Hyper-codes: High-performance Low-complexity Error-correcting Codes

by

Andrew W. Hunt, B.A.Sc. (Eng.)

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Engineering

Ottawa-Carleton Institute of Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario

May, 1998

0-612-32401-X

The undersigned recommend to the Faculty of Graduate Studies and Research acceptance of the thesis

Hyper-codes: High-performance Low-complexity Error-correcting Codes

submitted by Andrew W. Hunt, B.A.Sc. (Eng)
in partial fulfillment of the requirements
for the degree of Master of Engineering

[illegible]

Dr. S. Crozier
Thesis Supervisor

[illegible]

Dr. D. Falconer
Thesis Supervisor

[illegible]

Dr. R. Goubran
Chair,
Department of Systems and
Computer Engineering

Carleton University
May 1, 1998

Abstract

The existence of a very low complexity soft-in / soft-out decoder for simple parity equations makes code structures composed of such equations attractive because they can be efficiently decoded using an iterative approach. A basic way of creating code structures from simple parity equations is to arrange the equations in N-dimensional "box" configurations. Codes created in this manner offer good, but not exceptional, error-rate performance, for a given block size and code rate. This thesis presents methods of augmenting such basic structures to produce codes with significantly enhanced distance properties. Also, novel enhancements to the decoding process have been developed that both improve performance and reduce implementation complexity. This new family of coding schemes has been given the name "hyper-codes". These codes can provide very good error-correcting performance for a wide range of communications applications, and are especially attractive for applications requiring short block lengths or high code rates.

Acknowledgements

First and foremost. I would like to thank my primary thesis supervisor. Dr. Stewart Crozier, who is a research scientist in the Digital Signal Processing group at the Communications Research Centre (CRC) in Ottawa. He has been the source of many ideas and a constant source of enthusiasm and encouragement. Working with him has made this research project a very rewarding experience both personally and professionally.

In addition. I would like to express my gratitude to my co-supervisor. Dr. David Falconer. for the assistance he has provided. and especially for the flexibility he has shown regarding the manner in which this Master's degree was conducted. I would also like to thank Dr. John Lodge. the project leader of the Digital Signal Processing group at CRC. for his initial suggestions on research topics that would be worth investigating.

Thanks to all those at the Communications Research Centre who graciously allowed me to use their computers for the many simulations that were part of this research undertaking.

Finally. I gratefully acknowledge the financial support provided by NSERC and Carleton University.

Table of Contents

List of Figures

Abbreviations

AWGN	additive white Gaussian noise
BER	bit error rate
BPS	bits per second
DSP	digital signal processor
FEC	forward error-correcting
HC	hyper-code
LLR	log-likelihood ratio
MAP	maximum *a-posteriori* probability
ML	maximum likelihood
SNR	signal-to-noise ratio
QPSK	quadrature phase shift keying
8PSK	8-ary phase shift keying
16QAM	16 quadrature amplitude modulation

Symbols

E_b	energy per information bit
E_c	energy per channel bit
$N_0/2$	two-sided noise power spectral density of an AWGN channel
P	power
B	bandwidth
R_b	bit rate
p	probability
$p(a\|b)$	conditional probability
$f(x)$	probability density function
$f(x\|b)$	conditional probability density function

Chapter 1

Introduction

This document describes what have been termed "hyper-codes". These are block error-correcting codes composed of simple codes arranged in a "multi-dimensional" aggregate structure, which are decoded from soft channel samples using an iterative approach based on MAP (maximum *a posteriori*) principles. These codes can provide very good error-correcting performance for a wide range of communications applications, but are especially attractive for applications requiring a channel utilization level higher than 1 information bit per symbol or channel use. For example, using QPSK modulation over an AWGN channel, a hyper-code of block size 4096 information bits and of rate=0.79 achieves a BER of 10^{-5} at an E_b/N_0 of 3 dB. This is only 2 dB from the Shannon capacity limit, and only 1 dB from the capacity limit for QPSK modulation. Further, the processing and memory requirements associated with decoding these codes are very reasonable.

Readers familiar with MAP decoding and not wishing to read this entire document are referred to sections 3.6 and 6.

1.1 Background

The "hyper-code" FEC coding approach that has been developed builds on previous work by other researchers. The following list outlines some prominent related work.

- Iterative probabilistic decoding of codes comprised of simple parity equations, using log-likelihood ratios (LLRs): Gallager [2]; however, there was no subtraction of "extrinsic" information, nor was the "max" approximation used.

- Max-log-MAP approach as applied to simple parity equations: Battail [5].

- Iterative decoding of multi-dimensional parity structures, using the max-log-MAP approximation as applied to simple parity equations, and including a method of subtracting off the extrinsic information: Lodge [3, 4].
- "Turbo codes", which use parallel, recursive, systematic convolutional encoders, and are decoded using iterative MAP techniques: Berrou *et al* [6]; of particular interest is that these researchers also adjust the extrinsic information in order to improve the BER performance.
- Iterative log-MAP decoding of multi-dimensional parity structures, with subtraction of extrinsic information: Hagenauer [1]; however, there was no parity-on-parity, and though the "max" approximation was discussed, the complexity reduction afforded by using this approximation in conjunction with simple parity equations was not exploited. Also, no scaling or other adjustment of the extrinsic information was performed.

1.2 Capacity theory

Shannon's capacity theorem states that, for an AWGN channel, the maximum reliable (i.e. error-free) transmission rate is given by (from[11]):

$$R_b \leq B \log_2 \left(1 + \frac{P}{N_0 B}\right) bps \tag{1.1}$$

where B represents the channel bandwidth, $N_0/2$ the (two-sided) noise power spectral density, P the transmitted power, and R_b the communication bit rate in bits-per-second (bps). This expression can be re-arranged to give the minimum E_b/N_0 required for reliable communication as a function of R_b/B:

$$\frac{E_b}{N_0} \geq \frac{2^{\frac{R_b}{B}} - 1}{\frac{R_b}{B}} \tag{1.2}$$

The minimum value for E_b/N_0 is obtained when R_b/B (the so-called "bandwidth efficiency") approaches zero. This provides a lower bound for E_b/N_0 below which "reliable" communication is not possible. This is the *Shannon limit* (again, from [11]):

$$\begin{aligned} \frac{E_b}{N_0} &\geq 10 \log_{10} (\log_e 2) \ dB \\ &= -1.6 \ dB \end{aligned} \tag{1.3}$$

If bandwidth efficiency *is* of concern, this limit rises. For a bandwidth efficiency of $R_b/\mathrm{B} =$ 1 bps/Hz, the limit for reliable communication is 0 dB. A bandwidth efficiency of 1 bps/Hz is a relevant figure, because of the prevalence of antipodal signaling and rate=1/2 coding[1]. Note, however, that the 0 dB theoretical limit does not include the antipodal signaling restriction. If this is taken into account, the E_b/N_0 required for reliable communication increases to about 0.18 dB.

It is imperative to realize that all of these channel capacity results assume arbitrarily long block lengths. For finite-length blocks, it is impossible to guarantee error-free communication. (The channel is assumed to be AWGN, with non-zero $N_0/2$, throughout this discussion.) However, for a particular block length and transmitted E_b/N_0 (and bandwidth efficiency and type of signaling), there must exist a minimum probability of block error that no coding scheme can do better than. It is interesting to note, however, that such theoretical results do not seem to be readily available in the coding literature, though some discussion of block error rate for finite-length blocks is given in [12]. Such results would be very useful when evaluating new coding schemes.

Further, it is theoretically possible to devise a finite-length block coding scheme that would provide acceptable performance even *below* the Shannon limit. The key word here is "acceptable", which does not imply error-free. That is, the Shannon limit (and the other limits given earlier) are boundaries that apply to infinite-length blocks, and for finite-length blocks these limits do not exist. Instead, the minimum achievable block error rate simply continues to decrease with increasing E_b/N_0.

Finally, it is important to point out that as the block size increases, it becomes "easier" to develop codes that achieve a certain BER performance at a given E_b/N_0. Thus, when comparing the performance of FEC schemes, it is only valid to compare approaches with similar block lengths (and code rates). In continuous coding schemes (e.g. convolutional codes) the history depth can be used to determine an "effective" block length.

[1]For a channel having a bandwidth of B Hz, the maximum signaling rate is 2B. This is a direct consequence of the Nyquist sampling theorem. With antipodal signaling, each symbol is only one bit. Thus, the maximum channel bit rate is 2B bps. If rate=1/2 coding is used, the information bit rate is exactly half of the channel bit rate. This gives an information bit rate of B bits per second over a bandwidth of B Hz, and hence a bandwidth efficiency of one.

Chapter 2

Fundamental concepts

There are several concepts that form the foundation on which hyper-codes are built. These concepts are introduced in this chapter. Only a brief discussion of each concept will be presented at this point; more detailed descriptions will be provided later.

2.1 Parity equations

The term **parity equation** refers to a set of bits to which a single parity bit has been added, the parity bit being chosen so as to make the overall parity even. Even parity means an even number of "1" bits.

2.2 Basic codes constructed from parity equations

The **minimum distance** (d_{min}) of a code is the fewest number of bits that can be flipped to convert one codeword into another codeword. The minimum distance of a parity equation, in isolation, is 2. This is a small minimum distance. Parity equations can be combined, however, to produce composite structures with much higher minimum distances. A straightforward way of achieving this is to arrange a collection of parity equations into an N-dimensional **hyper-cube** structure. A basic example of a hyper-cube is a two-dimensional parity square. Consider the following codeword, consisting of 9 information bits and 7 parity bits (the parity bits are italicized):

1 0 0 *1*

1 0 1 *0*

1 1 0 *0*

1 1 1 1

The nine information bits have been arranged into a 3x3 square. A parity bit has been added to each of the 3 rows, making every row have even parity. Subsequently, a parity bit has been added to each column, making each column have even parity. The minimum distance of the resulting structure is four. It is important to observe that a parity bit has been added for the column of row-parity bits. This "parity-on-parity" bit could have been omitted, but then the minimum distance of the code would only be three. Note that the final row also forms a valid parity equation. The final row of bits is the (modulo-2) sum of all of the other rows, and each of the other rows has an even number of ones, and therefore the final row must have an even number of ones, making it a valid parity equation.

The example given was for a two-dimensional code. It should be fairly clear, however, to see that this approach of combining parity equations can be generalized to N dimensions. For example, one could take three parity squares like the ones described above, arrange them one behind another, and apply parity across the three planes to produce a fourth plane consisting of 16 more parity bits, the final result being a structure having a minimum distance of 8. The new parity plane is also a valid 2-D codeword, meaning that each of its rows has an even number of ones, and each of its columns has an even number of ones. Every time a new dimension is added, the minimum distance doubles, so that a parity hyper-cube with N dimensions has a minimum distance of 2^N.

It is important to mention that though the term "hyper-cube" has been used throughout this discussion, a more accurate term would perhaps be "hyper-box", since the sides of the structure need not be of equal length. This more rigorous terminology is somewhat awkward, however, and so the term "hyper-cube" is used instead, with it understood that the sides of the structure need not be of equal length, unless explicitly stated.

In subsequent discussions reference will frequently be made to the idea of a **set of parity equations**. for a two-dimensional parity rectangle, there are two *sets* of parity equations: the set of row parity equations and the set of column parity equations. Similarly, for a three-dimensional parity box, there are three sets of parity equations. In general, for an N-dimensional hyper-cube, there are N sets of parity equations, one set associated with each dimension of the hyper-cube. That is, for a hyper-cube parity structure, a *set* of parity equations is analagous to a *dimension* of parity equations. This equivalency no longer holds, however, with more complex parity structures, such as many hyper-codes. For this reason, the term "set of parity equations" is preferred over referring to dimensions.

2.3 Soft-in / soft-out decoder for parity equations

Consider transmitting a set of bits corresponding to a parity equation across a communications channel. Further, consider a receiver that does not output ones and zeros (**hard decisions**), but rather generates a real value for each channel bit (**soft decisions**), the value being a measure of confidence in the bit, with large positive numbers indicating a high degree of confidence that the corresponding bit is a zero, and large negative numbers indicating a high degree of confidence that the bit is a one. (To be precise, the numbers should be **log-likelihood ratios**, or **LLRs** for short. These quantities will be described later in this document.) Given these soft samples from the receiver, it is desired to "improve" these confidence estimates, using the additional information that the bits form a valid parity equation. Of course, such an "improvement" in the estimates would be of little utility if only a single parity equation was transmitted. When parity equations are part of a larger code structure, however, a means of improving the bit estimates becomes very useful since this provides a way to decode the code in an iterative manner.

The question, therefore, is how can the LLRs output by the receiver be improved (i.e. be made more reliable), using the information that the set of bits was transmitted with even parity. A simple yet effective way of achieving this is the following. First, the parity of the set of soft samples is established, based on how many of the LLRs are negative. This result determines whether the estimates will be strengthened or weakened. If there are an odd number of negative samples, then all of the estimates will be reduced in magnitude, and if there is an even number of negative samples, then all of the estimates will be increased in magnitude. The magnitude of the adjustment for a particular bit is determined by the LLR with the smallest magnitude, not including the bit in consideration.

The following example shows the soft-in, soft-out decoding of a 5-bit parity equation. Note that it does not matter which bit is the parity bit.

```
 0.123   1.295  -0.528  -0.789  -3.475      Input LLRs
                ------  ------  ------      Number of negative samples = 3
-0.528  -0.123   0.123   0.123   0.123      Values that are added (extrinsic information)
--------------------------------------
-0.405   1.172  -0.405  -0.666  -3.352      Output LLRs
```

Note that this decoding operation has flipped the bit that was most likely in error (the bit with the weakest LLR). The term **extrinsic information** is used to describe the values that are added to the input LLRs to produce the (hopefully) improved output LLRs.

The method described above is derived from calculating the ***a posteriori* probability (APP)** for each bit, expressing the calculations in terms of LLR quantities, and then approximating the operation of taking the sum of two exponentials by simply taking the maximum of the two exponents. This simple method for decoding parity equations is called the **max-log-APP** method and will be described in more detail later. In the literature this method is commonly referred to as the **maximum *a posteriori* (MAP)** method. This terminology is perhaps somewhat erroneous but because of its widespread usage in the related literature this document will often use the term MAP where APP may in fact be the more precise term. The simplicity of the max-log-APP method for soft-in / soft-out decoding of simple parity equations is one of the primary reasons why building code structures out of simple parity equations is so attractive.

2.4 Iterative decoding example

With a simple soft-in / soft-out decoder for parity equations, codes constructed out of parity equations can be efficiently decoded in an iterative manner. The following example illustrates how the iterative decoding process works. The sample code consists of 20 information bits and 10 parity bits, making it an (n,k)=(30,20) code. The information bits are encoded into a parity rectangle (i.e. a 2-D parity hyper-box), producing a code with a minimum distance $d_{min} = 4$. Note that the iterative decoding process illustrated here is by no means a maximum-likelihood sequence estimator (MLSE) decoder. Iterative decoded works well, however, and makes it possible to decode code structures for which MLSE decoding would be computationally intractable. The term **iteration** refers to one decoding pass through all of the parity equations making up a code structure; the term **cycle** is also sometimes used in the literature to refer to the same idea.

```
ENCODING
--------
Info bits          0      0      0      0      0             The information bits
                   1      0      0      0      1             are arranged into a
                   0      0      1      1      1             4x5 rectangle.
                   1      1      1      1      0

Add row parity     0      0      0      0      0      0      A parity bit is added
                   1      0      0      0      1      0      to each row so that
                   0      0      1      1      1      1      every row has even
                   1      1      1      1      0      0      parity.

Add col parity     0      0      0      0      0      0      A parity bit is added
                   1      0      0      0      1      0      to each column,
                   0      0      1      1      1      1      including the column
                   1      1      1      1      0      0      of row-parity bits.
                   0      1      0      0      0      1

TRANSMISSION
------------
Symbol mapping   1.00   1.00   1.00   1.00   1.00   1.00     0 bits are mapped
                -1.00   1.00   1.00   1.00  -1.00   1.00     to +1.0
                 1.00   1.00  -1.00  -1.00  -1.00  -1.00     1 bits are mapped
                -1.00  -1.00  -1.00  -1.00   1.00   1.00     to -1.0
                 1.00  -1.00   1.00   1.00   1.00  -1.00

With noise       0.57   0.61   0.26   2.49   1.43   2.65     The noise on the
                -0.74   1.86 <-0.03>  1.97  -2.25   1.26     channel has caused 3
               <-0.01>  1.82  -1.76  -2.77  -0.91 < 0.71>    bit errors: 2 info-bit
                -2.36  -0.35  -2.04  -1.76   3.77   0.44     errors and 1 parity-
                 2.84  -1.42   2.10   1.42   3.02  -1.11     bit error.

DECODING
--------
Iter. 1, rows    0.83   0.87   0.83   2.75   1.69   2.91     Performing soft-in
                -0.71   1.83   0.71   1.94  -2.22   1.23     soft-out decoding on
               <-0.72>  1.83  -1.77  -2.78  -0.92 < 0.72>    the row parity eqn's
                -2.71  -0.79  -2.39  -2.11   4.12   0.79     has corrected one of
                 3.95  -2.53   3.21   2.53   4.13  -2.53     the bit errors.

Iter. 1, cols    0.12   1.66   1.54   4.69   2.61   2.19     Observe that one error
               < 0.01>  2.62   1.54   4.05  -3.14   0.51     has been corrected,
               <-0.01>  2.62  -2.48  -4.72  -2.61  -0.07     but a new one has been
                -2.00  -1.66  -3.10  -4.05   5.04   0.07     introduced.
                 3.24  -3.32   3.92   4.47   5.05  -1.81

Iter. 2, rows    0.83   1.26   0.83   4.29   2.21   1.79     The block is now
                -0.56   2.67   0.82   4.10  -3.19   0.56     correct, including the
                 0.78   2.69  -2.55  -4.79  -2.68  -0.78     parity bits.  Decoding
                -1.37  -0.94  -2.47  -3.42   4.41   0.94     can stop because all
                 2.52  -2.60   3.20   3.75   4.33  -2.52     eqn's are satisfied.
```

Whenever a set of parity equations is returned to, it is important that the associated extrinsic information added in the previous iteration is subtracted off before new extrinsic information is calculated and added in. For instance, in the above example, before new extrinsic information can be calculated for the row parity equations in the second iteration, the extrinsic information for the row equations from the first iteration must be subtracted off. The intermediate steps involved in processing the row parity equations for the second time are shown below.

