors are chosen randomly) and very long block lengths to achieve very reliable communication. Let us suppose that we choose $K = 1000$, corresponding to a block length of $N = 2000$ is sufficient. At this smallish block length (and even at much smaller ones) the storage requirements at the transmitter are impossibly large: to store the table of mappings from $K$-vectors to $N$-vectors, we need $2^K N$ bits, which is $2000 \times 2^{1000}$ bits. It is not hard to imagine the difficulties and the time involved in decoding this kind of random code! Clearly, some more practical codes must be used.

Many practical means of coding have now been invented, which, as a rule, involve using some structure in the choice of mappings between information and code words. The gap between the performance of practical codes and the theoretical limit has been shrinking, due to developments in coding too numerous to list. However, no development has come closer to closing the gap to capacity than turbo-codes.
The information block length was $K=65535$, $R=0.5$ and RSC codes were 21/37. The number of decoding iterations performed shown on the curves.

1.2 Turbo Codes

Turbo codes, introduced in 1993 by Berrou, Glavieux and Thitimajshima [5] allow communications over an AWGN channel unprecedentedly close to the capacity of the channel, with a low error rate and manageable complexity. In the words of MacKay, Rodemich and Cheng [18]:

"Turbo codes [...] are the most exciting and potentially important development in coding theory in many years. [They] represent a genuine, and perhaps historic breakthrough."

Figure 1.1 shows the result of a rate 1/2 turbo code simulation from [5]. Substituting $R = 1/2$ in (1.2), we find that error free communications can be obtained down to an SNR of 0 dB. Taking a bit error rate (BER) of $10^{-5}$ as close to error free, Berrou's result is only 0.7 dB away from the Shannon limit!
To form a turbo code, one takes the stream of information bits and makes two identical copies of it. The original stream is transmitted as is. The first copy is fed through a recursive systematic convolutional (RSC) encoder [5] before being transmitted, and the last is scrambled deterministically (interleaved) before being fed through a RSC encoder identical to that of the previous stream and finally being transmitted also. Note that only the parity stream of each RSC code is transmitted, making each of these encoders effectively rate-1 scramblers. Figure 1.2 shows a block diagram of a turbo encoder.

An example of the binary RSC codes alluded to in the previous paragraph is shown in schematic form in Fig. 1.3. These codes have an infinite impulse response because of their feed-back component. The response of these “filters” to different discrete time binary input sequences is completely characterized by the choice of taps corresponding to different delay terms in the forward and backward sums. These sums correspond to polynomials in $D$, the delay operator, and the rational polynomial formed by the feed-forward over the feed-back polynomials constitutes the RSC code’s impulse response in terms of $D$. For the remainder of this document, feed-forward (numerator) and feed-back (denominator) generator polynomials are denoted in octal notation, and their memory order with $\nu$. For example, a 23/35 code has $\nu = 4$ (and is therefore a 16-state encoder) for which the feed-forward and feed-back polynomials (in indeterminate $D$) are $10011 \equiv 1 + D^3 + D^4$ and $11101 \equiv 1 + D + D^2 + D^4$.

The rate of a turbo code is naturally 1/3 because of the three streams transmitted at the same rate as the information stream, but it may be increased by puncturing, or omitting to
transmit some of the bits in the parity streams. For example, by omitting the transmission of every second parity bit at the output of each RSC code, we obtain a rate 1/2 turbo code.

Although the RSC components of the turbo code allow data to be encoded continuously, the code is often broken into blocks. The breaks are used to delimit the data that is decoded at one time.\footnote{Blocks were used in Berrou's paper, but this is not strictly necessary.} Henceforth, information block sizes will be denoted by $K$, and the corresponding number of transmitted symbols by $N$, both expressed in the equivalent number of bits.

Turbo decoding is “soft” and iterative. Soft decoding means that within the receiver, information about bits includes a value and a reliability estimate. The decoder consists of two sub-decoders, one for each of the RSC codes. Each of the sub-decoders take turns decoding the data, but they use each other’s estimates of the reliability of each bit. An iteration is when each of the sub-decoders has had its turn decoding, and a half-iteration implies that just one has decoded. Hard decisions are taken only at the very end of the decoding process, which is usually after several iterations.

This decoding scheme is sub-optimal, i.e., it does not minimize the probability of bit error or the probability of block error, but it has a complexity that is much lower than decoding the combined code (the code that is not separated into two sub-codes) and performs extremely well \cite{5}.

Turbo codes are not the answer to all coding needs; one of their limitations is that they require relatively long block lengths to achieve good performance. Needless to say, much effort has been put into understanding and improving turbo codes. Many papers have

\begin{figure}
\centering
\begin{tikzpicture}
\node (input) at (0,0) [draw,shape=circle] {1};
\node (first) at (1,-1) [draw,shape=circle] {2};
\node (second) at (2,-2) [draw,shape=circle] {3};
\node (third) at (3,-3) [draw,shape=circle] {4};
\node (output) at (4,-4) [draw,shape=circle] {5};
\draw (input) -- (first);
\draw (first) -- (second);
\draw (second) -- (third);
\draw (third) -- (output);
\end{tikzpicture}
\caption{21/37 RSC code}
\end{figure}
presented some solid design rules (see for example [20], [3]), and some propose modifications to the original decoding algorithm (see [14] for a survey).

Although developments such as serial concatenated turbo codes improve on the high-SNR performance of turbo codes [4], developments in the capacity challenging low-SNR region of the performance curve are scarce.

1.3 Multibit decoding

This thesis presents a modification to the original turbo decoding algorithm which gives an improvement in performance large enough to allow the number of decoding iterations to be reduced. Memory requirements are lower, and more operations may be made in parallel.

The modified decoding algorithm, which will be called “multibit turbo decoding” for the rest of this document, can be thought of as turbo-codes over symbols in $GF(2^q)$. All the soft information involved in the decoding concerns “joint reliabilities” of groups of bits. Since the encoding process introduces correlations between adjacent bits and this information cannot be used by binary decoding, the multibit decoding can improve performance.

Fig. 1.4 gives a preview of some simulation results presented in the next chapters: the multibit decoder did about 66% of the computations and used about 60% of the memory requirements of the conventional code, for a very similar BER vs. SNR performance.

The work on multibit turbo decoding for this thesis was motivated by several interesting papers. First is the work of Davey and MacKay [8], who show that the performance of low-density parity-check (LDPC) codes can be improved remarkably by using symbols in $GF(2^q), q > 1$, over the performance of the binary LDPC codes. LDPC codes were first described by Gallager [12], and they are closely related to turbo codes because, like them, they use a soft-valued iterative decoding algorithm. A recent paper which further motivates the exploration of multibit decoding is Hoeher’s [14]. He suggests (without showing any results) that a turbo code on symbols in $GF(2^q)$ might outperform classical turbo codes in low SNR. Next, Chang [7] published an algorithm and some results on decoding turbo codes in $GF(4)$ on very short blocks. He discusses the possible advantages of the algorithm because of increased parallelism. Further, Bingeman and Khandani [6] presented a paper on
Figure 1.4: BER vs. SNR performance of a conventional turbo code after 12 iterations (dotted) and a quaternary multibit code after 8 iterations (solid). Components were 23/35 RSC codes. Rate is 1/3 and information block length is $K=16384$ bits.
combined turbo codes and modulation. This paper describes turbo encoding and decoding with symbols in $GF(2^q)$, with various memoryless modulation schemes taking $n$-bit symbols. Finally, the author and Kschischang [21] presented a paper which became the basis of the material of this thesis. The focus was on the merits of multibit turbo decoding, using simple antipodal signaling.

1.4 Thesis Organization

The aims of this thesis are two: first, to describe multibit decoding in a simple, accessible manner and second, to make improvements that allow multibit decoding to be an attractive alternative to conventional turbo decoding, in low and moderately high SNR.

The document is organized for sequential reading. To properly prepare the reader, the next chapter introduces factor graphs, compound codes, and turbo codes. Multibit decoding and multibit code design are presented in Chapter 3. Chapter 4 describes the multibit code simulation experiments and their results. Finally, Chapter 5 concludes and provides some suggestions for further study. A description of the simulation software developed for and used in this thesis is included in Appendix A; Appendix B discusses the tolerance of the BER estimations made in simulations and Appendix C is a record of most of the recent simulation results.
Chapter 2

Background

The information in this section provides the foundation that will allow us to describe multibit turbo decoding. This information will include a section on decoding on factor graphs, a section on compound codes, and a section on turbo codes.

2.1 Decoding On Graphical Models: Factor Graphs

This section will present factor graphs, a useful representational tool. Factor graphs allow simple, elegant descriptions of codes’ structures and decoding algorithms, which makes them well suited to describing multibit decoding.

2.1.1 Basic Concepts

A factor graph is a bipartite graph representing how a “global” function of many arguments can be factored into a product of many local functions[15]. The two kinds of vertices are variables, which are represented as white circles, and local functions, represented as black squares. An edge between a variable and a local function implies that the variable is an argument of the function sharing the same edge.

For example, the global function $g$ with the following factorization

$$g(x_1, x_2, x_3, x_4, x_5, x_6, x_7) = A(x_1, x_2, x_7) \cdot B(x_3, x_4, x_7) \cdot C(x_5, x_6, x_7),$$

can be represented by the factor graph in Fig. 2.1.
Two kinds of functions are most interesting when using factor graphs in coding and decoding: indicator functions, and probability density functions (pdfs).

We will now discuss each of these in turn.

### 2.1.2 Representing Codes With Indicator Functions

An *indicator function* indicates whether a particular object is part of a set. It takes a value of one if the object is a member of the set, and a value of zero otherwise. Before going further, let us define a useful notation called “Iverson's convention” [13, 15]. If $P$ is a Boolean proposition, then $[P]$ denotes a binary function indicating the truth of $P$:

$$[P] = \begin{cases} 
1 & \text{if } P \text{ true}, \\
0 & \text{otherwise}.
\end{cases}$$

Using this new notation, an indicator function for the set $C$ would be

$$I_C(x) = [x \in C].$$

A code is a set of codewords, which means that there is an indicator function for any code. We can draw a factor graph of the code's indicator function.

For example, let us take the linear block code defined by the parity check matrix

$$H = \begin{bmatrix}
1 & 1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1
\end{bmatrix}. \quad (2.1)$$

---

1It is understood that the term pdf will also be used to stand for its discrete counterpart, the probability mass function (pmf).
The set $C$ of its codewords must satisfy

$$xH^T = 0 \text{ in } GF(2).$$

This is a global function which factors into three local functions:

$$g(x_1, \ldots, x_7) = [(x_1, \ldots, x_7) \in C] = [x_1 \oplus x_2 \oplus x_7 = 0] \cdot [x_3 \oplus x_4 \oplus x_7 = 0] \cdot [x_5 \oplus a_6 \oplus x_7 = 0].$$

The factor graph in Fig. 2.1 is a factor graph representing $g$, the indicator function of the block code in 2.1. The factor graph makes the local constraints very clear. The factor graph makes the local constraints very clear, as its indicator function factors into the three local functions corresponding to the parity equations generated by the three rows of the parity check matrix.

Let us look at another example, the factor graph representation of a truncated convolutional code, via its indicator function. Convolutional encoders are discrete linear systems, and one may visualize the encoding process as the convolution of the binary input data by the impulse response of the encoder. By truncated, we mean that the input is finite, and that the output will considered only for the time instants where there was an input. Because this truncated convolutional code is linear and of finite length, it may be represented in the same way as the block code, with a parity check matrix. The factorization dictated by the parity check matrix contains one function for each variable and is very complicated, especially if the impulse response of the encoder is long. If the parity check matrix for a rate-1 code formed by the parity stream of the $21/37$ RSC code shown in Fig. 1.3 on p. 5, truncated after an input of seven bits is:

$$H = \begin{bmatrix}
1 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix},$$

\begin{equation}
(2.2)
\end{equation}
Figure 2.2: Factor graph of a truncated convolutional code, showing the use of auxiliary variables (double circles).

which will correspond to a factor graph with 17 edges and seven different local functions. There is a more elegant way of representing this code, and that is by using the fact that given the state of the encoder (it can be seen as a finite state machine), the next output is only a function of the state and the next input. We can draw a much more uniform factor graph (all local functions are identical) by including these new “state” variables. Variables that are not observed and are included to simplify the factorization of the global function are called auxiliary variables; in factor graphs, they are often drawn as double circles, as we will do. Figure 2.2 shows a factor graph representing the factorization of the indicator function for the convolutional code using auxiliary state variables.

2.1.3 Factor Graphs of Probability Density Functions

Because noisy channels are described stochastically, the representation of a complete error control coding system requires probability distribution (or mass) functions. Fortunately, these functional descriptions of channels may co-exist with our code descriptions in a factor graph representation.

The transmission of codewords \((x_1, x_2, \ldots, x_n)\) through a memoryless channel to form a received word \((y_1, y_2, \ldots, y_n)\) has the likelihood function

\[
P(y_1, y_2, \ldots, y_n|x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} f(y_i|x_i),
\]

which factors in this way because the channel is memoryless, making individual symbol transmissions independent. If we assume equiprobable codewords, we can express the joint pdf as:

\[
P(x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n) = P(x_1, \ldots, x_n) \prod_{i=1}^{n} f(y_i|x_i).
\]
Figure 2.3: Factor graph of a simple block code, including the variables and local functions for a memoryless channel.

To see how the encoder may be included in this model, it suffices to notice that indicator functions are pmfs scaled so that each non-zero entry is equal to one. Augmenting our block code example with a memoryless channel results in the factor graph shown in Fig. 2.3.

A factor graph of a code and channel can be simplified somewhat (when decoding) when the channel outputs have been observed. The channel outputs $y_i$ are no longer variable arguments to the channel functions, and may be removed from the graph. The channel functions can now be regarded as pmfs, each corresponding to $P(y_i|x_i)$ evaluated at $y_i = a_i$. We will refer to functions of a single variable as constant functions.

One can assume without changing the global function, that every variable is connected to an added constant function with a value of one over all values of its argument. In terms of probabilities, this can correspond to a uniform a priori probability distribution for every variable. If the a priori probability is observed to be different (as in the case of observing the channel output), the function should be changed accordingly. The factor graphs can be further simplified if there is an understanding about which variables are observed: the constant functions corresponding to the a priori probability distribution of a variable are simply left out. This convention reduces visual clutter in the more complicated factor graphs and therefore, will often be used in the following sections.

Fig. 2.4a shows a factor graph for a general communication system. The information
Figure 2.4: Various levels of simplification of a coding system shown as factor graph: a) a general coding system b) simplifications after \( y \) has been observed c) simplification for a memoryless channel

vector \( u \) is encoded into codeword vector \( x \) before being transmitted on a noisy channel giving the received word vector \( y \). This graph can be simplified somewhat (this is necessary when the channel is not discrete) after \( y \) is observed (Fig. 2.4b), and even further if the channel is memoryless to the graph in Fig. 2.4c.

### 2.1.4 The Sum-Product Algorithm

The sum-product algorithm is a general algorithm that allows the efficient computation of the marginal functions\(^2\)

\[
G_i(x_i) \equiv \sum_{x \setminus \{x_i\}} g(x_1, \ldots, x_n)
\]

for every variable \( x_i \) which is an argument of a global function \( g \). The marginals are of particular interest in decoding when the global function is the joint pdf of channel and code; in that case, the marginals are the basic computations required to do maximum a posteriori (MAP) decoding\(^3\), which forms the basis of turbo decoding, and as we will see, multibit turbo decoding.

Apart from possible visual insights gained through the graphical representation it allows, the factor graph approach's main advantage is that its structure outlines the information paths required to apply the sum-product algorithm. To understand the sum-product algorithm, it is useful to visualize the factor graph in the following way: the nodes are inde-\(^2\)When the graph is cycle free.

\(^3\)MAP decoding is also known as the BCJR [1] algorithm.
Figure 2.5: Factor graph fragment showing the messages on one edge, as updated by the sum-product algorithm.

Independent computers, and the vertices connecting the nodes are communication pathways on which nodes may send messages to each other. The messages are functions, where all arguments to the global function not pertaining to the part of the graph the message is heading to are marginalized out. Only the information needed is transmitted on the edges, and that is why this is an efficient algorithm. The messages are computed using the sum-product update rule [15]:

The message sent from a node \( v \) on an edge \( e \) is the product of the local function at \( v \) (or the unit function if \( v \) is a variable node) with all messages received at \( v \) on edges other than \( e \), marginalized for the variable associated with \( e \).

The two messages passing on a single edge are shown in Fig. 2.5 where \( A, B, C \) are the local functions that have \( x \) as an argument. Similarly, the nodes \( x, y, z \) are the arguments of the function \( A \). In terms of these variables, the messages from variable \( x \) to local function \( A \) is:

\[
\mu_{z \to A}(x) = \mu_{B \to x}(x) \cdot \mu_{C \to x}(x). \tag{2.3}
\]

Similarly, the messages from local function \( A \) to variable \( x \) is:

\[
\mu_{A \to x}(x) = \sum_{y, z} A(x, y, z) \cdot \mu_{y \to A}(y) \cdot \mu_{z \to A}(z). \tag{2.4}
\]

Note that messages on an edge connected to variable \( x \) will always be a function of the alphabet of \( x \).
A schedule is used to orchestrate the message passing necessary for the distributed processing taking place during the application of the sum-product algorithm. Many message passing schedules are possible, but for the purpose of this document, we will limit ourselves to generalized forward backward (GFB) schedules or the “all but one” rule for graphs that are trees. According to this rule, a node may only send a message on an edge if it has received a message on all other edges. Initially, the only nodes that may send messages are the leaf nodes of the tree, those with only one edge. If the rule is followed, two messages will have been sent on each edge, one in each direction. At this point, no more messages need to be sent.

One very important property of the sum-product algorithm when applied to factor graphs representing discrete random variables (on a tree) is that the messages are proportional to conditional probability mass functions (pmfs). More specifically, for a GFB schedule, a message coming into a variable node is proportional to the pmf of this variable, conditioned on all observed variables that can be reached via edge on which the message is coming. The marginalization operation, if taken to be addition on the reals, is the same the marginalizing operation in probability, up to a scaling factor.

When all messages have been sent, the termination stage of the sum-product algorithm has been reached and the marginalizations of the global function with respect to each of the variables may be calculated. The marginal function for a variable may be calculated simply by taking the product of the two messages on one of its edges. For example,

\[ G_x(x) = \mu_{A\rightarrow x}(x) \cdot \mu_{x\rightarrow A}(x). \]

2.1.5 Sum-Product Algorithm on graphs with cycles

The sum-product algorithm is guaranteed to give exact marginalizations only when the graph is a tree. When the graph contains cycles, the schedule outlined above does not terminate, and the marginal functions do not necessarily converge to the exact value. Many practical decoding schemes such as those used for turbo codes and LDPC codes are equivalent to the application the sum-product on graphs with cycle, and have shown that although convergence
2.1.6 Application To Coding: the MAP Algorithm

For the particular case where the minimum symbol error rate is desired, the appropriate decoding criterion is the maximum a posteriori (MAP) rule,

\[ \hat{u}_i = \max \text{arg} P(u_i|y), \]  

(2.6)

where

\[ P(u_i|y) = \sum_{\text{all vars} \setminus \{u_i\}} P(u|y), \]  

(2.7)

which can be evaluated by applying the sum-product algorithm on the system's factor graph.