Iter. 1, cols	0.12	1.66	1.54	4.69	2.61	2.19	These are the LLR's
	< 0.01>	2.62	1.54	4.05	-3.14	0.51	after the first
	<-0.01>	2.62	-2.48	-4.72	-2.61	-0.07	decoding iteration.
	-2.00	-1.66	-3.10	-4.05	5.04	0.07	
	3.24	-3.32	3.92	4.47	5.05	-1.81	
Old extrinsic	0.26	0.26	0.57	0.26	0.26	0.26	This is the extrinsic
	0.03	-0.03	0.74	-0.03	0.03	-0.03	information that was
	-0.71	0.01	-0.01	-0.01	-0.01	0.01	calculated in the
	-0.35	-0.44	-0.35	-0.35	0.35	0.35	first iteration for the
	1.11	-1.11	1.11	1.11	1.11	-1.42	row parity eqn's.
With old	-0.14	1.40	0.97	4.43	2.35	1.93	The extrinsic info
extrinsic	-0.02	2.65	0.80	4.08	-3.17	0.54	associated with the
subtracted off.	0.70	2.61	-2.47	-4.71	-2.60	-0.08	row parity eqn's must
	-1.65	-1.22	-2.75	-3.70	4.69	-0.28	be subtracted off.
	2.13	-2.21	2.81	3.36	3.94	-0.39	
New extrinsic	0.97	-0.14	-0.14	-0.14	-0.14	-0.14	This is the new
	-0.54	0.02	0.02	0.02	-0.02	0.02	extrinsic information
	0.08	0.08	-0.08	-0.08	-0.08	-0.70	associated with the
	0.28	0.28	0.28	0.28	-0.28	1.22	row parity eqn's.
	0.39	-0.39	0.39	0.39	0.39	-2.13	
Iter. 2, rows	0.83	1.26	0.83	4.29	2.21	1.79	The new extrinsic
	-0.56	2.67	0.82	4.10	-3.19	0.56	information has been
	0.78	2.69	-2.55	-4.79	-2.68	-0.78	added in.
	-1.37	-0.94	-2.47	-3.42	4.41	0.94	
	2.52	-2.60	3.20	3.75	4.33	-2.52	

It is important to recognize that the need to subtract off the extrinsic information from the previous iteration necessitates storing the extrinsic information associated with each set of parity equations from one iteration to the next.

2.5 What are hyper-codes?

Using the ideas presented thus far, various error-control coding schemes can be devised. Unfortunately, such coding schemes are generally neither very powerful in terms of their error-correcting ability, nor very practical in terms of the memory requirements associated with their decoding. Several new techniques have been developed in this thesis, however, which when combined with these existing ideas can produce coding schemes that have powerful error-correcting capability and are also practical to implement. The most significant of these new developments are the following:

- Methods of constructing parity structures from parity equations that have much better distance properties than simple N-dimensional parity cubes. The most significant development in this area is the idea of "roll parity".

- An enhancement to the approximate soft-in / soft-out decoder for parity equations that significantly improves error-rate performance with iterative decoding. This enhancement is an appropriate scaling of the extrinsic information output by the max-log-MAP decoder.

- Compression techniques that greatly reduce the amount of memory required to store the extrinsic information.

The expanded family of coding schemes that can be created using these new ideas has been given the name "hyper-codes". The origin of this name is associated with the hyper-cube parity structures commonly used as basic building blocks. The subsequent chapters describe these new developments in detail.

Chapter 3

Hyper-code structures

This section describes an approach to error-control coding based on multi-dimensional parity structures. Iterative decoding of these codes using MAP (maximum *a posteriori* probability) techniques is described in the next section.

3.1 Multi-dimensional parity structures

The coding scheme is based on what can be called "simple parity equations". This term is used here to refer to a set of bits to which a single parity bit is added, in order to force the overall parity to be either even or odd. Even parity means that there are an even number of bits that are equal to one. Similarly, odd parity means that there are an odd number of bits that are equal to one. For simplicity and concreteness, the remainder of this document will always assume that the parity equations have even parity. Adopting a convention of odd parity would offer no benefits and would introduce unnecessary complications.

Constructing a code using simple parity equations is very practical because highly efficient iterative techniques can be used in the decoding process. This allows the development of decoders that are both fast and inexpensive. The details of the decoding process are described in the next section.

A simple parity equation is a one-dimensional structure. Higher-dimensional structures, however, can be created by the appropriate arrangement of simple parity equations. For example, if several rows of parity equations (each of equal length) are stacked one on top of each other, and then a parity bit is added to each column of this arrangement, a two-dimensional "plane" parity structure results. Note that the new row of parity bits for the columns is itself a simple parity equation; that is, it will always have even parity.

This process of creating higher-dimensional structures can be continued indefinitely. If several "plane" structures are stacked, and parity bits are added across these planes (creating an additional plane), a three-dimensional box-like structure results. If several box-structures are stacked, a four-dimensional structure is created, and so forth.

An important concept in coding theory is the minimum (Hamming) distance of a code. This is the closest "distance" any two codewords can be from each other, where the "distance" is measured in terms of how many bits must be flipped in one codeword to produce the other codeword. The minimum distance of a code gives, to some extent, an indication of how effective the code will be at correcting errors introduced by the communications channel. A trivial coding scheme consisting of only a simple parity equation has a minimum distance of 2. A two-dimensional "plane" parity structure, as described earlier, has a minimum distance of 4. The three-dimensional structure has a minimum distance of 8. In general, an n-dimensional "box" structure has a minimum distance of 2^n.

The multi-dimensional parity structure described up to this point is not new and has been studied in the scientific literature in some detail.

3.2 Motivation for more advanced code structures

With simple "box-like" code structures, each increase in the minimum distance of the code is accompanied by a significant increase in the block size, given the same code rate. For example, consider the construction of code geometries where the code rate is required to be 0.5 (or higher). A 4x4x4/5x5x5 three-dimensional structure has a minimum distance of 8 with a block length of only 64 information bits. Moving to a four-dimensional structure, a 6x6x6x4/7x7x7x5 code has a rate higher than 1/2 and has a minimum distance of 16, but now the block size has increased to 864 bits, more than 10 times larger than the three-dimensional block. To achieve a minimum distance of 32 with a "box" parity structure, a five-dimensional block would be required, such as a 7x7x7x7x6/8x8x8x8x7 code. This structure has a size of 14,406 information bits. These examples illustrate that although impressive minimum distances are achievable with simple multi-dimensional "box" structures, each increase in the minimum distance is accompanied by a large jump in the block size. This raises the question as to whether it is possible to add extra parity bits to the basic "box" structure and increase the minimum distance, without incurring as stiff a block-size penalty as is associated with adding a whole new dimension.

3.3 Desirable properties for the extra parity

When adding extra parity to an N-dimensional parity hyper-cube, it is desirable for the extra parity to have certain properties. First, the addition of the extra parity should increase the minimum distance of the overall code. The minimum distance of a code is a significant, though by no means the only, determining factor of a code's error-rate performance, and so a higher minimum distance will usually mean that the code will perform better. Second, it would be advantageous for the extra parity bits to themselves have some sort of inherent structure, assuming that they were arranged appropriately. There are two reasons why the second property is advantageous. First, if the extra parity bits have some sort of inherent structure, then the log-likelihood ratios (LLR's) associated with these extra parity bits can be "improved" during the course of the iterative decoding process. Such an improvement of the LLR's aids convergence, which in turn results in better-error performance (for a given number of iterations). Second, if the LLR's associated with the extra parity bits can be improved during the decoding process, then tests can be performed to determine whether a codeword has been reached. The ability to do such tests allows both early termination, which increases average decoding throughput, and basic decoder failure detection for blocks that did not terminate early.

It is desired, therefore, for the extra parity to have these properties:

- The extra parity bits should increase the minimum distance of the code.
- The extra parity bits, if arranged appropriately, should have some sort of inherent code structure, which allows them to be improved during the course of the iterative decoding process.

This section presents guidelines for adding extra parity that makes the parity have the second property; subsequent sections will go on to describe specific methods of adding extra parity that adhere to these guidelines and also result in an increase in the minimum distance.

Consider an N-dimensional parity hyper-cube. Such a structure can be viewed as a collection of N-1 dimensional structures, each of which is itself a valid N-1 dimensional parity hyper-cube. Within each N-1 dimensional parity hyper-cube, any re-arrangement of N-2 dimensional units will not disturb the validity of the parity structure of the N-1 dimensional unit. Thus, if parity is applied across the N-1 dimensional units, and the "mixing-up" of the bits within each N-1 dimensional unit is limited to a re-arrangement of N-2 dimensional units within each N-1 dimensional unit, then the extra parity bits will themselves form a valid N-1 dimensional parity hyper-cube, assuming that they are arranged in a manner consistent with how they were generated. This general approach to adding extra parity that results in the extra parity itself having a code structure has been termed "shuffle parity". Further, it is important to recognize that the shuffled N-1 dimensional units across which parity is applied need not necessarily come from the same N dimensional structure for the extra parity bits to form a valid N-1 dimensional parity hyper-cube. The above description is necessarily quite abstract, and so two examples will now be presented to clarify the concept.

Consider adding 3 extra parity bits to a two-dimensional 2x2/3x3 code, making a 2x2/3x4 code. The numbers are position indices.

```
Before      After shuffling
0  1  2     0  1  2
3  4  5     1  2  0
6  7  8     2  0  1
            a  b  c   <-  Will have even parity.
```

In the above example, the minimum distance of the code before the addition of the extra parity is $d_{min} = 4$. With the particular shuffling pattern used, the addition of the extra parity bits increases the minimum distance of the code to $d_{min} = 6$. A simple proof of this is shown below.

```
Patterns    xx.   x.x   xx.   x.x   .xx   .xx   ...   ...   ...
before      xx.   x.x   ...   ...   .xx   ...   xx.   x.x   .xx
shuffling   ...   ...   xx.   x.x   ...   .xx   xx.   x.x   .xx


After       xx.   x.x   xx.   x.x   .xx   .xx   ...   ...   ...
shuffling   x.x   .xx   ...   ...   xx.   ...   x.x   .xx   xx.
            ...   ...   .xx   xx.   ...   x.x   .xx   xx.   x.x
```

As a larger example, consider a two-dimensional parity square having a size of 5x5 / 6x6, meaning that 25 information bits have been arranged into a 5x5 pattern, and parity bits have been added to each row and each column of this square, including a parity bit for the column of parity bits (which is also the parity bit for the row of parity bits), producing a 6x6 "parity square". For the 2-D case, the "N-1" dimensional units within which the shuffling is restricted to are either the rows or the columns of the 6x6 square. The rows and the columns are entirely equivalent, and so either one can be picked: for this example it will be assumed that the extra parity will be applied across the rows (i.e. down the columns), and so the shuffling will be restricted to be within each row. Further, the "N-2" dimensional units that get shuffled are, for the 2-D case, individual bits, and there is only one dimension along which the shuffling can occur. Thus, if the 6 bits making up each row of the original parity square are shuffled, and then extra parity bits are added by applying parity across these shuffled rows, the resulting row of six parity bits will itself have even parity, and hence be a valid parity equation. Each row in the original parity square has even parity, since a parity bit was added to ensure this. Shuffling of the bits within each row does not affect the number of "1" bits in each row, and hence after shuffling, each row still has an even number of "1" bits. Applying parity across the rows is equivalent to adding up the rows, in a modulo-2 sense. This means that the new row of extra parity bits will necessarily have an even number of ones, and hence be a valid parity equation.

An example of how a 6x6 parity square could be shuffled is shown below. The block on the left is the parity square before shuffling, and the block on the right is the parity square after the rows have been shuffled. The numbers are column indices, starting at 0.

0	1	2	3	4	5		0	1	2	3	4	5
0	1	2	3	4	5		1	2	0	4	5	3
0	1	2	3	4	5		2	0	1	5	3	4
0	1	2	3	4	5		3	5	4	1	0	2
0	1	2	3	4	5		4	3	5	2	1	0
0	1	2	3	4	5		5	4	3	0	2	1

With this particular shuffling pattern, the addition of the extra row of parity bits increases the minimum distance of the code from 4 to 6, and increases the number of rows to 7. What has not been explained here is *how* the particular shuffling pattern was chosen. *Any* shuffling pattern will produce an extra row of parity bits that has even parity; however, only *particular* shuffling patterns will result in an increase in the minimum distance of the code. The sections that follow will describe various ways of selecting shuffling patterns so that the addition of the extra parity results in an increase in the minimum distance of the code. A simple way of selecting a shuffle pattern is to shuffle "diagonally". This approach is conceptually simple, but this method of shuffling is not optimal for all block sizes. A somewhat more complicated but more general approach is the concept of using "rolling" to shuffle. Using "rolling" as a means of shuffling produces good shuffle patterns for a wider range of block sizes than simple diagonal parity, but the selection of an appropriate roll pattern is more involved than using the "diagonal" approach. Note that diagonal shuffling is a special case of roll shuffling. Diagonal and roll parity will be described in more detail in later sections.

It is important to realise that diagonal parity, and more generally, roll parity, are simply specific cases of "shuffle parity". *Shuffle parity is the more general and fundamental concept: by shuffling N-2 dimensional units within N-1 dimensional units of an N dimensional parity structure, and then applying parity across the shuffled N-1 dimensional units, the extra parity so generated will itself have an inherent N-1 dimensional code structure.*

3.4 Diagonal parity

A highly effective and simple way of adding extra parity to N-dimensional "box" parity structures is through the use of what will be termed "diagonal parity". This is a special case of shuffle parity introduced above. The shuffle pattern shown for the first example in the preceding section was in fact a diagonal shuffling pattern. With diagonal parity, parity equations are created that move across all dimensions at once, with modulo wrap-around occuring whenever a side of the n-dimensional "box" is encountered. This concept is perhaps best explained using pseudo-code. The following pseudo-code segment illustrates how diagonal parity could be applied to a three-dimensional parity structure.

```
ArrayOfBits[Len2+1][Len1][Len0]  // +1 is for diagonal parity bits

for StartInx0 = 0 to Len0-1
   for StartInx1 = 0 to Len1-1
      Inx0 = StartInx0
      Inx1 = StartInx1
      Inx2 = 0
      Parity = 0
      while(Inx2 < Len2)
         Parity = Parity XOR ArrayOfBits[Inx2][Inx1][Inx0]
         Inx0 = (Inx0 + 1) MODULO Len0
         Inx1 = (Inx1 + 1) MODULO Len1
         Inx2 = Inx2 + 1
      end while
      ArrayOfBits[Inx2][Inx1][Inx0] = Parity
   end for
end for
```

This code segment illustrates that in moving along a diagonal parity equation, the index of each dimension is incremented by one, with modulo wrap-around occuring when an index reaches the end of a dimension. Since diagonal parity can be applied to parity structures of arbitrary dimension, the term "hyper-diagonal parity" is sometimes used to emphasize that the technique is not restricted to two or three dimensional parity structures.

The application of diagonal parity can also be expressed in mathematical notation as follows. A is a three-dimensional array with dimension lengths I, J and K+1, bit addition is modulo-2, and array index arithmetic is modulo the length of each respective dimension.

$$A_{i,j,K} = \sum_{k=0}^{K-1} A_{i+k,j+k,k} \quad \forall \quad i \in [0, I-1], \;\; j \in [0, J-1] \tag{3.1}$$

With diagonal parity, the equations are applied, in a certain sense, along a particular dimension of the basic box structure. This dimension is termed the "primary diagonal dimension". In the example above, the highest dimension (i.e. the dimension corresponding to **Inx2**) is the primary dimension. The total number of diagonal parity equations (and hence the number of diagonal parity bits) is equal to the product of the lengths of all of the dimensions, except for the primary diagonal dimension. This means that if the original "box" parity structure does not have sides of equal length, choosing a different primary dimension can result in a different number of diagonal parity bits.

Once the "primary diagonal dimension" has been established, there is still a choice as to the direction or orientation of the diagonals. With an n-dimensional parity structure, there are 2^{n-1} different possible directions for the diagonal parity. However, since a box structure is symmetric along every dimension, the different diagonal directions are equivalent. The direction *does* become significant, however, if diagonal parity is applied more than once.

3.4.1 Preferred geometry for diagonal parity

While in concept diagonal parity can be applied to any shape of n-dimensional parity structure, it has been found that applying a certain constraint to the overall shape makes the diagonal parity more effective. This constraint is that every dimension of the parity structure be equal in length, and that this length be odd. That is, n-dimensional "cube" structures with odd-length sides work well with diagonal parity. Note that having odd-length dimensions for the diagonal parity means that the number of information bits per dimension is even. This is an alternate way to express the constraint. Note that shortening can be used, however, to adjust the block size while still satisfying this constraint. This is discussed in greater detail in section 3.5.

This constraint limits the selection of block sizes that can be created using diagonal parity.

The reason why odd-length sides are preferable relates to the modulo wrap-around of the diagonal parity equations. An understanding of this behaviour can be gained by analysing the simple two-dimensional case. Consider applying diagonal parity to a 2-D parity structure (i.e. a parity square). It is desired to show that if the sides of the square are odd in length, then the addition of diagonal parity will increase the minimum distance from 4 to 6. For the sake of concreteness, it will be assumed that the diagonal parity equations all begin in the first row, and that they extend down and to the right. There is no loss in generality in making this assumption. It is helpful to keep in mind that no diagonal parity equation will ever return to a row through which it has already passed, and likewise for the columns. The minimum-distance error patterns for 2-D parity structures are rectangles and the minimum distance is 4. Consider the diagonal parity equation passing through the upper-left error location. If the error pattern *is not* a square, this diagonal equation will miss the lower-right error location. This is the only other location it could hit, since the other two locations either share a row or a column with the original location. Similarly, the diagonal parity equation passing through the lower-right error location will miss the upper-left location if the pattern is not a square. Thus, if the error pattern is a rectangle but not a square, the addition of diagonal parity will increase the weight of the error pattern to at least 6. On the other hand, if the original error pattern *is* a square, both the upper-left and lower-right error locations will land in the same diagonal parity equation, and so these locations do not contribute any increase to the overall distance. In this case, the interaction of the other two error locations with the diagonal parity equations must be examined to determine whether they will result in an increased distance. Consider the diagonal parity equation passing through the upper-right error location. The only way it could possibly encounter the lower-left location is after wrap-around (since the diagonals go down and to the right). For this diagonal to pass through the lower-left location, the following relation must be true:

$$x + \Delta x + \Delta y - LENGTH = x \tag{3.2}$$

where x represents the column number of the left side of the error rectangle. Δx and Δy are the lengths of the sides of the error rectangle, and $LENGTH$ represents the length of the sides of the original parity square to which the diagonal parity was applied. Since the error pattern is a square $\Delta x = \Delta y$, and so this relation can be re-written as:

$$2\Delta x = LENGTH \tag{3.3}$$

From this equation, it is apparent that if $LENGTH$ is odd, the relationship cannot hold true, and hence the minimum distance of a "2-D plus diagonal" structure is indeed 6. This makes it clear why the odd-length specification is important: it ensures that if two (minimum-distance) error locations end up in the same diagonal parity equation, then the other two error locations will fall in *distinct* diagonal equations.

The preceding argument can be extended to higher-dimensional parity structures to show that if a minimum-distance event occurs within the original structure, then the addition of diagonal parity almost doubles the distance (only two error-locations can be paired in the diagonals). This result, however, is not sufficient to allow any conclusions to be made about the minimum distance of the overall code. This is because it is also necessary to consider error events within the original parity structure other than the minimum distance events. In the 2-D case, it was only desired to show that the addition of diagonal parity increased the minimum distance from 4 to 6, and so it was not necessary to consider any error patterns within the original square other than distance-4 patterns. There can be no distance-5 error patterns within a parity square since each row must have even parity. In contrast, for the 3-D case, if it is desired to show that the addition of diagonal parity increases the minimum distance from 8 to 14, then distance-12 error events within the original cube must be analysed in addition to the distance-8 events. Note that distance 9, 10, 11, and 13 events cannot happen in the original parity cube. Distance-10 cannot occur because this would mean that there would be 5 error locations in a plane. The need to consider other error patterns within the basic parity structure makes the analysis of higher-dimensional structures much more difficult than the 2-D case. Distance-12 error patterns have been found, however, for the 2x2x2/3x3x4 cube-plus-diagonal code. For example, the following distance-12 error pattern within a 3x3x3 parity cube will not go up in weight with the addition of diagonal parity (applied either upper-left to lower-right or upper-right to lower-left, moving through the planes left to right):

```
x . x   . . .   x . x
x . x   x . x   . . .
. . .   x . x   x . x
```

Another example is the following error pattern:

```
. x x   . x x   . . .
x . x   x . x   . . .
x x .   x x .   . . .
```

It is believed, however, that the existence of these distance-12 error patterns relates to the fact that, in this example, both the number of rows and the number of columns is divisible by 3. An exhaustive computer search was performed on a 4x4x4/5x5x6 cube-plus-diagonal code structure and the minimum distance of this structure is 14, not 12. This leads to the conjecture that if the lengths of the sides of the cube are not divisible by 3 (nor 2 as discussed earlier), then the minimum distance of the cube-plus- diagonal code will be 14. Further, with the appropriate *roll interleaving* (to be discussed in section 3.6), it may be possible to achieve $d_{min} = 14$ even when the lengths of the sides of the cube *are* divisible by 3 by ensuring that no roll-difference has two roll-counts equal to plus (or minus) 1/3 of the length of the corresponding side.

For the 4-D case, an exhaustive computer search of a 2x2x2x2/3x3x3x4 4-D plus diagonal code structure found a minimum distance of 30. Also, when simulating larger 4-D block codes, if the decoder converged to a codeword which was not the correct codeword, the distance was calculated between the correct codeword and the decoded codeword. In all of these cases, no distance less than 30 was ever observed.

The analysis presented in this section all reduces to the following simple conclusion about applying diagonal parity: *"Use odd-length hyper-cubes."* That is, the preferred geometry for diagonal parity is an n-dimensional structure whose sides are all of equal length (i.e. a hyper-cube), where this length is odd. Note that an odd overall length means that the number of information bits per dimension is even. It will be shown later (in section 3.5) that these regular-shaped structures can be "shortened", so that codes can be constructed having any arbitrary block length.

3.4.2 Properties of diagonal parity

The diagonal parity bits, if arranged appropriately, form an $n-1$ dimensional parity "box". For example, if the basic parity structure is a cube, then the diagonal parity bits will form a parity square. That is, all of the rows will have even parity, and all of the columns will have even parity. This means that each diagonal parity bit is involved in not just one but n different parity equations. This property can be exploited in a MAP decoder, allowing the diagonal parity LLRs to be "improved" during the iterative decoding process. Being able to "improve" the extra parity bits helps the decoder both to converge to the correct (i.e. closest) codeword, and to converge faster. Also, if the extra parity bits could not be improved, the implementation of a simple convergence test (see section 4.7) would no longer be as straight-forward or effective.

In order to exhibit this property, the diagonal parity bits simply have to be arranged in a manner consistent with the diagonal parity equations themselves. There is more than one placement that will give rise to this property; one arrangement that will work is to place the parity bits as though each diagonal "continued" one bit further.

To understand why this property arises requires analysing parity structures in greater detail. Consider a single parity equation (i.e. a "row" of bits having even parity). Any cyclic shift of this row will yield another valid parity row. In fact, regardless of how the bits in the row are "mixed-up", the result will still be a valid parity row. Further, if two parity rows are "added" modulo-2, the result will also be a valid row, since 1's are only eliminated in pairs. Similarly, consider a parity plane, which is a rectangle of bits with each row and each column having even parity. Any re-arrangment of the rows and/or columns will produce another valid parity plane, and any modulo-2 sum of parity planes will produce another parity plane. These properties naturally extend to higher dimensions.

Now consider the specific case of applying diagonal parity to a three-dimensional (i.e. cube) parity structure. The generation of the parity bits is equivalent to adding (modulo-2) specific re-arrangments of the parity planes. The first plane is left unchanged. In the second parity plane, the rows are effectively "rolled" one row up, and the columns are "rolled" one column to the left (this assumes a particular direction of the diagonal). In the third parity plane, the rows are rolled two rows up, and the columns are rolled two columns to the left, and so forth for the rest of the planes. The re-arrangement of each plane maintains the parity structure: each row still has even parity, and each column still has even parity. Further, adding up valid parity planes produces another valid parity plane. Thus, the diagonal parity bits naturally form a valid parity plane.

The ability to "improve" the extra parity bits is a key advantage associated with using diagonal parity as compared with using a random interleaver followed by a face of parity (this approach is described later). With a random interleaver, the extra parity bits do not inherit as much parity structure, and hence each bit is involved in fewer parity equations. This makes it more difficult to improve the estimates associated with these bits during the decoding process.

3.5 Shortening the block

Up to this point, diagonal parity has only been discussed in the context of N-dimensional blocks with equal (and odd) length sides. With this restriction, the choice of block size is very limited. For example, consider applying diagonal parity to 3-D parity cubes. The cube dimensions that can be used are 2x2x2/3x3x3, 4x4x4/5x5x5, 6x6x6/7x7x7, 8x8x8/9x9x9, and so forth. The corresponding block lengths are 8, 64, 216 and 512 information bits, and the corresponding rates including the diagonal parity bits are 0.22, 0.43, 0.55 and 0.63. This shows that the choice of available block sizes and rates is limited, with large gaps between one size and the next larger size.

There is an easy way, however, to "fill in" these gaps. Starting with a basic "regular" structure (i.e. an n-dimensional parity structure with equal and odd length sides), the block can be "shortened" by forcing any number of information bits to zero. Of course, any bits that are always zero are not transmitted, and in the decoding operation, these bits are simply skipped (since zero bits do not affect the parity). Moreover, if the choice of forced information bits is such that certain *parity* bits are always zero, then these bits can also be dropped, and ignored in the decoding process.

This block "shortening" is best understood by means of an example. Consider, for instance, an 8x8/9x9 parity square to which diagonal parity is added. If 8 information bits are always forced to zero, the block size is reduced from 64 to 56 information bits. Also, if these bits are chosen so that they all come from a single row, then the parity bit associated with this row will always be zero, and so it does not need to be transmitted (which helps the code rate). The overall result is an 8x7/9x9 code, which has a block size of 56 information bits and a code rate of 0.69. The block size notation used here indicates the dimensions of the code structure *with* diagonal parity included. This notation is convenient for defining the code structure, and will be used in later discussions as well.

The example above illustrates one means of "shortening" a block: namely, reducing the length of the primary diagonal dimension of the structure. This approach has the advantage of eliminating all of the parity bits associated with the rows (or planes, etc. depending on the dimension of the structure) that were eliminated. This makes the code rate decrease to a lesser extent than if only information bits had been eliminated.

As alluded to earlier, the penalty associated with "shortening" a regular (i.e. equal-length sides) structure is a lowering of the code rate. For this reason, when selecting a code geometry for a particular application, the degree of shortening should be kept as low as possible. As an example, consider choosing between two "3-D + diagonal" structures: a 6x6x6/7x7x8 cube-plus-diagonal code, and a 8x8x4/9x9x6 shortened cube-plus-diagonal code. The first code has a block size of 216 information bits and a code rate of 0.55. With the second code, though the block size has increased to 256 bits, the code rate has in fact dropped to 0.53. If an application could use either code, the 6x6x6 code would be the preferred choice.

The "shortening" of code blocks by forcing information bits to be zero is of course not limited to code structures using diagonal parity. It has been described in relation to such structures because of the restrictions on the dimension lengths required to make the diagonal parity work most effectively.

3.6 Roll interleaving

Diagonal parity is a highly effective way of increasing the minimum distance of basic parity "boxes". Diagonal parity has the added advantage that the extra parity bits form an n-1 dimensional sub-codeword, and this property can be used to aid decoder convergence. The limitation of diagonal parity, however, is that for it to be most effective, the size of the basic parity box has to satisfy a certain criterion: namely, the total length (with parity) of each side must be the same (except for shortening) and odd (see section 3.4.1). This restriction introduces large gaps in the block sizes and code rates that can be created using diagonal parity. The diagonal parity concept can be generalized, however, in a manner that eliminates this size constraint, while still achieving essentially the full performance improvement offered by diagonal parity. This generalization is called "roll" interleaving, and is depicted in Figure 3.1.

3.6.1 Explanation using a simple example

The concept of "roll" interleaving can best be explained using an example. Consider a parity cube as the basic parity structure, to which it is desired to add extra parity bits, with the objective being to increase the minimum distance of the code. For the sake of concreteness, consider a 3x3x3/4x4x4 parity cube. Observe that the total length of each side of this cube is even, making it a poor candidate for diagonal parity. A good starting point in creating the extra set of parity equations is to specify that, for each parity equation, every bit must have a distinct column index, a distinct row index, and a distinct depth index. This requirement ensures that no two parity equations making up the overall code overlap by more than 1 bit. Further, let the equations be as long as possible, in order that the reduction in code rate be as small as possible. This means that the new parity equations will all have length 4, since there are only 4 different depth indices, row indices, and column indices to choose from. Finally, if all bits are to be treated in the same manner, the extra set of parity equations must cover all of the bits in the regular cube, and this means that there will be a total of 16 new parity equations in the extra set being designed.

Since the order of bits within a parity equation is of no consequence, each parity equation can be re-ordered so that the first bit comes from the first depth plane, the second bit from the second depth plane, and so forth (each parity equation includes one and only one bit from each depth plane). This means that generating the extra parity bits can be thought of as interleaving within each depth plane of the original parity cube, and then applying parity across these new planes. Further, the order of the parity equations themselves is irrelevant, which means that they can be arranged in the order of the bits in the first depth

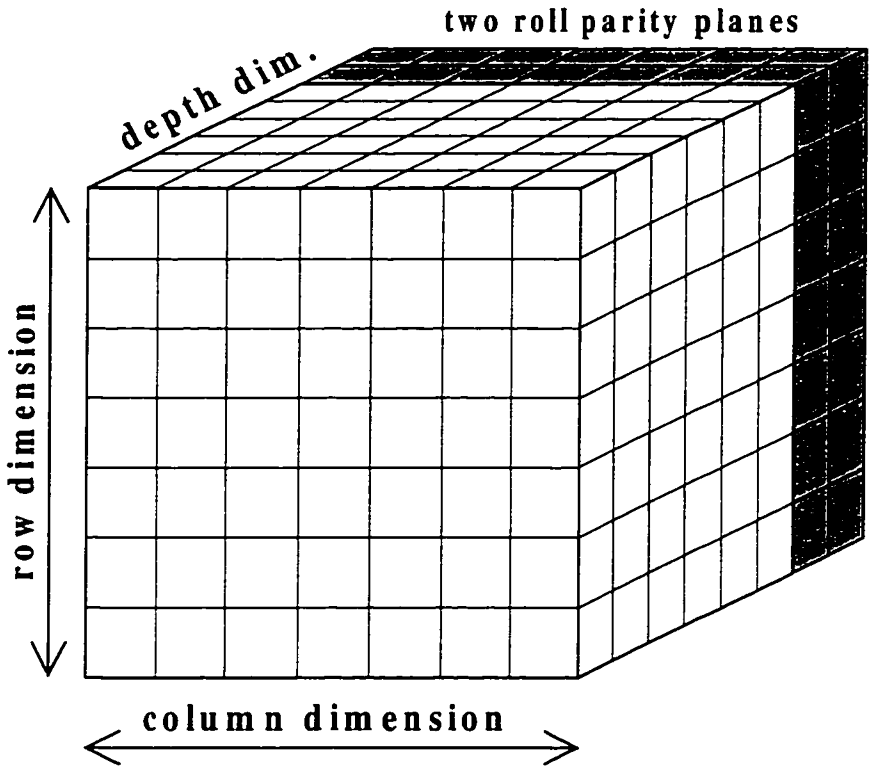

Figure 3.1: A 3D parity box to which two layers of roll parity have been added, resulting in a 6x6x5/7x7x8 code structure.

plane. With this arrangement, the first plane remains unaltered, and only planes 2,3, and 4 need interleaving. This makes sense since it is only the *relative* positioning of the bits that is of consequence.

Now, it is required to determine a way of interleaving each plane (other than the first plane) in a manner that will assure that when parity is applied across the planes, each new parity equation created in this manner will have each bit coming from a different row and different column of the original parity cube. Having only one bit come from each depth plane has already been taken care of by interleaving only within each plane and applying parity across the new interleaved planes. A way this can be achieved is by "rolling" the rows and columns of each depth plane by a different amount. "Rolling" simply means shifting with modulo wrap-around. For example, the second plane could be rolled up by one row and rolled left by one column, the third plane rolled up by two rows and rolled left by two columns, and the fourth plane rolled up by three rows and rolled left by three columns. When the parity is applied across these new planes, each equation so created will have one and only one bit coming from each row and column of the original parity cube. Further, the new plane of parity bits so created will have even parity along each row, and even parity along each column; that is, it is a valid two-dimensional sub-codeword. This results because the rolling of the planes does not disrupt the parity properties of the plane: each row still retains even parity, and each column still retains even parity, and the modulo-2 addition (xor-ing) of valid parity planes produces another valid parity plane. Recall that producing parity bits that themselves have a regular parity structure helps the convergence of the decoder, which in turn improves error-rate performance.

Next, let us examine the different roll patterns that are available for this simple example. As stated earlier, the parity equations and the bits making up these equations can always be re-arranged in such a way so that the first plane can always be left unaltered (i.e. roll the rows by zero and roll the columns by zero). This leaves only planes 2, 3, and 4 to consider, and the available roll counts for both the rows and columns are 1, 2, and 3. Note that a roll of 4 is the same as a roll of 0 since the sides of the cube have length 4. Since all planes are equivalent, the roll pattern for one of the dimensions can simply be assigned in ascending order. Choosing rows, this means that the second plane is rolled up by one row, the third plane is rolled up by two rows, and the fourth plane is rolled up by three rows. This leaves only the column rolling to be assigned. The available choices are as follows, represented as (row-roll, col-roll):

```
a:  (0,0) (1,1) (2,2) (3,3)
b:  (0,0) (1,1) (2,3) (3,2)
c:  (0,0) (1,2) (2,1) (3,3)
d:  (0,0) (1,2) (2,3) (3,1)
e:  (0,0) (1,3) (2,1) (3,2)
f:  (0,0) (1,3) (2,2) (3,1)
```

Not all of these are distinct choices. The roles of the rows and columns in the original parity cube can be swapped and the cube is still the same: this makes cases *d* and *e* equivalent. Further, if the columns are rolled right instead of left, the end result is still the same: this makes cases *a* and *f* equivalent, *b* and *e* equivalent, and *c* and *d* equivalent. Combining all of this leaves only two cases remaining for this simple example:

```
a:  (0,0) (1,1) (2,2) (3,3)
b:  (0,0) (1,1) (2,3) (3,2)
```

At this point, a critical observation can be made. Case *a* is identical to applying diagonal parity. That is, *diagonal parity is simply a special case of roll interleaving.* Moreover, when considered from this viewpoint, it becomes clear why the performance of diagonal parity degrades when the total lengths of the sides of the basic box structure are the same and even. The diagonal parity case includes a (2,2) roll pattern for one of the planes; that is, the rows are rolled by half the total number of rows, and the columns are rolled by half the total number of columns. What this means is that a square error pattern within a plane can land exactly on top of itself after rolling. This in turn means that a weight-8 error pattern can exist in the original cube which will not generate any extra parity even after roll interleaving and adding the extra parity plane. Thus, case *a* has a minimum distance no greater than that of the original parity cube; that is, $d_{min} = 8$.