2.1.7 Example: Forward Backward Algorithm on a Convolutional Code

A factor graph of a "terminated" convolutional code is shown in Fig. 2.6. By terminated, we mean that the output of the code has been segmented into blocks, and that the state machine always ends in a known state. The function nodes representing the channel information have been omitted. Note that the auxiliary state variables have been marked by double circles to distinguish them from the others.

The function nodes here are indicator function of the allowable state transition quadruples \((u_i, s_i, s_{i+1}, x_i)\), which are from the definition of the state machine.

The decoding algorithm for MAP decoding of a convolutional code is often referred to as the BCJR algorithm after the authors of the paper first describing its use for coding [1].
Figure 2.7: Fragment of the factor graph of one component of a turbo code, including observations from the channel. This fragment represents one state transition of the component and its associated messages.

From our point of view, one may decode this code by applying the sum-product algorithm and applying the rule in (2.6) on the information variables' \((u_i)\) marginals.

The message passing schedule is as follows (refer to Fig. 2.7):

1. reception: all leaf function nodes send their message \(\gamma_i(x_i)\) (channel info) and the \(x_i\) send messages toward the state transition local functions

2. forward pass: starting from the “earliest” \(s_0\) state, messages \(\alpha_i(s_i)\) are passed forward from state variable to state variable

3. backward pass: starting from the “most recent” \(s_n\) state, messages \(\beta_i(s_i)\) are passed backward from state to state.

4. upward pass: messages \(\delta_i(u_i)\) are passed from each of the local function nodes to the \(u_i\) variables.

The desired \(P(u_i|y)\) is simply the message incoming to the \(u_i\) multiplied by \(P(u_i)\) (which is not shown in Fig. 2.6, but is an implied uniform pdf and has no effect).

### 2.2 Compound Codes And Iterative Decoding

A compound code is a code that can be described by a set of constituent codes, each of which is tractably decodable on its own [16].
The challenge in error control coding is to make a good\textsuperscript{4} code that can be decoded with reasonable complexity. We know that random codes with a very long block length would be “good”, but the computing power required to decode them increases exponentially with block length. Practical codes have some structure that simplifies their decoding. Compound codes achieve this goal by creating a complex code from a combination of two or more smaller codes, which may be decoded “pseudo-independently” with great savings in complexity over the decoding of the code as one corresponding to the “product of the realizations.”

Some examples of common compound codes are Gallager’s low-density parity-check (LDPC) codes [12], Forney’s concatenated codes [11], and the “parallel concatenation” used in turbo codes [5].

In terms of factor graphs, compound codes are special because they are usually not factored as trees. The component codes’ graphs are trees, but the combination of a number of these components which share the same information variables invariably creates cycles in the graph. This is not to say that there exists no cycle free factor graph representation for these codes, but only that we are willing to settle for the inexact iterative decoding because of its smaller decoding complexity.

The marginals calculated by the sum-product algorithm applied to a graph with loops are not guaranteed to converge to their exact values. In practice though, high performance compound codes such as turbo and LDPC codes using instances of the sum-product algorithm to decode iteratively on graph with cycles have proven that the approach is a very useful one.

The decoding of compound codes will now be described. The usual practice is to treat each component code as a separate graph on which the sum-product algorithm is applied. The messages (usually called “extrinsic information” in the literature) emerging from the common variables nodes carry information from the other codes. These messages are treated as the \textit{a priori} probabilities of the common variables by the component being decoded. We call an \textit{iteration} one application of the sum-product algorithm on each component code. Usually, many iterations are performed since this usually causes the pseudo-pmf estimates

\textsuperscript{4}Let’s say reasonable block length, rate and acceptable performance.
Figure 2.8: Example of iterative schedule for a two component compound code: one iteration is shown.

at the variable nodes to improve, as the information from each of the component code trickles in.

Figure 2.8 shows an example of a schedule for iteratively decoding a two-component compound code iteratively. One iteration is shown. Messages being sent are shown as dots with arrows.

Now, we have the basic knowledge needed to understand the main subject of the thesis: turbo codes and multibit decoding.

2.3 Turbo Codes

The subject of turbo codes was briefly addressed in the introduction: we got an idea of the performance that can be expected for long block lengths, and how the encoder works. Now, turbo codes and their decoding algorithm will be presented from the factor graph point of view. The explanations are greatly simplified by the examples given in the previous sections since turbo codes are compound codes made up of two convolutional codes.

Figure 2.9 shows a factor graph for a turbo code. One can see that it is formed of two convolutional codes sharing the same input data variables, although one of the codes uses an interleaved (or scrambled) version of these inputs.

Although it is not specified by the factor graph, binary Recursive Systematic Convolutional (RSC) codes are used as components.

The decoding schedule is that of a typical two-component compound code: one applies the sum-product algorithm to each of the convolutional codes (as seen in a previous section)
in turn, for a suitable number of iterations. This number is either fixed by experiment, or variable and depending on the confidence level of the symbols to be decoded. More iterations usually improve performance up to a certain point. Upon termination, the product of the messages on one of the information bit variables is taken as a pseudo-pmf. The maximum mass in each bit’s pseudo-pmf is the symbol corresponding to the APP decoding rule.

Note that in the decoding of each of the compounds, the shared variables are bits, which are assumed to be independent. This, of course, is only an approximation, because we choose to decode the graph with cycles on the components rather than the much more complex tree graph of the whole code.

We will discuss puncturing of multibit codes in other chapters. Puncturing is a common means of increasing the rate of a turbo code. Basically, it means that one simply omits transmitting some of the parity bits in a repeating pattern. For example, a rate 1/2 turbo code is made by omitting one of every two parity bits for each of the two component of an ordinary rate 1/3 turbo code. Decoding a punctured code is very simple: since no channel information is available, the punctured symbols are assumed to have uniform pmfs.

Let us briefly examine the shape of the typical turbo code’s BER vs. SNR curves. Figure 2.10 shows the typical shape of the turbo performance curve. In future sections we will often refer to the error floor, which corresponds to segment B of the figure. This reduction in the

---

5 “Error floor” is the name often used in the literature; however it is not a real “floor”, since it is sloping.
Figure 2.10: Schematic showing the steep descent (A) and "error floor" (B) regions of a typical turbo code's BER vs. SNR curve.

slope of the curve at high SNRs occurs because the dominance of the few hard to discern codewords (at or close to the minimum distance) cause most of the errors. Since they are similar, it takes a larger increase in SNR to eliminate the confusion at the receiver. Most of the turbo code's codewords are easily discernible at the SNRs corresponding to the "floor." and have long ceased to be misidentified by the decoder.

Now that we have discussed factor graphs, the sum-product algorithm, compound codes and turbo codes, we will see in the next chapter that multibit decoding can be described very succinctly.
Chapter 3

Multibit Turbo Decoding

This chapter will introduce the concept, explain the construction, and calculate the complexity of Multibit Turbo Decoding. Multibit turbo codes will be compared to conventional turbo codes.

3.1 General Concept

The algorithm used in decoding many compound codes including conventional turbo codes, LDPC codes and serial concatenated convolutional codes is the sum product algorithm, which involves estimating the probability $P(u_i|y)$ in an iterative process before making a hard decision on the value of bit $u_i$. In this scheme, the compound codes exchange information in the form of (or equivalent to) pseudo probability mass functions\(^1\) (pmf) of bits. Multibit decoding means that component codes exchange information in the form of pseudo joint-pmfs of groups of bits. Any dependence between bits within a group will be used to help the decoding process. Figure 3.1 shows information variables common to two component codes with no grouping, and then with data groups of 2 and 4 bits.

For identical codes, then, multibit decoding can only improve the decoding performance. There will be an improvement if there is a dependence between the newly grouped information symbols. There has to be such a dependency in a compound code, and ignoring it totally

---

\(^1\)We say pseudo because the function is proportional to a pmf. The often used log likelihood ratio can be found from the pmf by finding $\log P(u_i = 1|y) - \log P(u_i = 0|y)$. 

23
Figure 3.1: Schematic representing multibit decoding for different group sizes \( g \), from left to right, \( g = 1, 2, 4 \). The number of dots on the variable nodes indicate the number of bits treated as single variables.

is what makes the code so simple to decode in conventional turbo decoding. On the other hand, if we took all dependencies into account, we would have to jointly decode the whole of the information vector, which would involve some enormous number of computations. We will use the parameter \( g \) to denote the size of the joint decoding group. By varying this parameter, we can change the complexity of decoding, and, depending on the dependence between variables in a group, achieve some gain in decoding performance.

### 3.2 Applying Multibit to Turbo Decoding

To apply the multibit idea to turbo codes, a change must be made to the code. Because grouped bits must remain together throughout the whole system so that the joint probabilities never need to be marginalized,\(^2\) the choice of interleavers is limited to those which preserve the bit groups. Fig. 3.2 shows an ordinary interleaver and one suitable for \( g = 2 \) decoding.

As we will soon see, this "limited" choice of interleaver can weaken the turbo code's performance. To be practical, multibit turbo decoding must more than overcome this performance loss with the advantage of its additional knowledge of joint symbol probabilities.

Figure 3.3 shows factor graphs for a turbo code, for both \( g = 1 \) and \( g = 2 \) representations.\(^2\)

---

\(^2\)This would destroy the advantage of the group, since the product of the marginals of a pdf is not equal to the original.
Figure 3.2: Schematic diagram of two interleavers: at left, a conventional turbo code interleaver and at right, an interleaver suitable for $g = 2$ decoding.

Figure 3.3: Factor graphs for conventional turbo code (left) and the equivalent $g = 2$ realization. Thicker edges show where messages have higher dimensionality.

The codes are identical, since both sport the same grouped interleaver. The multibit code’s graph is shorter since the number of independent variables is reduced by a factor of $g$. The thicker lines on the multibit graph are a reminder that the input and parity symbols are now variables with $2^g$ possible values. The function nodes in the two factor graphs are different functions. In the conventional code, the local functions are indicator functions for valid state transitions but in a multibit code, these functions are indicator function for $g$ consecutive valid state transitions. Fig. 3.4 shows the trellis for two state transitions of a 5/7 RSC code. The computations associated with these two transitions are effected in one local function in a $g = 2$ decoder.