This leaves case *b* as the only remaining roll assignment to be considered. In this case, however, there are no two planes that have a "roll-difference" of (2,2). Note that the roll-difference between each pair of planes must be considered, and not just the roll-difference between the first plane and the other planes. Since case *b* has no (2,2) roll-difference between any two planes, there is no possibility of a weight-4 error pattern in a plane landing on top of itself after rolling. This in turn means that when the basic parity cube has a minimum-distance error pattern, at most two error locations from one plane will line up with two error locations from another plane. Thus, the extra parity bits added after rolling will increase the weight of the error pattern by four, to a total of twelve. Finally, the lowest-weight error patterns in the original parity cube, after the weight-8 patterns, are weight-12 patterns. There can be no weight-10 patterns, as pointed out in section 3.4.1. In conclusion, the (0,0) (1,1) (2,3) (3,2) roll pattern for the 3x3x3/4x4x4 parity cube results in an overall code having a minimum distance of 12, even though the lengths of the sides of the cube are even. This has also been confirmed using an exhaustive computer search of all codewords for this case. Note that the asymptotic coding gain for this example is $10 * \log_{10}(12 * 27/80) = 6.07$ dB. This is fairly impressive for such a simple code with such a short block size.

3.6.2 Generalization to larger cubes

The analysis given above shows that, for parity cubes, a "roll" interleaver having no roll-difference equal to (n/2, n/2) will produce a code structure having a minimum distance of 12 ('n' refers to the total length of the sides of the cube). A way of assigning roll values for arbitrary-size cubes such that this requirement is always met is as follows: Begin with roll assignments of (0,0) (1,1) (2,2) ... etc. up to but not including a roll assignment of (n/2,n/2). At this plane, increase the column roll count by one, giving (n/2, n/2+1). Continue by increasing both the row and the column counts by one until the last plane is reached. The last plane uses the column count that was skipped, resulting in a roll assignment of (n-1, n/2). Another way of stating the above approach is that the last $n/2$ roll numbers for the columns are themselves rolled, and we have rolled them by 1 position. Observe that using this approach would give roll-assignment *b* in the example described earlier.

How is it known that no roll-difference in such an assignment scheme will equal exactly (n/2,n/2)? The first set of planes have roll assignments of (0,0) (1,1) (2,2) ... (n/2-1, n/2-1). Certainly, no pair of planes chosen from within this first set can have a roll-difference of (n/2,n/2), since the largest possible roll-difference is (n/2-1, n/2-1). Similary, from the second set of roll assignments, (n/2, n/2+1) (n/2+1, n/2+2) ... (n-1, n/2), no pair of planes can be chosen that gives a roll-difference of (n/2,n/2). Finally, a roll-difference of (n/2,n/2) cannot occur between the two sets because of the offset roll in the second set.

3.6.3 Generalization to boxes with sides of different lengths

The above approach also generalizes to boxes with sides having different lengths. The only restriction that is required is that the depth dimension be no longer than either the row or column dimensions. With this restriction, there are enough different roll possibilities for both the row and column dimensions that each depth plane can have a distinct row roll count and a distinct column roll count. Again, the row and column roll counts should be assigned so that no depth plane has both a row roll count equal to half the number of rows, and at the same time, a column roll count equal to half the number of columns, in order to achieve a minimum distance of 12. It is interesting to note that if the number of rows or the number of columns is odd, then standard diagonal parity will automatically satisfy this requirement.

Being able to adjust not only the length of the depth dimension, but the length of the other dimensions as well, further expands the range of code block sizes and rates that can be created using roll interleaving. Of course, if the desired block size can be created using a box having equal-length sides (i.e. a cube), then this is preferable to a box having sides of differing lengths, since the rate will be higher.

3.6.4 Four or more dimensions

All of the discussion up to this point has only considered roll interleaving as applied to three-dimensional parity structures. The concept easily extends to higher dimensional parity structures, and will be analysed here for the case of four-dimensional parity hyper-cubes. To simplify this description, only four-dimensional parity structures with sides of equal length are considered in the following discussion, but the points mentioned in the previous section relating to unequal-length sides apply also to parity structures of four and more dimensions.

As in the 3-D case, the interleaving is performed only within the parity structures associated with the highest dimension. Specifically, this means that the interleaving is restricted to within each cube. The first cube is left unaltered, and only the other cubes are interleaved. The roll patterns for the depth planes within each cube can be assigned in ascending order, just as the row roll counts were assigned in the 3-D case. This leaves the row roll counts and the column roll counts to be assigned.

As in the 3-D case, the roll-differences between each pair of cubes must be considered, and it is roll counts equal to half the length of the associated dimension that are of concern. If no roll-difference has all three roll counts exactly equal to half of the length of each corresponding dimension, then the minimum distance of the 4-D hyper-cube plus extra parity is at least 24. This requires that any given roll difference have at most two roll counts equal to half of the lengths of the associated dimensions. If there *is* a roll-difference that has *two* "half" roll-counts, then the minimum distance of the composite code structure will be 24. This results because if the original hyper-cube has a minimum-distance error pattern (weight-16), after roll interleaving, four error locations from one error "box" can end up lining up with four error locations from the other error "box". Performance can perhaps be further improved by imposing an even stronger restriction: namely, that any given roll-difference have at most *one* roll-count equal to half of the length of the associated dimension. While it has not been proven that this stronger restriction increases the minimum distance above 24, at the very least the number of codewords at this distance is reduced. What can be stated, however, is that the minimum distance of the overall code is certainly no greater than 28, even with this stronger restriction on the roll assignment, if any roll-difference has a roll count of half the dimension length.

It is interesting to note that if none of the roll differences have a roll count equal to exactly half of the length of the associated dimension, then minimum distance error patterns within the original 4-D hyper-cube increase in weight to at least 30 after roll interleaving and adding the extra parity. This is the case when diagonal parity is used with a four-dimensional hyper-cube with sides of odd length. Of course, error patterns other than minimum-distance events within the original 4-D hyper-cube would have to be considered before one could conclude that such composite parity structures do indeed have a minimum distance of 30. As mentioned in section 3.4.1, however, a computer search of a 2x2x2x2/3x3x3x4 4-D hyper-cube-plus-diagonal code determined that the minimum distance for this simple case is indeed 30. This suggests that hyper-cubes with odd-length sides to which diagonal parity is applied may have a performance advantage over hyper-cubes with even-length sides, even with an appropriate roll interleaver.

As an example, the following is a good roll assignment to use with a 4-D hyper-cube of size 3x3x3x3/4x4x4x4: `(0,0,0) (1,1,2) (2,3,1) (3,2,3)`

In conclusion, when selecting a roll assigment for 4-D hyper-cubes, the values should be selected so that no roll difference has more than one dimension with a roll count equal to exactly half of the length of the dimension. With such a roll assignment, the minimum distance of the overall code is at least 24, and possibly as high as 28. It could also be as high as 30 if all sides are odd. It might be possible to impose even more stringent restrictions on the selection of roll counts that would further improve performance, but none are known of at this time.

3.6.5 Two dimensions

Roll interleaving offers no advantage over standard diagonal parity for 2-D parity squares, even when the lengths of the sides are even. This is because if the sides of the square are even in length, it is impossible to avoid a roll-difference of half of the length of the side, regardless of the roll assignment chosen. It is a roll-difference of half of the length of a side that allows a square error pattern to become another square error pattern after roll-interleaving, keeping the minimum distance at only 4. For parity squares with sides having odd length, diagonal parity achieves $d_{min} = 6$, and this is the most that can be achieved with one extra row of parity bits, *regardless* of the interleaver chosen.

3.7 Random interleaving

Another way of adding extra parity bits (and effective distance) to an existing code structure is by using random interleaving. The concept involves starting with one code structure (the first *layer*), randomly interleaving these bits, and then arranging them into some form onto which more parity is applied, creating a second code structure (the second *layer*). That is, the channel bits from the first code serve the purpose of information bits for the second code. Since the interleaving is random, the minimum distance of the aggregate code cannot be established, but the idea is that the "effective distance" of the code approaches the product of the minimum distances of the first-layer and second-layer codes.

In this discussion, the terms "first-layer code" and "second-layer" code are being used, rather than using "inner code" and "outer code". In conventional usage, the term "inner code" refers to the code closest to the channel, and "outer code" refers to the code furthest from the channel [9]. In the code structures being described here, however, the second-layer code is geometrically "outside" of the first-layer code, and so there is potential for confusion if "inner/outer" terminology was used. Moreover, the "layer" terminology generalizes easily to more than two layers. Also, the order of decoding in the receiver is arbitrary and iterative, making the usual inner/outer terminology inappropriate.

3.7.1 Configurations of random interleaving

There are many possible variations when using random interleaving.

- Not all of the bits of the first-layer code need necessarily be involved in the second-layer code. For example, an interleaving scheme might only randomize the information bits from the first code (i.e. the parity bits from the first layer are not included in the second layer). This is often called "parallel concatenated coding". When all of the bits from the first layer *are* used in the second layer, the technique is often referred to as "serial concatenated coding".

- The second layer of coding need not resemble the first layer of coding.

- Parity structures in the first and/or second layers can be repeated. When only a single layer of coding is being used, repetition serves no purpose, but with random interleaving, repetition can become quite useful. To be completely general, the first and/or second layers could even consist of a mix of different parity structures. Repetition is useful because random interleaving tends to work better as the number of bits being interleaved increases. Using repetition allows the interleaving to involve a large number of bits even if the parity structures being used are small.

- The set of parity bits applied before or after interleaving need not be "complete". A situation could arise where, for code rate reasons, it is convenient to omit one or more of the parity bits that would normally be associated with the first or second layer coding structures. Deleting parity bits is called *puncturing.*

- The interleaving can be completely random, or constrained in some manner. For example, consider a 2-D first-layer code, and a second-layer code structure consisting of repeated 1-D codes (i.e. a collection of simple parity equations). In such a case, it might be advantageous to restrict the randomization of the first code to within each row, and then orient the equations of the second-layer code along the columns. The rationale behind such an approach is that error locations must occur in pairs in the rows, and hence by constraining the interleaving to within each row, any two error locations within a row cannot both end up in a single column of the randomized structure.

- Random interleaving can be used more than once in constructing a code geometry (i.e. the code can have more than two "layers").

3.7.2 Analysis of a simple code

It is instructive to examine in greater detail a simple code structure that employs random interleaving. Consider a 5x5/6x6 "plane" parity structure as a first-layer code, which is then randomly interleaved and formed into a 6x6/7x7 plane structure. The overall code is a 5x5/7x7 code. A simple argument shows that the minimum distance of such a structure cannot be greater than 10, regardless of the interleaver chosen. Pick any two "info" bits from any parity equation of the second layer. Map these two bits back to the first-layer code (i.e. de-interleave the two bits). A minimum-distance (i.e. weight=4) error pattern can always be created that includes two specific locations. The other two locations making up the error pattern in the first layer could end up anywhere in the second layer (other than on top of the original two bits selected). At best, these two locations would end up in distinct rows, in columns different than those where the first two bits ended up. Thus, the minimum distance of the aggregate code cannot be greater than 10.

In this example, there will always be an even number of ones in the "information bits" for the second-layer code (since the bits from the first-layer code will always have even parity). This means that the corner parity bit in the second code will always be a zero. Since the value of this bit is independent of the message sent, there is no need to transmit it (thus saving the associated overhead). However, because the second-layer corner parity bit is redundant, the second-layer code is in some sense multiplying the first-layer code distances by a factor of 3, and not 4. Note that the row and column parity of the second-layer code will both be even. This can be used in the decoding process to improve these parity bits, even without the corner parity bit.

The above arguments do not hold if multiple codes are interleaved together. Intuitively, one can see why performance should improve in such cases.

3.7.3 Observations

Some observations can be made about the use of random interleaving.

The larger the block size, the more effective random interleaving becomes. This property arises because as the block size becomes larger, there is less chance of two errors from the first-layer code ending up in a single parity equation of the second-layer code. As an example, consider the 5x5/7x7 2-D 2-layer code discussed earlier. Given any minimum-distance error pattern in the first-layer code, the probability that each of the four error locations will end up in distinct rows in the second-layer code is:

$$\frac{36 * 30 * 24 * 18}{36 * 35 * 34 * 33} = 0.33 \tag{3.4}$$

which is quite low. By comparison, if the first-layer code was repeated say 4 times, then the four error locations would be distributed amongst 24 different rows, and so the probability that they all would end up in distinct rows is much greater:

$$\frac{144 * 138 * 132 * 124}{144 * 143 * 142 * 141} = 0.80 \tag{3.5}$$

There are other factors involved here: multiple repetitions in the first-layer code could have error patterns, and the first-layer error patterns need not be minimum-distance patterns. The analysis given is very rudimentary, but nonetheless provides some insight into why random interleaving becomes more effective as the number of bits involved becomes larger. This does not imply, however, that the minimum distance of the larger code is necessarily any greater than that of the smaller code, only that the error-rate performance will be better. If 36 repetitions were used, however, random interleaving would no longer be necessary; instead, *complete* interleaving could be used, where only one bit from each first-layer repetition ends up in any given second-layer repetition. Such interleaving would indeed guarantee a minimum distance equal to the product of the distances of the component codes ($4 * 4 = 16$). As the number of repetitions is increased in a coding scheme using random interleaving, the error performance gradually approaches that of a code structure using complete interleaving. (The "complete" interleaving example given here is equivalent to a 5x5x6x6/6x6x7x7 4-dimensional code, and so does not represent a new code, but simply a different viewpoint).

Minimum distance is only one of the contibuting factors to error rate performance; the other codeword distances are also factors, as well as the number of codewords at each distance. The minimum distance becomes increasingly important in determining the error performance of the code as the SNR increases. Often, however, at the operating SNR, the minimum distance of the code is not the determining factor of the error-rate performance. This is especially true when there are very few codewords at the minimum distance relative to the total number of codewords.

Random interleaving is often appropriate as the last component of an aggregate code structure. A typical coding scheme might consist of an n-dimensional parity structure with diagonal parity, that is randomly interleaved and arranged into a set of simple parity equations. The number of parity equations in this final layer is completely flexible, and so this layer can use up all of the parity bits that remain (the total number of parity bits that can be used may be determined by the desired code rate). In this way, the "random" parity layer enhances the distance properties of the code, while at the same time allows for fine adjustment of the code rate. Further, the set of parity bits associated with this last layer will always have even parity since the original structure was forced to even parity. This means that the final set of parity bits can be treated as a simple parity equation in the iterative decoding process. In other words, each of these bits is involved in not just one, but two different simple parity equations.

It can often be counter-productive to try to get "too much" from a second-layer code. For an increase in the complexity of the second-layer code to be worthwhile, the increased error-correcting ability of the code must more than make up for the loss in code rate. For this reason, a simple set of one-dimensional parity equations may often be the best choice as the final coding layer.

For some code structures. computer searches can be performed to find a random interleaver that meets a particular minimum-distance requirement. For example. for a two-layer, two-dimensional repetition code. a program was developed that evaluates whether the overall code has minimum distance 4. 6 or 8. While this task may at first appear to be computationally intractable. culling techniques can be employed that make this process feasible. Note that without such an evaluation of a potential interleaver. the only guarantee that can be made about the minimum distance of the aggregate code is that it is at least 4. In some instances (specifically. codes having only two repetitions) no random interleaver could be found that would give a minimum distance of 8: only $d_{min} = 6$ could be guaranteed. In these cases. however. the number of distance-6 patterns was small. and if the block could be shortened by a few bits. these patterns could be eliminated. thereby guaranteeing a minimum distance of 8. This shows that block shortening can in fact be very useful when the code structure includes random interleaving.

Chapter 4

Decoding of hyper-codes

4.1 Log-likelihood ratios (log-MAP)

One natural way to express MAP calculations is in terms of probability values. If MAP operations are implemented in this form, however, problems can be encountered associated with finite-precision arithmetic. In a MAP decoder, these precision problems can result in a degradation of the error-rate performance. It is possible, however, to greatly reduce these precision problems by not working with the probabilities directly, but rather with quantitites called "log-likelihood ratios" (LLRs). These quantities are commonly used in MAP decoder implementations. The LLR of a probability p is defined as:

$$llr = \log\left(\frac{p}{1-p}\right) \tag{4.1}$$

If the probability p was identically zero or identically one, the above expression would be ill-defined. A physical communication channel, however, can never provide such absolute certainty, and hence these special cases are never really an issue.

Consider now the calculation of the LLR for a particular channel bit. It will be assumed that the signal component at the receiver output is $\pm\sqrt{E_c}$ (i.e. binary antipodal signaling where E_c is the energy per channel bit), and that the noise at the receiver output is white and Gaussian with zero mean and variance $\sigma^2 = N_0/2$. The probability density function of the receiver output, x, conditioned on the bit that was transmitted, is:

$$\begin{aligned} f_x(x|0) &= \frac{1}{\sqrt{\pi N_0}} \exp\left[-\frac{1}{N_0}\left(x - \sqrt{E_c}\right)^2\right] \\ f_x(x|1) &= \frac{1}{\sqrt{\pi N_0}} \exp\left[-\frac{1}{N_0}\left(x + \sqrt{E_c}\right)^2\right] \end{aligned} \tag{4.2}$$

The bit-to-symbol mapping convention used is the natural "sign-bit" mapping:

$$\begin{aligned} '0' &\leftrightarrow +\sqrt{E_c} \\ '1' &\leftrightarrow -\sqrt{E_c} \end{aligned} \tag{4.3}$$

The probabilities for the two possible transmitted bits, given the received signal sample x, are:

$$\begin{aligned} p(0|x) &= \frac{f_x(x|0)}{f_x(x|0)+f_x(x|1)} \\ p(1|x) &= \frac{f_x(x|1)}{f_x(x|0)+f_x(x|1)} \end{aligned} \tag{4.4}$$

The LLR is calculated from these probability values:

$$llr = \log\left(\frac{p(0|x)}{p(1|x)}\right) \tag{4.5}$$

where the probability that a zero was transmitted is placed in the numerator (convention). Substituting for the probability values and simplifying gives:

$$llr = 4\frac{E_c}{N_0}\frac{x}{\sqrt{E_c}} \tag{4.6}$$

This shows that the log-likelihood ratio for a particular channel bit is calculated by multiplying the channel sample by the factor $4\frac{E_c}{N_0}\frac{1}{\sqrt{E_c}}$ This conversion requires knowledge of the received signal power and signal-to-noise ratio. It will be shown in section 4.3, however, that for the case of simple parity equations, an approximation can be used in the log-MAP calculations that makes the MAP operations independent of this initial scaling, thus eliminating the need for estimates of the channel conditions.