Note that the number of symbols in the termination is identical in Fig. 3.3. Although one function node on the $g = 2$ code can accommodate two termination trellis sections, only one is filled and the other will always be a zero state transition. This implies a slight waste of rate unless $g$ is a factor of $\nu$.\textsuperscript{3}

\textsuperscript{3}Alternatively, the remaining bit(s) can be punctured away.
Figure 3.4: Trellis diagram showing the allowed state transition of a 5/7 code for two local function nodes of the conventional (left) realization and for one transition of the $g = 2$ multibit realization (right).

### 3.3 Decoding Algorithm

The decoding algorithm is identical to that of the conventional turbo code: for each iteration, the “forward-backward” algorithm is performed once on each RSC code (see Section 2.1.7), taking into account the information incoming from the other code via the information variables. As for conventional turbo decoding, the product of the messages entering each $u_i$ variable is taken after the required number of iterations. The most probable symbol is detected from the pseudo pdf resulting from this product operation.

An interesting property of multibit decoding is that each step in the forward and backward passes of the forward backward schedule processes $g$ state transitions at once [7]. One may take advantage of the higher parallelism of the computations to increase a decoder’s throughput.

### 3.4 Decoding Complexity

The number of operations required for decoding turbo codes (including multibit) will now be derived. The variables $|S|$, $|U|$ and $|E| = |U||S|$ will be used to represent the number of possible states, the size of the input alphabet, and the number of edges in one trellis section, respectively.
In the following, we will refer to messages corresponding to Fig. 3.5. Since the decoding algorithm is symmetrical in each component code, let us simply calculate the number of operations required for the forward backward schedule on one component code only, which is also known as a "half iteration":

1. We will assume that channel information messages for parity and systematic bits are produced by the demodulator and incur no computational cost for the decoder.

2. The extrinsic information from the other component must be multiplied with the channel information at the systematic bit. This involves \(|U|\) multiplications.

3. The calculation of the state probabilities given the past information (forward pass: \(\alpha(s_i)\)) requires a product of three factors for each edge in the trellis, and a sum of \(|U|\) terms for each state. This gives \(2|E|\) multiplies and \(|S|(|U| - 1)\) adds per trellis section.

4. The backward pass requires the same number of operations as the forward pass.

5. To calculate the \(\delta(u_i)\) message, the product \(\alpha(s_i)\beta(s_{i+1})\gamma(x_i)\) must be found for each edge and marginalized over the \(|U|\) possible values for the input symbols. This gives us \(|U|\) sums of \(|S| - 1\) terms each and an additional \(2|E|\) multiplies.

In total, we find that each half iteration requires

\[
3|S||U| - 2|S| - |U| \text{ additions, and}
\]

\[
6|S||U| + |U| \text{ multiplications}
\]

per symbol.

For a constituent code with \(|S|\) states and a symbol alphabet of size \(|U| = 2^g\) where \(g\) is the multibit group size, comparing with \(g = 1\), the complexity increases by a factor of approximately \(2^{g-1}/g\) relative to single-bit probability propagation. For \(g = 2\), this implies approximately no increase in decoding complexity. For \(g = 3\), \(g = 4\), the increases in complexity are approximately factors of 4/3 and two, respectively.
Figure 3.5: Fragment of a convolutional code's factor graph corresponding to one state transition and the associated messages.

### 3.5 Memory Requirements

The memory requirements of multibit decoding for small $g$ can sometimes be less than those of conventional decoding. While the number of variables is reduced by a factor of $g$, only the information variables and parity variables $(u_i, x_i)$ suffer an increase in dimension. Table 3.5 summarizes the memory requirements of multibit turbo decoding, for a simple internal representation: the pmf of each variable is represented by a number of scalar values equal to the size of the alphabet of this variable.

<table>
<thead>
<tr>
<th>node type</th>
<th>number of nodes</th>
<th>memory per node</th>
<th>memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_i$</td>
<td>$2K/g$</td>
<td>$2g^{-1}$</td>
<td>$K/g \cdot 2^g$</td>
</tr>
<tr>
<td>$u_i$</td>
<td>$K/g$</td>
<td>$2^g$</td>
<td>$K/g \cdot 2^g$</td>
</tr>
<tr>
<td>$s_i$</td>
<td>$2K/g + 2$</td>
<td>$</td>
<td>S</td>
</tr>
<tr>
<td>total:</td>
<td></td>
<td></td>
<td>$\frac{2K}{g}(2^g +</td>
</tr>
</tbody>
</table>

Table 3.1: Memory requirements for multibit turbo decoding as a function of the number of information bits $K$, the grouping factor $g$, and $|S|$, the number of states. No likelihoods ratios are used in the representation.

As an example, if we evaluate the memory required for for a block length of say $K = 1024$ bits, a 16 state code and $g = 2$, we find that we use about 56% of the memory required for the $g = 1$ case. These savings are for the simple pmf representation described above;
the savings are reduced if the conventional turbo code uses likelihood ratios to represent bit
probabilities. The memory required for \( g = 2 \) is then about 59% compared to the \( g = 1 \) code
that uses likelihood ratios.

These savings are substantial. In VLSI implementations of turbo decoders, this may
mean important area or power savings.

3.6 Choosing Grouped Interleavers

The choice of interleaver is very critical for a multibit turbo code. As we will see in the next
chapter, a random selection of grouped interleaver will likely produce a code suffering of
severe flattening of the BER vs. SNR curve. Some interleaver selection criteria are required
to ensure that multibit codes work well.

In the next sections, we look at why some interleavers are bad, and then we present two
criteria to choose interleavers. Although these criteria apply to any turbo code, a simple
method of modifying them to grouped interleavers is presented.

3.6.1 RSC Code response to weight two inputs

Recursive systematic convolutional (RSC) codes may be viewed as infinite impulse response
filters. An input consisting of single one followed by a stream of zeros produces a high
weight periodic binary stream at the (non-systematic) output of an RSC coder. What is the
minimum weight output with a weight two input? Let us first consider (3.1) which shows
the impulse response of the 21/37 RSC code in Figure 3.6. We see that the period \( \tau \) of the
repeating tail is five bits long:

\[
\begin{align*}
\text{input} & \quad 1000000000000000000000000000 \cdots \\
\text{output} & \quad 11001 \underbrace{01001}_\tau \underbrace{01001}_\tau \underbrace{01001}_\tau \cdots
\end{align*}
\]  

(3.1)

We know that this IIR RSC code is a linear system in modulo 2 arithmetic \((f(a \oplus b) = f(a) \oplus f(b))\) so that we can cancel most of the high weight tail of the impulse response by
adding to it a shifted copy of itself:

\[
\begin{align*}
\text{input} &: \quad 1000010000000000000000 \cdots \\
\tau &\quad (3.2) \\
\text{output} &: \quad 110010100101001010001 \cdots \\
\oplus &\quad 000001100101001001001 \cdots \\
&\quad 1100110000000000000 \cdots 
\end{align*}
\]

We can easily extend this little experiment to any weight two input with a spacing that is a multiple of \(\tau\).

### 3.6.2 Avoiding low-weight code words in turbo-codes

A bound to the performance of a (maximum likelihood decoded) code can be computed if its codeword’s weight distribution and the channel are known. Different regions of the BER vs. SNR curve are affected by the different weights, and the low weight codewords especially affect the “error-floor” part of the performance curve [19, 2]. As we will see in the next chapters, some grouped interleavers have a tendency to flatten the BER vs. SNR curve, and flatten early. The flattening is probably partly due the effect of the low weight components of the code weight spectrum. By increasing the minimum distance of the turbo code and reducing the number of other low weight code words, the flattening effect can be diminished. We can do this by first understanding how the turbo code’s two RSC component codes interact through the interleaver to produce low-weight codewords.

Turbo codes combine two RSC codes in a scheme dubbed parallel concatenation (see
Figure 1.2). A critical component of this scheme is the interleaver block. Without it, there is no need for the second RSC constituent code, as it would be producing the same output as the first one. We can easily see that this would be a bad code (from the minimum distance and low weight multiplicity points of view). Take for example an input of two bits separated by $k\tau$, the period of the RSC code. The resulting code word has weight (at best, with an all-1 tail) $2 + 2k\tau$.

If an interleaver (say randomly chosen) is used, the situation changes completely. Most $k\tau$ separated weight-two inputs will be mapped to two bits with a different spacing. This means that the 'tail' of bits induced in the interleaved RSC code by one of the 1's will not be canceled by the other, giving a high-weight code word which is far from the all-zero code word.

The other weight inputs may also be considered. Normally, only the first few will affect the code's performance since as the input weight increases, the chance of a low weight output at the output of either interleaver diminishes quickly. If the code's trellis is truncated at the end of a block, then there are more odd low weight cases to be considered: the case when one of the ones is very close to the end of the block and is mapped to the same position in the second code. This produces (at worst, if the trellis is truncated) a weight three code word for an input weight of one, or $2 + 2k\tau + 2$ for a weight three input.

3.6.3 Simple "Weight-Two" criterion

Following the previous observations, we can develop a simple method of choosing interleavers of a given size $K$ and for a given $\tau$ for a conventional turbo code. We will ignore weight-three and higher inputs for the time being.

For every $K - d_{in}$ weight-two inputs spaced by $d_{in} \in \{\tau, 2\tau, \ldots, M_{max}\tau\}$, verify that their spacing at the output of interleaver satisfies $d_{out} \notin \{\tau, 2\tau, \ldots, M_{max}\tau\}$

The number of RSC impulse response periods to be considered is chosen with the $M_{max}$ parameter. In practice, an interleaver search using this criterion is implemented using some
kind of optimization scheme. For example, genetic algorithms could be used. When used in conjunction with a fitness function embodying the optimization rules, one can breed an increasingly successful crop of interleavers.