4.2 Extrinsic information

The term "extrinsic information" refers to the difference between the LLRs of the bits at the decoder input and the LLRs of the output bits from a soft-input, soft-output decoder. That is, the "extrinsic information" is the adjustment made to the LLRs. The next section describes how the "extrinsic information" for a simple parity equation can be calculated.

4.3 The max approximation (max-log-MAP)

A detailed description of "true MAP" calculations will not be presented as part of this report, since the topic is well covered in the standard literature: for example, see [1]. The "max" approximation, however, warrants some description since it is an important factor in making hyper-codes practical to implement. In the log-MAP equations, an expression that frequently arises is $\log(e^x + e^y)$. This can be approximated as:

$$\log(e^x + e^y) \approx \max(x, y) \tag{4.7}$$

When this approximation is substituted into the log-MAP equations for a simple parity equation, the equations are dramatically simplified. This idea is mentioned in [5]. The final result is as follows. When the "max approximation" is substituted into the log-MAP equations for a simple parity equation, the calculation of the extrinsic information for each element of the parity equation reduces to:

- The *magnitude* of the extrinsic information for a particular element is equal to the minimum magnitude of all of the other elements.
- The *sign* of the extrinsic information for a particular element is equal to the sign of the element itself, if the parity of the overall equation is even, and opposite to the sign of the element, if the overall parity is odd.

That is, each element is "strengthened" or "weakened" depending on whether the overall parity worked or not, and the amount of adjustment is determined by the smallest other element. The smallest magnitude of the other elements can be considered as something of a confidence measure in the overall parity result. At this point in the discussion, the scaling of the extrinsic information is being ignored. This is described later in section 4.6.

The simplicity of these calculations greatly reduces the computational requirements associated with determining the extrinsic information, as compared to using the "true MAP" equations. Further, when the "max approximation" is employed, there is no need to scale the channel samples to come up with log-likelihood ratios. This is because any scaling of the input samples would have no effect on the output of the decoder; scaling of the input values simply results in a scaling of all the values throughout the decoding process[1]. The

[1]Scaling is still important in time-varying channels, such as fading channels.

benefit of this is that the decoder no longer requires estimates of the channel signal and noise power levels. This also means that the error-rate performance of the decoder is not affected by the accuracy of these channel estimates. Finally, it will be shown later in section 4.5 that using the max-log-MAP approach allows for a significant reduction in the amount of memory required for storing the extrinsic information.

Initial investigations did not reveal any error-rate performance degradation associated with using the "max approximation", as compared to using the "true MAP" equations in decoding (provided that the appropriate extrinsic information scale factor was selected; see section 4.6). For this reason, and because of the many implementation advantages associated with its use, the max-log-MAP approach was primarily used for further investigations. It must be emphasized, however, that the error-control coding scheme that has been developed is in no way dependent on the use of this approximate method.

4.4 The iterative decoding process

In the decoding process, the basic MAP operation is applied to each simple parity equation in the first set of parity equations. Then, the second set of parity equations is processed, incorporating the extrinsic information from the first set. After this, the third set of parity equations is processed, using the extrinsic information from the first and second sets, and so forth, until all the sets have been processed. At this point, the decoding returns to the first set and a new cycle begins. Before processing the first set again, however, *it is important that the extrinsic information added in the first cycle be removed before new extrinsic information is calculated.* It has been observed that if this key step is omitted, the error-rate performance of the decoder will degrade significantly. This is the reason why it is necessary to save the extrinsic information from one cycle until the next cycle. The iterative decoding process continues until a specified number of cycles have been completed, or until some other stopping criterion is satisfied (for example, further decoding cycles will not change the final outcome; see section 4.7). The number of decoding cycles that should be performed depends on numerous factors, including the code structure itself, the operating signal-to-noise ratio, and the desired error-rate performance.

There are several variations to the basic MAP decoding approach described above. One aspect that can vary is the order in which the sets of parity equations are processed.

Changing the order of processing is generally of no consequence. though in certain situations a particular order might be advantageous. Another variation is to wait until the end of each cycle before adding in the extrinsic information. This approach certainly removes any bias associated with the order in which the sets are processed. When this method was tested, however, no improvement in the error-rate performance was observed. and convergence was slightly slower. Also, this method increases the memory requirements of the decoder. Such an approach could be useful. however. in a multi-processor decoder to help manage shared memory.

There are two minor implementation optimizations related to the extrinsic information storage that can be used to save processing. In the first cycle. there is usually no extrinsic information to subtract off. While this could be accommodated by simply zeroing all extrinsic information before beginning decoding, if the first cycle is treated as a special case, the subtraction is not required in this cycle. and the extrinsic information does not need to be zeroed. This saves some processing. Another cycle that can also be treated as a special case. to save processing. is the last cycle. In this case. there is no need to store the extrinsic information. since it will never be made use of.

It is important to recognize that this iterative decoding approach (even with the true MAP equations on the parity equations) is *not* equivalent to true maximum-likelihood (ML) or true MAP decoding. The advantage of using an iterative approach. however. is that more complex code structures can be handled. The ability to use more powerful codes often more than compensates for the sub-optimality of the decoding process. That is. even though iterative decoding is not true ML or true MAP. the combination of a more complex code structure and iterative decoding can provide better error-rate performance than using a simpler code structure with true ML or MAP decoding.

This iterative decoding approach is not a new development and is discussed in several papers. A good description is given in [1].

4.5 Reducing the memory requirement for extrinsic information storage

When the max-log-MAP approximation is used in decoding, the extrinsic information associated with each parity equation can be stored in a compressed form. Rather than storing a value for each element of the parity equation, only the two minimum magnitudes are stored, along with a "sign word" and a location index. The "sign word" records the sign bit of each element of the parity equation, and the index indicates the location of the minimum magnitude within the parity equation. This information is sufficient to allow subtraction of the extrinsic information when the parity equation is processed in the next decoding cycle.

This reduction in the amount of memory required to store the extrinsic information is possible because, when the "max" approximation is employed, the magnitudes of the extrinsic information for all but one of the elements of any given parity equation are the same. (Recall from section 4.3 that the magnitude of the extrinsic information for a particular element is equal to the minimum magnitude of all of the other elements. This means that the element of the parity equation having the smallest magnitude gives the magnitude of the extrinsic information for all of the other elements in the parity equation.)

Of course, many variations are possible based on this fundamental idea. For example, instead of storing the two minimum magnitudes, the minimum magnitude and a difference value could be stored. As another example, in some DSP implementations it is desirable to pack both the sign information and the location index into a single word. Such variations are simply different embodiments of the same idea. The specific manner of storage will depend on what is most convenient for the implementation being developed.

When stored in this compressed form, the space required to store the extrinsic information associated with a single parity equation increases only slightly as the parity equation becomes longer. This means that as the lengths of the parity equations go up, the compression factor becomes greater.

The ability to compress the extrinsic information when using the max-log-MAP approximation is another significant advantage associated with this approximation. The approximation not only reduces the processing requirements, but also allows significant savings in the memory required to store the extrinsic information.

4.6 Scaling of the extrinsic information

It has been found that the error rate performance of the decoder is significantly improved if the extrinsic information is scaled down before being added to the composite values. In particular, a scale factor of 0.625 has been used in testing, and gives good performance. The specific value of 0.625 was used in testing because multiplication by this number can be realised using two shifts and an add. This property could be useful in implementations where multiplication by an arbitrary constant is undesirable (e.g. a hardware implementation). In implementations where there is no penalty associated with multiplication (e.g. a DSP implementation), the value of the scaling factor can be optimized, taking into consideration the number of cycles the decoder will perform. The optimal choice of scale factor depends somewhat on the number of cycles the decoder executes before outputing decisions. Typically, the more decoding cycles, the smaller the factor should be, and vice versa. This means that final selection of the scale factor should only be made after the number of decoding cycles has been decided upon.

Improving performance by scaling the extrinsic information is not unique to max-log-MAP decoding of simple parity equations, nor even to MAP decoding of simple parity equations in general. Other researchers have reported similar findings related to various MAP decoding applications (for example, see [6]). In some cases, the adjustments made to the extrinsic information are more involved than simple multiplication by a scale factor. In this application, however, the simplicity of the core decoding process is a key feature, and so the minor performance improvements that might be gained by using a more complicated adjustment approach would probably not justify the extra computational complexity that an alternate approach would entail. Thus, for this approach, simple multiplication by a scale factor will usually be the most appropriate choice for adjusting the extrinsic information.

4.7 Convergence tests

A simple test can be implemented that gives a result which, if true, guarantees that the decoder has converged to a valid codeword, and that further processing is redundant since it will not affect the final outcome. Note that convergence to a *valid* codeword does not necessarily imply convergence to the *correct* codeword. This test can be used to stop the decoding before all of the specified cycles are complete, and hence reduce the average processing requirements associated with the decoding operation.

The test involves checking whether the parity of each equation, with extrinsic information removed, is even, and that none of the elements of the equation have changed sign since the last cycle. If these conditions are satisfied for all parity equations making up the codeword, then no further decoding cycles are executed.

The implementation of this test requires negligible extra processing. If significant extra processing was required, the test would not be worthwhile, since the whole point is to reduce the amount of processing. As part of the max-log-MAP processing of a simple parity equation, it is already necessary to determine the parity of the equation, with extrinsic information removed, and so there is no extra processing associated with this step. Further, when the extrinsic information is stored in compressed form (see section 4.5), a "sign word" is assembled which indicates the sign of each element in the parity equation, with extrinsic information removed. Performing an exclusive-or (XOR) operation between the old and new sign words gives a result that indicates whether any signs have changed since last time. If no signs have changed, *and* the parity works this time, then the parity must have worked last time. If the parity worked last time, and no signs have changed since last time, then subtracting the extrinsic information off this time did not flip any signs. This means that the equation, with extrinsic information included, also had even parity. The conclusion is that this approach will not affect the final decoding outcome. The implementation of the test is in fact very simple. At the beginning of a decoding cycle, a global flag is set. During the max-log-MAP processing of each parity equation, if the parity of the equation (with extrinsic information removed) was odd, the flag is cleared. Also, if the parity worked but the XOR result was non-zero, the flag is cleared. At the end of the cycle, if the flag is still set, then the decoding process is halted.

This approach can be easily extended to allow processing to stop not only at cycle boundaries, but also after any set of parity equations within a cycle. In addition to the flag variable, a counter is required that is incremented after each set if the flag is still true, and zeroed if the flag is clear. In this way, the counter keeps track of how many sequential sets of parity equations satisfied the test criterion. When the counter reaches the total number of sets making up the overall structure, then the decoding process can be stopped. In this approach, the global flag must be set before each set of parity equations, and not only at the beginning of each cycle. This extension further reduces the average processing requirements of the decoder.

It is important to realise that the failure of this test does not necessarily indicate that the decoder has not yet converged to a codeword. Rather, if the test passes, it is *guaranteed* that the decoder has indeed converged. That is, this is a conservative test. Other stopping criteria could be used that would halt the decoding earlier, while still achieving the same or only slightly degraded error-rate performance. For example, after every cycle the composite samples could be tested to determine whether they form a valid codeword. If so, the decoding process would be stopped. It has not been shown that such a stopping criterion would never alter the final outcome of the decoding operation. In practice, however, such an approach would probably cause no significant degradation in error-rate performance, but would reduce the average processing to a greater extent than the "guaranteed" approach described earlier.

It is interesting to observe that if the decoder exits early, the sign words that are part of the compressed extrinsic information storage can be used as the hard-decision output of the decoder. Only the sign words associated with one set of parity equations are required, and the first set of parity equations will probably have the bits in an order that is convenient for output.

Chapter 5

Different modulation schemes

5.1 Starting LLRs for arbitrary two-dimensional signaling constellations

Up to this point, the calculation of log-likelihood ratios (LLRs) from channel samples has only been discussed in relation to binary antipodal signaling (for example, BPSK or QPSK modulation). In this section, a method will be presented for calculating LLRs from channel samples for any arbitrary two-dimensional signaling constellation. Having an efficient means of calculating starting LLRs for more complicated modulation schemes is particularly important in light of the fact that hyper-codes are well-suited to higher-rate applications. This is because, with hyper-codes, higher rates are achieved not by puncturing parity bits, but by simply increasing the lengths of the dimensions, and this preserves the minimum distance property of the code.

The calculation of LLRs from channel samples essentially amounts to determining, for each bit in the symbol, the probability that it is a zero, given the received point. To calculate this exactly involves significant log-likelihood algebra, which is computationally intensive. However, an approximation has been developed that greatly reduces the associated computational requirements. The approximation involves finding, for each bit position within the symbol, the closest constellation point to the received point that has a "0" in that position, and similarly the closest constellation point to the received point with a "1" in that position. The difference between the squared distances from the received point to each of these two constellation points is used as the starting LLR for the bit. There is also a scale factor involved, which is a function of the signal power and the channel noise power. If max-log-MAP decoding is being used, however, this scale factor is of no consequence and can be ignored.

The necessary operations for each received point amount to the following. The squared distances to each constellation point are first calculated and recorded, and the shortest distance is determined, giving the closest point. Then, for each bit position in the symbol, the next closest constellation point is determined, *the search being restricted to points that are opposite in polarity to the closest point at the bit position being considered.* Finally, the two squared distances are subtracted, and the difference is the LLR estimate for that bit. The "direction" of the subtraction is determined by which point has a "0" in the relevant bit position: the distance associated with this point is the value that is subtracted. This convention results in "0" bits being associated with positive LLR values, and "1" bits with negative values. It is worthwhile to note that, with this method of calculating LLRs, not only is the number of required operations small, but also the complexity of the operations themselves is low. There are no exponentials or square-root operations, only multiplications, additions, and subtractions.

The intuitive justification for ignoring all but two of the consellation points, for each bit, is that the two points that are kept would normally dominate the exact log-likelihood calculations.

5.2 Analysis

A more detailed mathematical analysis will now be presented describing the calculation of log-likelihood ratios from received channel samples for arbitrary two-dimensional signaling constellations. Assume that all transmitted symbols are equi-probable, and that both the x and y signal components are corrupted by independent Gaussian noise having zero mean and average power $\sigma^2 = \frac{N_0}{2}$. Let (t_{j_x}, t_{j_y}) represent the j^{th} possible transmitted point (i.e. constellation point), and let (r_x, r_y) represent the received point. The (exact) LLR for the i^{th} bit in the symbol is given by:

$$\begin{aligned} llr_i &= \log\left(\frac{p_0}{p_1}\right) \\ &= \log\left\{\frac{\sum_j^0 \left[\frac{1}{\sqrt{\pi N_0}} e^{-\frac{1}{N_0}(r_x - t_{j_x})^2}\right]\left[\frac{1}{\sqrt{\pi N_0}} e^{-\frac{1}{N_0}(r_y - t_{j_y})^2}\right]}{\sum_j^1 \left[\frac{1}{\sqrt{\pi N_0}} e^{-\frac{1}{N_0}(r_x - t_{j_x})^2}\right]\left[\frac{1}{\sqrt{\pi N_0}} e^{-\frac{1}{N_0}(r_y - t_{j_y})^2}\right]}\right\} \end{aligned} \tag{5.1}$$

where the summation in the numerator is over all of the constellation points that have a "0" in the i^{th} bit poisition, and the summation in the denominator is over all of the constellation points that have a "1" in the i^{th} position. The above formula follows directly from the definition of a LLR and the fact that the two Gaussian noise sources are independent. The LLR can also be expressed in vector notation:

$$llr_i = \log\left[\frac{\sum_j^0 \frac{1}{\pi N_0} e^{-\frac{1}{N_0}\left|\vec{r}-\vec{t_j}\right|^2}}{\sum_j^1 \frac{1}{\pi N_0} e^{-\frac{1}{N_0}\left|\vec{r}-\vec{t_j}\right|^2}}\right] \tag{5.2}$$

At this point the approximation is applied. The summation in the numerator is approximated by keeping only the largest term, and likewise for the denominator. This gives:

$$llr_i \approx \log\left[\frac{\frac{1}{\pi N_0} e^{-\frac{1}{N_0}\left|\vec{r}-\vec{t_0}\right|^2}}{\frac{1}{\pi N_0} e^{-\frac{1}{N_0}\left|\vec{r}-\vec{t_1}\right|^2}}\right] \tag{5.3}$$

where $\vec{t_0}$ represents the closest constellation point to the received point having a "0" in the i^{th} bit position, and $\vec{t_1}$ represents the closest point with a "1" at the i^{th} position. Simplifying,

$$llr_i \approx \frac{1}{N_0}\left[\left|\vec{r}-\vec{t_1}\right|^2 - \left|\vec{r}-\vec{t_0}\right|^2\right] \tag{5.4}$$

This is precisely the equation described earlier for obtaining the (approximate) LLR from the channel samples; namely, a difference is taken between two squared distances. Note the scale factor of $\frac{1}{N_0}$ that was not mentioned earlier. In the case of max-log-MAP decoding, however, this scale factor is of no consequence.

It is interesting to re-express the difference of squared distances in an alternate form in order to compare this method of determining LLRs with the approach described earlier for binary antipodal signaling; namely, simply using the channel samples themselves. Let x represent the distance from the midpoint of the line segment joining $\vec{t_0}$ and $\vec{t_1}$ to the projection of $\vec{r}$ onto the line passing through $\vec{t_0}$ and $\vec{t_1}$. Further, let x be positive if the projection of $\vec{r}$ is closer to $\vec{t_0}$ than $\vec{t_1}$, and negative if the projection is closer to $\vec{t_1}$ than $\vec{t_0}$ (that is, x is a "signed distance"). Finally, let h represent the distance from $\vec{r}$ to the line passing through $\vec{t_0}$ and $\vec{t_1}$. Then:

$$\left|\vec{r}-\vec{t_1}\right|^2 = \left[\frac{\left|\vec{t_1}-\vec{t_0}\right|}{2} + x\right]^2 + h^2$$

$$\left|\vec{r}-\vec{t_0}\right|^2 = \left[\frac{\left|\vec{t_1}-\vec{t_0}\right|}{2} - x\right]^2 + h^2 \tag{5.5}$$

Subtracting these two equations, simplifying, and multiplying by $\frac{1}{N_0}$ gives:

$$\frac{1}{N_0}\left[\left|\vec{r}-\vec{t_1}\right|^2 - \left|\vec{r}-\vec{t_0}\right|^2\right] = 2\frac{\left|\vec{t_1}-\vec{t_0}\right|}{N_0}x \tag{5.6}$$

That is, taking the difference of squared distances is equivalent to determining the distance between the midpoint of the line segment joining the two constellation points and the projection of the received point onto this line (times a scale factor). Of course, in practice one would not calculate the projected point and the distance from it to the mid-point, since this would involve a lot of extra computation that is completely unnecessary. Recall that the squared distances are already available since they were required to determine the two closest constellation points to use. The above derivation is informative, however, in showing the relationship between the general method of determining LLRs and the earlier method described for use with antipodal signaling. In fact, finding the starting LLRs for BPSK and QPSK modulation is simply a special case of the above general method. For BPSK, taking the difference of the squared distances from the received sample to each of the two constellation points is equivalent to taking the channel sample itself, within a scale factor. For QPSK, provided that the bit assignment is made in a Gray-coded manner, the same reasoning applies.

The above result may lead one to the false conclusion that, for any arbitrary two-dimensional constellation, finding the starting LLR for a particular bit can be likened to the antipodal case, with the two symbols chosen from the arbitrary constellation playing the roles of the two antipodal signaling points. The reason this conclusion is not correct is that there is a scale factor involved that is dependent on the distance between the two constellation points being used in the calculations. Thus, it can be misleading to think of calculating an LLR for a particular bit in an arbitrary constellation in terms of a rotated, translated antipodal signal.

Another distinction that is worth emphasizing is that taking the difference between the two squared distances is not necessarily equivalent to finding the distance to a decision boundary. Finding the distance to the decision boundary is *not* the recommended way of calculating the starting LLRs. even though it might seem like a natural extension from the antipodal signaling case. An analysis of the decision-boundary approach will not be presented here. but if this topic is of interest to the reader. a good case to examine is Gray-coded 16QAM modulation.

In the above discussion. scale factors were routinely ignored. since they have no effect on max-log-MAP processing. which will normally be used for decoding.

5.3 Performance

In testing, this method of computing the starting LLRs has been found to provide excellent performance. This is not to say that this approximation will always give a value that is very close to the exact LLR. but rather that using the approximation causes no significant degradation in error-rate performance as compared to using the exact LLRs. Because of its efficiency and effectiveness. this approximation is considered one of the significant innovations coming out of this development effort.

5.4 Guidelines related to larger constellations

5.4.1 Selection of the constellation size and FEC code rate

The number of bits per symbol of the modulation scheme and the code rate of the FEC code should be analysed as a pair to find the combination that provides the best overall error-rate performance. subject to the channel utilization requirements imposed by the application (i.e. bits/Hz). Better error-rate performance will often be achieved by using a smaller constellation and a higher-rate code. as opposed to a larger constellation and a lower-rate code. That is. it can be difficult for an FEC code to make up the energy loss associated with using a larger signaling constellation.

5.4.2 Arrangement of constellation points, and assignment of bits

The arrangement of the constellation points and the assignment of bit patterns to these points should be considered together so as to minimize the raw channel bit error rate, which in turn will minimize the decoded error rate. It is not only the Euclidean distances between constellation points that impacts the final error-rate performance, but also the number of bits that change in going from one point to another. For example, with 16QAM, the following Gray-coded assignment is a good choice:

```
1101 1001 1000 1100
0101 0001 0000 0100
0111 0011 0010 0110
1111 1011 1010 1110
```

Observe that for any constellation point, all of the neighbouring points differ by only one bit. Using a random bit assignment instead of a pattern like this would cause a significant degradation in decoded error-rate performance.

5.4.3 Interleaving of bits before grouping into symbols

Certain combinations of symbol constellations and code structures can result in unfavourable distributions of the less-reliable channel bits within the code structure. If the arrangement of the less-reliable symbol bits within the code structure is such that complete or partial low-distance error patterns can be formed from these less-reliable bits, error-rate performance will be degraded.

For example, consider Gray-coded 8PSK modulation in combination with an 8x8/9x9 parity square. If the bits are grouped into symbols by simply moving along each row, taking three bits at a time, all of the least-reliable channel bits will come from only 3 of the 9 columns of the code. This occurs because 3 divides evenly into 9. Since the minimum-distance error patterns for a parity square are rectangles, there will be many minimum-distance error patterns that only involve these least-reliable bits. This ordered arrangement of the least-reliable channel bits will degrade the error-rate performance of the code.

To avoid such performance degradation, the channel bits should be randomly interleaved before they are grouped into symbols for transmission. This eliminates any regular relationship between the position of bits within the symbols, and their positions within the overall code structure.

5.4.4 Grouping of symbol bits into different codes according to reliability

For modulation schemes where the symbol bits are not all equally reliable, an option that is available when devising an FEC approach is to divide the channel bits into two or more codes, using more powerful codes for the less-reliable channel bits, and simpler codes for the more-reliable channel bits.

Chapter 6

Performance results

For all the simulation results presented, a factor of 0.625 was used in scaling the extrinsic information (see section 4.6). The number of decoding cycles varied depending on the code structure being used, but was typically between 8 and 16 cycles. For small block sizes, good results were often obtained with 8 cycles. Note that the number of cycles used to decode hyper-codes cannot be compared with the number of cycles used with other iterative decoding schemes, such as Turbo codes, since the complexity of a hyper-code cycle is usually much less than of other schemes.

6.1 BER performance for a representative hyper-code

Figure 6.1 shows the bit error-rate performance of a 7x7x7x7/8x8x8x9 4D plus "roll-parity" hyper-code. This code has a block size of 2401 information bits, and a rate of r=0.52. The number of decoding cycles used was 12. Observe that a BER of 10^{-5} is attained at an E_b/N_0 of 1.75 dB.

For comparison purposes, the performance of standard k=7 and k=9 rate=1/2 convolutional codes with soft Viterbi decoding is also indicated on the plot.

In addition, the performance of a concatenated Viterbi/Reed-Solomon coding scheme is shown on the figure. The overall block size of the concatenated code is 15,040 information bits, and the overall rate is $r = 0.46$ [13]. Observe that the hyper-code, in addition to providing better performance, is a higher rate code and uses a much smaller block size.

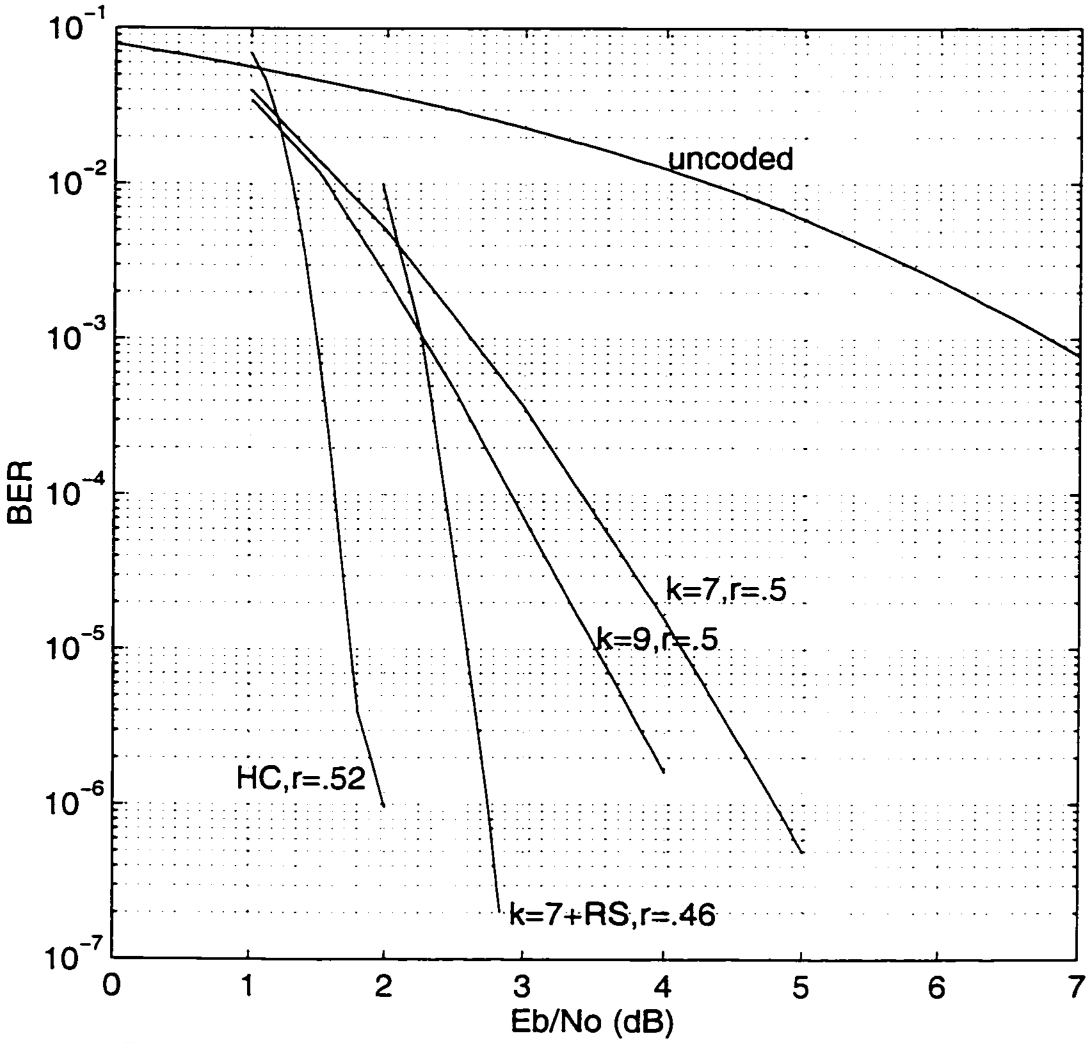

Figure 6.1: BER vs. E_b/N_0 in AWGN for a 2401 information bit, rate=0.52 hyper-code.

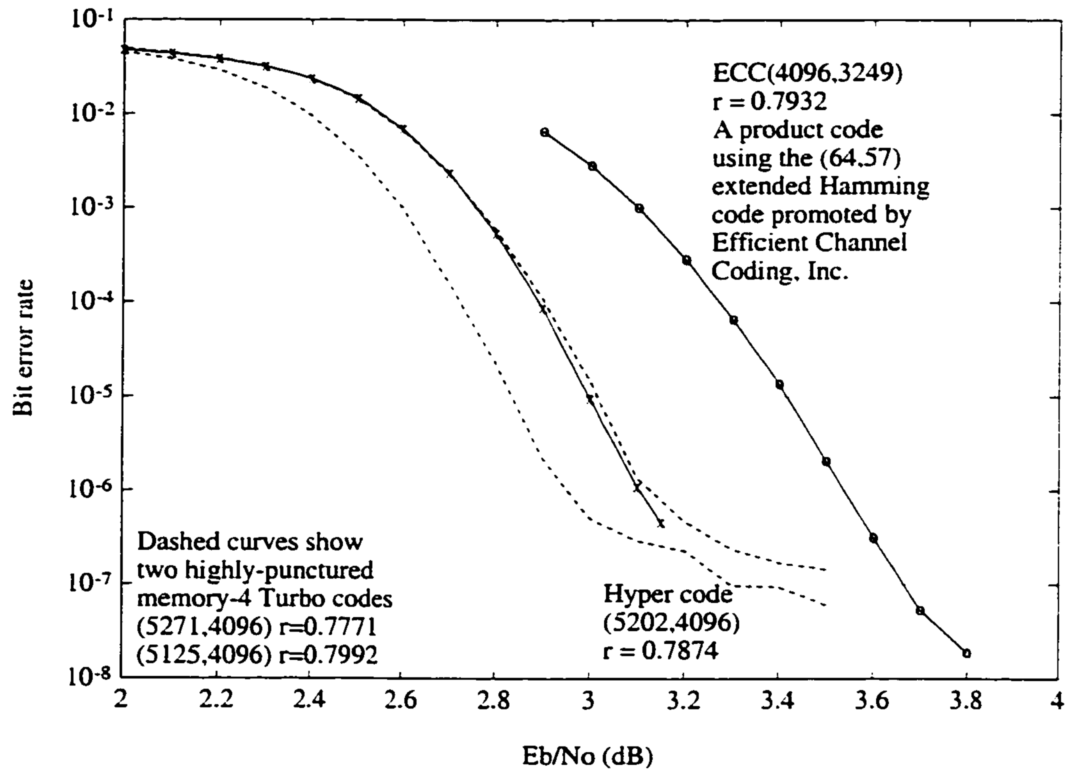

Figure 6.2: BER vs. E_b/N_0 in AWGN for a 4096 information bit, rate $r \approx 4/5$ hyper-code.

6.2 BER performance for a higher-rate hyper-code

Figure 6.2 shows the BER performance of a rate $r \approx 4/5$ hyper-code, as well as the performance of competitive coding schemes with similar blocks sizes and code rates. One of the comparison codes is a code promoted by a company called *Efficient Channel Coding, Inc.* The performance figures were obtained from the web page *www.eccincorp.com.* The other comparison coding scheme is "Turbo coding". Two curves are given for Turbo-code performance, showing the performance of codes with lower and higher code rates than that of the hyper-code. It is difficult to match the code rate of the Turbo code exactly to that of the hyper-code.

6.3 Performance for a variety of hyper-codes and modulation schemes

Figures 6.3 through 6.5 summarize the BER performance in Gaussian noise of a wide variety of hyper-codes, for various modulation schemes. The vertical axis of each plot has the units "information bits per use", which indicates the bandwidth efficiency of the combination of modulation scheme and code structure. For example, a rate=1/2 code used with QPSK modulation has a bandwidth efficiency of 1 information bit per use.

The figures show only a single point for each hyper-code structure: namely, the E_b/N_0 required to achieve a BER of 10^{-5}.

On each plot the Shannon capacity limit is also indicated [11], as well as the capacity limit associated with the modulation scheme being considered [10].

The curve labels indicate the number of dimensions of the parity structure. For example, the curve label "3D" indicates a three-dimensional cube structure. A "+" suffix on the curve label indicates the addition of diagonal parity. For block sizes for which diagonal parity would be less effective, roll-interleaving has been used, with an appropriate roll assignment. See section 3.6 for a discussion on roll interleaving.

The point labels indicate the lengths of the sides of the parity structure, in information bits. For example, the point labelled with an "8" along the "3D" curve corresponds to an 8x8x8/9x9x9 parity cube code with 512 information bits and 729 total bits. The point labelled with an 8 along the "3D+" curve corresponds to an 8x8x8/9x9x10 parity-cube-plus-diagonal code with 512 information bits and 810 total bits.

6.3.1 QPSK modulation

Figure 6.3 includes many points, and so at first may appear somewhat cluttered. It has been found, however, that having all of this data presented on the same plot facilitates comparisons between many coding alternatives, and also allows trends to be established. For these reasons, the plot has not been subdivided into a set of simpler plots.

For comparison purposes, the performance of standard k=7 convolutional encoding (G1=171, G2=133) is marked on the plot. Results are given for no puncturing (rate=1/2), as well as puncturing to rates 2/3, 3/4, and 7/8. Soft Viterbi decoding is assumed.

Recall that with QPSK modulation, no more than 2 information bits can be transmitted per channel use.

As an example, consider the performance of the 16x16x16/17x17x18 code, labelled as "16" along the "3D+" curve. This code has a block size of 4096 bits and a code rate of 0.79. The required E_b/N_0 is only 2dB from the Shannon capacity limit, and only 1dB from the capacity limit for this modulation format. It is also about 2.5 dB better than the punctured k=7 curve at the same code rate.

Note that this code performs almost as well as a "4D" code of block size $16^4 = 65536$ information bits. In general, for three or more dimensions, the "nD+" codes perform very close to the corresponding "(n+1)D" codes with the same length parity equations, but with a much smaller block size.

6.3.2 8PSK modulation

Figure 6.4 shows the results for 8PSK. The codes with sides of length 8 information bits required random interleaving of the channel bits before the channel bits were grouped into symbols for transmission. This was necessary in order to avoid a performance degradation associated with the relationship between the placement of the least-reliable channel bits within the code structure, and the shapes of the low-distance error-patterns of the code structure. See section 5.4.3 for a discussion on this issue. Without random interleaving, performance is about 0.5 dB worse.

The bit patterns were assigned to the constellation points in a Gray-coded manner, as outlined in section 5.4.2.

Recall that with 8PSK modulation, no more than 3 information bits can be transmitted per channel use.

6.3.3 16QAM modulation

Figure 6.5 shows the results for 16QAM. The bit patterns were assigned to the symbol points using a Gray code in both signaling dimensions.

Recall that with 16QAM modulation, no more than 4 information bits can be transmitted per channel use.

As an example, the performance of the 16x16x16/17x17x18 code is within 1.1 dB of channel capacity for this modulation format. This is typically about 2 dB better than standard trellis coding techniques, with similar complexity [10].