A picture makes the mappings to avoid more obvious. Figure 3.7 shows an interleaver "taboo plot" for the weight-two criterion for $M_{\text{max}} = 2$: the topmost row of squares represents the input to the interleaver, and the bottom row is the output and each square represents an interleaved symbol. The weight-two criterion given above may be restated as follows: given an input symbol (hashed in top row) and its mapping through the interleaver (hashed bottom row), any other mapping from black squares in the top row to black squares in the bottom row are disallowed mappings.

![Figure 3.7: Block diagram of taboos for increasing minimum distance and reducing multiplicity of second minimum distance in an interleaver](image)

3.6.4 Modifying Interleaver Selection Criteria for use with Group Interleavers

The method to modify an interleaver selection criterion so that it applies to grouped interleaver can be summarized in two main points. To begin, if bit $a$ should not map to bit $b$ according to the selection criterion in a $g = 1$ interleaver, then the mapping should not be allowed in the grouped interleaver. Secondly, if one of the bits in a group is part of a disallowed mapping, the mapping of the whole group is disallowed. In terms of costs of mappings, the cost of a grouped mapping is sum of the costs of the individual bit mappings. In this discussion, we are trying to find the zero cost interleavers, and that is why we use

---

4The search algorithms used with this thesis were developed after some work with Warren Gross and Vincent Gaudet.
such words as "disallowed mappings" and "taboo."

Figure 3.8 shows how the weight-two criterion on a $K$ symbol interleaver can be applied to a group interleaver with $g = 2$ of length $K/g = K/2$ symbols.

![Diagram of a group interleaver](image)

Figure 3.8: Block diagram showing derivation of taboos for "weight-2" criterion in an $g = 2$ blocked interleaver for $\tau = 5$

### 3.6.5 The Spread Rule

In iterative decoding of turbo codes, as with all compound codes, distance spectrum of the code is not the only factor affecting the decoding performance. As mentioned in Chapter 1, iterative decoding is an approximation to ML decoding of the whole block, and it is not guaranteed that the sum-product algorithm on the graph will converge to the ML value, because the graph has cycles.

It is generally agreed that the number and "length" of cycles in a code's factor graph will affect decoding performance, and the fewer and longer cycles, the better [24].

A very interesting interleaver choosing rule is the spread method [9]. The spread interleaver ensures that every weight-two input to the interleaver that is spaced less or equal to $\sigma$ maps to a spacing of more than $\sigma$ (see Fig. 3.9 for a schematic). Although this method is not clearly motivated in [9], it produces an interleaver with interesting properties. First, if $\sigma \geq 2\tau$, it improves the code's minimum distance for weight-2 inputs (satisfies the "weight-2" optimization rule). Furthermore, it ensures that error events occurring near the end of the interleaver (esp. for odd weight inputs) have at least length $\sigma$. The weight-2 rule for smaller $\tau$ and increasing $g$ becomes identical to spread interleavers, as the taboos merge into a continuous region. An advantage of the $g > 1$ code when finding spread interleavers is that
one can always find a spread interleaver that has a higher $\sigma$ in terms of bits. This is because among interleavers of size $K$, the largest spreads that can be found in a reasonable amount of time are $\sigma \leq \sqrt{\frac{K}{2}}$ [9]. This means that for a group size of $g$, the largest spreads available (in bits) are a factor of $\sqrt{g}$ larger than the equivalent $g = 1$ spread.

**Combining rules**

The spread criterion seems to be a good all-around solution to interleaver optimization if we can find an interleaver with $\sigma \geq \tau$, or even better, $\sigma \geq \tau$ since it will improve the low-end of the distance spectrum of the so constructed turbo codes, which seems to be a good thing. In practice, these large spreads are often impossible to achieve for small interleavers. Table 3.2 shows the minimum interleaver size $K$ in symbols, satisfying $\sigma = \sqrt{K/2}$ for different $\sigma$. We can see that for larger $\tau$, it can be impossible to find an interleaver with the desired spread. An interesting solution is to combine the spread and weight-2 rules so that the critical even weight error events are eliminated by the weight-2 taboos, and some of the other very low weight events are eliminated by the spread taboos.

**3.7 Puncturing Methods**

Multibit turbo codes offer two kinds of puncturing methods: by block, where a whole block of $g$ bits are punctured, and by sub-block, where only certain bits out of the block are transmitted.

One advantage of full block puncturing is that there is a reduced number of messages to be processed in the application of the sum product algorithm because the non-transmitted bits require no information from the demodulator. Some experimental results on different
Table 3.2: Minimum interleaver size to find a spread interleaver for various combinations of $g$, $\tau$ and $\sigma$

<table>
<thead>
<tr>
<th>$\tau$ (code)</th>
<th>$\sigma = \tau$</th>
<th>$\sigma = 2\tau$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3(5/7)$</td>
<td>18</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>72</td>
</tr>
<tr>
<td>$5(21/37)$</td>
<td>50</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>200</td>
</tr>
<tr>
<td>$7(7/5)$</td>
<td>98</td>
<td>196</td>
</tr>
<tr>
<td></td>
<td>49</td>
<td>392</td>
</tr>
<tr>
<td>$15(23/35)$</td>
<td>450</td>
<td>900</td>
</tr>
<tr>
<td></td>
<td>225</td>
<td>450</td>
</tr>
</tbody>
</table>

puncturing schemes are shown on p. 44, in section 4.4.

### 3.8 Sub-Block Operations

Interleavers are weight preserving transformations that are limited to permuting multi-bit groups. Bits within symbols could also be interleaved. Going even further, the interleaver can be replaced by a non-weight preserving transformation. The next section explores these ideas, made possible by multi-bit decoding.

Sub-block operation refer to operations that affect the bits inside a multibit “block”, or symbol. These operations may be used to implement sub-block interleaving, for example. Fig. 3.10 shows how the interleaver in a turbo code can be augmented to allow sub-block interleaving, or any sub-block operation. The heart of the modifications is the addition of the function nodes which effect the “operation”. A row of variable nodes was added since function nodes cannot be directly connected to other function nodes. The added variable nodes have only two edges and thereby require no operations during the sum product algorithm. The operation that actually has to be performed at the function nodes is a change of indices in the messages: a permutation of these. The operations can be made in place (no extra memory requirements) and are quite inexpensive computationally.
Sub-Block Interleaving

Sub-block interleaving is simply a permutation of the bits inside a block. The difference here is that in a multibit decoder, this takes the form of a symbol substitution. For example: if in a $g = 2$ code the bit order is flipped for a particular symbol, the function that has to be implemented is as follows:

\[
\begin{align*}
00 (0) & \rightarrow 00 (0) \\
01 (1) & \rightarrow 10 (2) \\
10 (2) & \rightarrow 01 (1) \\
11 (3) & \rightarrow 11 (3)
\end{align*}
\] (3.3)

How can this improve performance? Given a random blocked interleaver, and a particular choice of permutation for each block, the interleaving may break a bad weight-two mapping, for example. For an ordinary grouped interleaver, a particularly bad weight-two mapping will yield $g$ minimum distance events to the weight spectrum of the code, thus reducing its performance at moderate to high SNRs. The sub-block interleaver has reduced and possibly eliminated the possibility of these $g$ minimum distance error events occurring.
Sub-block linear functions

Sub-block linear functions \(^5\) subsume sub-block interleaving and have the ability to change the weight of the output. A weight changing property is interesting because it can help us in achieving our goal of achieving the performance of a random code in a structured way.

With more general, non-weight preserving functions, the problem of avoiding low weight code words changes completely. It suffices to say that with careful design, an added element of “randomness” will decrease the correlation between the two constituent codes and improve performance.

This scheme may not be used to advantage with conventional turbo codes for two reasons: first, because of complexity, and second, because of confusion arising from marginalization. The complexity of the computations at the function nodes is increased because marginalization is now required. Multiple messages must be sent, and many marginalizations must be performed. These marginalization cause the second problem: the bits at the function’s input and output ports are assumed independent, which they certainly are not. This independence assumption allows the bits to assume, if detected, arrangements impossible from the function’s point of view. Information is “confused” and this will cause errors.

\(^5\)The reason for limiting ourselves to linear functions is because is easier to use the existing simulation program.
Chapter 4

Experimental Results

This chapter presents the most salient results of the several experiments conducted. It begins with a description of the simulation software used to produce all the results, which was developed during the course of the research. Next, is a section describing the details of the system model used for the simulations. Finally, the results are presented, organized by experiment. For further results, the reader may find the more complete catalog of results in Appendix C of interest.

4.1 Simulation Software

The performance of codes and their decoding algorithms in noise may be estimated by simulation. A sizeable amount of time devoted to the present thesis project was devoted to the development of a simulation software package. The aim was to develop a tool which was general enough to be useful for applying the sum-product algorithm to arbitrary finite factor graphs, and especially to provide high level constructs to facilitate the simulation of codes and their decoding algorithms.

It was decided that the C++ programming language was to be used [23], since it supports the object-oriented programming paradigm. Object-oriented (OO) programming languages allow the user to define objects (called classes in C++); these objects encapsulate both data structures and behavior. The OO programming paradigm is an alternative the very
well established functional programming paradigm supported by such languages as C and Fortran, which separates the definitions of data and the functions that act on them.

Object oriented languages allow complex objects (such as component codes) to be defined and manipulated which as much ease as the built in data types (like the integer) used in most programming languages.

The result of this software development endeavor is a series of C++ classes which can easily be expanded, but at the moment allow the user to define and manipulate at the highest level:

- Turbo Codes, both with conventional and multibit decoding
- RSC and ordinary convolutional encoders for use as components

At a slightly lower level of abstraction, one may use vectors of variable nodes, vectors of function nodes or vectors of constant functions (useful to represent memoryless channel variables), and interconnect them either normally, or through interleavers, both blocked and ordinary. At the lowest level, we can describe individual local functions and variable nodes. There is also an object called a “function representation”, which allows a function to be defined once, and to be used by many local function nodes.

A more detailed description of the classes and their relationships, as well as some source code examples can be found in Appendix A.

4.2 System Model

This section describes the system setup which is used to evaluate multibit decoding in the remainder of this document.

In order to propagate multibit probability distributions in iterative decoding, we will assume that the interleaver in the turbo code has the effect of permuting bits in multibit groups; in other words, we assume that the interleaver operates on symbols over $GF(2^g)$. where $g$ is the number of bits in a group.

The trellises for each constituent code were terminated by appending a tail of $\nu$ “zero-feedback” sections to them [10]. This means that the input to the constituent codes’ shift
register is set to zero. After \( \nu \) clock cycles, \( \nu \) parity bits will have been generated for a binary code, and the state will be zero, as the shift register will have been flushed of any remaining non-zero values.

The rate of the resulting (non-punctured) code will be slightly less than \( 1/3 \) since the output symbols generated during trellis termination are transmitted. If \( K \) is the interleaver size, then a total of \( Kg \) information bits are transmitted in a block of \( 3Kg + 2\nu_s \) bits, resulting in a transmitted bit rate \( R \) given by:

\[
R = \frac{Kg}{3Kg + 2\nu_s} = \frac{1}{3 + 2\nu_s/Kg} \approx 1/3,
\]

where \( \nu_s \) is \([\nu/g]\).

**Channel Model** Symbols in \( GF(2^g) \) are transmitted as \( g \) separate bits over an additive white Gaussian noise channel. The channel inputs are \( \pm 1 \), so a code of rate \( R \) yields a transmitted bit energy \( E_b = 1/R \). The noise variance per sample is \( \sigma^2 = N_0/2 \), where \( N_0 \) is the one-sided power spectral density of the noise. The common practice of expressing the results in terms of \( E_b/N_0 = (2R\sigma^2)^{-1} \), expressed in decibels, will be followed.

At the receiver front-end, the likelihoods of the individual bit transmissions are grouped into a vector expressing their joint probabilities before being sent to the decoder.

**Interleaver Selection** A “random” choice of interleaver means that the interleaver was selected uniformly from all possible interleavers of the prescribed size. Grouped interleavers of size \( N \) and group size \( g \) are simply interleavers of size \( N/g \), permuting multibit symbols of size \( g \) without any permutations of the bits within a symbol. Optimized interleavers were randomly selected from all interleavers satisfying a particular optimization criterion. An important assumption was made when generating results. It is assumed that that there is not much variance in performance amongst the members of a class of randomly chosen interleavers, and that the probability of picking an especially good or bad interleaver is minute. The reason for making this assumption is practical: simulations take a long time to perform, and we want to test as many interesting configurations as is possible.
4.3 Experiments

This section will present some actual simulation results. The BER vs. SNR plots presented include error bars on all points. These represent an estimated 95% confidence interval. Their derivation is described at length in Appendix B.

4.3.1 Impact of grouped interleaver on code performance

A multibit system differs from the conventional system in two ways: the decoding, and the interleaver. Let us now observe (in a somewhat artificial setup) the effect of the interleaver alone.

The multibit system requires an interleaver that preserves groups of bits. This interleaving strategy has a negative impact on the code and its performance. By evaluating (by simulation) the effect of grouped interleavers on a conventional turbo code's \((g = 1)\) performance, we can try to see the effect of grouping in the interleaver. Figure 4.1 shows the effect of grouping on a rate 1/3, \(K = 1024\) information bit turbo code.

We see that grouping in the interleaver has a large effect on the code. The error “floor” rises by almost an order of magnitude with the group size going from \(g = 1\) to \(g = 2\), and \(g = 2\) to \(g = 4\). We must take the result of this experiment with a grain of salt however, since along with the code's distance spectrum is not the only change effected by the grouping of the interleaver: the bitwise independence assumption of the MAP decoding is also weakened with the use of a grouped interleaver. The two effects cannot be totally separated.

4.3.2 Effect of grouping on decoding performance

How does multibit decoding compare to conventional decoding in performance? Let's use identical codes (identical component codes and grouped interleavers) for a representative example:

The results show that there is a substantial \(Eb/No\) gain (with these particular interleavers) in going from \(g = 1\) to \(g = 2\) and, although the grouped interleaver gives a bad code, an even more substantial gain in going from \(g = 1\) to \(g = 4\). This gain increases with block
Figure 4.1: Effect of blocked interleaver on conventional turbo code for blocks of $b = 1$ (dotted), $b = 2$ (solid) and $b = 4$ (dashed) bits. Code is $23/35$, $K = 1024$, 8 iterations.

There is a change in shape in the curves that is more apparent in the $g = 4$ case, where the steepness of the descent is greater. The $g = 4$ curve for the second iteration crosses the $g = 1$ eight iterations. This means that at that point, the $g = 4$ decoding can achieve a performance equal to the $g=1$ decoding, with approximately half the number of operations.

### 4.3.3 Real Systems with Random Interleavers

The last two sections have artificially separated the code and the decoding aspects of multi-bit systems. Figures 4.4, 4.5 show results comparing multibit decoding with conventional decoding in a more “fair” way, meaning that the $g = 1$ is a conventional turbo code with a random interleaver, and the $g > 1$ have randomly chosen blocked interleavers.

For short block lengths, the weakness of the grouped interleaver code dominates, and causes the performance of the multibit code to lag behind the conventional code in all but the lowest SNRs. The multibit code improves to the point of surpassing the $g = 1$ code on
Figure 4.2: Identical $K=1024$ bit $23/35$ turbo codes. Decoded as $g = 1$ with interleaver in blocks of $b = 2$ (dotted), and $g = 2$ (solid) at 8 iterations.

Figure 4.3: Identical $K = 1024$ bit $23/35$ turbo codes. Decoded as $g = 1$ with interleaver in blocks of $b = 4$ (dotted), and $g = 4$ (solid) at 4.8 iterations.
Figure 4.4: 23/35 turbo code, $K = 1024$ information bits, randomly chosen interleavers, $g = 1$ (dotted), $g = 2$ (solid) and $g = 4$ (dashed) at 8 iterations.

most the SNR, BER plane for long block lengths. We see that a random choice of interleaver, although it gives a fair performance for conventional turbo codes, is only adequate for very long block lengths for multibit codes.

### 4.4 Puncturing Methods

Some experiments were made comparing rate 1/2, $g = 1$ codes to both sub-block and block punctured $g = 2$ codes. Fig. 4.6 shows the results of such an experiment on some 23/35 codes.

For this particular case, the method of puncturing seems to make little difference in performance, except maybe at high SNR.
Figure 4.5: 23/35 turbo code, $K = 16384$ information bits, randomly chosen interleavers showing iterations 8, 12 and 16 for $g = 1$ (dotted) and $g = 2$ (solid)

4.5 Interleaver Optimization

Since we are looking for performance improvements mostly for relatively short blocks, the following experiments will all be of codes with $K = 256$ information bits. Figures 4.7, 4.8 and 4.9 show various attempts to improve interleavers for each of $g = 1, 2$ and 4. We notice immediately that the large spread surpasses all others in performance, and that the weight-2 method either worsens the results or provides only a marginal improvement.
Figure 4.6: Comparison of 23/35 codes rate 1/2 punctured turbo codes with information block length of 256 bits: $g = 1$ (dotted), $g = 2$ sub-block puncturing (solid), and $g = 2$ block puncturing (dashed).
Figure 4.7: Comparison of various attempts to improve interleavers for $g = 1$, $K = 256$, decoded with 2 and 8 iterations: a) is random choice, b) is simple optimization avoiding $d = \tau$ only, and c) is a spread interleaver with $\sigma = 10$. 
Figure 4.8: Comparison of various attempts to improve interleavers for $g = 2$, $K = 256$ bits, 8 iterations: a) avoids $d = \tau$, b) avoids $d = \tau, 2\tau$, c) is a random interleaver for comparison, d) has $\sigma = 7$ and e) has $\sigma = 4$ while avoiding $d = \tau$. 
Figure 4.9: Comparison of various attempts to improve interleavers for $g = 4$, $K = 256$ bits, 8 iterations: 

a) is random interleaver for comparison, 
b) avoids $d = \tau, 2\tau$, 
c) has $\sigma = 4$ while avoiding $d = \tau, 2\tau$ and 
d) has $\sigma = 5$. 
Figure 4.10: Comparison of two codes having $K = 256$ information bits for iterations 2, 4 and 8. Both use spread interleavers. The dotted line is a $g = 1$ code with $\sigma = 10$ and the solid line is for a $g = 2$ code with $\sigma = 7$.

4.6 Multibit vs. Conventional for short blocks

To make a fair comparison of the performance of multibit and conventional turbo decoding, both should be optimized. Fig. 4.10 shows the best $g = 1$ pitted against the best out of the various $g = 2, 4$ simulations shown in the previous sections: the $g = 1$, with a spread of 10 and the $g = 2$ with a spread of 7. Only iterations 4, 8 are shown. It is clear that we have overcome the original flattening problem, and that the multibit code now shows an almost constant gain in performance with respect to the conventional one. At the eighth iteration, the $g = 2$ code has about half the BER of its $g = 1$ counterpart. This gain can be traded off for decoding complexity. Since the complexity per iteration for $g = 2$ is about the same as for $g = 1$, we can say that this $g = 2$ code can achieve the performance of a $g = 1$ code at eight iterations, using half as many operations.
### Table 4.1: Complexity increase factor for different values of $g$, assuming equal cost for addition and multiplication. The values are normalized with respect to $\kappa$ for $g = 1$.