6.4 Decoding complexity

The decoding complexity for hyper-codes is very low. Preliminary DSP implementation results indicate that the complexity of the "3D+" codes and "4D+" codes is about the same as that required for Viterbi decoding of k=7 and k=8 convolutional codes, respectively.

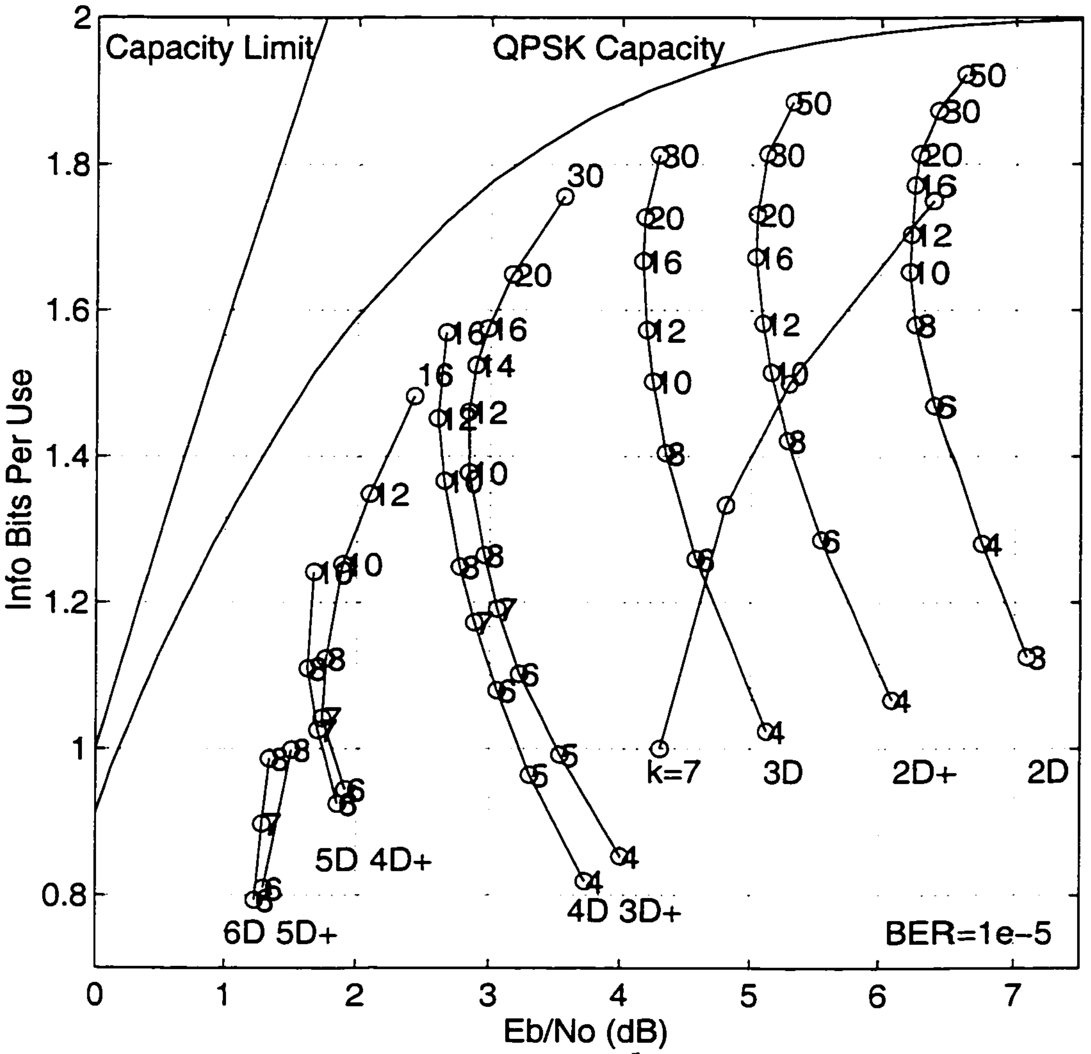

Figure 6.3: E_b/N_0 required to achieve a BER of 10^{-5} for various hyper-codes, using QPSK modulation.

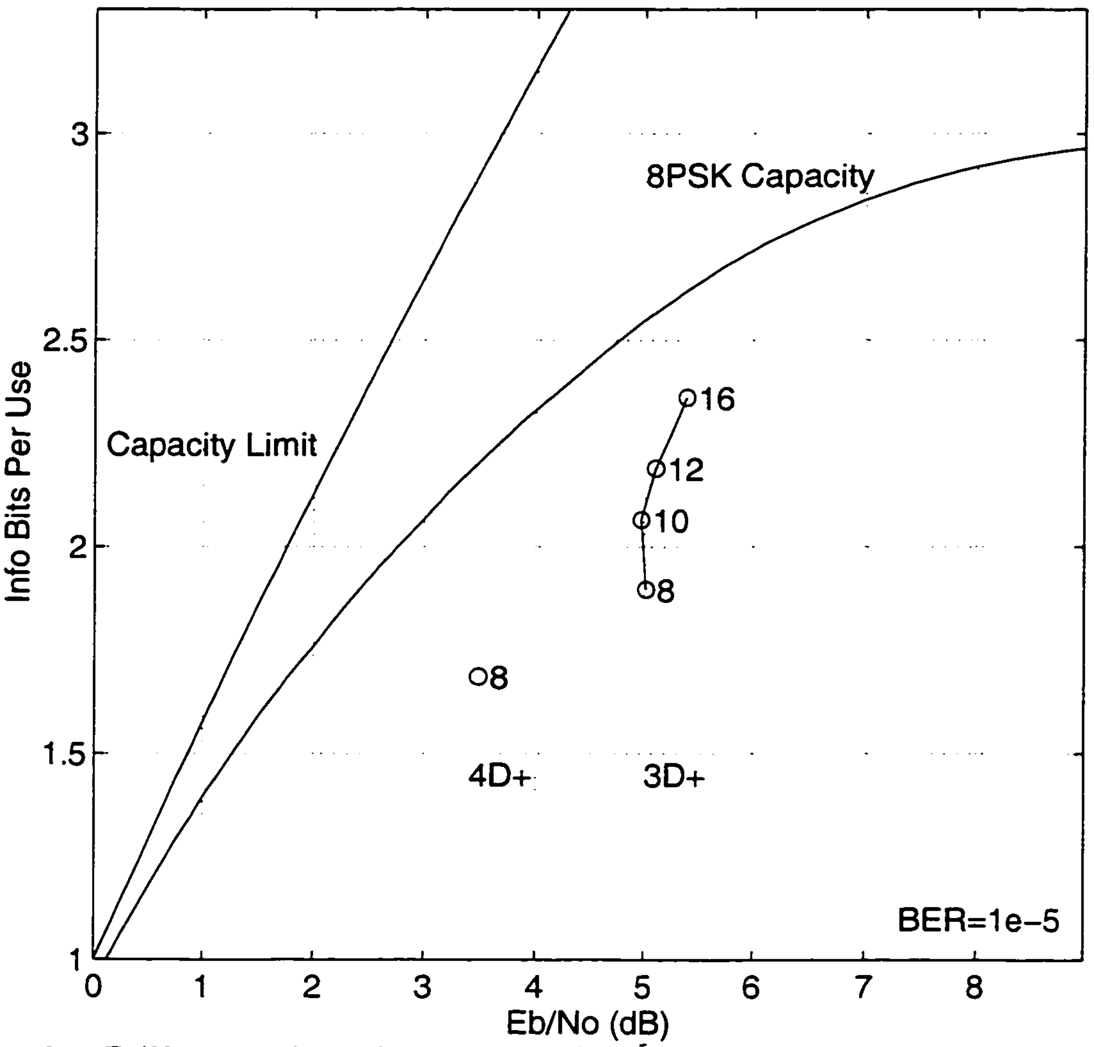

Figure 6.4: E_b/N_0 required to achieve a BER of 10^{-5} for various hyper-codes, using 8PSK modulation.

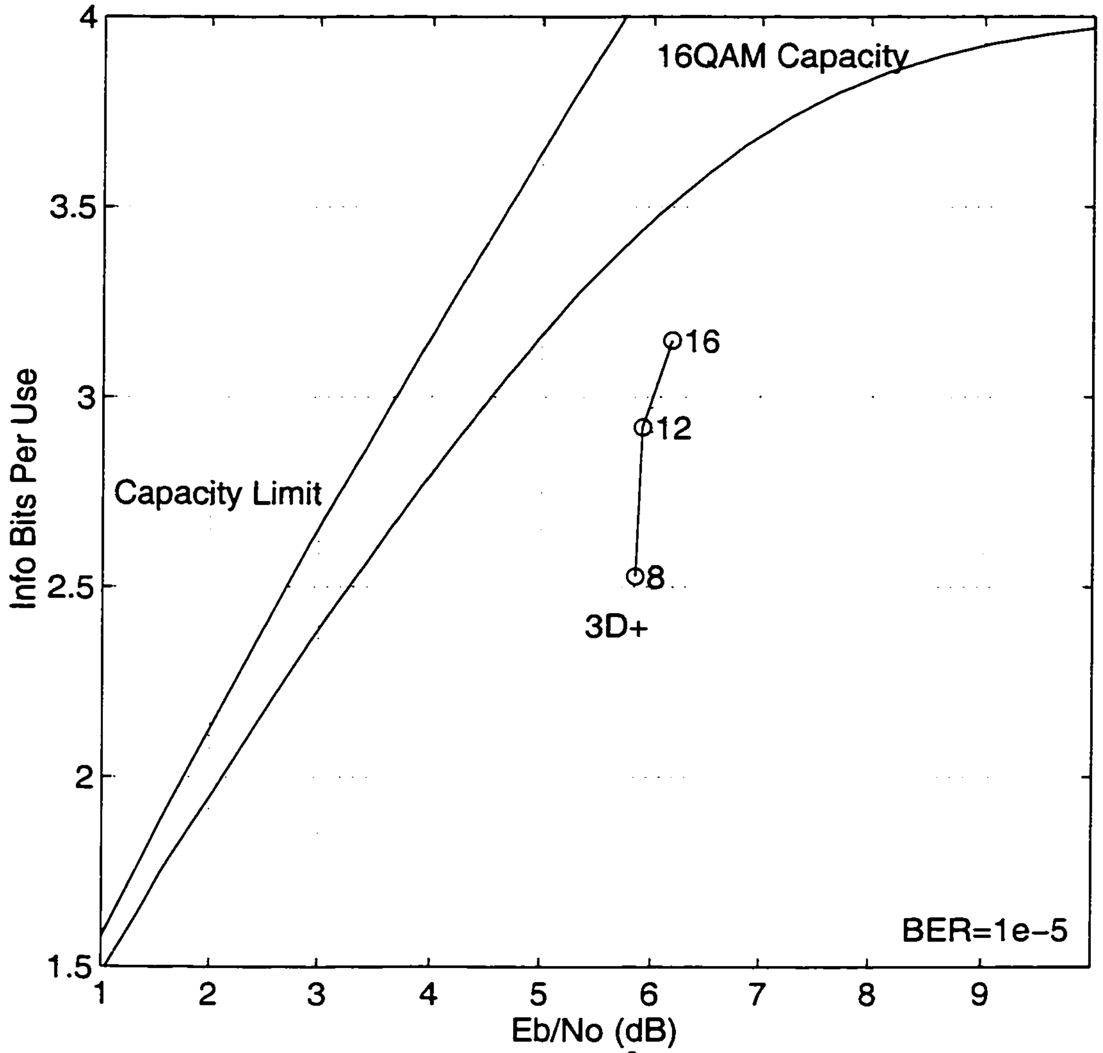

Figure 6.5: E_b/N_0 required to achieve a BER of 10^{-5} for various hyper-codes, using 16QAM modulation.

Chapter 7

Conclusions and suggestions for further research

This thesis has presented a new family of FEC codes, called "hyper-codes". These codes offer powerful error-correcting performance with low decoding complexity. For example, using QPSK modulation over an AWGN channel, a hyper-code of block size 4096 information bits and of rate=0.79 achieves a BER of 10^{-5} at an E_b/N_0 of 3 dB. This is only 2 dB from the Shannon capacity limit, and only 1 dB from the capacity limit for QPSK modulation. The decoding complexity is about the same as that required for a k=7 Viterbi decoder, based on a DSP implementation.

There are numerous areas related to hyper-codes that could be investigated further. The following list indicates some of these areas.

- Diagonal interleaving, and more generally, roll interleaving, are effective methods of shuffle interleaving for a wide variety of block sizes. In some cases, however, the best roll interleaver may not achieve the performance of the best shuffle interleaver. For example, consider a 3-D parity hyper-cube of size 5x5x5 / 6x6x6. Since the lengths of the sides of this cube are even, the addition of diagonal parity does not increase the minimum distance beyond the 8 of the original cube. With appropriate roll interleaving, the minimum distance can be raised to 12, which is a significant improvement over diagonal parity, but still not as high as the minimum distance of 14 that is attained when diagonal parity is added to a cube having odd-length sides. This raises the question as to how to shuffle hyper-cubes where roll parity does not achieve the highest possible minimum distance. Some work has been done in this area, especially for the two-dimensional case, but there is certainly more research that could be done in this area.

- Multiple sets of shuffle parity (and should the first layer of extra shuffle parity be included in the second shuffle interleaver?)
- Decoder convergence: Why is the decoder performance so dependent on the scaling of the extrinsic information? Should a smaller scale factor not simply slow down decoder convergence? This issue is part of the larger problem of improving the error-rate performance at high bit error rates (e.g. 10^{-3}).
- Performance with a final layer consisting of a set of 1-D parity equations, applied after random interleaving.
- Performance with shortening and puncturing.
- Performance in fading channels.
- More work could be done using larger signaling constellations, and the design of these signaling constellations.
- DSP implementation / FPGA implementation / VLSI implementation.

Bibliography

[1] J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Transactions on Information Theory*, Vol. 42, No. 2, pp. 429-445, March 1996.

[2] R.G. Gallager, "Low-density parity-check codes," *IRE Transactions on Information Theory*, pp.21-28, January 1962.

[3] J. Lodge, P. Hoeher, and J. Hagenauer, "The decoding of multi-dimensional codes using separable MAP 'filters'," in *Proc., 16th Biennial Symp. on Communications* (Queen's University, Kingston, Canada, May 1992), pp. 343-346.

[4] J. Lodge, R. Young, P. Hoeher, and J. Hagenauer, "Separable MAP 'filters' for the decoding of product and concatenated codes," in *Proc., IEEE Int. Conf. on Communications* (Geneva, Switzerland, May 1993), pp. 1740-1745.

[5] G. Battail, "Building long codes by combination of simple ones, thanks to weighted-output decoding," in *Proc., URSI ISSSE* (Erlangen, Germany, Sept. 1989), pp. 634-637.

[6] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo codes," in *Proc., IEEE Int. Conf. on Communications* (Geneva, Switzerland, May 1993), pp. 1064-1070.

[7] C. Berrou, A. Glavieux, "Near optimum error-correcting coding and decoding: Turbo codes," *IEEE Transactions on Communications*, Vol. 44, No. 10, pp. 1261-1271, October 1996.

[8] C. Berrou and M. Jézéquel, "Frame-oriented convolutional turbo codes," *Electronics Letters*, Vol. 22, No. 15, pp. 1362-1364, 1996.

[9] J. Proakis, *Digital Communications, Third Edition*. McGraw-Hill, 1995.

[10] G. Ungerboeck, "Channel coding with multi-level/phase signals," *IEEE Transactions on Information Theory*, Vol. IT-28, No. 1, pp. 55-67, January 1982.

[11] S. Haykin, *Digital Communications*. John Wiley and Sons, 1988.

[12] R. Gallager, *Information Theory and Reliable Communication*. John Wiley and Sons, 1968.

[13] *AHA4210 Viterbi / Reed-Solomon Decoder*. Advanced Hardware Architectures, Inc., 1995.

[14] J. Erfanian, S. Pasupathy, G. Gulak, "Reduced complexity symbol detectors with parallel structures for ISI channels," *IEEE Transactions on Communications*, Vol. 42, No. 2/3/4, pp. 1661-1671, Feb/Mar/Apr 1994.