<table>
<thead>
<tr>
<th>$g$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\kappa_g/\kappa_{g=1}$</td>
<td>1.00</td>
<td>1.06</td>
<td>1.46</td>
<td>2.22</td>
</tr>
</tbody>
</table>

#### 4.7 Performance and Complexity

Although multibit decoding coupled with a good choice of interleaver gives a consistently lower bit error rate than conventional turbo codes with conventional decoding, the gain in required SNR for equivalent performance seems to hover around 0.1 to 0.2 dB, which is quite small. As will be shown in this section, the advantage of using multibit decoding is more likely the reduced number of iterations required for equivalent performance, and the reduced memory requirements rather than the raw SNR gains.

The computations involved in decoding a turbo code include a number of additions and multiplications (see Eqns. 3.4 and 3.4). To allow easy comparisons between the complexity of different turbo codes with grouping factor $g$, and convolutional codes with $|S|$ states, let us define $\kappa$ to be the total number of computations required by one half-iteration of multibit decoding

$$\kappa = \frac{|S|}{g} \left( 9 \cdot 2^g - 2 \right) \text{ computations per bit}$$

Table 4.7 shows the complexity increase factor, $\kappa_g/\kappa_{g=1}$ associated with different values of $g$. The performance vs. complexity graphs that follow will use these factors to compare complexities associated to multibit decoding and to conventional turbo decoding.

Figures 4.11, 4.12, 4.13 show complexity vs. performance for BER=10^{-3}, 10^{-4} and 10^{-5} for 23/35 codes, $g = 1, 2$ for $K = 16k$ bits, and 4.14, 4.15, 4.16 show the same for $g = 1, 2, 4$ for the shorter block length of $K = 256$. The short codes use spread interleavers ($\sigma = 10, 7, 5$ for $g = 1, 2, 4$ respectively) and the long codes simply use a random interleaver.

We observe that there is quite a reduction in complexity required in going from $g = 1$ to $g = 2$, for the whole range of BERs for the long block length. It would seem that the complexity of $g = 2$ multibit decoding can drop in the range of 2/3 to 1/2 the equivalent
Figure 4.11: Relative Complexity vs. SNR required to achieve a BER of $10^{-3}$ for 23/35 codes, $K=16384$ bits, $g = 1, 2$.

Figure 4.12: Relative Complexity vs. SNR required to achieve a BER of $10^{-4}$ for 23/35 codes, $K=16384$ bits, $g = 1, 2$. 
Figure 4.13: Relative Complexity vs. SNR required to achieve a BER of $10^{-5}$ for 23/35 codes, $K=16384$ bits, $g = 1, 2$.

Figure 4.14: Relative Complexity vs. SNR required to achieve a BER of $10^{-3}$ for 23/35 codes, $K=256$ bits, $g = 1, 2, 4$. 
Figure 4.15: Relative Complexity vs. SNR required to achieve a BER of 10^{-4} for 23/35 codes, K=256 bits, g = 1, 2, 4.

Figure 4.16: Relative Complexity vs. SNR required to achieve a BER of 10^{-5} for 23/35 codes, K=256 bits, g = 1, 2, 4.
complexity of $g = 1$ decoding. For the short block lengths, the advantage of multibit decoding was mostly near the steep (low SNR) part of the curve corresponding to diminishing returns for the $g = 1$ code; the complexity could be reduced in half for an equivalent performance. The $g = 4$ code's performance in the short block does not justify the extra computations it requires.

### 4.8 Lessons Learned

From the results presented in this chapter, the main lessons that we learned are:

- Multibit codes need to have carefully chosen interleavers to perform adequately because the grouped interleavers weaken the code. Spread interleavers gave the best performance of those tested.

- Multibit decoding for $g = 2$ is a "free lunch": it requires practically the same complexity, $\approx 40\%$ less memory, and better BER than $g = 1$ decoding through the whole spectrum of measured SNRs, for short and long block lengths.

- There is a sizeable reduction (30-50\%) in the complexity required to decode long block lengths for equivalent performance in going from $g = 1$ to $g = 2$.

- For $g = 2$ rate 1/2 turbo codes with random interleavers, sub-block and symbol puncturing produce almost the same BERs. Symbol puncturing should be used because it requires less computation.
Chapter 5

Conclusions

5.1 Summary

We have found that a suitably designed turbo code using a spread interleaver decoded by the multibit method (especially the $g = 2$ or decoding using quaternary symbols) offers several advantages over its conventional counterpart. Multibit decoding for $g = 2$ offers performances surpassing ordinary turbo decoding over the whole range of block lengths from $K = 256$ to $K = 16384$ information bits, without requiring any additional expense in computing. In fact, for many cases, the same performance may be achieved with half the computations. It was also shown that memory requirements for such a decoding scheme were equal or lower than the conventional method, and that the interleaver description was simpler. The throughput of such a decoding scheme, if it were suitably implemented, could be higher since more computations may be done in parallel [7].

It is felt that the increase in dimensionality of the basic information symbol in compound codes and other graphs with cycles is a subject worthy of further investigations.

5.2 Contributions

This thesis makes three main contributions. First, multibit decoding was explored through extensive simulations for a variety of block lengths and interleaver types. Second, the ex-
Experiments performed showed that spread interleaving allows multibit decoding to work well for relatively short blocks. Third, software was developed allowing one to create arbitrary factor graphs, and to apply the sum-product algorithm to them. The software allows quick code prototyping and is a powerful simulation tool.

5.3 Suggestions for Further Work

This thesis is by no means the last word on multibit turbo decoding. Much more time and many more simulations need to be performed to fully explore the potential of this approach. Some interesting avenues one might explore include the exploration of the use of non-binary codes as components of turbo codes, the investigation of the effect of multibit decoding on serial concatenation, the investigation of joint optimization of the interleaver and sub-block operations, and to find the effect of using the max-log-map decoding on performance of multibit decoding.
Appendix A

Simulation Software

A large part of the work that went into this thesis was devoted to developing the software that allowed us to perform simulations of various codes. The aim was not only to allow simulation of various turbo codes, but also to be general enough to allow the sum product algorithm to be applied to any finite factor graph. This appendix describes some of the details concerning “facTor”, its design and implementation, and finally some recommendations for further work.

A.1 Description

FacTor is a general factor graph simulation software library: a series of class definitions to be used as construction blocks. Various higher level building blocks are included for the simulation of compound convolutional codes on memoryless channels.

A.2 Programmer’s Guide

The object oriented (OO) programming paradigm and the C++ programming language were used for designing and implementing facTor. For a good reference on OO design, I recommend the book by Martin [17]; the standard reference for the C++ programming language is Stroustrup [23].
A.2.1 Design

Why use an object oriented design approach? An object oriented approach was taken because of the powerful abstraction mechanisms available to a programmer. This programming paradigm makes (if the design is good) the definition and manipulation of objects such as generic memoryless channels, or vectors of edges, or component codes, almost as easy as the native types like the int. The key difficulty in creating such a high level work environment is the design: choosing what is an object and its responsibilities.

Choice Of Objects Figures A.1, A.2 and A.3 show my choice of objects: what they contain and the structure of the inheritance. The notation used is the Booch diagram [17] and can be summarized by the following points:

- A class is represented by a dotted outline.
- An abstract class is marked by an “A” in a triangle.
- Inheritance is shown by an arrow pointing to the parent.
- A “has a” relationship is shown by a line with a filled circle at the owner class and a filled square in the other class.
- A class “has a reference to a” is the same as the “has a” except that the square is white.
- Templatized classes have a square on the upper right, where the template parameters are written.

Many classes were made templates, in the hopes that the code could be better optimized. Templates also give the advantage of letting the compiler do so-called static type checking. This means that many foolish mistakes can be avoided at compile-time, which saves time. The unfortunate side effect is that some unwanted interdependencies developed, and these require some awkward notation in many classes.
Figure A.1: Booch diagram of the single-node construction generic factor graph classes.

Figure A.2: Booch diagram of the vector related generic factor graph classes.
A.2.2 Implementation

The author chose the C++ programming language to implement the facTor classes because it is widely available, can be well optimized, and is becoming standardized. The new standards include the Standard Template Library (STL) which provides a multitude of standard, useful classes such as containers and simple algorithms.

To show what code using the facTor classes looks like, two examples classes are included: one which works at a low level of abstraction to implement the local function node in a convolutional code, and one which is at a high level of abstraction which allows the simulation of a conventional turbo code.

The first source code example on the next page illustrates the high level of abstraction possible (and hopefully, its elegance) with the use of an object oriented approach. A conventional turbo code is implemented and ready to simulate in only a few lines. The interface provided is made of three main calls: `receiveIn()`, `oneIter()` and `detMap()`. The only low level knowledge needed in this piece of code relates to "CC", the convolutional code specification class.

The second source code example shows the RSCFnRep (RSC Function Representation)
class, which is gives the functionality to all function nodes in a convolutional code. Its most important (and prominent) feature is the implementation of the `marginalize()` function. This function calculates outgoing messages to any of this local function's argument nodes.
Example 1

```cpp
#include"../blocks/compCodes/CCcodeSpec.hh"
#include"../blocks/compCodes/CCcode.h"
#include"../blocks/compCodes/CCStateSpec.hh"
#include"../blocks/channels/ChannelVec.hh"
#include"../blocks/plumbing/PlumbAny.h"
#include"CompCodeBase.hh"

template<CC>
class TurboConventional1: public CompoundCodeBase {
public:
  typedef typename CC::Edge DST;
  typedef typename CC::DataType DT;
  typedef EdgeIterator<typename CC::Edge> EdgeIter;
  typedef enum { systritchan, codetrack, codetrack2 };
private:
  int itSize;
  VarNodeVarNode<DataType, CC::idim, 3> in;
  ASTATE.dst dst;
  ICchannelVec<CC> itsChnVec;
  float itsRate;
  // constructor
  TurboConventional1(int size, Interleaver inter):
    itsSize(size),
    itsRate(1.0/1.0-1.0*itsRate/1.0, itsRate)
  { itSize = CC::codeTable[0];
    // destructor
    TurboConventional1();
    // implementation of inherited functionality
    inline void receiveIn(ChannelBase chan) {
      // receive the all zero systematic
      in.clearmark();
      itsChnVec.setChannel(chan);
      itsChnVec.sendall1();
      itSize = CC::codeTable[0];
      break;
      recv(sysbitchan);
      break;
      default:
      return;
    }
    inline void onIter() {
      in.receive(codetrack);
      in.forwardBackward(1);
      in.receive(codetrack);
      in.forwardBackward(1);
    }
    inline void setMAP() { return in.setMAP();
    }
    inline void decodeBlock() { return itsSize=CC::bitsPerSym;
    }
};

Example 2

```
A.3 Recommendations for future work

Here is list of things that could be improved, or that should be done:

- Some higher level blocks are implemented by using the fundamental factor graph classes which may add unnecessary memory and computational overhead. The critical blocks, such as the ones implementing RSC codes, should be made to have an interface compatible with the rest of facTor, but have optimized “guts”.

- A factor graph description language and its associated compiler or interpreter could be developed for quick code prototyping.

- facTor’s code should rely less on templates so that it is more flexible, and to increase the knowledge separation between high and low level classes. This may allow facTor to be used in conjunction with an interpreter.

- A user’s manual for facTor should be written.
Appendix B

Tolerance of Results

This Appendix discusses how error bars for the results were generated and under which assumptions.

**BER Estimator** The results that we want are an accurate estimate of the BER of the codes, for a useful range of SNRs. A simulator was developed for this purpose. What follows is a description of the details surrounding the estimation of BER and how to get an idea of the accuracy of this estimate. For a detailed description of the simulation software that was developed in the context of this thesis, please refer to Appendix A.

In the simulation, many identical experiments are performed which consist of receiving a noisy block of data (at a particular noise level in which we are interested in). The result of each decoding experiment is the number of bit errors made by the decoder. The number of bit errors made in a block $w$ is a random variable drawn from an unknown probability mass function $W(x)$ which depends on the particular code and noise level. The BER of the code is defined to be $E(W(x))/K$, where $E()$ stands for the expectation operator, and $K$ for the number of information bits in a block. The unbiased estimator for the BER based on $n$ samples is simply the scaled mean of the number of errors recorded is

$$\hat{BER} = \frac{1}{K \cdot N} \cdot \sum_{i=1}^{N} w_i$$
How close is $B\overline{E}R$ to $BER$? Since the estimate is a random variable, this information can be found from the probability mass function of the estimate. Let us use the central limit theorem to help:

$$S_n = W_1 + W_2 + \ldots W_n$$ (B.1)

$$\frac{S_n - n\mu_w}{\sqrt{n\sigma_w^2}} \xrightarrow{distr} \mathcal{N}(0,1) \text{ as } n \to \infty$$ (B.2)

Proceeding to find an expression for the mean,

$$S_n - n\mu_w \xrightarrow{distr} \sqrt{n\sigma_w^2} \mathcal{N}(0,1)$$

$$\frac{S_n}{n} \xrightarrow{distr} \mathcal{N}\left(\frac{\mu_w}{\sqrt{n}}, \frac{\sigma_w}{\sqrt{n}}\right)$$ (B.3)

Finally, we find what we are interested in, a scaled mean:

$$B\overline{E}R = \frac{S_n}{nK} \xrightarrow{distr} \mathcal{N}\left(\frac{\mu_w}{K}, \frac{\sigma_w}{K\sqrt{n}}\right)$$ (B.4)

As the number of terms in the sum tends to infinity, the scaled sum tends to have a distribution that is closer and closer to a normal distribution with a mean which is $E(W)/K = \mu_w/K$ and a variance which tends to zero. In the simulation, only a finite number of experiments are performed. Two factors affect the pmf of our estimator: the number of terms in the sum (number of experiments) and the distribution of $W(x)$. If the number of experiments is small, or if the variance of $W(x)$ is large, the variance of the estimator will be correspondingly large. Furthermore, (and this is not apparent in the formula) if the distribution of $W(x)$ is badly skewed, (as is the case for a system that makes very few errors), the estimator's distribution will converge more slowly to that of a Gaussian variable.

In practice, the distribution of $W(x)$ is unknown. However, all is not lost if we are willing to use an estimate of this distribution, so that we may find an empirical variance to estimate the variance of our estimator (!) The error bars that are plotted on the results are derived from the empirical variance of $W(x)$ and giving an interval of $\pm 2\frac{\sigma_{\text{emp}}}{\sqrt{nK}} = \pm 2\sigma_{B\overline{E}R}$, which corresponds to a confidence interval of about 95% for the $B\overline{E}R$, assuming that it has a pmf close to a Gaussian distribution.

A simple mechanism was devised to adaptively adjust the number of experiments performed so that $\sigma_w$ could be estimated properly and to keep $\sigma_{B\overline{E}R}$ low. The mechanism basically counts the number of erroneous blocks, but with a subtlety:
1. specify a number $x$ of erroneous blocks to decode in the first round.

2. decode blocks until the number of erroneous blocks is greater than $x$. Let $n$ be the number of decoded blocks after “the first round”.

3. decode $3n$ more blocks (three more rounds)

In this way, the stopping rule is somewhat decoupled from the last error. Stopping the simulation after $4x$ errors would make the BER estimate artificially high because the last block was erroneous. This rule also ensures that the non-zero-error mass in the empirical $W(x)$ is significant.

\footnote{Note that the number of mistakes is counted for the results of the highest iteration performed.}
Appendix C

Full Results

This appendix contains the most important simulation results, in the form of BER vs. SNR curves. They are sorted first by generator polynomial, then in descending rate, descending block size, and increasing $g$. 
23/35 rate 0.33 turbo codes, $K = 16384$ information bits

Figure C.1: 23/35, $K=16384$, $g=1$

Figure C.2: 23/35, $K=16384$, $g=2$

Figure C.3: 23/35, $K=16384$, $g=3$

Figure C.4: 23/35, $K=16384$, $g=4$
23/35 rate 0.33 turbo codes, $K = 16384$ information bits

![Graphs showing bit error rate versus Eb/No dB for different iterations.](image-url)

**Figure C.5:** 23/35, $K=16386$, $g=3$, random sub-block interleaving

**Figure C.6:** 23/35, $K=16386$, $g=3$, random sub-block lin. substitutions
23/35 rate 0.33 turbo codes, K=4096 information bits

Figure C.7: 23/35, K=4096, g=1

Figure C.8: 23/35, K=4096, g=1

Figure C.9: 23/35, K=4096, g=2
23/35 rate 0.33 turbo codes. \( K=1024 \) information bits

Figure C.10: 23/35, \( K=1024 \), \( g=1 \)

Figure C.11: 23/35, \( K=1024 \), \( g=1 \), inter \( \sigma = 4 \), avoids \( d = r, 2r \)

Figure C.12: 23/35, \( K=1024 \), \( g=1 \), inter \( \sigma = 10 \)

Figure C.13: 23/35, \( K=1024 \), \( g=2 \)
23/35 rate 0.33 turbo codes. K=1024 information bits

Figure C.14: 23/35, K=1024, g=1, inter. blocked by 2

Figure C.15: 23/35, K=1024, g=1, inter. blocked by 2

Figure C.16: 23/35, K=1024, g=1, inter. blocked by 1
23/35 rate 0.33 turbo codes, K=256 information bits, g=1,

blocked interleaver

Figure C.17: 23/35, K=256, g=1

Figure C.18: 23/35, K=256, g=1, inter. avoids d = r, 2r

Figure C.19: 23/35, K=256, g=1, inter. σ = 10
23/35 rate 0.33 turbo codes, \( K = 256 \), information bits, \( g = 1 \)

Figure C.21: 23/35, \( K = 256 \), \( g = 1 \), inter. blocked by 2

Figure C.20: 23/35, \( K = 256 \), \( g = 1 \), inter. blocked by 2

Figure C.22: 23/35, \( K = 256 \), \( g = 1 \), inter. blocked by 1

75
23/35 rate 0.33 turbo codes, K=256 information bits, g=2

Figure C.23: 23/35, K=256, g=2

Figure C.24: 23/35, K=256, g=2, inter. avoids d = r

Figure C.25: 23/35, K=256, g=2, inter. has σ = 4, avoids d = r
23/35 rate 0.33 turbo codes, $K=256$ information bits, $g=2$

Figure C.26: 23/35, $K=256$, $g=2$, inter. has $\sigma = 7$

Figure C.27: 23/35, $K=256$, $g=2$, inter. has $\sigma = 7$

Figure C.28: 23/35, $K=256$, $g=2$, inter. avoids $d = r, 2r$
Figure C.38. 23/35, rate 0.686 turbo codes, K=256 information bits, g=-1
Figure C.39. 23/35, K=256, g=1, inter. lines a = -1, avoid d = r, 0 mistakes
Figure C.39. 23/35, K=256, g=1, inter. lines a = -1, avoid d = r, 0 mistakes
Figure C.39. 23/35, K=256, g=1, inter. lines a = -1, avoid d = r, 0 mistakes

23/35 rate 0.686 turbo codes, K=256 information bits, g=-1
23/35 rate 0.33 turbo codes, K=256 information bits, g=4

Figure C.35: 23/35, K=256, g=4

Figure C.36: 23/35, K=256, g=4, const. sub-block lin. subst.

Figure C.37: 23/35, K=256, g=4, inter. $\sigma = 10$
5/7, rate 0.5 turbo codes

Figure C.44: rate 0.33, 5/7, K=512, g=2, iter: σ = 8 with two mistakes

Figure C.45: rate 0.5, 5/7, K=1024, g=1

Figure C.46: rate 0.5, 5/7 K=1024, g=2, symbol puncturing
21/37 coded, rate 0.33 turbo codes

Figure C.47: rate 0.33, 21/37, K=256, g=1

Figure C.48: rate 0.33, 21/37, K=256, g=2
References