Appendix A

Core hyper-code functions

A.1 Header file

```
#ifndef _HC_H_
#define _HC_H_

void HCencode(
int *SrcAndDstPtr,
const unsigned short *CodeDefnArray);

int HCdecode(
double *InputAndOutputLlrs,
double *ExtrinsicArray,
const unsigned short *CodeDefnArray,
unsigned NumCycles,
int& ZeroTheExtrinsicFlag);

void HCsanity(
const unsigned short *CodeDefnArray,
unsigned NumChanBits,
unsigned NumInfoBits,
unsigned NumExtrinsics);

#endif
```

A.2 "C" code

```
/*
Hyper-code core routines.
Andrew Hunt 1997
*/

typedef int            bit;  // LinkSim uses type int for bits.
typedef unsigned short  inx;  // This should be kept an unsigned type.

#define MAX_LEN 64
#define FIXED_POINT 0

//***************************************************************************

#if FIXED_POINT
   typedef int llr;
   #define Scale(x) (((x) >> 1) + ((x) >> 3))
   #error Have you included normalization logic elsewhere?
#else
   typedef double llr;
   #define Scale(x) ((x) * 0.625)
#endif

#include <assert.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include "hc.h"
static int Map(llr *OutBuf, const llr *InBuf, size_t Len);
static inx Overlap(const inx *Ptr1, inx Len1, const inx *Ptr2, inx Len2);

//***************************************************************************

void HCencode(
bit *SrcAndDstPtr,
const inx *CodeDefnArray)  // Format is described at the end of this file.
{
register bit *DataArray, Parity;
register const inx *DefnPtr, *End;
inx Len;
/**/
DataArray = SrcAndDstPtr;
DefnPtr = CodeDefnArray;
while((Len=*DefnPtr++) != 0)
{
End = DefnPtr + Len - 1;
Parity = 0;
while(DefnPtr < End)
   Parity ^= DataArray[*DefnPtr++];
DataArray[*DefnPtr++] = Parity;
} /* end while */
} /* end function */

//***************************************************************************

/*
Hard decisions are not made because the caller may
want hard decisions for all of the channel bits or for only the
```

```
information bits.

Scaling of the input log-likelihood ratios is unnecessary because
max-log-MAP processing is being used, as long as the scale factor
is the same for all of the samples.  In time-varying channels,
however, this will not be the case, and so scaling will be
required.  An example of a time-varying channel is a fading
channel.  The conversion from an antipodal signal sample to an LLR is
as follows:
llr = 4*(Ec/N0)*(1/sqrt(Ec)) * x;
where "x" is the channel sample.  This equation assumes that
zero and positive signal samples correspond with "0" bits, and that
negative signal samples correspond with "1" bits.  This requirement
arises from the standard definition of a LLR, which is ln(p(0)/p(1)).
*/

int HCdecode(                // Return value indicates convergence.
llr *InputAndOutputLlrs,     // Observe that there is no const qualifier.
llr *ExtrinsicArray,         // Must have length = number of extrinsics.
const inx *CodeDefnArray,    // Format is described at the end of this file.
unsigned NumCycles,          // Will stop early if converged, but will perform
                             // at least one cycle if this is non-zero.
int& ZeroTheExtrinsicFlag)   // Can use this and specify decoding cycles.
                             // This flag is cleared if set.  (for RWK)
{
static llr InBuf[MAX_LEN];
llr *LlrPtr, *LlrEnd;
llr *ExtrinsicPtr;

const inx *DefnPtr, *InxEnd;
inx Len;

unsigned Cycle;
int SomeEqnHasOddParity;

/**/

// Zero the extrinsic, if requested.
if(ZeroTheExtrinsicFlag)
{
LlrPtr = ExtrinsicArray;
DefnPtr = CodeDefnArray;
while((Len=*DefnPtr++) != 0)
{
assert(Len <= MAX_LEN);
LlrEnd = LlrPtr + Len;
while(LlrPtr < LlrEnd)
   *LlrPtr++ = 0;
DefnPtr += Len;
} /* end while */
ZeroTheExtrinsicFlag = 0;
} /* end if */

// Decode for the specified number of cycles, or until convergence is achieved.
SomeEqnHasOddParity = 1;
for(Cycle=0; SomeEqnHasOddParity && (Cycle < NumCycles); ++Cycle)
{
SomeEqnHasOddParity = 0;
DefnPtr=CodeDefnArray, ExtrinsicPtr=ExtrinsicArray;
```

```
while((Len=*DefnPtr++) != 0)
{
InxEnd = DefnPtr + Len;
LlrPtr = InBuf;
while(DefnPtr < InxEnd)
   *LlrPtr++ = InputAndOutputLlrs[*DefnPtr++] - *ExtrinsicPtr++;
ExtrinsicPtr -= Len;
SomeEqnHasOddParity = Map(ExtrinsicPtr, InBuf, Len) || SomeEqnHasOddParity;
// DO NOT CHANGE THE ORDER OF THE || OPERANDS IN THE ABOVE LINE!
LlrPtr=InBuf, DefnPtr-=Len;
while(DefnPtr < InxEnd)
   InputAndOutputLlrs[*DefnPtr++] = *LlrPtr++ + *ExtrinsicPtr++;
} /* end while */
} /* end for */

return !SomeEqnHasOddParity;  // i.e. true if converged.
} /* end function */

//***************************************************************************

/*
Basic checking of the code definition.  Call this function whenever
a new code definition is established.
*/

void HCsanity(
const inx *CodeDefnArray,  // Format is described at the end of this file.
unsigned NumChanBits,
unsigned NumInfoBits,
unsigned NumExtrinsics)
{
const inx *DefnPtr, *InxEnd;
inx Len, Val1;
const inx *DefnPtr2;
inx Len2;
int *ScratchArray, *ScratchPtr, *ScratchEnd;
unsigned Total, OnesCount, ManyCount;
int Defined;

/**/

// Basic argument checking.
assert(NumInfoBits <= NumChanBits);

// Allocate scratch memory (freed later).
ScratchArray = (int *)malloc(NumChanBits * sizeof(int));
if(!ScratchArray)
   fprintf(stderr,"ERROR HCsanity(): Unable to allocate memory.\n"), exit(1);
ScratchEnd = ScratchArray + NumChanBits;

// Sum of lengths must equal the number of extrinsics,
// and each length must be greater than 1.
Total = 0;
DefnPtr = CodeDefnArray;
while((Len=*DefnPtr++) != 0)
{
assert(Len > 1);
if(Len < 3 || Len > 64)
   fprintf(stderr,"WARNING HCsanity():  Suspicious equation length (%u).\n",
      (unsigned int)Len);
Total += Len, DefnPtr += Len;
} /* end while */
assert(Total == NumExtrinsics);

// All indices must be less than the number of channel bits.
// All indices from 0 to NumChanBits-1 should be used at least once.
ScratchPtr = ScratchArray;
while(ScratchPtr < ScratchEnd)
   *ScratchPtr++ = 0;
DefnPtr = CodeDefnArray;
while((Len=*DefnPtr++) != 0)
{
InxEnd = DefnPtr + Len;
while(DefnPtr < InxEnd)
   {
   assert(*DefnPtr < NumChanBits);
   ScratchArray[*DefnPtr++]++;
   } /* end while */
} /* end while */
OnesCount = 0, ManyCount = 0;
ScratchPtr = ScratchArray;
while(ScratchPtr < ScratchEnd)
   {
   assert(*ScratchPtr);
   if(*ScratchPtr == 1)
      OnesCount++;
   if(*ScratchPtr > 6)
      ManyCount++;
   ScratchPtr++;
   } /* end while */
if(OnesCount)
   fprintf(stderr,
   "WARNING HCsanity():  %u channel bits only involved in one equation.\n",
   OnesCount);
if(ManyCount)
   fprintf(stderr,
   "WARNING HCsanity(): %u channel bits involved in more than 6 equations.\n",
   ManyCount);

// No parity bit should have an index less than the number of info bits.
DefnPtr = CodeDefnArray;
while((Len=*DefnPtr++) != 0)
   DefnPtr += Len-1, assert(*DefnPtr++ >= NumInfoBits);

// The ordering of the equations must work for encoding (not only decoding).
ScratchPtr = ScratchArray;
ScratchEnd = ScratchArray + NumInfoBits;  // Temporarily.
while(ScratchPtr < ScratchEnd)
   *ScratchPtr++ = 1;
ScratchEnd = ScratchArray + NumChanBits;  // Restore to normal.
while(ScratchPtr < ScratchEnd)
   *ScratchPtr++ = 0;
DefnPtr = CodeDefnArray;
while((Len=*DefnPtr++) != 0)
{
Defined = 1;
InxEnd = DefnPtr + Len - 1;
while(DefnPtr < InxEnd)
```

```
   if(!ScratchArray[*DefnPtr++])
      Defined = 0;
 ScratchArray[*DefnPtr++] = Defined;
 } /* end while */
 ScratchPtr = ScratchArray;
 while(ScratchPtr < ScratchEnd)
    assert(*ScratchPtr++);

 // No two parity equations should overlap by more than one bit.
 Total = 0;
 DefnPtr = CodeDefnArray;
 while((Len=*DefnPtr++) !=0 )
 {
 DefnPtr2 = DefnPtr + Len;
 while((Len2=*DefnPtr2++) != 0)
 {
 if(Overlap(DefnPtr,Len, DefnPtr2, Len2) > 1)
    Total++;
 DefnPtr2 += Len2;
 } /* end while */
 DefnPtr += Len;
 } /* end while */
 if(Total)
    fprintf(stderr,
    "WARNING HCsanity():  %u equations overlap by more than 1 bit.\n",
    Total);

 // An index can only occur once in any given parity equation.
 Total = 0;
 DefnPtr = CodeDefnArray;
 while((Len=*DefnPtr++) != 0)
 {
 InxEnd = DefnPtr + Len;
 while(DefnPtr < InxEnd)
 {
 Val1 = *DefnPtr++;
 DefnPtr2 = DefnPtr;
 while(DefnPtr2 < InxEnd)
    if(*DefnPtr2++ == Val1)
       Total++;
 } /* end while */
 } /* end while */
 assert(!Total);

 // Free memory.
 free(ScratchArray);
 } /* end function */

 //**************************************************************************

 /*
 The return value indicates the parity of the input equation; non-zero is
 odd parity and zero is even parity.

 The output LLRs are the extrinsic information, not improved LLRs.
 */

 static int Map(
 llr *OutBuf,
const llr *InBuf,
size_t Len)
{
register llr Tmp;
register const llr *Ptr, *End;
register llr Min, Min2;
register const llr *Loc;
int Parity;
llr *Dst;

/**/

assert(Len >= 2);  // Otherwise you can't have two minimums.
End = InBuf + Len;

// Calculate Min, Loc, and Parity.
Ptr = InBuf;
Parity = 0;
if((Min=*Ptr) < 0)
   Min = -Min, Parity ^= 1;
Loc = Ptr++;
while(Ptr < End)
{
if((Tmp=*Ptr) < 0)
   Tmp = -Tmp, Parity ^= 1;
if(Tmp < Min)
   Min = Tmp, Loc = Ptr;
Ptr++;
} /* end while */

// Calculate Min2.
Min2 = Loc==InBuf ? InBuf[1] : *InBuf;
if(Min2 < 0)
   Min2 = -Min2;
for(Ptr=InBuf; Ptr < End; ++Ptr)
{
if((Tmp=*Ptr) < 0)
   Tmp = -Tmp;
if(Tmp < Min2 && Ptr != Loc)
   Min2 = Tmp;
} /* end for */

// Account for whether parity worked or not, and scale values.
if(Parity)
   Min = -Min, Min2 = -Min2;
Min = Scale(Min), Min2 = Scale(Min2);

// Fill output buffer.
Ptr = InBuf;
Dst = OutBuf;
while(Ptr < End)
   *Dst++ = *Ptr++ >= 0 ? Min : -Min;
OutBuf[Loc-InBuf] = *Loc >= 0 ? Min2 : -Min2;

return Parity;
} /* end function */

//*****************************************************************************
```

```
static inx Overlap(
const inx *Ptr1,
inx Len1,
const inx *Ptr2,
inx Len2)
{
const inx *End1, *End2;
inx Val1;
const inx *Tmp2;
inx Total;
/**/
Total = 0;
End1 = Ptr1 + Len1;
End2 = Ptr2 + Len2;
while(Ptr1 < End1)
{
Val1 = *Ptr1++;
Tmp2 = Ptr2;
while(Tmp2 < End2)
   if(*Tmp2++ == Val1)
      Total++;
} /* end while */
return Total;
} /* end function */

//****************************************************************************

/*

Documentation

1 Code definition
The code definition is a mixed array.  The first element is the length of
the first parity equation.  This is followed by the indices for the
first parity equation (there will be "length" indices).  The last index is
the position of the parity bit for the parity equation.  The indices begin
at 0, as always in "C".  After the first equation, the length of the second
equation is given, followed by the corresponding indices.  This is repeated
for all of the parity equations.  A length of 0 terminates the array.

1.1 Bit ordering
The information bits must come first, followed by the parity bits.  That is,
indices [0,NumInfoBits-1] are information bits, and
indices [NumInfoBits, NumChanBits-1] are parity bits.

1.2 Equation ordering
It is important that the equations are ordered so that they
can be used for encoding, and not only decoding.

1.3 Maximum block size.
The data type used to store the code definition will determine the
maximum block size that can be accommodated.  For example, with GNU C++
running on an Intel 386 platform, the "unsigned short" data type occupies
16 bits which means that, if this data type is used for the code definition,
the maximum block size that can be simulated is 65,536 channel bits.

*/
```

Appendix B

The Golay code in two dimensions

This appendix presents results for a code that is not a hyper-code. The code was created by applying the (24,12) extended Golay code in two dimensions, resulting in a (576,144) block code. This code was investigated early in the course of this research project. The relation to hyper-codes is that iterative max-log-MAP processing is used for decoding. The results are of interest in that they show the error-rate performance that can be achieved with a very short block, in situations where a low code rate is acceptable. Such a coding scheme could be useful, for example, in a CDMA cellular system where the data blocks must be short due to delay concerns, and the spread-spectrum modulation makes the low coding rate acceptable.

Figure B.1 shows the bit and frame error-rate performance for this "2D" Golay code. The code has a block size of only 144 information bits, a code rate of r=1/4, and a minimum distance of $d_{min} = 64$. Note that this minimum distance is equivalent, asymptotically, to a rate 1/2 code with a minimum distance of 32. This is very high for such a small block size. The figure shows that at an E_b/N_0 of only 1.5 dB, the frame error rate is better than 10^{-2}, and the bit error rate is better than 10^{-3}.

The decoder written to generate this data made use of the "max" approximation in implementing the MAP algorithm. This allowed various efficiency improvements to be incorporated as compared to a straight-forward MAP decoder. While the computational requirements of such an approach are still somewhat high for present-day VLSI technology, given the ongoing advances in semiconductor technology, such an approach may soon become quite viable. This implementation was coded entirely in 'C', without any assembler hand coding, yet was able to provide a throughput of better than 125 bps when run on a

E_b/N_0	Frame errors	FER	BER
0.25 dB	2202	3.6e-1	7.2e-2
0.50 dB	4329	2.2e-1	4.1e-2
0.75 dB	2510	1.1e-1	2.1e-2
1.00 dB	1088	5.1e-2	8.7e-3
1.25 dB	307	1.8e-2	3.0e-3
1.50 dB	1107	5.5e-3	8.7e-4
1.75 dB	653	1.4e-3	2.2e-4

Table B.1: Bit and frame error rates for an extended Golay code applied in two dimensions, decoded using max-log-MAP. Frame size = 144 information bits, rate = 1/4, decoding cycles = 5. The table shows the number of frame errors that were counted in order to give an indication of the statistical validity of the error rates reported. The variation in the number of frame errors counted is due to the simulations being run on different computers.

desktop PC (200 MHz PentiumPro). Of course, a hardware implementation could provide a much higher data rate. Also, in applications where the frame size must be small, the data rate is typically low, as in CDMA cellular systems.

The simulation results for the two-dimensional (extended) Golay code are given in Table B.1.

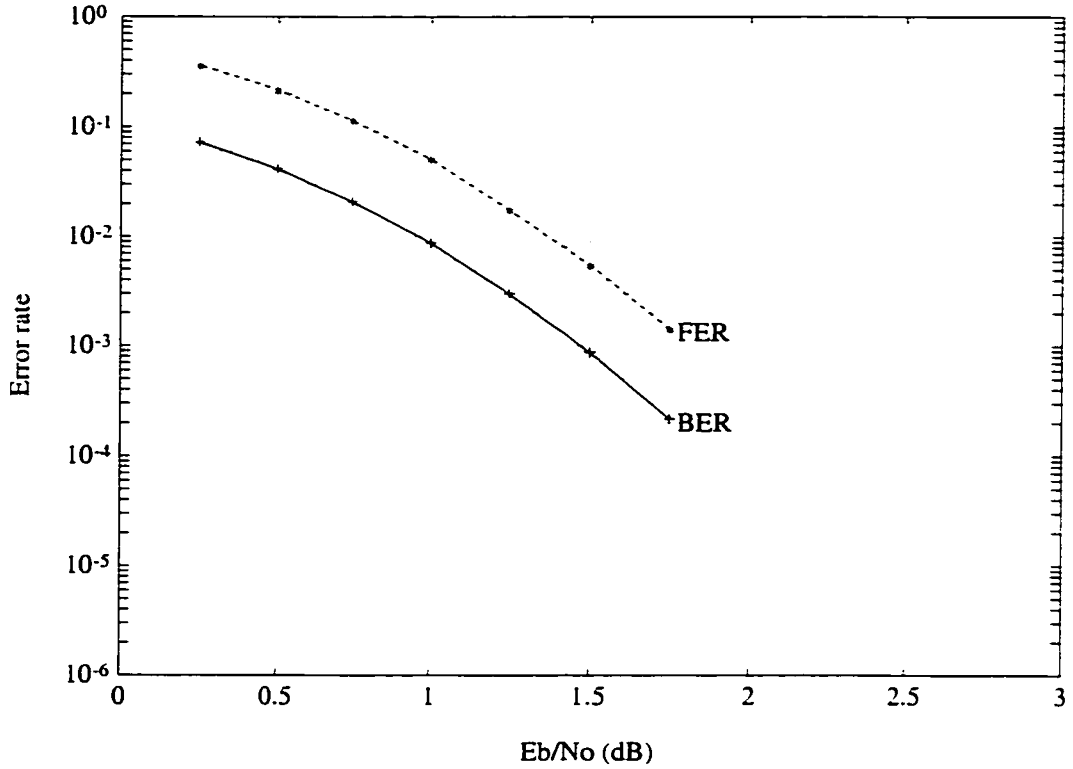

Figure B.1: Bit and frame error rates for an extended Golay code applied in two dimensions, decoded using max-log-MAP.

Vita

Personal Data

Name :	Andrew W. Hunt
Place and Date of Birth :	Guelph, Ontario, January 27, 1970

Post Secondary Education

1995-1997	Master of Engineering in Systems and Computer Engineering Carleton University Ottawa, Ontario
1988-1993	Bachelor of Applied Science in Electrical Engineering (Co-op) University of Waterloo Waterloo, Ontario

Work Experience

1993-1995	Signal Processing Engineer Algorithm design and implementation for satellite telephony applications. Gemtronics Engineering Nepean, Ontario
Co-op 6	VLSI Design Engineer Logic and SPICE simulation of full-custom digital VLSI integrated circuits. Cypress Semiconductor Starkville, Mississippi
Co-op 5	Research Assistant Software development for an electro-magnetic circuit simulation package. Department of Electrical Engineering McGill University Montréal, Québec
Co-op 3,4	Digital Hardware Engineer Programmable logic design and micro-controller programming. Hardware Design Lab University of Waterloo Waterloo, Ontario
Co-op 1,2	Test Engineer Bell-Northern Research Ottawa, Ontario

IMAGE EVALUATION
TEST TARGET (QA-3)